AOCL-Sparse

Release 4.2.0.0

Advanced Micro Devices, Inc

Dec 14, 2023

AOCL APIS

1 AOCL-Sparse Analysis Functions 3
2 AOCL-Sparse Auxiliary Functions 7
3 AOCL-Sparse Conversion Subprogram 25
4 AOCL-Sparse Level 1,2,3 Functions 39
5 AOCL-Sparse Iterative Linear System Solvers 125
6 AOCL-Sparse Types 137
7 Search the documentation 145

The AMD Optimized CPU Library AOCL-Sparse is a library that contains Basic Linear Algebra Subroutines for sparse matrices and vectors (Sparse BLAS) and is optimized for AMD EPYC and RYZEN family of CPU processors. It implements numerical algorithms in C++ while providing a public-facing C interface so it is can be used with C, C++ and compatible languages.

The current functionality of AOCL-Sparse is organized in the following categories:

- **Sparse Level 1** functions perform vector operations such as dot product, vector additions on sparse vectors, gather, scatter, and other similar operations.
- **Sparse Level 2** functions describe the operations between a matrix in a sparse format and a vector in the dense format, including matrix-vector product (SpMV), triangular solve (TRSV) and similar.
- **Sparse Level 3** functions describe the operations between a matrix in a sparse format and one or more dense/sparse matrices. The operations comprise of matrix additions (SpADD), matrix-matrix product (SpMM, Sp2M), and triangular solver with multiple right-hand sides (TRSM).
- **Iterative sparse solvers** based on Krylov subspace methods (CGM, GMRES) and preconditioners (such as, SymGS, ILU0).
- Sparse format conversion functions for translating matrices in a variety of sparse storage formats.
- Auxiliary functions to allow basic operations, including create, copy, destroy and modify matrix handles and descriptors.

Additional highlights:

- Supported data types: single, double, and the complex variants
- 0-based and 1-based indexing of sparse formats
- **Hint & Optimize framework** to accelerate supported functions by a prior matrix analysis based on users' hints of expected operations.

CHAPTER

ONE

AOCL-SPARSE ANALYSIS FUNCTIONS

aoclsparse_status **aoclsparse_optimize**(*aoclsparse_matrix* mat)

Performs data allocations and restructuring operations related to sparse matrices.

`aoclsparse_optimize` Sparse matrices are restructured based on matrix analysis, into different storage formats to improve data access and thus performance.

Parameters

mat – **[in]** sparse matrix in CSR format and sparse format information inside

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` is invalid.
- **aoclsparse_status_invalid_pointer** –
- **aoclsparse_status_internal_error** – an internal error occurred.

aoclsparse_status **aoclsparse_set_mv_hint**(*aoclsparse_matrix* mat, *aoclsparse_operation* trans, const *aoclsparse_mat_descr* descr, *aoclsparse_int* expected_no_of_calls)

Provides any hints such as the type of routine, expected no of calls etc.

`aoclsparse_set_mv_hint` sets a hint id for analysis and execute phases of the program to analyse and perform ILU factorization and Solution

Parameters

- **mat** – **[in]** sparse matrix in CSR format and sparse format information inside
- **trans** – **[in]** Whether in transposed state or not. Transpose operation is not yet supported.
- **descr** – **[in]** descriptor of the sparse CSR matrix. Currently, only *aoclsparse_matrix_type_general* and *aoclsparse_matrix_type_symmetric* is supported.
- **expected_no_of_calls** – **[in]** unused parameter

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` is invalid.
- **aoclsparse_status_invalid_pointer** –
- **aoclsparse_status_internal_error** – an internal error occurred.

aoclsparse_status **aoclsparse_set_sv_hint**(*aoclsparse_matrix* mat, *aoclsparse_operation* trans, const *aoclsparse_mat_descr* descr, *aoclsparse_int* expected_no_of_calls) *aoclsparse_status* **aoclsparse_set_mm_hint**(*aoclsparse_matrix* mat, *aoclsparse_operation* trans, const *aoclsparse_mat_descr* descr, *aoclsparse_int* expected_no_of_calls)

aoclsparse_status **aoclsparse_set_2m_hint**(*aoclsparse_matrix* mat, *aoclsparse_operation* trans, const *aoclsparse_mat_descr* descr, *aoclsparse_int* expected_no_of_calls)

aoclsparse_status **aoclsparse_set_lu_smoother_hint**(*aoclsparse_matrix* mat, *aoclsparse_operation* trans, const *aoclsparse_mat_descr* descr, *aoclsparse_int* expected_no_of_calls)

Provides any hints such as the type of routine, expected no of calls etc.

`aoclsparse_set_lu_smoother_hint` sets a hint id for analysis and execute phases of the program to analyse and perform ILU factorization and Solution

Parameters

- **mat** – **[in]** A sparse matrix and ILU related information inside
- **trans** – **[in]** Whether in transposed state or not. Transpose operation is not yet supported.
- **descr** – **[in]** Descriptor of the sparse matrix.
- **expected_no_of_calls** – **[in]** unused parameter

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – indicates that `m` is invalid, expecting m>=0.
- **aoclsparse_status_invalid_pointer.** –
- **aoclsparse_status_internal_error** – Indicates that an internal error occurred.

aoclsparse_status **aoclsparse_set_sm_hint**(*aoclsparse_matrix* A, *aoclsparse_operation* trans, const *aoclsparse_mat_descr* descr, const *aoclsparse_order* order, const *aoclsparse_int* dense_matrix_dim, const *aoclsparse_int* expected_no_of_calls)

Store user-hints to accelerate the `aoclsparse_?trsm` triangular-solvers.

This function stores user-provided hints related to the structures of the matrices involved in a triangular linear system of equations and its solvers. The hints are for the problem

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

where A is a sparse matrix, $op()$ is a linear operator, X and B are dense matrices, while $alpha$ is a scalar. The hints are used in order to perform certain optimizations over the input data that can potentially accelerate the solve operation. The hints include, expected number of calls to the API, matrix layout, dimension of dense right-hand-side matrix, etc.

Parameters

- **A** – **[in]** A sparse matrix A.
- **trans** – **[in]** Operation to perform on the sparse matrix A, valid options are *aoclsparse_operation_none*, aoclsparse_operation_transpose, and aoclsparse_operation_conjugate_transpose.
- **descr** – **[in]** Descriptor of the sparse matrix A.
- **order** – **[in]** Layout of the right-hand-side matrix B, valid options are *aoclsparse_order_row* and *aoclsparse_order_column*.

- **dense_matrix_dim** – **[in]** number of columns of the dense matrix B.
- **expected_no_of_calls** – **[in]** Hint on the potential number of calls to the solver API, e.g., calls to *aoclsparse_strsm()*.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m, n, nnz, ldb or ldx is invalid. Expecting m>0, n>0, m==n, nnz>0, ldb>=n, ldx>=n
- **aoclsparse_status_invalid_value** – Sparse matrix is not square, or expected_no_of_calls or dense_matrix_dim or matrix type are invalid.
- **aoclsparse_status_invalid_pointer** – Pointers to sparse matrix A or dense matrices B or X or descriptor are null
- **aoclsparse_status_internal_error** – Indicates that an internal error occurred.

CHAPTER

TWO

AOCL-SPARSE AUXILIARY FUNCTIONS

const char ***aoclsparse_get_version**()

Get AOCL-Sparse version.

`aoclsparse_get_version` gets the aoclsparse library version number. in the format "AOCL-Sparse <major>.<minor>.<patch>"

Parameters

version – **[out]** the version string of the aoclsparse library.

aoclsparse_status **aoclsparse_create_mat_descr**(*aoclsparse_mat_descr* *descr)

Create a matrix descriptor.

`aoclsparse_create_mat_descr` creates a matrix descriptor. It initializes *aoclsparse_matrix_type* to *aoclsparse_matrix_type_general* and *aoclsparse_index_base* to *aoclsparse_index_base_zero*. It should be destroyed at the end using *aoclsparse_destroy_mat_descr()*.

Parameters

descr – **[out]** the pointer to the matrix descriptor.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `descr` pointer is invalid.

aoclsparse_status **aoclsparse_copy_mat_descr**(*aoclsparse_mat_descr* dest, const *aoclsparse_mat_descr* src)

Copy a matrix descriptor.

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

Parameters

- **dest** – **[out]** the pointer to the destination matrix descriptor.
- **src** – **[in]** the pointer to the source matrix descriptor.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `src` or `dest` pointer is invalid.

aoclsparse_status **aoclsparse_destroy_mat_descr**(*aoclsparse_mat_descr* descr)

Destroy a matrix descriptor.

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

Parameters
`descr` – **[in]** the matrix descriptor.

Return values

- `aoclsparse_status_success` – the operation completed successfully.
- `aoclsparse_status_invalid_pointer` – `descr` is invalid.

aoclsparse_status **`aoclsparse_set_mat_index_base`**(*aoclsparse_mat_descr* descr, *aoclsparse_index_base* base)

Specify the index base of a matrix descriptor.

`aoclsparse_set_mat_index_base` sets the index base of a matrix descriptor. Valid options are *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.

Parameters

- `descr` – **[inout]** the matrix descriptor.
- `base` – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.

Return values

- `aoclsparse_status_success` – the operation completed successfully.
- `aoclsparse_status_invalid_pointer` – `descr` pointer is invalid.
- `aoclsparse_status_invalid_value` – `base` is invalid.

aoclsparse_index_base **`aoclsparse_get_mat_index_base`**(const *aoclsparse_mat_descr* descr)

Get the index base of a matrix descriptor.

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

Parameters
`descr` – **[in]** the matrix descriptor.

Returns
aoclsparse_index_base_zero or *aoclsparse_index_base_one*.

aoclsparse_status **`aoclsparse_set_mat_type`**(*aoclsparse_mat_descr* descr, *aoclsparse_matrix_type* type)

Specify the matrix type of a matrix descriptor.

`aoclsparse_set_mat_type` sets the matrix type of a matrix descriptor. Valid matrix types are *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_symmetric*, *aoclsparse_matrix_type_hermitian* or *aoclsparse_matrix_type_triangular*.

Parameters

- `descr` – **[inout]** the matrix descriptor.
- `type` – **[in]** *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_symmetric*, *aoclsparse_matrix_type_hermitian* or *aoclsparse_matrix_type_triangular*.

Return values

- `aoclsparse_status_success` – the operation completed successfully.
- `aoclsparse_status_invalid_pointer` – `descr` pointer is invalid.
- `aoclsparse_status_invalid_value` – `type` is invalid.

aoclsparse_matrix_type **`aoclsparse_get_mat_type`**(const *aoclsparse_mat_descr* descr)

Get the matrix type of a matrix descriptor.

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

Parameters
descr – **[in]** the matrix descriptor.

Returns
aoclsparse_matrix_type_general, *aoclsparse_matrix_type_symmetric*, *aoclsparse_matrix_type_hermitian* or *aoclsparse_matrix_type_triangular*.

aoclsparse_status **aoclsparse_set_mat_fill_mode**(*aoclsparse_mat_descr* descr, *aoclsparse_fill_mode* fill_mode)

Specify the matrix fill mode of a matrix descriptor.

`aoclsparse_set_mat_fill_mode` sets the matrix fill mode of a matrix descriptor. Valid fill modes are *aoclsparse_fill_mode_lower* or *aoclsparse_fill_mode_upper*.

Parameters

- **descr** – **[inout]** the matrix descriptor.
- **fill_mode** – **[in]** *aoclsparse_fill_mode_lower* or *aoclsparse_fill_mode_upper*.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `descr` pointer is invalid.
- **aoclsparse_status_invalid_value** – `fill\_mode` is invalid.

aoclsparse_fill_mode **aoclsparse_get_mat_fill_mode**(const *aoclsparse_mat_descr* descr)

Get the matrix fill mode of a matrix descriptor.

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

Parameters
descr – **[in]** the matrix descriptor.

Returns
aoclsparse_fill_mode_lower or *aoclsparse_fill_mode_upper*.

aoclsparse_status **aoclsparse_set_mat_diag_type**(*aoclsparse_mat_descr* descr, *aoclsparse_diag_type* diag_type)

Specify the matrix diagonal type of a matrix descriptor.

`aoclsparse_set_mat_diag_type` sets the matrix diagonal type of a matrix descriptor. Valid diagonal types are *aoclsparse_diag_type_unit*, *aoclsparse_diag_type_non_unit* or *aoclsparse_diag_type_zero*.

Parameters

- **descr** – **[inout]** the matrix descriptor.
- **diag_type** – **[in]** *aoclsparse_diag_type_unit* or *aoclsparse_diag_type_non_unit* or *aoclsparse_diag_type_zero*.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `descr` pointer is invalid.
- **aoclsparse_status_invalid_value** – `diag_type` is invalid.

aoclsparse_diag_type **aoclsparse_get_mat_diag_type**(const *aoclsparse_mat_descr* descr)

Get the matrix diagonal type of a matrix descriptor.

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

Parameters
descr – **[in]** the matrix descriptor.

Returns
aoclsparse_diag_type_unit or *aoclsparse_diag_type_non_unit* or *aoclsparse_diag_type_zero*.

aoclsparse_status **aoclsparse_create_scsr**(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr, *aoclsparse_int* *col_idx, float *val)

Creates a new `aoclsparse_matrix` based on CSR (Compressed Sparse Row) format.

`aoclsparse_create_(s/d/c/z)csr` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of column indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. To convert any other format to CSR, `aoclsparse_convert` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[out]** the pointer to the CSR sparse matrix allocated in the API.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** number of rows of the sparse CSR matrix.
- **N** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **col_idx** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – at least one of `row_ptr`, `col_idx` or `val` pointer is NULL.
- **aoclsparse_status_invalid_size** – at least one of `M`, `N` or `nnz` has a negative value.
- **aoclsparse_status_invalid_index_value** – any `col_idx` value is not within N.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_dcsr**(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr, *aoclsparse_int* *col_idx, double *val)

Creates a new `aoclsparse_matrix` based on CSR (Compressed Sparse Row) format.

`aoclsparse_create_(s/d/c/z)csr` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of column indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. To convert any other format to CSR, `aoclsparse_convert` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[out]** the pointer to the CSR sparse matrix allocated in the API.

- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** number of rows of the sparse CSR matrix.
- **N** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **col_idx** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – at least one of `row_ptr`, `col_idx` or `val` pointer is NULL.
- **aoclsparse_status_invalid_size** – at least one of `M`, `N` or `nnz` has a negative value.
- **aoclsparse_status_invalid_index_value** – any `col_idx` value is not within N.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_ccsr**(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr, *aoclsparse_int* *col_idx, aoclsparse_float_complex *val)

Creates a new `aoclsparse_matrix` based on CSR (Compressed Sparse Row) format.

`aoclsparse_create_(s/d/c/z)csr` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of column indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. To convert any other format to CSR, `aoclsparse_convert` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[out]** the pointer to the CSR sparse matrix allocated in the API.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** number of rows of the sparse CSR matrix.
- **N** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **col_idx** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – at least one of `row_ptr`, `col_idx` or `val` pointer is NULL.

- **aoclsparse_status_invalid_size** – at least one of `M`, `N` or `nnz` has a negative value.
- **aoclsparse_status_invalid_index_value** – any `col_idx` value is not within N.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_zcsr**(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr, *aoclsparse_int* *col_idx, aoclsparse_double_complex *val)

Creates a new `aoclsparse_matrix` based on CSR (Compressed Sparse Row) format.

`aoclsparse_create_(s/d/c/z)csr` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of column indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. To convert any other format to CSR, `aoclsparse_convert` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[out]** the pointer to the CSR sparse matrix allocated in the API.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** number of rows of the sparse CSR matrix.
- **N** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **col_idx** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – at least one of `row_ptr`, `col_idx` or `val` pointer is NULL.
- **aoclsparse_status_invalid_size** – at least one of `M`, `N` or `nnz` has a negative value.
- **aoclsparse_status_invalid_index_value** – any `col_idx` value is not within N.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_scoo**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ind, *aoclsparse_int* *col_ind, float *val)

Creates a new `aoclsparse_matrix` based on COO (Co-ordinate format).

`aoclsparse_create_(s/d/c/z)coo` creates `aoclsparse_matrix` and initializes it with input parameters passed. Array data must not be modified by the user while matrix is alive as the pointers are copied, not the data. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[inout]** the pointer to the COO sparse matrix.

- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one* depending on whether the index first element starts from 0 or 1.
- **M** – **[in]** total number of rows of the sparse COO matrix.
- **N** – **[in]** total number of columns of the sparse COO matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse COO matrix.
- **row_ind** – **[in]** array of `nnz` elements that point to the row of the element in co-ordinate Format.
- **col_ind** – **[in]** array of `nnz` elements that point to the column of the element in co-ordinate Format.
- **val** – **[in]** array of `nnz` elements of the sparse COO matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – pointer given to API is invalid or nullptr.
- **aoclsparse_status_invalid_size** – `coo` dimension of matrix or non-zero elements is invalid.
- **aoclsparse_status_invalid_index_value** – index given for `coo` is out of matrix bounds depending on base given
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_dcoo**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ind, *aoclsparse_int* *col_ind, double *val)

Creates a new `aoclsparse_matrix` based on COO (Co-ordinate format).

`aoclsparse_create_(s/d/c/z)coo` creates `aoclsparse_matrix` and initializes it with input parameters passed. Array data must not be modified by the user while matrix is alive as the pointers are copied, not the data. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[inout]** the pointer to the COO sparse matrix.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one* depending on whether the index first element starts from 0 or 1.
- **M** – **[in]** total number of rows of the sparse COO matrix.
- **N** – **[in]** total number of columns of the sparse COO matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse COO matrix.
- **row_ind** – **[in]** array of `nnz` elements that point to the row of the element in co-ordinate Format.
- **col_ind** – **[in]** array of `nnz` elements that point to the column of the element in co-ordinate Format.
- **val** – **[in]** array of `nnz` elements of the sparse COO matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – pointer given to API is invalid or nullptr.

- **aoclsparse_status_invalid_size** – coo dimension of matrix or non-zero elements is invalid.
- **aoclsparse_status_invalid_index_value** – index given for coo is out of matrix bounds depending on base given
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_ccoo**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ind, *aoclsparse_int* *col_ind, aoclsparse_float_complex *val)

Creates a new `aoclsparse_matrix` based on COO (Co-ordinate format).

`aoclsparse_create_(s/d/c/z)coo` creates `aoclsparse_matrix` and initializes it with input parameters passed. Array data must not be modified by the user while matrix is alive as the pointers are copied, not the data. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[inout]** the pointer to the COO sparse matrix.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one* depending on whether the index first element starts from 0 or 1.
- **M** – **[in]** total number of rows of the sparse COO matrix.
- **N** – **[in]** total number of columns of the sparse COO matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse COO matrix.
- **row_ind** – **[in]** array of `nnz` elements that point to the row of the element in co-ordinate Format.
- **col_ind** – **[in]** array of `nnz` elements that point to the column of the element in co-ordinate Format.
- **val** – **[in]** array of `nnz` elements of the sparse COO matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – pointer given to API is invalid or nullptr.
- **aoclsparse_status_invalid_size** – coo dimension of matrix or non-zero elements is invalid.
- **aoclsparse_status_invalid_index_value** – index given for coo is out of matrix bounds depending on base given
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_zcoo**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ind, *aoclsparse_int* *col_ind, aoclsparse_double_complex *val)

Creates a new `aoclsparse_matrix` based on COO (Co-ordinate format).

`aoclsparse_create_(s/d/c/z)coo` creates `aoclsparse_matrix` and initializes it with input parameters passed. Array data must not be modified by the user while matrix is alive as the pointers are copied, not the data. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[inout]** the pointer to the COO sparse matrix.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one* depending on whether the index first element starts from 0 or 1.
- **M** – **[in]** total number of rows of the sparse COO matrix.
- **N** – **[in]** total number of columns of the sparse COO matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse COO matrix.
- **row_ind** – **[in]** array of `nnz` elements that point to the row of the element in co-ordinate Format.
- **col_ind** – **[in]** array of `nnz` elements that point to the column of the element in co-ordinate Format.
- **val** – **[in]** array of `nnz` elements of the sparse COO matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – pointer given to API is invalid or nullptr.
- **aoclsparse_status_invalid_size** – `coo` dimension of matrix or non-zero elements is invalid.
- **aoclsparse_status_invalid_index_value** – index given for `coo` is out of matrix bounds depending on base given
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_export_scsr**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **row_ptr, *aoclsparse_int* **col_ind, float **val)

Export a CSR matrix.

`aoclsparse_export_(s/d/c/z)csr` exposes the components defining the CSR matrix in `mat` structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once `aoclsparse_destroy()` is called to free `mat`, these arrays will become inaccessible. If the matrix is not in CSR format, an error is obtained. *aoclsparse_convert_csr* can be used to convert non-CSR format to CSR format.

Parameters

- **mat** – **[in]** the pointer to the CSR sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSR matrix.
- **n** – **[out]** number of columns of the sparse CSR matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[out]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **col_ind** – **[out]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **val** – **[out]** array of `nnz` elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.

- **aoclsparse_status_invalid_pointer** – `mat` or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – `mat` is not in CSR format.
- **aoclsparse_status_wrong_type** – data type of `mat` does not match the function.

aoclsparse_status **aoclsparse_export_dcsr**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **row_ptr, *aoclsparse_int* **col_ind, double **val)

Export a CSR matrix.

`aoclsparse_export_(s/d/c/z)csr` exposes the components defining the CSR matrix in `mat` structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once `aoclsparse_destroy()` is called to free `mat`, these arrays will become inaccessible. If the matrix is not in CSR format, an error is obtained. *aoclsparse_convert_csr* can be used to convert non-CSR format to CSR format.

Parameters

- **mat** – **[in]** the pointer to the CSR sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSR matrix.
- **n** – **[out]** number of columns of the sparse CSR matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[out]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **col_ind** – **[out]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **val** – **[out]** array of `nnz` elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `mat` or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – `mat` is not in CSR format.
- **aoclsparse_status_wrong_type** – data type of `mat` does not match the function.

aoclsparse_status **aoclsparse_export_ccsr**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **row_ptr, *aoclsparse_int* **col_ind, aoclsparse_float_complex **val)

Export a CSR matrix.

`aoclsparse_export_(s/d/c/z)csr` exposes the components defining the CSR matrix in `mat` structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once `aoclsparse_destroy()` is called to free `mat`, these arrays will become inaccessible. If the matrix is not in CSR format, an error is obtained. *aoclsparse_convert_csr* can be used to convert non-CSR format to CSR format.

Parameters

- **mat** – **[in]** the pointer to the CSR sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSR matrix.
- **n** – **[out]** number of columns of the sparse CSR matrix.

- **nnz** – **[out]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[out]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **col_ind** – **[out]** array of nnz elements containing the column indices of the sparse CSR matrix.
- **val** – **[out]** array of nnz elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – mat or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – mat is not in CSR format.
- **aoclsparse_status_wrong_type** – data type of mat does not match the function.

aoclsparse_status **aoclsparse_export_zcsr**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **row_ptr, *aoclsparse_int* **col_ind, aoclsparse_double_complex **val)

Export a CSR matrix.

aoclsparse_export_(s/d/c/z)csr exposes the components defining the CSR matrix in mat structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once *aoclsparse_destroy()* is called to free mat, these arrays will become inaccessible. If the matrix is not in CSR format, an error is obtained. *aoclsparse_convert_csr* can be used to convert non-CSR format to CSR format.

Parameters

- **mat** – **[in]** the pointer to the CSR sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSR matrix.
- **n** – **[out]** number of columns of the sparse CSR matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[out]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **col_ind** – **[out]** array of nnz elements containing the column indices of the sparse CSR matrix.
- **val** – **[out]** array of nnz elements of the sparse CSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – mat or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – mat is not in CSR format.
- **aoclsparse_status_wrong_type** – data type of mat does not match the function.

aoclsparse_status **aoclsparse_destroy**(*aoclsparse_matrix* *mat)

Destroy a sparse matrix structure.

aoclsparse_destroy destroys a structure that holds the matrix

Parameters
`mat` – **[in]** the pointer to the sparse matrix.

Return values

- `aoclsparse_status_success` – the operation completed successfully.
- `aoclsparse_status_invalid_pointer` – `matrix` structure pointer is invalid.

aoclsparse_status `aoclsparse_create_scsc`(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *col_ptr, *aoclsparse_int* *row_idx, float *val)

Creates a new `aoclsparse_matrix` based on CSC (Compressed Sparse Column) format.

`aoclsparse_create_(s/d/c/z)csc` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of row indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- `mat` – **[inout]** the pointer to the CSC sparse matrix allocated in the API.
- `base` – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- `M` – **[in]** number of rows of the sparse CSC matrix.
- `N` – **[in]** number of columns of the sparse CSC matrix.
- `nnz` – **[in]** number of non-zero entries of the sparse CSC matrix.
- `col_ptr` – **[in]** array of `n+1` elements that points to the start of every column in row_idx array of the sparse CSC matrix.
- `row_idx` – **[in]** array of `nnz` elements containing the row indices of the sparse CSC matrix.
- `val` – **[in]** array of `nnz` elements of the sparse CSC matrix.

Return values

- `aoclsparse_status_success` – the operation completed successfully.
- `aoclsparse_status_invalid_pointer` – `col_ptr`, `row_idx` or `val` pointer is NULL.
- `aoclsparse_status_invalid_size` – `M`, `N` or `nnz` are negative values.
- `aoclsparse_status_invalid_index_value` – any `row_idx` value is not within M.
- `aoclsparse_status_memory_error` – memory allocation for matrix failed.

aoclsparse_status `aoclsparse_create_dcsc`(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *col_ptr, *aoclsparse_int* *row_idx, double *val)

Creates a new `aoclsparse_matrix` based on CSC (Compressed Sparse Column) format.

`aoclsparse_create_(s/d/c/z)csc` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of row indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- `mat` – **[inout]** the pointer to the CSC sparse matrix allocated in the API.
- `base` – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.

- **M** – **[in]** number of rows of the sparse CSC matrix.
- **N** – **[in]** number of columns of the sparse CSC matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[in]** array of `n+1` elements that points to the start of every column in row_idx array of the sparse CSC matrix.
- **row_idx** – **[in]** array of `nnz` elements containing the row indices of the sparse CSC matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `col_ptr`, `row_idx` or `val` pointer is NULL.
- **aoclsparse_status_invalid_size** – `M`, `N` or `nnz` are negative values.
- **aoclsparse_status_invalid_index_value** – any `row_idx` value is not within M.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_ccsc**(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *col_ptr, *aoclsparse_int* *row_idx, aoclsparse_float_complex *val)

Creates a new `aoclsparse_matrix` based on CSC (Compressed Sparse Column) format.

`aoclsparse_create_(s/d/c/z)csc` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of row indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[inout]** the pointer to the CSC sparse matrix allocated in the API.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** number of rows of the sparse CSC matrix.
- **N** – **[in]** number of columns of the sparse CSC matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[in]** array of `n+1` elements that points to the start of every column in row_idx array of the sparse CSC matrix.
- **row_idx** – **[in]** array of `nnz` elements containing the row indices of the sparse CSC matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `col_ptr`, `row_idx` or `val` pointer is NULL.
- **aoclsparse_status_invalid_size** – `M`, `N` or `nnz` are negative values.
- **aoclsparse_status_invalid_index_value** – any `row_idx` value is not within M.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_create_zcsc**(*aoclsparse_matrix* *mat, *aoclsparse_index_base* base, *aoclsparse_int* M, *aoclsparse_int* N, *aoclsparse_int* nnz, *aoclsparse_int* *col_ptr, *aoclsparse_int* *row_idx, aoclsparse_double_complex *val)

Creates a new `aoclsparse_matrix` based on CSC (Compressed Sparse Column) format.

`aoclsparse_create_(s/d/c/z)csc` creates `aoclsparse_matrix` and initializes it with input parameters passed. The input arrays are left unchanged except for the call to `aoclsparse_order_mat`, which performs ordering of row indices of the matrix. To avoid any changes to the input data, `aoclsparse_copy` can be used. Matrix should be destroyed at the end using `aoclsparse_destroy`.

Parameters

- **mat** – **[inout]** the pointer to the CSC sparse matrix allocated in the API.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** number of rows of the sparse CSC matrix.
- **N** – **[in]** number of columns of the sparse CSC matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[in]** array of `n+1` elements that points to the start of every column in row_idx array of the sparse CSC matrix.
- **row_idx** – **[in]** array of `nnz` elements containing the row indices of the sparse CSC matrix.
- **val** – **[in]** array of `nnz` elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `col_ptr`, `row_idx` or `val` pointer is NULL.
- **aoclsparse_status_invalid_size** – `M`, `N` or `nnz` are negative values.
- **aoclsparse_status_invalid_index_value** – any `row_idx` value is not within M.
- **aoclsparse_status_memory_error** – memory allocation for matrix failed.

aoclsparse_status **aoclsparse_copy**(const *aoclsparse_matrix* src, const *aoclsparse_mat_descr* descr, *aoclsparse_matrix* *dest)

Creates a copy of source `aoclsparse_matrix`.

`aoclsparse_copy` creates a deep copy of source `aoclsparse_matrix` (hints and optimized data are not copied). Matrix should be destroyed using *aoclsparse_destroy()*. *aoclsparse_convert_csr()* can also be used to create a copy of the source matrix while converting it in CSR format.

Parameters

- **src** – **[in]** the source `aoclsparse_matrix` to copy.
- **descr** – **[in]** the source matrix descriptor, this argument is reserved for future releases and it will not be referenced.
- **dest** – **[out]** pointer to the newly allocated copied `aoclsparse_matrix`.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `src`, `dest` or internal pointers are NULL or dest points to src.

- **aoclsparse_status_memory_error** – memory allocation for matrix failed.
- **aoclsparse_status_invalid_value** – src matrix type is invalid.
- **aoclsparse_status_wrong_type** – src matrix data type is invalid.

aoclsparse_status **aoclsparse_order_mat**(*aoclsparse_matrix* mat)

Performs ordering of index array of the matrix.

aoclsparse_order orders column indices within a row for matrix in CSR format and row indices within a column for CSC format. It also adjusts value array accordingly. Ordering is implemented only for CSR and CSC format. aoclsparse_copy can be used to get exact copy of data aoclsparse_convert can be used to convert any format to CSR. Matrix should be destroyed at the end using aoclsparse_destroy.

Parameters

mat – **[inout]** pointer to matrix in either CSR or CSC format

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – mat pointer is NULL.
- **aoclsparse_status_memory_error** – internal memory allocation failed.
- **aoclsparse_status_not_implemented** – matrix is not in CSR format.

aoclsparse_status **aoclsparse_export_scsc**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **col_ptr, *aoclsparse_int* **row_ind, float **val)

Export CSC matrix.

aoclsparse_export_(s/d/c/z)csc exposes the components defining the CSC matrix in mat structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once *aoclsparse_destroy()* is called to free mat, these arrays will become inaccessible. If the matrix is not in CSC format, an error is obtained.

Parameters

- **mat** – **[in]** the pointer to the CSC sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSC matrix.
- **n** – **[out]** number of columns of the sparse CSC matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[out]** array of n+1 elements that point to the start of every col of the sparse CSC matrix.
- **row_ind** – **[out]** array of nnz elements containing the row indices of the sparse CSC matrix.
- **val** – **[out]** array of nnz elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – mat or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – mat is not in CSC format.
- **aoclsparse_status_wrong_type** – data type of mat does not match the function.

aoclsparse_status **aoclsparse_export_dcsc**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **col_ptr, *aoclsparse_int* **row_ind, double **val)

Export CSC matrix.

`aoclsparse_export_(s/d/c/z)csc` exposes the components defining the CSC matrix in `mat` structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once `aoclsparse_destroy()` is called to free `mat`, these arrays will become inaccessible. If the matrix is not in CSC format, an error is obtained.

Parameters

- **mat** – **[in]** the pointer to the CSC sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSC matrix.
- **n** – **[out]** number of columns of the sparse CSC matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[out]** array of `n+1` elements that point to the start of every col of the sparse CSC matrix.
- **row_ind** – **[out]** array of `nnz` elements containing the row indices of the sparse CSC matrix.
- **val** – **[out]** array of `nnz` elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `mat` or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – `mat` is not in CSC format.
- **aoclsparse_status_wrong_type** – data type of `mat` does not match the function.

aoclsparse_status **aoclsparse_export_ccsc**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **col_ptr, *aoclsparse_int* **row_ind, aoclsparse_float_complex **val)

Export CSC matrix.

`aoclsparse_export_(s/d/c/z)csc` exposes the components defining the CSC matrix in `mat` structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once `aoclsparse_destroy()` is called to free `mat`, these arrays will become inaccessible. If the matrix is not in CSC format, an error is obtained.

Parameters

- **mat** – **[in]** the pointer to the CSC sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSC matrix.
- **n** – **[out]** number of columns of the sparse CSC matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[out]** array of `n+1` elements that point to the start of every col of the sparse CSC matrix.
- **row_ind** – **[out]** array of `nnz` elements containing the row indices of the sparse CSC matrix.

- **val** – **[out]** array of nnz elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – mat or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – mat is not in CSC format.
- **aoclsparse_status_wrong_type** – data type of mat does not match the function.

aoclsparse_status **aoclsparse_export_zcsc**(const *aoclsparse_matrix* mat, *aoclsparse_index_base* *base, *aoclsparse_int* *m, *aoclsparse_int* *n, *aoclsparse_int* *nnz, *aoclsparse_int* **col_ptr, *aoclsparse_int* **row_ind, aoclsparse_double_complex **val)

Export CSC matrix.

aoclsparse_export_(s/d/c/z)csc exposes the components defining the CSC matrix in mat structure by copying out the data pointers. No additional memory is allocated. User should not modify the arrays and once *aoclsparse_destroy()* is called to free mat, these arrays will become inaccessible. If the matrix is not in CSC format, an error is obtained.

Parameters

- **mat** – **[in]** the pointer to the CSC sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse CSC matrix.
- **n** – **[out]** number of columns of the sparse CSC matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSC matrix.
- **col_ptr** – **[out]** array of n+1 elements that point to the start of every col of the sparse CSC matrix.
- **row_ind** – **[out]** array of nnz elements containing the row indices of the sparse CSC matrix.
- **val** – **[out]** array of nnz elements of the sparse CSC matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – mat or any of the output arguments are NULL.
- **aoclsparse_status_invalid_value** – mat is not in CSC format.
- **aoclsparse_status_wrong_type** – data type of mat does not match the function.

CHAPTER

THREE

AOCL-SPARSE CONVERSION SUBPROGRAM

`aoclsparse_convert.h` provides sparse format conversion functions.

aoclsparse_status **aoclsparse_csr2ell_width**(*aoclsparse_int* m, *aoclsparse_int* nnz, const *aoclsparse_int* *csr_row_ptr, *aoclsparse_int* *ell_width)

Convert a sparse CSR matrix into a sparse ELL matrix.

`aoclsparse_csr2ell_width` computes the maximum of the per row non-zero elements over all rows, the ELL `width`, for a given CSR matrix.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **ell_width** – **[out]** pointer to the number of non-zero elements per row in ELL storage format.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` is invalid.
- **aoclsparse_status_invalid_pointer** – `csr\_row\_ptr`, or `ell\_width` pointer is invalid.
- **aoclsparse_status_internal_error** – an internal error occurred.

aoclsparse_status **aoclsparse_scsr2ell**(*aoclsparse_int* m, const *aoclsparse_mat_descr* descr, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const float *csr_val, *aoclsparse_int* *ell_col_ind, float *ell_val, *aoclsparse_int* ell_width)

Convert a sparse CSR matrix into a sparse ELLPACK matrix.

`aoclsparse_csr2ell` converts a CSR matrix into an ELL matrix. It is assumed, that `ell_val` and `ell_col_ind` are allocated. Allocation size is computed by the number of rows times the number of ELL non-zero elements per row, such that $\text{nnz}_{\text{ELL}} = m \cdot \text{ell\_width}$. The number of ELL non-zero elements per row is obtained by *aoclsparse_csr2ell_width()*. The index base is preserved during the conversion.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in the conversion process, the remaining descriptor elements are ignored.

- **csr_val** – **[in]** array containing the values of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array containing the column indices of the sparse CSR matrix.
- **ell_width** – **[in]** number of non-zero elements per row in ELL storage format.
- **ell_val** – **[out]** array of m times ell_width elements of the sparse ELL matrix.
- **ell_col_ind** – **[out]** array of m times ell_width elements containing the column indices of the sparse ELL matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_handle** – the library context was not initialized.
- **aoclsparse_status_invalid_size** – m or ell_width is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row_ptr, csr_col _ind, ell_val or ell_col _ind pointer is invalid.

aoclsparse_status **aoclsparse_dcsr2ell**(*aoclsparse_int* m, const *aoclsparse_mat_descr* descr, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const double *csr_val, *aoclsparse_int* *ell_col_ind, double *ell_val, *aoclsparse_int* ell_width)

Convert a sparse CSR matrix into a sparse ELLPACK matrix.

aoclsparse_csr2ell converts a CSR matrix into an ELL matrix. It is assumed, that ell_val and ell_col_ind are allocated. Allocation size is computed by the number of rows times the number of ELL non-zero elements per row, such that $\text{nnz}_{\text{ELL}} = m \cdot \text{ell\_width}$. The number of ELL non-zero elements per row is obtained by *aoclsparse_csr2ell_width()*. The index base is preserved during the conversion.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **csr_val** – **[in]** array containing the values of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array containing the column indices of the sparse CSR matrix.
- **ell_width** – **[in]** number of non-zero elements per row in ELL storage format.
- **ell_val** – **[out]** array of m times ell_width elements of the sparse ELL matrix.
- **ell_col_ind** – **[out]** array of m times ell_width elements containing the column indices of the sparse ELL matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_handle** – the library context was not initialized.
- **aoclsparse_status_invalid_size** – m or ell_width is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row_ptr, csr_col _ind, ell_val or ell_col _ind pointer is invalid.

aoclsparse_status **aoclsparse_csr2dia_ndiag**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, *aoclsparse_int* nnz, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, *aoclsparse_int* *dia_num_diag)

Convert a sparse CSR matrix into a sparse DIA matrix.

`aoclsparse_csr2dia_ndiag` computes the number of the diagonals for a given CSR matrix.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of cols of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in computing the diagonals, the remaining descriptor elements are ignored.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array containing the column indices of the sparse CSR matrix.
- **dia_num_diag** – **[out]** pointer to the number of diagonals with non-zeroes in DIA storage format.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` is invalid.
- **aoclsparse_status_invalid_pointer** – `csr\_row\_ptr`, or `ell\_width` pointer is invalid.
- **aoclsparse_status_internal_error** – an internal error occurred.

aoclsparse_status **aoclsparse_scsr2dia**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const float *csr_val, *aoclsparse_int* dia_num_diag, *aoclsparse_int* *dia_offset, float *dia_val)

Convert a sparse CSR matrix into a sparse DIA matrix.

`aoclsparse_csr2dia` converts a CSR matrix into an DIA matrix. It is assumed, that `dia_val` and `dia_offset` are allocated. Allocation size is computed by the number of rows times the number of diagonals. The number of DIA diagonals is obtained by *aoclsparse_csr2dia_ndiag()*.The index base is preserved during the conversion.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of cols of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array containing the column indices of the sparse CSR matrix.
- **csr_val** – **[in]** array containing the values of the sparse CSR matrix.
- **dia_num_diag** – **[in]** number of diagoanls in ELL storage format.

- **dia_offset** – **[out]** array of dia_num_diag elements containing the diagonal offsets from main diagonal.
- **dia_val** – **[out]** array of m times dia_num_diag elements of the sparse DIA matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_handle** – the library context was not initialized.
- **aoclsparse_status_invalid_size** – m or ell_width is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row _ptr, csr_col _ind, ell_val or ell_col _ind pointer is invalid.

aoclsparse_status **aoclsparse_dcsr2dia**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const double *csr_val, *aoclsparse_int* dia_num_diag, *aoclsparse_int* *dia_offset, double *dia_val)

Convert a sparse CSR matrix into a sparse DIA matrix.

aoclsparse_csr2dia converts a CSR matrix into an DIA matrix. It is assumed, that dia_val and dia_offset are allocated. Allocation size is computed by the number of rows times the number of diagonals. The number of DIA diagonals is obtained by *aoclsparse_csr2dia_ndiag()*.The index base is preserved during the conversion.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of cols of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **csr_row_ptr** – **[in]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array containing the column indices of the sparse CSR matrix.
- **csr_val** – **[in]** array containing the values of the sparse CSR matrix.
- **dia_num_diag** – **[in]** number of diagoanls in ELL storage format.
- **dia_offset** – **[out]** array of dia_num_diag elements containing the diagonal offsets from main diagonal.
- **dia_val** – **[out]** array of m times dia_num_diag elements of the sparse DIA matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_handle** – the library context was not initialized.
- **aoclsparse_status_invalid_size** – m or ell_width is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row _ptr, csr_col _ind, ell_val or ell_col _ind pointer is invalid.

aoclsparse_status **aoclsparse_csr2bsr_nnz**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, *aoclsparse_int* block_dim, *aoclsparse_int* *bsr_row_ptr, *aoclsparse_int* *bsr_nnz) aoclsparse_csr2bsr_nnz 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.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in computing the nnz blocks, the remaining descriptor elements are ignored.
- **csr_row_ptr** – **[in]** integer array containing `m+1` elements that point to the start of each row of the CSR matrix
- **csr_col_ind** – **[in]** integer array of the column indices for each non-zero element in the CSR matrix
- **block_dim** – **[in]** the block dimension of the BSR matrix. Between 1 and min(m, n)
- **bsr_row_ptr** – **[out]** integer array containing `mb+1` elements that point to the start of each block row of the BSR matrix
- **bsr_nnz** – **[out]** total number of nonzero elements in device or host memory.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` or `n` or `block\_dim` is invalid.
- **aoclsparse_status_invalid_pointer** – `csr\_row` _ptr or `csr\_col` _ind or `bsr\_row` _ptr or `bsr\_nnz` pointer is invalid.

aoclsparse_status **aoclsparse_scsr2bsr**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const float *csr_val, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, *aoclsparse_int* block_dim, float *bsr_val, *aoclsparse_int* *bsr_row_ptr, *aoclsparse_int* *bsr_col_ind)

Convert a sparse CSR matrix into a sparse BSR matrix.

`aoclsparse_csr2bsr` 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`. The index base is preserved during the conversion.

Parameters

- **m** – **[in]** number of rows in the sparse CSR matrix.
- **n** – **[in]** number of columns in the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **csr_val** – **[in]** array of `nnz` elements containing the values of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **block_dim** – **[in]** size of the blocks in the sparse BSR matrix.

- **bsr_val** – **[out]** array of `nnzb*block_dim*block_dim` containing the values of the sparse BSR matrix.
- **bsr_row_ptr** – **[out]** array of `mb+1` elements that point to the start of every block row of the sparse BSR matrix.
- **bsr_col_ind** – **[out]** array of `nnzb` elements containing the block column indices of the sparse BSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` or `n` or `block\_dim` is invalid.
- **aoclsparse_status_invalid_pointer** – `bsr\_val` , `bsr\_row` _ptr, `bsr\_col_ind` , `csr\_val` , `csr\_row` _ptr or `csr\_col` _ind pointer is invalid.

aoclsparse_status **aoclsparse_dcsr2bsr**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const double *csr_val, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, *aoclsparse_int* block_dim, double *bsr_val, *aoclsparse_int* *bsr_row_ptr, *aoclsparse_int* *bsr_col_ind)

Convert a sparse CSR matrix into a sparse BSR matrix.

`aoclsparse_csr2bsr` 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`. The index base is preserved during the conversion.

Parameters

- **m** – **[in]** number of rows in the sparse CSR matrix.
- **n** – **[in]** number of columns in the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix. Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **csr_val** – **[in]** array of `nnz` elements containing the values of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **block_dim** – **[in]** size of the blocks in the sparse BSR matrix.
- **bsr_val** – **[out]** array of `nnzb*block_dim*block_dim` containing the values of the sparse BSR matrix.
- **bsr_row_ptr** – **[out]** array of `mb+1` elements that point to the start of every block row of the sparse BSR matrix.
- **bsr_col_ind** – **[out]** array of `nnzb` elements containing the block column indices of the sparse BSR matrix.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` or `n` or `block\_dim` is invalid.
- **aoclsparse_status_invalid_pointer** – `bsr\_val` , `bsr\_row` _ptr, `bsr\_col_ind` , `csr\_val` , `csr\_row` _ptr or `csr\_col` _ind pointer is invalid.

aoclsparse_status **aoclsparse_scsr2csc**(*aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const *aoclsparse_mat_descr* descr, *aoclsparse_index_base* baseCSC, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const float *csr_val, *aoclsparse_int* *csc_row_ind, *aoclsparse_int* *csc_col_ptr, float *csc_val)

Convert a sparse CSR matrix into a sparse CSC matrix.

`aoclsparse_csr2csc` converts a CSR matrix into a CSC matrix. `aoclsparse_csr2csc` can also be used to convert a CSC matrix into a CSR matrix. The index base can be modified during the conversion.

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix.Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **baseCSC** – **[in]** the desired index base (zero or one) for the converted matrix.
- **csr_val** – **[in]** array of `nnz` elements of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **csc_val** – **[out]** array of `nnz` elements of the sparse CSC matrix.
- **csc_row_ind** – **[out]** array of `nnz` elements containing the row indices of the sparse CSC matrix.
- **csc_col_ptr** – **[out]** array of `n+1` elements that point to the start of every column of the sparse CSC matrix. aoclsparse_csr2csc_buffer_size().

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_pointer** – `csr\_val` , `csr\_row` _ptr, `csr\_col` _ind, `csc\_val` , `csc\_row` _ind, `csc\_col` _ptr is invalid.

aoclsparse_status **aoclsparse_dcsr2csc**(*aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const *aoclsparse_mat_descr* descr, *aoclsparse_index_base* baseCSC, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const double *csr_val, *aoclsparse_int* *csc_row_ind, *aoclsparse_int* *csc_col_ptr, double *csc_val)

Convert a sparse CSR matrix into a sparse CSC matrix.

`aoclsparse_csr2csc` converts a CSR matrix into a CSC matrix. `aoclsparse_csr2csc` can also be used to convert a CSC matrix into a CSR matrix. The index base can be modified during the conversion.

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix.Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **baseCSC** – **[in]** the desired index base (zero or one) for the converted matrix.
- **csr_val** – **[in]** array of nnz elements of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of nnz elements containing the column indices of the sparse CSR matrix.
- **csc_val** – **[out]** array of nnz elements of the sparse CSC matrix.
- **csc_row_ind** – **[out]** array of nnz elements containing the row indices of the sparse CSC matrix.
- **csc_col_ptr** – **[out]** array of n+1 elements that point to the start of every column of the sparse CSC matrix. aoclsparse_csr2csc_buffer_size().

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m, n or nnz is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row _ptr, csr_col _ind, csc_val , csc_row _ind, csc_col _ptr is invalid.

aoclsparse_status **aoclsparse_ccsr2csc**(*aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const *aoclsparse_mat_descr* descr, *aoclsparse_index_base* baseCSC, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const aoclsparse_float_complex *csr_val, *aoclsparse_int* *csc_row_ind, *aoclsparse_int* *csc_col_ptr, aoclsparse_float_complex *csc_val)

Convert a sparse CSR matrix into a sparse CSC matrix.

aoclsparse_csr2csc converts a CSR matrix into a CSC matrix. aoclsparse_csr2csc can also be used to convert a CSC matrix into a CSR matrix. The index base can be modified during the conversion.

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.

- **descr** – **[in]** descriptor of the input sparse CSR matrix.Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **baseCSC** – **[in]** the desired index base (zero or one) for the converted matrix.
- **csr_val** – **[in]** array of nnz elements of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of nnz elements containing the column indices of the sparse CSR matrix.
- **csc_val** – **[out]** array of nnz elements of the sparse CSC matrix.
- **csc_row_ind** – **[out]** array of nnz elements containing the row indices of the sparse CSC matrix.
- **csc_col_ptr** – **[out]** array of n+1 elements that point to the start of every column of the sparse CSC matrix. aoclsparse_csr2csc_buffer_size().

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m, n or nnz is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row _ptr, csr_col _ind, csc_val , csc_row _ind, csc_col _ptr is invalid.

aoclsparse_status **aoclsparse_zcsr2csc**(*aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const *aoclsparse_mat_descr* descr, *aoclsparse_index_base* baseCSC, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, const aoclsparse_double_complex *csr_val, *aoclsparse_int* *csc_row_ind, *aoclsparse_int* *csc_col_ptr, aoclsparse_double_complex *csc_val)

Convert a sparse CSR matrix into a sparse CSC matrix.

aoclsparse_csr2csc converts a CSR matrix into a CSC matrix. aoclsparse_csr2csc can also be used to convert a CSC matrix into a CSR matrix. The index base can be modified during the conversion.

Note: The resulting matrix can also be seen as the transpose of the input matrix.

Parameters

- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the input sparse CSR matrix.Only the base index is used in the conversion process, the remaining descriptor elements are ignored.
- **baseCSC** – **[in]** the desired index base (zero or one) for the converted matrix.
- **csr_val** – **[in]** array of nnz elements of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of m+1 elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of nnz elements containing the column indices of the sparse CSR matrix.

- **csc_val** – **[out]** array of nnz elements of the sparse CSC matrix.
- **csc_row_ind** – **[out]** array of nnz elements containing the row indices of the sparse CSC matrix.
- **csc_col_ptr** – **[out]** array of n+1 elements that point to the start of every column of the sparse CSC matrix. aoclsparse_csr2csc_buffer_size().

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m, n or nnz is invalid.
- **aoclsparse_status_invalid_pointer** – csr_val , csr_row _ptr, csr_col _ind, csc_val , csc_row _ind, csc_col _ptr is invalid.

aoclsparse_status **aoclsparse_scsr2dense**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const float *csr_val, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, float *A, *aoclsparse_int* ld, *aoclsparse_order* order)

This function converts the sparse matrix in CSR format into a dense matrix.

Parameters

- **m** – **[in]** number of rows of the dense matrix A.
- **n** – **[in]** number of columns of the dense matrix A.
- **descr** – **[in]** the descriptor of the dense matrix A, the supported matrix type is *aoclsparse_matrix_type_general*. Base index from the descriptor is used in the conversion process.
- **csr_val** – **[in]** array of nnz (= csr_row_ptr[m] - csr_row_ptr[0]) nonzero elements of matrix A.
- **csr_row_ptr** – **[in]** integer array of m+1 elements that contains the start of every row and the end of the last row plus one.
- **csr_col_ind** – **[in]** integer array of nnz (= csr_row_ptr[m] - csr_row_ptr[0]) column indices of the non-zero elements of matrix A.
- **A** – **[out]** array of dimensions (ld, n)
- **ld** – **[in]** leading dimension of dense array A.
- **order** – **[in]** memory layout of a dense matrix A.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m or n or ld is invalid.
- **aoclsparse_status_invalid_pointer** – A or csr_valcsr_row_ptr or csr_col_ind pointer is invalid.

aoclsparse_status **aoclsparse_dcsr2dense**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const double *csr_val, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, double *A, *aoclsparse_int* ld, *aoclsparse_order* order)

This function converts the sparse matrix in CSR format into a dense matrix.

Parameters

- **m** – **[in]** number of rows of the dense matrix A.
- **n** – **[in]** number of columns of the dense matrix A.
- **descr** – **[in]** the descriptor of the dense matrix A, the supported matrix type is *aoclsparse_matrix_type_general*. Base index from the descriptor is used in the conversion process.
- **csr_val** – **[in]** array of nnz (= csr_row_ptr[m] - csr_row_ptr[0]) nonzero elements of matrix A.
- **csr_row_ptr** – **[in]** integer array of m+1 elements that contains the start of every row and the end of the last row plus one.
- **csr_col_ind** – **[in]** integer array of nnz (= csr_row_ptr[m] - csr_row_ptr[0]) column indices of the non-zero elements of matrix A.
- **A** – **[out]** array of dimensions (ld, n)
- **ld** – **[in]** leading dimension of dense array A.
- **order** – **[in]** memory layout of a dense matrix A.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m or n or ld is invalid.
- **aoclsparse_status_invalid_pointer** – A or csr_valcsr_row_ptr or csr_col_ind pointer is invalid.

aoclsparse_status **aoclsparse_ccsr2dense**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const aoclsparse_float_complex *csr_val, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, aoclsparse_float_complex *A, *aoclsparse_int* ld, *aoclsparse_order* order)

This function converts the sparse matrix in CSR format into a dense matrix.

Parameters

- **m** – **[in]** number of rows of the dense matrix A.
- **n** – **[in]** number of columns of the dense matrix A.
- **descr** – **[in]** the descriptor of the dense matrix A, the supported matrix type is *aoclsparse_matrix_type_general*. Base index from the descriptor is used in the conversion process.
- **csr_val** – **[in]** array of nnz (= csr_row_ptr[m] - csr_row_ptr[0]) nonzero elements of matrix A.
- **csr_row_ptr** – **[in]** integer array of m+1 elements that contains the start of every row and the end of the last row plus one.
- **csr_col_ind** – **[in]** integer array of nnz (= csr_row_ptr[m] - csr_row_ptr[0]) column indices of the non-zero elements of matrix A.
- **A** – **[out]** array of dimensions (ld, n)
- **ld** – **[in]** leading dimension of dense array A.
- **order** – **[in]** memory layout of a dense matrix A.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` or `n` or `ld` is invalid.
- **aoclsparse_status_invalid_pointer** – `A` or `csr\_valcsr_row_ptr` or `csr_col_ind` pointer is invalid.

aoclsparse_status **aoclsparse_zcsr2dense**(*aoclsparse_int* m, *aoclsparse_int* n, const *aoclsparse_mat_descr* descr, const aoclsparse_double_complex *csr_val, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_int* *csr_col_ind, aoclsparse_double_complex *A, *aoclsparse_int* ld, *aoclsparse_order* order)

This function converts the sparse matrix in CSR format into a dense matrix.

Parameters

- **m** – **[in]** number of rows of the dense matrix `A`.
- **n** – **[in]** number of columns of the dense matrix `A`.
- **descr** – **[in]** the descriptor of the dense matrix `A`, the supported matrix type is *aoclsparse_matrix_type_general*. Base index from the descriptor is used in the conversion process.
- **csr_val** – **[in]** array of nnz (= `csr_row_ptr`[m] - `csr_row_ptr`[0]) nonzero elements of matrix `A`.
- **csr_row_ptr** – **[in]** integer array of m+1 elements that contains the start of every row and the end of the last row plus one.
- **csr_col_ind** – **[in]** integer array of nnz (= `csr_row_ptr`[m] - csr_row_ptr[0]) column indices of the non-zero elements of matrix `A`.
- **A** – **[out]** array of dimensions (`ld`, `n`)
- **ld** – **[in]** leading dimension of dense array `A`.
- **order** – **[in]** memory layout of a dense matrix `A`.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` or `n` or `ld` is invalid.
- **aoclsparse_status_invalid_pointer** – `A` or `csr\_valcsr_row_ptr` or `csr_col_ind` pointer is invalid.

aoclsparse_status **aoclsparse_convert_csr**(const *aoclsparse_matrix* src_mat, const *aoclsparse_operation* op, *aoclsparse_matrix* *dest_mat)

Convert internal representation of matrix into a sparse CSR matrix.

`aoclsparse_convert_csr` converts any supported matrix format into a CSR format matrix and returns it as a new `aoclsparse_matrix`. The new matrix can also be transposed or conjugate transposed during the conversion. It should be freed by calling `aoclsparse_destroy`. The source matrix needs to be initalized using `aoclsparse_create_(d/s/c/z)(coo/csc/csr)` and it is not modified here.

Parameters

- **src_mat** – **[in]** source matrix used for conversion.
- **op** – **[in]** operation to be performed on destination matrix
- **dest_mat** – **[out]** destination matrix output in CSR Format of the src_mat.

Return values

- `aoclsparse_status_success` – the operation completed successfully
- `aoclsparse_status_invalid_size` – matrix dimension are invalid
- `aoclsparse_status_invalid_pointer` – pointers in `src_mat` or `dest_mat` are invalid
- `aoclsparse_status_not_implemented` – conversion of the src_mat format given is not implemented
- `aoclsparse_status_memory_error` – memory allocation for destination matrix failed

CHAPTER

FOUR

AOCL-SPARSE LEVEL 1,2,3 FUNCTIONS

aoclsparse_functions.h provides AMD CPU hardware optimized level 1, 2, and 3 Sparse Linear Algebra Subprograms (Sparse BLAS).

4.1 Level 1

aoclsparse_status **aoclsparse_saxpyi**(const *aoclsparse_int* nnz, const float a, const float *x, const *aoclsparse_int* *indx, float *y)

A variant of sparse vector-vector addition between a compressed sparse vector and a dense vector.

aoclsparse_(s/d/c/z)axpyi adds a scalar multiple of compressed sparse vector to a dense vector.

Let $y \in C^m$ be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least nnz described by indx, then

$$y_{I_{x_i}} = a * x_i + y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = a*x[i] + y[indx[i]];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **a** – **[in]** Scalar value.
- **x** – **[in]** Sparse vector stored in compressed form of nnz elements.
- **indx** – **[in]** Indices of nnz elements. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of y. Array should follow 0-based indexing.
- **y** – **[inout]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.

- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, indx, y is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided nnz is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_daxpyi**(const *aoclsparse_int* nnz, const double a, const double *x, const *aoclsparse_int* *indx, double *y)

A variant of sparse vector-vector addition between a compressed sparse vector and a dense vector.

`aoclsparse_(s/d/c/z)axpyi` adds a scalar multiple of compressed sparse vector to a dense vector.

Let $y \in C^m$ be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least nnz described by indx, then

$$y_{I_{x_i}} = a * x_i + y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = a*x[i] + y[indx[i]];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **a** – **[in]** Scalar value.
- **x** – **[in]** Sparse vector stored in compressed form of nnz elements.
- **indx** – **[in]** Indices of nnz elements. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of y. Array should follow 0-based indexing.
- **y** – **[inout]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, indx, y is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided nnz is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_caxpyi**(const *aoclsparse_int* nnz, const void *a, const void *x, const *aoclsparse_int* *indx, void *y)

A variant of sparse vector-vector addition between a compressed sparse vector and a dense vector.

`aoclsparse_(s/d/c/z)axpyi` adds a scalar multiple of compressed sparse vector to a dense vector.

Let $y \in C^m$ be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$y_{I_{x_i}} = a * x_i + y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = a*x[i] + y[indx[i]];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **a** – **[in]** Scalar value.
- **x** – **[in]** Sparse vector stored in compressed form of nnz elements.
- **indx** – **[in]** Indices of nnz elements. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y`. Array should follow 0-based indexing.
- **y** – **[inout]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_zaxpyi**(const *aoclsparse_int* nnz, const void *a, const void *x, const *aoclsparse_int* *indx, void *y)

A variant of sparse vector-vector addition between a compressed sparse vector and a dense vector.

`aoclsparse_(s/d/c/z)axpyi` adds a scalar multiple of compressed sparse vector to a dense vector.

Let $y \in C^m$ be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$y_{I_{x_i}} = a * x_i + y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = a*x[i] + y[indx[i]];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **a** – **[in]** Scalar value.
- **x** – **[in]** Sparse vector stored in compressed form of nnz elements.
- **indx** – **[in]** Indices of nnz elements. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y`. Array should follow 0-based indexing.
- **y** – **[inout]** Array of at least $\max(indx_i, i \in \{1, \dots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_cdotci**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, const void *y, void *dot)

Sparse conjugate dot product for single and double data precision complex types.

`aoclsparse_cdotci` (complex float) and `aoclsparse_zdotci` (complex double) compute the dot product of the conjugate of a complex vector stored in a compressed format and a complex dense vector. Let x and y be respectively a sparse and dense vectors in C^m with `indx` an indices vector of length at least nnz that is used to index into the entries of dense vector y, then these functions return

$$\text{dot} = \sum_{i=0}^{nnz-1} \text{conj}(x_i) * y_{indx_i}.$$

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements (length) of vectors x and $indx$.
- **x** – **[in]** Array of at least nnz complex elements.
- **indx** – **[in]** Vector of indices of length at least nnz. Each entry of this vector must contain a valid index into y and be unique. The entries of `indx` are not checked for validity.
- **y** – **[in]** Array of at least $\max(indx_i, i \in \{1, \dots, nnz\})$ complex elements.
- **dot** – **[out]** The dot product of conjugate of x and y when $nnz > 0$. If $nnz \leq 0$, `dot` is set to 0.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, indx, y, dot is invalid.
- **aoclsparse_status_invalid_size** – Indicates that the provided nnz is not positive.

aoclsparse_status **aoclsparse_zdotci**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, const void *y, void *dot)

Sparse conjugate dot product for single and double data precision complex types.

aoclsparse_cdotci (complex float) and aoclsparse_zdotci (complex double) compute the dot product of the conjugate of a complex vector stored in a compressed format and a complex dense vector. Let x and y be respectively a sparse and dense vectors in C^m with indx an indices vector of length at least nnz that is used to index into the entries of dense vector y, then these functions return

$$\text{dot} = \sum_{i=0}^{nnz-1} \text{conj}(x_i) * y_{indx_i}.$$

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements (length) of vectors x and $indx$.
- **x** – **[in]** Array of at least nnz complex elements.
- **indx** – **[in]** Vector of indices of length at least nnz. Each entry of this vector must contain a valid index into y and be unique. The entries of indx are not checked for validity.
- **y** – **[in]** Array of at least $\max(indx_i, i \in \{1, \dots, nnz\})$ complex elements.
- **dot** – **[out]** The dot product of conjugate of x and y when $nnz > 0$. If $nnz \leq 0$, dot is set to 0.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, indx, y, dot is invalid.
- **aoclsparse_status_invalid_size** – Indicates that the provided nnz is not positive.

aoclsparse_status **aoclsparse_cdotui**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, const void *y, void *dot)

Sparse dot product for single and double data precision complex types.

aoclsparse_cdotui (complex float) and aoclsparse_zdotui (complex double) compute the dot product of a complex vector stored in a compressed format and a complex dense vector. Let x and y be respectively a sparse and dense vectors in C^m with indx an indices vector of length at least nnz that is used to index into the entries of dense vector y, then these functions return

$$\text{dot} = \sum_{i=0}^{nnz-1} x_i * y_{indx_i}.$$

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements (length) of vectors x and $indx$.
- **x** – **[in]** Array of at least nnz complex elements.
- **indx** – **[in]** Vector of indices of length at least nnz. Each entry of this vector must contain a valid index into y and be unique. The entries of `indx` are not checked for validity.
- **y** – **[in]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ complex elements.
- **dot** – **[out]** The dot product of x and y when $nnz > 0$. If $nnz \leq 0$, `dot` is set to 0.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y`, `dot` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that the provided `nnz` is not positive.

aoclsparse_status **aoclsparse_zdotui**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, const void *y, void *dot)

Sparse dot product for single and double data precision complex types.

`aoclsparse_cdotui` (complex float) and `aoclsparse_zdotui` (complex double) compute the dot product of a complex vector stored in a compressed format and a complex dense vector. Let x and y be respectively a sparse and dense vectors in C^m with `indx` an indices vector of length at least nnz that is used to index into the entries of dense vector y, then these functions return

$$\text{dot} = \sum_{i=0}^{nnz-1} x_i * y_{indx_i}.$$

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements (length) of vectors x and $indx$.
- **x** – **[in]** Array of at least nnz complex elements.
- **indx** – **[in]** Vector of indices of length at least nnz. Each entry of this vector must contain a valid index into y and be unique. The entries of `indx` are not checked for validity.
- **y** – **[in]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ complex elements.
- **dot** – **[out]** The dot product of x and y when $nnz > 0$. If $nnz \leq 0$, `dot` is set to 0.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y`, `dot` is invalid.

- **aoclsparse_status_invalid_size** – Indicates that the provided nnz is not positive.

float **aoclsparse_sdoti**(const *aoclsparse_int* nnz, const float *x, const *aoclsparse_int* *indx, const float *y)

Sparse dot product for single and double data precision real types.

aoclsparse_sdoti (float) and aoclsparse_ddoti (double) compute the dot product of a real vector stored in a compressed format and a real dense vector. Let x and y be respectively a sparse and dense vectors in R^m with indx an indices vector of length at least nnz that is used to index into the entries of dense vector y, then these functions return

$$\text{dot} = \sum_{i=0}^{nnz-1} x_i * y_{indx_i}.$$

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements (length) of vectors x and $indx$.
- **x** – **[in]** Array of at least nnz real elements.
- **indx** – **[in]** Vector of indices of length at least nnz. Each entry of this vector must contain a valid index into y and be unique. The entries of indx are not checked for validity.
- **y** – **[in]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ complex elements.

Return values

Float/double – Value of the dot product if nnz is positive, otherwise it is set to 0.

double **aoclsparse_ddoti**(const *aoclsparse_int* nnz, const double *x, const *aoclsparse_int* *indx, const double *y)

Sparse dot product for single and double data precision real types.

aoclsparse_sdoti (float) and aoclsparse_ddoti (double) compute the dot product of a real vector stored in a compressed format and a real dense vector. Let x and y be respectively a sparse and dense vectors in R^m with indx an indices vector of length at least nnz that is used to index into the entries of dense vector y, then these functions return

$$\text{dot} = \sum_{i=0}^{nnz-1} x_i * y_{indx_i}.$$

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements (length) of vectors x and $indx$.
- **x** – **[in]** Array of at least nnz real elements.
- **indx** – **[in]** Vector of indices of length at least nnz. Each entry of this vector must contain a valid index into y and be unique. The entries of indx are not checked for validity.
- **y** – **[in]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ complex elements.

Return values

`Float/double` – Value of the dot product if `nnz` is positive, otherwise it is set to 0.

aoclsparse_status **aoclsparse_ssctr**(const *aoclsparse_int* nnz, const float *x, const *aoclsparse_int* *indx, float *y)

Sparse scatter for single and double precision real and complex types.

`aoclsparse_?sctr` scatter the elements of a compressed sparse vector into a dense vector.

Let $y \in R^m$ (or C^m) be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$y_{I_{x_i}} = x_i, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = x[i];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **x** – **[in]** Array of nnz elements to be scattered.
- **indx** – **[in]** Indices of nnz elements to be scattered. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y`.
- **y** – **[out]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_dsctr**(const *aoclsparse_int* nnz, const double *x, const *aoclsparse_int* *indx, double *y)

Sparse scatter for single and double precision real and complex types.

`aoclsparse_?sctr` scatter the elements of a compressed sparse vector into a dense vector.

Let $y \in R^m$ (or C^m) be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$y_{I_{x_i}} = x_i, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = x[i];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **x** – **[in]** Array of nnz elements to be scattered.
- **indx** – **[in]** Indices of nnz elements to be scattered. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y`.
- **y** – **[out]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_csctr**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, void *y)

Sparse scatter for single and double precision real and complex types.

`aoclsparse_?sctr` scatter the elements of a compressed sparse vector into a dense vector.

Let $y \in R^m$ (or C^m) be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$y_{I_{x_i}} = x_i, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = x[i];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **x** – **[in]** Array of nnz elements to be scattered.
- **indx** – **[in]** Indices of nnz elements to be scattered. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y`.

- **y** – **[out]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_zsctr**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, void *y)

Sparse scatter for single and double precision real and complex types.

`aoclsparse_?sctr` scatter the elements of a compressed sparse vector into a dense vector.

Let $y \in R^m$ (or C^m) be a dense vector, x be a compressed sparse vector and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$y_{I_{x_i}} = x_i, i \in \{1, \ldots, \text{nnz}\}.$$

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[indx[i]] = x[i];
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **x** – **[in]** Array of nnz elements to be scattered.
- **indx** – **[in]** Indices of nnz elements to be scattered. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y`.
- **y** – **[out]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative.

aoclsparse_status **aoclsparse_ssctrs**(const *aoclsparse_int* nnz, const float *x, *aoclsparse_int* stride, float *y)

Sparse scatter with stride for real/complex single and double data precisions.

`aoclsparse_?sctrs` scatters the elements of a compressed sparse vector into a dense vector using a stride.

Let y be a dense vector of length $n > 0$, x be a compressed sparse vector with `nnz` > 0 nonzeros, and `stride` be a striding distance, then $y_{\text{stride} \times i} = x_i, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[stride * i] = x[i];
```

Note: Contents of the vector `x` are accessed but not checked.

Parameters

- **nnz** – **[in]** Number of nonzero elements in x.
- **x** – **[in]** Array of `nnz` elements to be scattered into `y`.
- **stride** – **[in]** (Positive) striding distance used to store elements in vector `y`.
- **y** – **[out]** Array of size at least `stride` $\times$ `nnz`.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that one or more of the values provided in `nnz` or `stride` is not positive.

aoclsparse_status **aoclsparse_dsctrs**(const *aoclsparse_int* nnz, const double *x, *aoclsparse_int* stride, double *y)

Sparse scatter with stride for real/complex single and double data precisions.

`aoclsparse_?sctrs` scatters the elements of a compressed sparse vector into a dense vector using a stride.

Let y be a dense vector of length $n > 0$, x be a compressed sparse vector with `nnz` > 0 nonzeros, and `stride` be a striding distance, then $y_{\text{stride} \times i} = x_i, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[stride * i] = x[i];
```

Note: Contents of the vector `x` are accessed but not checked.

Parameters

- **nnz** – **[in]** Number of nonzero elements in x.
- **x** – **[in]** Array of `nnz` elements to be scattered into `y`.
- **stride** – **[in]** (Positive) striding distance used to store elements in vector `y`.
- **y** – **[out]** Array of size at least `stride` $\times$ `nnz`.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, y is invalid.
- **aoclsparse_status_invalid_size** – Indicates that one or more of the values provided in nnz or stride is not positive.

aoclsparse_status **aoclsparse_csctrs**(const *aoclsparse_int* nnz, const void *x, *aoclsparse_int* stride, void *y)

Sparse scatter with stride for real/complex single and double data precisions.

aoclsparse_?sctrs scatters the elements of a compressed sparse vector into a dense vector using a stride.

Let y be a dense vector of length $n > 0$, x be a compressed sparse vector with nnz > 0 nonzeros, and stride be a striding distance, then $y_{\text{stride}\times i} = x_i, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[stride * i] = x[i];
```

Note: Contents of the vector x are accessed but not checked.

Parameters

- **nnz** – **[in]** Number of nonzero elements in x.
- **x** – **[in]** Array of nnz elements to be scattered into y.
- **stride** – **[in]** (Positive) striding distance used to store elements in vector y.
- **y** – **[out]** Array of size at least stride $\times$ nnz.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, y is invalid.
- **aoclsparse_status_invalid_size** – Indicates that one or more of the values provided in nnz or stride is not positive.

aoclsparse_status **aoclsparse_zsctrs**(const *aoclsparse_int* nnz, const void *x, *aoclsparse_int* stride, void *y)

Sparse scatter with stride for real/complex single and double data precisions.

aoclsparse_?sctrs scatters the elements of a compressed sparse vector into a dense vector using a stride.

Let y be a dense vector of length $n > 0$, x be a compressed sparse vector with nnz > 0 nonzeros, and stride be a striding distance, then $y_{\text{stride}\times i} = x_i, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   y[stride * i] = x[i];
```

Note: Contents of the vector x are accessed but not checked.

Parameters

- **nnz** – **[in]** Number of nonzero elements in x.
- **x** – **[in]** Array of nnz elements to be scattered into y.
- **stride** – **[in]** (Positive) striding distance used to store elements in vector y.
- **y** – **[out]** Array of size at least stride $\times$ nnz.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers x, y is invalid.
- **aoclsparse_status_invalid_size** – Indicates that one or more of the values provided in nnz or stride is not positive.

aoclsparse_status **aoclsparse_sroti**(const *aoclsparse_int* nnz, float *x, const *aoclsparse_int* *indx, float *y, const float c, const float s)

Applies Givens rotations to single and double precision real vectors.

aoclsparse_sroti (float) and aoclsparse_droti (double) apply the Givens rotations on elements of two real vectors.

Let $y \in R^m$ be a vector in full storage form, x be a vector in a compressed form and I_x be an indices vector of length at least nnz described by indx, then

$$x_i = c * x_i + s * y_{I_{x_i}}$$

$$y_{I_{x_i}} = c * y_{I_{x_i}} - s * x_i$$

where c, s are scalars.

A possible C implementation could be

```
for(i = 0; i < nnz; ++i)
{
   temp = x[i];
   x[i] = c * x[i] + s * y[indx[i]];
   y[indx[i]] = c * y[indx[i]] - s * temp;
}
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **x** – **[inout]** Array of at least nnz elements in compressed form. The elements of the array are updated after applying Givens rotation.
- **indx** – **[in]** Indices of nnz elements used for Givens rotation. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of y and are distinct.

- **y** – **[inout]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements in full storage form. The elements of the array are updated after applying Givens rotation.
- **c** – **[in]** A scalar.
- **s** – **[in]** A scalar.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative. With this error, the values of vectors x and y are undefined.

aoclsparse_status **aoclsparse_droti**(const *aoclsparse_int* nnz, double *x, const *aoclsparse_int* *indx, double *y, const double c, const double s)

Applies Givens rotations to single and double precision real vectors.

`aoclsparse_sroti` (float) and `aoclsparse_droti` (double) apply the Givens rotations on elements of two real vectors.

Let $y \in R^m$ be a vector in full storage form, x be a vector in a compressed form and I_x be an indices vector of length at least `nnz` described by `indx`, then

$$x_i = c * x_i + s * y_{I_{x_i}}$$

$$y_{I_{x_i}} = c * y_{I_{x_i}} - s * x_i$$

where `c`, `s` are scalars.

A possible C implementation could be

```
for(i = 0; i < nnz; ++i)
{
   temp = x[i];
   x[i] = c * x[i] + s * y[indx[i]];
   y[indx[i]] = c * y[indx[i]] - s * temp;
}
```

Note: The contents of the vectors are not checked for NaNs.

Parameters

- **nnz** – **[in]** The number of elements in x and $indx$.
- **x** – **[inout]** Array of at least nnz elements in compressed form. The elements of the array are updated after applying Givens rotation.
- **indx** – **[in]** Indices of nnz elements used for Givens rotation. The elements in this vector are only checked for non-negativity. The user should make sure that index is less than the size of `y` and are distinct.

- **y** – **[inout]** Array of at least $\max(indx_i, i \in \{1, \ldots, nnz\})$ elements in full storage form. The elements of the array are updated after applying Givens rotation.
- **c** – **[in]** A scalar.
- **s** – **[in]** A scalar.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – At least one of the pointers `x`, `indx`, `y` is invalid.
- **aoclsparse_status_invalid_size** – Indicates that provided `nnz` is less than zero.
- **aoclsparse_status_invalid_index_value** – At least one of the indices in indx is negative. With this error, the values of vectors x and y are undefined.

aoclsparse_status **aoclsparse_sgthr**(*aoclsparse_int* nnz, const float *y, float *x, const *aoclsparse_int* *indx)

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

The `aoclsparse_?gthr` is a group of functions that gather the elements indexed in `indx` from the dense vector `y` into the sparse vector `x`.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 <$ `nnz` $\leq m$ described by `indx`, then

$$x_i = y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

For double precision complex vectors use `aoclsparse_zgthr` and for single precision complex vectors use `aoclsparse_cgthr`.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in `indx` are less than m without duplicate elements, and that `x` and `indx` are pointers to vectors of size at least `nnz`.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x`, `y`, and `indx` are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least `nnz` non-zero elements.
- **indx** – **[in]** index vector of size `nnz`, containing the indices of the non-zero values of x. Indices should range from 0 to $m - 1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully

- **aoclsparse_status_invalid_size** – nnz parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, x or indx is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in indx is negative

aoclsparse_status **aoclsparse_dgthr**(*aoclsparse_int* nnz, const double *y, double *x, const *aoclsparse_int* *indx)

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

The aoclsparse_?gthr is a group of functions that gather the elements indexed in indx from the dense vector y into the sparse vector x.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 <$ nnz $\leq m$ described by indx, then

$$x_i = y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

For double precision complex vectors use aoclsparse_zgthr and for single precision complex vectors use aoclsparse_cgthr.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in indx are less than m without duplicate elements, and that x and indx are pointers to vectors of size at least nnz.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If nnz is zero, then none of the entries of vectors x, y, and indx are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least nnz non-zero elements.
- **indx** – **[in]** index vector of size nnz, containing the indices of the non-zero values of x. Indices should range from 0 to $m - 1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – nnz parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, x or indx is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in indx is negative

aoclsparse_status **aoclsparse_cgthr**(*aoclsparse_int* nnz, const void *y, void *x, const *aoclsparse_int* *indx)

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

The `aoclsparse_?gthr` is a group of functions that gather the elements indexed in `indx` from the dense vector `y` into the sparse vector `x`.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 < \texttt{nnz} \leq m$ described by `indx`, then

$$x_i = y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

For double precision complex vectors use `aoclsparse_zgthr` and for single precision complex vectors use `aoclsparse_cgthr`.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in `indx` are less than m without duplicate elements, and that `x` and `indx` are pointers to vectors of size at least `nnz`.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x`, `y`, and `indx` are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least `nnz` non-zero elements.
- **indx** – **[in]** index vector of size `nnz`, containing the indices of the non-zero values of x. Indices should range from 0 to $m - 1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – nnz parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers `y`, `x` or `indx` is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in `indx` is negative

aoclsparse_status **aoclsparse_zgthr**(*aoclsparse_int* nnz, const void *y, void *x, const *aoclsparse_int* *indx)

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

The `aoclsparse_?gthr` is a group of functions that gather the elements indexed in `indx` from the dense vector `y` into the sparse vector `x`.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 < \texttt{nnz} \leq m$ described by `indx`, then

$$x_i = y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}.$$

For double precision complex vectors use `aoclsparse_zgthr` and for single precision complex vectors use `aoclsparse_cgthr`.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in `indx` are less than m without duplicate elements, and that `x` and `indx` are pointers to vectors of size at least `nnz`.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x`, `y`, and `indx` are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least `nnz` non-zero elements.
- **indx** – **[in]** index vector of size `nnz`, containing the indices of the non-zero values of x. Indices should range from 0 to $m-1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – nnz parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers `y`, `x` or `indx` is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in `indx` is negative

aoclsparse_status **aoclsparse_sgthrz**(*aoclsparse_int* nnz, float *y, float *x, const *aoclsparse_int* *indx)

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

The `aoclsparse_?gthrz` is a group of functions that gather the elements

indexed in `indx` from the dense vector `y` into the sparse vector `x`. The gathered elements in y are replaced by zero.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 < \texttt{nnz} \leq m$ described by `indx`, then

$$x_i = y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}, \text{ and after the assignment, } y_{I_{x_i}} = 0, i \in \{1, \ldots, \text{nnz}\}.$$

For double precision complex vectors use `aoclsparse_zgthrz` and for single precision complex vectors use `aoclsparse_cgthrz`.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in `indx` are less than m without duplicate elements, and that `x` and `indx` are pointers to vectors of size at least `nnz`.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x`, `y`, and `indx` are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least `nnz` non-zero elements.
- **indx** – **[in]** index vector of size `nnz`, containing the indices of the non-zero values of x. Indices should range from 0 to $m - 1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – `nnz` parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers `y`, `x` or `indx` is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in `indx` is negative

aoclsparse_status **aoclsparse_dgthrz**(*aoclsparse_int* nnz, double *y, double *x, const *aoclsparse_int* *indx)

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

The `aoclsparse_?gthrz` is a group of functions that gather the elements

indexed in `indx` from the dense vector `y` into the sparse vector `x`. The gathered elements in y are replaced by zero.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 < \text{nnz} \leq m$ described by `indx`, then

$$x_i = y_{I_{x_i}}, i \in \{1, \dots, \text{nnz}\}, \text{ and after the assignment, } y_{I_{x_i}} = 0, i \in \{1, \dots, \text{nnz}\}.$$

For double precision complex vectors use `aoclsparse_zgthrz` and for single precision complex vectors use `aoclsparse_cgthrz`.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in `indx` are less than m without duplicate elements, and that `x` and `indx` are pointers to vectors of size at least `nnz`.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x`, `y`, and `indx` are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least `nnz` non-zero elements.
- **indx** – **[in]** index vector of size `nnz`, containing the indices of the non-zero values of x. Indices should range from 0 to $m-1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – `nnz` parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers `y`, `x` or `indx` is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in `indx` is negative

aoclsparse_status **aoclsparse_cgthrz**(*aoclsparse_int* nnz, void *y, void *x, const *aoclsparse_int* *indx)

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

The `aoclsparse_?gthrz` is a group of functions that gather the elements

indexed in `indx` from the dense vector `y` into the sparse vector `x`. The gathered elements in y are replaced by zero.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 < \text{nnz} \leq m$ described by `indx`, then

$$x_i = y_{I_{x_i}}, i \in \{1, \ldots, \text{nnz}\}, \text{ and after the assignment, } y_{I_{x_i}} = 0, i \in \{1, \ldots, \text{nnz}\}.$$

For double precision complex vectors use `aoclsparse_zgthrz` and for single precision complex vectors use `aoclsparse_cgthrz`.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in `indx` are less than m without duplicate elements, and that `x` and `indx` are pointers to vectors of size at least `nnz`.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If nnz is zero, then none of the entries of vectors x, y, and indx are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least nnz non-zero elements.
- **indx** – **[in]** index vector of size nnz, containing the indices of the non-zero values of x. Indices should range from 0 to $m-1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – nnz parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, x or indx is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in indx is negative

aoclsparse_status **aoclsparse_zgthrz**(*aoclsparse_int* nnz, void *y, void *x, const *aoclsparse_int* *indx)

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

The aoclsparse_?gthrz is a group of functions that gather the elements

indexed in indx from the dense vector y into the sparse vector x. The gathered elements in y are replaced by zero.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and I_x be a set of indices of size $0 < \text{nnz} \le m$ described by indx, then

$$x_i = y_{I_{x_i}}, i \in \{1, \dots, \text{nnz}\}, \text{ and after the assignment, } y_{I_{x_i}} = 0, i \in \{1, \dots, \text{nnz}\}.$$

For double precision complex vectors use aoclsparse_zgthrz and for single precision complex vectors use aoclsparse_cgthrz.

A possible C implementation for real vectors could be

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

Note: These functions assume that the indices stored in indx are less than m without duplicate elements, and that x and indx are pointers to vectors of size at least nnz.

Parameters

- **nnz** – **[in]** number of non-zero entries of x. If nnz is zero, then none of the entries of vectors x, y, and indx are touched.
- **y** – **[in]** pointer to dense vector y of size at least m.
- **x** – **[out]** pointer to sparse vector x with at least nnz non-zero elements.

- **indx** – **[in]** index vector of size nnz, containing the indices of the non-zero values of x. Indices should range from 0 to $m-1$, need not be ordered. The elements in this vector are only checked for non-negativity. The user should make sure that no index is out-of-bound and that it does not contains any duplicates.

Return values

- **aoclsparse_status_success** – the operation completed successfully
- **aoclsparse_status_invalid_size** – nnz parameter value is negative
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, x or indx is invalid
- **aoclsparse_status_invalid_index_value** – at least one of the indices in indx is negative

aoclsparse_status **aoclsparse_sgthrs**(*aoclsparse_int* nnz, const float *y, float *x, *aoclsparse_int* stride)

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

The aoclsparse_?gthrs is a group of functions that gather the elements from the dense vector y using a fixed stride distance and copies them into the sparse vector x.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and stride be a (positive) striding distance, then $x_i = y_{\text{stride}\times i}, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   x[i] = y[stride *i];
```

Note: These functions are taylored for the case where stride is greater than 1. If stride is 1, then it is recommended to use the aoclsparse_?gthr set of functions.

Parameters

- **nnz** – **[in]** Number of non-zero entries of x. If nnz is zero, then none of the entries of vectors x and y are accessed.
- **y** – **[in]** Pointer to dense vector y of size at least m.
- **x** – **[out]** Pointer to sparse vector x with at least nnz non-zero elements.
- **stride** – **[in]** Striding distance used to access elements in the dense vector y.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – either nnz or the stride parameter values are not positive.
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, or x is invalid.

aoclsparse_status **aoclsparse_dgthrs**(*aoclsparse_int* nnz, const double *y, double *x, *aoclsparse_int* stride)

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

The aoclsparse_?gthrs is a group of functions that gather the elements from the dense vector y using a fixed stride distance and copies them into the sparse vector x.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and `stride` be a (positive) striding distance, then $x_i = y_{\text{stride} \times i}, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   x[i] = y[stride *i];
```

Note: These functions are taylored for the case where `stride` is greater than 1. If stride is 1, then it is recommended to use the `aoclsparse_?gthr` set of functions.

Parameters

- **nnz** – **[in]** Number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x` and `y` are accessed.
- **y** – **[in]** Pointer to dense vector y of size at least m.
- **x** – **[out]** Pointer to sparse vector x with at least `nnz` non-zero elements.
- **stride** – **[in]** Striding distance used to access elements in the dense vector `y`.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – either `nnz` or the `stride` parameter values are not positive.
- **aoclsparse_status_invalid_pointer** – at least one of the pointers `y`, or `x` is invalid.

aoclsparse_status **aoclsparse_cgthrs**(*aoclsparse_int* nnz, const void *y, void *x, *aoclsparse_int* stride)

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

The `aoclsparse_?gthrs` is a group of functions that gather the elements from the dense vector `y` using a fixed stride distance and copies them into the sparse vector `x`.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and `stride` be a (positive) striding distance, then $x_i = y_{\text{stride} \times i}, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   x[i] = y[stride *i];
```

Note: These functions are taylored for the case where `stride` is greater than 1. If stride is 1, then it is recommended to use the `aoclsparse_?gthr` set of functions.

Parameters

- **nnz** – **[in]** Number of non-zero entries of x. If `nnz` is zero, then none of the entries of vectors `x` and `y` are accessed.
- **y** – **[in]** Pointer to dense vector y of size at least m.
- **x** – **[out]** Pointer to sparse vector x with at least `nnz` non-zero elements.
- **stride** – **[in]** Striding distance used to access elements in the dense vector `y`.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – either nnz or the stride parameter values are not positive.
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, or x is invalid.

aoclsparse_status **aoclsparse_zgthrs**(*aoclsparse_int* nnz, const void *y, void *x, *aoclsparse_int* stride)

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

The aoclsparse_?gthrs is a group of functions that gather the elements from the dense vector y using a fixed stride distance and copies them into the sparse vector x.

Let $y \in R^m$ (or C^m) be a dense vector, x be a sparse vector from the same space and stride be a (positive) striding distance, then $x_i = y_{\text{stride} \times i}, \quad i \in \{1, \ldots, \text{nnz}\}$.

A possible C implementation for real vectors could be

```
for(i = 0; i < nnz; ++i)
   x[i] = y[stride *i];
```

Note: These functions are taylored for the case where stride is greater than 1. If stride is 1, then it is recommended to use the aoclsparse_?gthr set of functions.

Parameters

- **nnz** – **[in]** Number of non-zero entries of x. If nnz is zero, then none of the entries of vectors x and y are accessed.
- **y** – **[in]** Pointer to dense vector y of size at least m.
- **x** – **[out]** Pointer to sparse vector x with at least nnz non-zero elements.
- **stride** – **[in]** Striding distance used to access elements in the dense vector y.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – either nnz or the stride parameter values are not positive.
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, or x is invalid.

4.2 Level 2

aoclsparse_status **aoclsparse_scsrmv**(*aoclsparse_operation* trans, const float *alpha, *aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const float *csr_val, const *aoclsparse_int* *csr_col_ind, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_mat_descr* descr, const float *x, const float *beta, float *y)

Single and double precision sparse matrix vector multiplication using CSR storage format.

aoclsparse_csrmv 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Example

This example performs a sparse matrix vector multiplication in CSR format using additional meta data to improve performance.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **csr_val** – **[in]** array of `nnz` elements of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the sparse CSR matrix. Currently, only *aoclsparse_matrix_type_general* and *aoclsparse_matrix_type_symmetric* is supported.
- **x** – **[in]** array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `csr_val`, `csr\_row` _ptr, `csr\_col` _ind, `x`, `beta` or `y` pointer is invalid.
- **aoclsparse_status_not_implemented** – `trans` is not *aoclsparse_operation_none* and `trans` is not *aoclsparse_operation_transpose*. *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*, or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_symmetric*.

aoclsparse_status **aoclsparse_dcsrmv**(*aoclsparse_operation* trans, const double *alpha, *aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const double *csr_val, const *aoclsparse_int* *csr_col_ind, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_mat_descr* descr, const double *x, const double *beta, double *y)

Single and double precision sparse matrix vector multiplication using CSR storage format.

`aoclsparse_csrmv` 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Example

This example performs a sparse matrix vector multiplication in CSR format using additional meta data to improve performance.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse CSR matrix.
- **n** – **[in]** number of columns of the sparse CSR matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse CSR matrix.
- **csr_val** – **[in]** array of `nnz` elements of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the sparse CSR matrix. Currently, only *aoclsparse_matrix_type_general* and *aoclsparse_matrix_type_symmetric* is supported.
- **x** – **[in]** array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `csr_val`, `csr\_row` _ptr, `csr\_col` _ind, `x`, `beta` or `y` pointer is invalid.
- **aoclsparse_status_not_implemented** – `trans` is not *aoclsparse_operation_none* and `trans` is not *aoclsparse_operation_transpose*. *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*, or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_symmetric*.

aoclsparse_status **aoclsparse_sellmv**(*aoclsparse_operation* trans, const float *alpha, *aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const float *ell_val, const *aoclsparse_int* *ell_col_ind, *aoclsparse_int* ell_width, const *aoclsparse_mat_descr* descr, const float *x, const float *beta, float *y)

Single & Double precision sparse matrix vector multiplication using ELL storage format.

`aoclsparse_ellmv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in ELL 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse ELL matrix.
- **n** – **[in]** number of columns of the sparse ELL matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse ELL matrix.
- **descr** – **[in]** descriptor of the sparse ELL matrix. Both, base-zero and base-one input arrays of ELL matrix are supported
- **ell_val** – **[in]** array that contains the elements of the sparse ELL matrix. Padded elements should be zero.
- **ell_col_ind** – **[in]** array that contains the column indices of the sparse ELL matrix. Padded column indices should be -1.
- **ell_width** – **[in]** number of non-zero elements per row of the sparse ELL matrix.
- **x** – **[in]** array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `ell_width` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `ell_val`, `ell_col_ind`, `x`, `beta` or `y` pointer is invalid.
- **aoclsparse_status_not_implemented** – `trans` != *aoclsparse_operation_none* or *aoclsparse_matrix_type* != *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_dellmv**(*aoclsparse_operation* trans, const double *alpha, *aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const double *ell_val, const *aoclsparse_int* *ell_col_ind, *aoclsparse_int* ell_width, const *aoclsparse_mat_descr* descr, const double *x, const double *beta, double *y)

Single & Double precision sparse matrix vector multiplication using ELL storage format.

`aoclsparse_ellmv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in ELL 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse ELL matrix.
- **n** – **[in]** number of columns of the sparse ELL matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse ELL matrix.
- **descr** – **[in]** descriptor of the sparse ELL matrix. Both, base-zero and base-one input arrays of ELL matrix are supported
- **ell_val** – **[in]** array that contains the elements of the sparse ELL matrix. Padded elements should be zero.
- **ell_col_ind** – **[in]** array that contains the column indices of the sparse ELL matrix. Padded column indices should be `-1`.
- **ell_width** – **[in]** number of non-zero elements per row of the sparse ELL matrix.
- **x** – **[in]** array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `ell_width` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `ell_val`, `ell_col_ind`, `x`, `beta` or `y` pointer is invalid.
- **aoclsparse_status_not_implemented** – `trans` != *aoclsparse_operation_none* or *aoclsparse_matrix_type* != *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_sdiamv**(*aoclsparse_operation* trans, const float *alpha, *aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const float *dia_val, const *aoclsparse_int* *dia_offset, *aoclsparse_int* dia_num_diag, const *aoclsparse_mat_descr* descr, const float *x, const float *beta, float *y)

Single & Double precision sparse matrix vector multiplication using DIA storage format.

`aoclsparse_diamv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in DIA 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse DIA matrix.
- **n** – **[in]** number of columns of the sparse DIA matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse DIA matrix.
- **descr** – **[in]** descriptor of the sparse DIA matrix.
- **dia_val** – **[in]** array that contains the elements of the sparse DIA matrix. Padded elements should be zero.
- **dia_offset** – **[in]** array that contains the offsets of each diagonal of the sparse DIAL matrix.
- **dia_num_diag** – **[in]** number of diagonals in the sparse DIA matrix.
- **x** – **[in]** array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `ell_width` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `ell_val`, `ell_col_ind`, `x`, `beta` or `y` pointer is invalid.
- **aoclsparse_status_not_implemented** – `trans` != *aoclsparse_operation_none* or *aoclsparse_matrix_type* != *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_ddiamv**(*aoclsparse_operation* trans, const double *alpha, *aoclsparse_int* m, *aoclsparse_int* n, *aoclsparse_int* nnz, const double *dia_val, const *aoclsparse_int* *dia_offset, *aoclsparse_int* dia_num_diag, const *aoclsparse_mat_descr* descr, const double *x, const double *beta, double *y)

Single & Double precision sparse matrix vector multiplication using DIA storage format.

`aoclsparse_diamv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in DIA 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse DIA matrix.
- **n** – **[in]** number of columns of the sparse DIA matrix.
- **nnz** – **[in]** number of non-zero entries of the sparse DIA matrix.
- **descr** – **[in]** descriptor of the sparse DIA matrix.
- **dia_val** – **[in]** array that contains the elements of the sparse DIA matrix. Padded elements should be zero.
- **dia_offset** – **[in]** array that contains the offsets of each diagonal of the sparse DIAL matrix.
- **dia_num_diag** – **[in]** number of diagonals in the sparse DIA matrix.
- **x** – **[in]** array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `ell_width` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `ell_val`, `ell_col_ind`, `x`, `beta` or `y` pointer is invalid.
- **aoclsparse_status_not_implemented** – `trans` != *aoclsparse_operation_none* or *aoclsparse_matrix_type* != *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_sbsrmv**(*aoclsparse_operation* trans, const float *alpha, *aoclsparse_int* mb, *aoclsparse_int* nb, *aoclsparse_int* bsr_dim, const float *bsr_val, const *aoclsparse_int* *bsr_col_ind, const *aoclsparse_int* *bsr_row_ptr, const *aoclsparse_mat_descr* descr, const float *x, const float *beta, float *y)

Single & Double precision Sparse matrix vector multiplication using BSR storage format.

`aoclsparse_bsrmv` multiplies the scalar α with a sparse $(mb \cdot \text{bsr\_dim}) \times (nb \cdot \text{bsr\_dim})$ 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Parameters

- **`trans`** – **[in]** matrix operation type.
- **`mb`** – **[in]** number of block rows of the sparse BSR matrix.
- **`nb`** – **[in]** number of block columns of the sparse BSR matrix.
- **`alpha`** – **[in]** scalar α.
- **`descr`** – **[in]** descriptor of the sparse BSR matrix. Both, base-zero and base-one input arrays of BSR matrix are supported
- **`bsr_val`** – **[in]** array of `nnzb` blocks of the sparse BSR matrix.
- **`bsr_row_ptr`** – **[in]** array of `mb+1` elements that point to the start of every block row of the sparse BSR matrix.
- **`bsr_col_ind`** – **[in]** array of `nnz` containing the block column indices of the sparse BSR matrix.
- **`bsr_dim`** – **[in]** block dimension of the sparse BSR matrix.
- **`x`** – **[in]** array of `nb*bsr_dim` elements ($op(A) = A$) or `mb*bsr_dim` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **`beta`** – **[in]** scalar β.
- **`y`** – **[inout]** array of `mb*bsr_dim` elements ($op(A) = A$) or `nb*bsr_dim` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **`aoclsparse_status_success`** – the operation completed successfully.
- **`aoclsparse_status_invalid_handle`** – the library context was not initialized.
- **`aoclsparse_status_invalid_size`** – `mb`, `nb`, `nnzb` or `bsr_dim` is invalid.
- **`aoclsparse_status_invalid_pointer`** – `descr`, `alpha`, `bsr_val`, `bsr_row_ind`, `bsr_col_ind`, `x`, `beta` or `y` pointer is invalid.
- **`aoclsparse_status_arch_mismatch`** – the device is not supported.
- **`aoclsparse_status_not_implemented`** – `trans` != *aoclsparse_operation_none* or *aoclsparse_matrix_type* != *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_dbsrmv**(*aoclsparse_operation* trans, const double *alpha, *aoclsparse_int* mb, *aoclsparse_int* nb, *aoclsparse_int* bsr_dim, const double *bsr_val, const *aoclsparse_int* *bsr_col_ind, const *aoclsparse_int* *bsr_row_ptr, const *aoclsparse_mat_descr* descr, const double *x, const double *beta, double *y)

Single & Double precision Sparse matrix vector multiplication using BSR storage format.

`aoclsparse_bsrmv` multiplies the scalar α with a sparse $(mb \cdot \text{bsr\_dim}) \times (nb \cdot \text{bsr\_dim})$ 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Parameters

- **trans** – **[in]** matrix operation type.
- **mb** – **[in]** number of block rows of the sparse BSR matrix.
- **nb** – **[in]** number of block columns of the sparse BSR matrix.
- **alpha** – **[in]** scalar α.
- **descr** – **[in]** descriptor of the sparse BSR matrix. Both, base-zero and base-one input arrays of BSR matrix are supported
- **bsr_val** – **[in]** array of `nnzb` blocks of the sparse BSR matrix.
- **bsr_row_ptr** – **[in]** array of `mb+1` elements that point to the start of every block row of the sparse BSR matrix.
- **bsr_col_ind** – **[in]** array of `nnz` containing the block column indices of the sparse BSR matrix.
- **bsr_dim** – **[in]** block dimension of the sparse BSR matrix.
- **x** – **[in]** array of `nb*bsr_dim` elements ($op(A) = A$) or `mb*bsr_dim` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of `mb*bsr_dim` elements ($op(A) = A$) or `nb*bsr_dim` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_handle** – the library context was not initialized.
- **aoclsparse_status_invalid_size** – `mb`, `nb`, `nnzb` or `bsr_dim` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `bsr_val`, `bsr_row_ind`, `bsr_col_ind`, `x`, `beta` or `y` pointer is invalid.

- **aoclsparse_status_arch_mismatch** – the device is not supported.
- **aoclsparse_status_not_implemented** – trans != *aoclsparse_operation_none* or *aoclsparse_matrix_type* != *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_smv**(*aoclsparse_operation* op, const float *alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const float *x, const float *beta, float *y)

Computes sparse matrix vector multiplication for real/complex single and double data precisions.

aoclsparse_(s/d/c/z)mv performs a sparse matrix vector multiplication such that

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

where, x and y are dense vectors, alpha and beta are scalars, and A is a sparse matrix structure. The matrix operation 'op' is defined as:

$$op(A) = \begin{cases} A, & \text{if op = aoclsparse\_operation\_none} \\ A^T, & \text{if op = aoclsparse\_operation\_transpose} \\ A^H, & \text{if op = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Parameters

- **op** – **[in]** Matrix operation.
- **alpha** – **[in]** Scalar α.
- **A** – **[in]** The sparse matrix structure containing a sparse matrix of dimension ($m \cdot n$) that is created using aoclsparse_create_?csr.
- **descr** – **[in]** Descriptor of the sparse matrix can be one of the following: *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_triangular*, *aoclsparse_matrix_type_symmetric*, and *aoclsparse_matrix_type_hermitian*. Both base-zero and base-one are supported, however, the index base needs to match the one used at when aoclsparse_matrix was created.
- **x** – **[in]** An array of n elements ($op(A) = A$) or m elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** Scalar β.
- **y** – **[inout]** An array of m elements ($op(A) = A$) or n elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of m, n or nnz is invalid.
- **aoclsparse_status_invalid_pointer** – descr, alpha, internal structures related to the sparse matrix A, x, beta or y has an invalid pointer.
- **aoclsparse_status_not_implemented** – The requested functionality is not implemented.

aoclsparse_status **aoclsparse_dmv**(*aoclsparse_operation* op, const double *alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const double *x, const double *beta, double *y)

Computes sparse matrix vector multiplication for real/complex single and double data precisions.

`aoclsparse_`(s/d/c/z)mv performs a sparse matrix vector multiplication such that

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

where, x and y are dense vectors, alpha and beta are scalars, and A is a sparse matrix structure. The matrix operation 'op' is defined as:

$$op(A) = \begin{cases} A, & \text{if op = aoclsparse\_operation\_none} \\ A^T, & \text{if op = aoclsparse\_operation\_transpose} \\ A^H, & \text{if op = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Parameters

- **op** – **[in]** Matrix operation.
- **alpha** – **[in]** Scalar α.
- **A** – **[in]** The sparse matrix structure containing a sparse matrix of dimension ($m \cdot n$) that is created using aoclsparse_create_?csr.
- **descr** – **[in]** Descriptor of the sparse matrix can be one of the following: *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_triangular*, *aoclsparse_matrix_type_symmetric*, and *aoclsparse_matrix_type_hermitian*. Both base-zero and base-one are supported, however, the index base needs to match the one used at when aoclsparse_matrix was created.
- **x** – **[in]** An array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** Scalar β.
- **y** – **[inout]** An array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, internal structures related to the sparse matrix `A`, `x`, `beta` or `y` has an invalid pointer.
- **aoclsparse_status_not_implemented** – The requested functionality is not implemented.

aoclsparse_status **aoclsparse_cmv**(*aoclsparse_operation* op, const aoclsparse_float_complex *alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_float_complex *x, const aoclsparse_float_complex *beta, aoclsparse_float_complex *y)

Computes sparse matrix vector multiplication for real/complex single and double data precisions.

`aoclsparse_`(s/d/c/z)mv performs a sparse matrix vector multiplication such that

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

where, x and y are dense vectors, alpha and beta are scalars, and A is a sparse matrix structure. The matrix operation 'op' is defined as:

$$op(A) = \begin{cases} A, & \text{if op = aoclsparse\_operation\_none} \\ A^T, & \text{if op = aoclsparse\_operation\_transpose} \\ A^H, & \text{if op = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Parameters

- **op** – **[in]** Matrix operation.
- **alpha** – **[in]** Scalar α.
- **A** – **[in]** The sparse matrix structure containing a sparse matrix of dimension ($m \cdot n$) that is created using aoclsparse_create_?csr.
- **descr** – **[in]** Descriptor of the sparse matrix can be one of the following: *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_triangular*, *aoclsparse_matrix_type_symmetric*, and *aoclsparse_matrix_type_hermitian*. Both base-zero and base-one are supported, however, the index base needs to match the one used at when aoclsparse_matrix was created.
- **x** – **[in]** An array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** Scalar β.
- **y** – **[inout]** An array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, internal structures related to the sparse matrix `A`, `x`, `beta` or `y` has an invalid pointer.
- **aoclsparse_status_not_implemented** – The requested functionality is not implemented.

aoclsparse_status **aoclsparse_zmv**(*aoclsparse_operation* op, const aoclsparse_double_complex *alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_double_complex *x, const aoclsparse_double_complex *beta, aoclsparse_double_complex *y)

Computes sparse matrix vector multiplication for real/complex single and double data precisions.

`aoclsparse_`(s/d/c/z)mv performs a sparse matrix vector multiplication such that

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

where, x and y are dense vectors, alpha and beta are scalars, and A is a sparse matrix structure. The matrix operation 'op' is defined as:

$$op(A) = \begin{cases} A, & \text{if op = aoclsparse\_operation\_none} \\ A^T, & \text{if op = aoclsparse\_operation\_transpose} \\ A^H, & \text{if op = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Parameters

- **op** – **[in]** Matrix operation.
- **alpha** – **[in]** Scalar α.
- **A** – **[in]** The sparse matrix structure containing a sparse matrix of dimension ($m \cdot n$) that is created using aoclsparse_create_?csr.
- **descr** – **[in]** Descriptor of the sparse matrix can be one of the following: *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_triangular*, *aoclsparse_matrix_type_symmetric*, and *aoclsparse_matrix_type_hermitian*. Both base-zero and base-one are supported, however, the index base needs to match the one used at when aoclsparse_matrix was created.
- **x** – **[in]** An array of `n` elements ($op(A) = A$) or `m` elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** Scalar β.
- **y** – **[inout]** An array of `m` elements ($op(A) = A$) or `n` elements ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, internal structures related to the sparse matrix `A`, `x`, `beta` or `y` has an invalid pointer.
- **aoclsparse_status_not_implemented** – The requested functionality is not implemented.

aoclsparse_status **aoclsparse_scsrsv**(*aoclsparse_operation* trans, const float *alpha, *aoclsparse_int* m, const float *csr_val, const *aoclsparse_int* *csr_col_ind, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_mat_descr* descr, const float *x, float *y)

Sparse triangular solve using CSR storage format for single and double data precisions.

`aoclsparse_`?srsv 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

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Note: The input matrix has to be sparse upper or lower triangular matrix with unit or non-unit main diagonal. Matrix has to be sorted. No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse CSR matrix.
- **csr_val** – **[in]** array of `nnz` elements of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the sparse CSR matrix.
- **x** – **[in]** array of `m` elements, holding the right-hand side.
- **y** – **[out]** array of `m` elements, holding the solution.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `csr_val`, `csr_row_ptr`, `csr_col_ind`, `x` or `y` pointer is invalid.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – `trans` = *aoclsparse_operation_conjugate_transpose* or `trans` = *aoclsparse_operation_transpose* or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_dcsrsv**(*aoclsparse_operation* trans, const double *alpha, *aoclsparse_int* m, const double *csr_val, const *aoclsparse_int* *csr_col_ind, const *aoclsparse_int* *csr_row_ptr, const *aoclsparse_mat_descr* descr, const double *x, double *y)

Sparse triangular solve using CSR storage format for single and double data precisions.

`aoclsparse_?srsv` 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

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: Currently, only `trans` = *aoclsparse_operation_none* is supported.

Note: The input matrix has to be sparse upper or lower triangular matrix with unit or non-unit main diagonal. Matrix has to be sorted. No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

Parameters

- **trans** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **m** – **[in]** number of rows of the sparse CSR matrix.
- **csr_val** – **[in]** array of `nnz` elements of the sparse CSR matrix.
- **csr_row_ptr** – **[in]** array of `m+1` elements that point to the start of every row of the sparse CSR matrix.
- **csr_col_ind** – **[in]** array of `nnz` elements containing the column indices of the sparse CSR matrix.
- **descr** – **[in]** descriptor of the sparse CSR matrix.
- **x** – **[in]** array of `m` elements, holding the right-hand side.
- **y** – **[out]** array of `m` elements, holding the solution.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m` is invalid.
- **aoclsparse_status_invalid_pointer** – `descr`, `alpha`, `csr_val`, `csr_row_ptr`, `csr_col_ind`, `x` or `y` pointer is invalid.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – `trans` = *aoclsparse_operation_conjugate_transpose* or `trans` = *aoclsparse_operation_transpose* or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*.

aoclsparse_status **aoclsparse_strsv**(*aoclsparse_operation* trans, const float alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const float *b, float *x)

Sparse triangular solver for real/complex single and double data precisions.

The functions `aoclsparse_?trsv` solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor `descr` specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the `descr`, the `fill_mode` entity specifies which triangle to consider, namely, if `fill_mode` = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **`trans`** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **`alpha`** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **`A`** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.
- **`descr`** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.
- **`b`** – **[in]** array of `m` elements, storing the right-hand side.
- **`x`** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **`kid`** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **`aoclsparse_status_success`** – the operation completed successfully and x contains the solution to the linear system of equations.
- **`aoclsparse_status_invalid_size`** – matrix A or $op(A)$ is invalid.
- **`aoclsparse_status_invalid_pointer`** – One or more of `A`, `descr`, `x`, `b` are invalid pointers.
- **`aoclsparse_status_internal_error`** – an internal error occurred.
- **`aoclsparse_status_not_implemented`** – the requested operation is not yet implemented.
- **`other`** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_dtrsv**(*aoclsparse_operation* trans, const double alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const double *b, double *x)

Sparse triangular solver for real/complex single and double data precisions.

The functions `aoclsparse_?trsv` solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor `descr` specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the `descr`, the `fill_mode` entity specifies which triangle to consider, namely, if `fill_mode` = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **trans** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.

- **alpha** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **A** – **[inout]** matrix data. A is modified only if solver requires to optimize matrix data.
- **descr** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.
- **b** – **[in]** array of m elements, storing the right-hand side.
- **x** – **[out]** array of m elements, storing the solution if solver returns *aoclsparse_status_success*.
- **kid** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **aoclsparse_status_success** – the operation completed successfully and x contains the solution to the linear system of equations.
- **aoclsparse_status_invalid_size** – matrix A or $op(A)$ is invalid.
- **aoclsparse_status_invalid_pointer** – One or more of A, descr, x, b are invalid pointers.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – the requested operation is not yet implemented.
- **other** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_ctrsv**(*aoclsparse_operation* trans, const aoclsparse_float_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_float_complex *b, aoclsparse_float_complex *x)

Sparse triangular solver for real/complex single and double data precisions.

The functions aoclsparse_?trsv solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor descr specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the descr, the fill_mode entity specifies which triangle to consider, namely, if fill_mode = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **`trans`** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **`alpha`** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **`A`** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.
- **`descr`** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.
- **`b`** – **[in]** array of `m` elements, storing the right-hand side.
- **`x`** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **`kid`** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **`aoclsparse_status_success`** – the operation completed successfully and x contains the solution to the linear system of equations.
- **`aoclsparse_status_invalid_size`** – matrix A or $op(A)$ is invalid.
- **`aoclsparse_status_invalid_pointer`** – One or more of `A`, `descr`, `x`, `b` are invalid pointers.
- **`aoclsparse_status_internal_error`** – an internal error occurred.
- **`aoclsparse_status_not_implemented`** – the requested operation is not yet implemented.
- **`other`** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **`aoclsparse_ztrsv`**(*aoclsparse_operation* trans, const aoclsparse_double_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_double_complex *b, aoclsparse_double_complex *x)

Sparse triangular solver for real/complex single and double data precisions.

The functions `aoclsparse_?trsv` solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor `descr` specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the `descr`, the `fill_mode` entity specifies which triangle to consider, namely, if `fill_mode` = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **`trans`** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **`alpha`** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **`A`** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.

- **descr** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.
- **b** – **[in]** array of `m` elements, storing the right-hand side.
- **x** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **kid** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **aoclsparse_status_success** – the operation completed successfully and x contains the solution to the linear system of equations.
- **aoclsparse_status_invalid_size** – matrix A or $op(A)$ is invalid.
- **aoclsparse_status_invalid_pointer** – One or more of `A`, `descr`, `x`, `b` are invalid pointers.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – the requested operation is not yet implemented.
- **other** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_strsv_kid**(*aoclsparse_operation* trans, const float alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const float *b, float *x, const *aoclsparse_int* kid)

Sparse triangular solver for real/complex single and double data precisions.

The functions `aoclsparse_?trsv` solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor `descr` specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the `descr`, the `fill_mode` entity specifies which triangle to consider, namely, if `fill_mode` = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **trans** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **alpha** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **A** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.
- **descr** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.
- **b** – **[in]** array of `m` elements, storing the right-hand side.
- **x** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **kid** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **aoclsparse_status_success** – the operation completed successfully and x contains the solution to the linear system of equations.
- **aoclsparse_status_invalid_size** – matrix A or $op(A)$ is invalid.
- **aoclsparse_status_invalid_pointer** – One or more of `A`, `descr`, `x`, `b` are invalid pointers.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – the requested operation is not yet implemented.
- **other** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_dtrsv_kid**(*aoclsparse_operation* trans, const double alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const double *b, double *x, const *aoclsparse_int* kid)

Sparse triangular solver for real/complex single and double data precisions.

The functions `aoclsparse_?trsv` solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense

m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor `descr` specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the `descr`, the `fill_mode` entity specifies which triangle to consider, namely, if `fill_mode` = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **`trans`** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **`alpha`** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **`A`** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.
- **`descr`** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.

- **b** – **[in]** array of m elements, storing the right-hand side.
- **x** – **[out]** array of m elements, storing the solution if solver returns *aoclsparse_status_success*.
- **kid** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **aoclsparse_status_success** – the operation completed successfully and x contains the solution to the linear system of equations.
- **aoclsparse_status_invalid_size** – matrix A or $op(A)$ is invalid.
- **aoclsparse_status_invalid_pointer** – One or more of A, descr, x, b are invalid pointers.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – the requested operation is not yet implemented.
- **other** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_ctrsv_kid**(*aoclsparse_operation* trans, const aoclsparse_float_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_float_complex *b, aoclsparse_float_complex *x, const *aoclsparse_int* kid)

Sparse triangular solver for real/complex single and double data precisions.

The functions aoclsparse_?trsv solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor descr specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the descr, the fill_mode entity specifies which triangle to consider, namely, if fill_mode = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if fill_mode = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **trans** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **alpha** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **A** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.
- **descr** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.
- **b** – **[in]** array of `m` elements, storing the right-hand side.
- **x** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **kid** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **aoclsparse_status_success** – the operation completed successfully and x contains the solution to the linear system of equations.
- **aoclsparse_status_invalid_size** – matrix A or $op(A)$ is invalid.
- **aoclsparse_status_invalid_pointer** – One or more of `A`, `descr`, `x`, `b` are invalid pointers.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – the requested operation is not yet implemented.
- **other** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_ztrsv_kid**(*aoclsparse_operation* trans, const aoclsparse_double_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_double_complex *b, aoclsparse_double_complex *x, const *aoclsparse_int* kid)

Sparse triangular solver for real/complex single and double data precisions.

The functions `aoclsparse_?trsv` solve sparse lower (or upper) triangular linear system of equations. The system is defined by the sparse $m \times m$ matrix A, the dense solution m-vector x, and the right-hand side dense

m-vector b. Vector b is multiplied by α. The solution x is estimated by solving

$$op(L) \cdot x = \alpha \cdot b, \quad \text{or} \quad op(U) \cdot x = \alpha \cdot b,$$

where $L = \text{tril}(A)$ is the lower triangle of matrix A, similarly, $U = \text{triu}(A)$ is the upper triangle of matrix A. The operator $op()$ is regarded as the matrix linear operation,

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Note: This routine supports only sparse matrices in CSR format.

Note: If the matrix descriptor `descr` specifies that the matrix A is to be regarded has having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are all considered to be unitary.

Note: The input matrix need not be (upper or lower) triangular matrix, in the `descr`, the `fill_mode` entity specifies which triangle to consider, namely, if `fill_mode` = *aoclsparse_fill_mode_lower*, then

$$op(L) \cdot x = \alpha \cdot b,$$

otherwise, if `fill_mode` = *aoclsparse_fill_mode_upper*, then

$$op(U) \cdot x = \alpha \cdot b$$

is solved.

Note: To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides b's, then it is encouraged to provide hints using `aoclsparse_set_sv_hint` and `aoclsparse_optimize`, otherwise, the optimization for the matrix will be done by the solver on entry.

Note: There is a `_kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, this solver allows to choose which TRSV kernel to use (if possible). Currently the possible choices are: `kid=0` Reference implementation (No explicit AVX instructions). `kid=1` Reference AVX 256-bit implementation only for double data precision and for operations *aoclsparse_operation_none* and *aoclsparse_operation_transpose*. `kid=2` Kernel Template version using AVX/AVX2 extensions. `kid=3` Kernel Template version using AVX512F+ CPU extensions. Any other Kernel ID value will default to `kid=0`.

Parameters

- **trans** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **alpha** – **[in]** scalar α, used to premultiply right-hand side vector b.
- **A** – **[inout]** matrix data. `A` is modified only if solver requires to optimize matrix data.
- **descr** – **[in]** matrix descriptor. Supported matrix types are *aoclsparse_matrix_type_symmetric* and *aoclsparse_matrix_type_triangular*.

- **b** – **[in]** array of `m` elements, storing the right-hand side.
- **x** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **kid** – **[in]** Kernel ID, hints a request on which TRSV kernel to use.

Return values

- **aoclsparse_status_success** – the operation completed successfully and x contains the solution to the linear system of equations.
- **aoclsparse_status_invalid_size** – matrix A or $op(A)$ is invalid.
- **aoclsparse_status_invalid_pointer** – One or more of `A`, `descr`, `x`, `b` are invalid pointers.
- **aoclsparse_status_internal_error** – an internal error occurred.
- **aoclsparse_status_not_implemented** – the requested operation is not yet implemented.
- **other** – possible failure values from a call to *aoclsparse_optimize*.

aoclsparse_status **aoclsparse_sdotmv**(const *aoclsparse_operation* op, const float alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const float *x, const float beta, float *y, float *d)

Performs sparse matrix-vector multiplication followed by vector-vector multiplication.

`aoclsparse_`?dotmv multiplies the scalar α with a sparse $m \times n$ matrix, defined in a sparse 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

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

followed by dot product of dense vectors x and y such that

$$\text{d} = \begin{cases} \sum_{i=0}^{min(m,n)-1} x_i * y_i, & \text{real case} \\ \sum_{i=0}^{min(m,n)-1} \text{conj}(x_i) * y_i, & \text{complex case} \end{cases}$$

Note: Currently, Hermitian matrix is not supported.

Parameters

- **op** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** the sparse $m \times n$ matrix structure that is created using `aoclsparse_create_(s/d/c/z)csr`
- **descr** – **[in]** descriptor of the sparse CSR matrix. Both base-zero and base-one are supported, however, the index base needs to match the one used when aoclsparse_matrix was created.
- **x** – **[in]** array of atleast `n` elements if $op(A) = A$ or atleast `m` elements if $op(A) = A^T or A^H$.

- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of atleast m elements if $op(A) = A$ or atleast n elements if $op(A) = A^T or A^H$.
- **d** – **[out]** dot product of y and x

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m, n or nnz is invalid.
- **aoclsparse_status_invalid_value** – (base != *aoclsparse_index_base_zero*) or, (base != *aoclsparse_index_base_one*) or, matrix base and descr base value do not match.
- **aoclsparse_status_invalid_pointer** – descr, internal structures related to the sparse matrix A, x, y or d are invalid pointer.
- **aoclsparse_status_wrong_type** – matrix data type is not supported.
- **aoclsparse_status_not_implemented** – (*aoclsparse_matrix_type* == *aoclsparse_matrix_type_hermitian*) or, (*aoclsparse_matrix_format_type* != *aoclsparse_csr_mat*)

aoclsparse_status **aoclsparse_ddotmv**(const *aoclsparse_operation* op, const double alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const double *x, const double beta, double *y, double *d)

Performs sparse matrix-vector multiplication followed by vector-vector multiplication.

aoclsparse_?dotmv multiplies the scalar α with a sparse $m \times n$ matrix, defined in a sparse 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

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

followed by dot product of dense vectors x and y such that

$$\text{d} = \begin{cases} \sum_{i=0}^{min(m,n)-1} x_i * y_i, & \text{real case} \\ \sum_{i=0}^{min(m,n)-1} \text{conj}(x_i) * y_i, & \text{complex case} \end{cases}$$

Note: Currently, Hermitian matrix is not supported.

Parameters

- **op** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** the sparse $m \times n$ matrix structure that is created using aoclsparse_create_(s/d/c/z)csr
- **descr** – **[in]** descriptor of the sparse CSR matrix. Both base-zero and base-one are supported, however, the index base needs to match the one used when aoclsparse_matrix was created.

- **x** – **[in]** array of atleast `n` elements if $op(A) = A$ or atleast `m` elements if $op(A) = A^T or A^H$.
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of atleast `m` elements if $op(A) = A$ or atleast `n` elements if $op(A) = A^T or A^H$.
- **d** – **[out]** dot product of y and x

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_value** – (base != *aoclsparse_index_base_zero*) or, (base != *aoclsparse_index_base_one*) or, matrix base and descr base value do not match.
- **aoclsparse_status_invalid_pointer** – `descr`, `internal` structures related to the sparse matrix `A`, `x`, `y` or `d` are invalid pointer.
- **aoclsparse_status_wrong_type** – matrix data type is not supported.
- **aoclsparse_status_not_implemented** – (*aoclsparse_matrix_type* == *aoclsparse_matrix_type_hermitian*) or, (*aoclsparse_matrix_format_type* != *aoclsparse_csr_mat*)

aoclsparse_status **aoclsparse_cdotmv**(const *aoclsparse_operation* op, const aoclsparse_float_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_float_complex *x, const aoclsparse_float_complex beta, aoclsparse_float_complex *y, aoclsparse_float_complex *d)

Performs sparse matrix-vector multiplication followed by vector-vector multiplication.

`aoclsparse_?dotmv` multiplies the scalar α with a sparse $m \times n$ matrix, defined in a sparse 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

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

followed by dot product of dense vectors x and y such that

$$\text{d} = \begin{cases} \sum_{i=0}^{min(m,n)-1} x_i * y_i, & \text{real case} \\ \sum_{i=0}^{min(m,n)-1} \text{conj}(x_i) * y_i, & \text{complex case} \end{cases}$$

Note: Currently, Hermitian matrix is not supported.

Parameters

- **op** – **[in]** matrix operation type.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** the sparse $m \times n$ matrix structure that is created using `aoclsparse_create_(s/d/c/z)csr`

- **descr** – **[in]** descriptor of the sparse CSR matrix. Both base-zero and base-one are supported, however, the index base needs to match the one used when aoclsparse_matrix was created.
- **x** – **[in]** array of atleast n elements if $op(A) = A$ or atleast m elements if $op(A) = A^T or A^H$.
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of atleast m elements if $op(A) = A$ or atleast n elements if $op(A) = A^T or A^H$.
- **d** – **[out]** dot product of y and x

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – m, n or nnz is invalid.
- **aoclsparse_status_invalid_value** – (base != *aoclsparse_index_base_zero*) or, (base != *aoclsparse_index_base_one*) or, matrix base and descr base value do not match.
- **aoclsparse_status_invalid_pointer** – descr, internal structures related to the sparse matrix A, x, y or d are invalid pointer.
- **aoclsparse_status_wrong_type** – matrix data type is not supported.
- **aoclsparse_status_not_implemented** – (*aoclsparse_matrix_type* == *aoclsparse_matrix_type_hermitian*) or, (*aoclsparse_matrix_format_type* != *aoclsparse_csr_mat*)

aoclsparse_status **aoclsparse_zdotmv**(const *aoclsparse_operation* op, const aoclsparse_double_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const aoclsparse_double_complex *x, const aoclsparse_double_complex beta, aoclsparse_double_complex *y, aoclsparse_double_complex *d)

Performs sparse matrix-vector multiplication followed by vector-vector multiplication.

aoclsparse_?dotmv multiplies the scalar α with a sparse $m \times n$ matrix, defined in a sparse 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

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

followed by dot product of dense vectors x and y such that

$$\text{d} = \begin{cases} \sum_{i=0}^{min(m,n)-1} x_i * y_i, & \text{real case} \\ \sum_{i=0}^{min(m,n)-1} \text{conj}(x_i) * y_i, & \text{complex case} \end{cases}$$

Note: Currently, Hermitian matrix is not supported.

Parameters

- **op** – **[in]** matrix operation type.

- **alpha** – **[in]** scalar α.
- **A** – **[in]** the sparse $m \times n$ matrix structure that is created using `aoclsparse_create_(s/d/c/z)csr`
- **descr** – **[in]** descriptor of the sparse CSR matrix. Both base-zero and base-one are supported, however, the index base needs to match the one used when aoclsparse_matrix was created.
- **x** – **[in]** array of atleast `n` elements if $op(A) = A$ or atleast `m` elements if $op(A) = A^T or A^H$.
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of atleast `m` elements if $op(A) = A$ or atleast `n` elements if $op(A) = A^T or A^H$.
- **d** – **[out]** dot product of y and x

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – `m`, `n` or `nnz` is invalid.
- **aoclsparse_status_invalid_value** – (base != *aoclsparse_index_base_zero*) or, (base != *aoclsparse_index_base_one*) or, matrix base and descr base value do not match.
- **aoclsparse_status_invalid_pointer** – `descr`, `internal` structures related to the sparse matrix `A`, `x`, `y` or `d` are invalid pointer.
- **aoclsparse_status_wrong_type** – matrix data type is not supported.
- **aoclsparse_status_not_implemented** – (*aoclsparse_matrix_type* == *aoclsparse_matrix_type_hermitian*) or, (*aoclsparse_matrix_format_type* != *aoclsparse_csr_mat*)

4.3 Level 3

aoclsparse_status **aoclsparse_strsm**(const *aoclsparse_operation* trans, const float alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const float *B, *aoclsparse_int* n, *aoclsparse_int* ldb, float *X, *aoclsparse_int* ldx)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\ X = \alpha\ B, \text{ with solution } X = \alpha\ (op(A)^{-1})\ B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none}, \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose}, \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose}. \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\ X = \alpha\ B, \text{ with solution } X = \alpha\ A^{-T}\ B, \text{ and }\ A^H\ X = \alpha\ B, \text{ with solution } X = \alpha\ A^{-H}\ B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **`trans`** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **`alpha`** – **[in]** scalar α.
- **`A`** – **[in]** sparse matrix A of size m.
- **`descr`** – **[in]** descriptor of the sparse matrix A.
- **`order`** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **`B`** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **`n`** – **[in]** n, number of columns of the dense matrix B.
- **`ldb`** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the

submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.
- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_dtrsm**(const *aoclsparse_operation* trans, const double alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const double *B, *aoclsparse_int* n, *aoclsparse_int* ldb, double *X, *aoclsparse_int* ldx)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\ X = \alpha\ B, \text{ with solution } X = \alpha\ (op(A)^{-1})\ B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none}, \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose}, \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose}. \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\ X = \alpha\ B, \text{ with solution } X = \alpha\ A^{-T}\ B, \text{ and }\ A^H\ X = \alpha\ B, \text{ with solution } X = \alpha\ A^{-H}\ B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **order** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **B** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **n** – **[in]** n, number of columns of the dense matrix B.
- **ldb** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.
- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_ctrsm**(*aoclsparse_operation* trans, const aoclsparse_float_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const aoclsparse_float_complex *B, *aoclsparse_int* n, *aoclsparse_int* ldb, aoclsparse_float_complex *X, *aoclsparse_int* ldx)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\;X = \alpha\;B, \text{with solution } X = \alpha\;(op(A)^{-1})\;B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none,} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose,} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose.} \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\;X = \alpha\;B, \text{with solution } X = \alpha\;A^{-T}\;B, \text{and }\; A^H\;X = \alpha\;B, \text{with solution } X = \alpha\;A^{-H}\;B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **order** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **B** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **n** – **[in]** n, number of columns of the dense matrix B.
- **ldb** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.
- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_ztrsm**(*aoclsparse_operation* trans, const aoclsparse_double_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const aoclsparse_double_complex *B, *aoclsparse_int* n, *aoclsparse_int* ldb, aoclsparse_double_complex *X, *aoclsparse_int* ldx)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\;X = \alpha\;B, \text{with solution } X = \alpha\;(op(A)^{-1})\;B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none}, \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose}, \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose}. \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\;X = \alpha\;B, \text{with solution } X = \alpha\;A^{-T}\;B, \text{and}\quad A^H\;X = \alpha\;B, \text{with solution } X = \alpha\;A^{-H}\;B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **order** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **B** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **n** – **[in]** n, number of columns of the dense matrix B.
- **ldb** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either m, n, nnz, ldb or ldx is invalid.
- **aoclsparse_status_invalid_pointer** – informs that either descr, alpha, A, B, or X pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix A is not in CSR format.

aoclsparse_status **aoclsparse_strsm_kid**(const *aoclsparse_operation* trans, const float alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const float *B, *aoclsparse_int* n, *aoclsparse_int* ldb, float *X, *aoclsparse_int* ldx, const *aoclsparse_int* kid)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

aoclsparse_?trsm solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by fill_mode from the matrix descriptor descr where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see ldb and ldx input parameters for further details.

Explicitly, this kernel solves

$$op(A)\;X = \alpha\;B, \text{ with solution } X = \alpha\;(op(A)^{-1})\;B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans = aoclsparse\_operation\_none,} \\ A^T, & \text{if trans = aoclsparse\_operation\_transpose,} \\ A^H, & \text{if trans = aoclsparse\_operation\_conjugate\_transpose.} \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-T}\;B, \text{ and } \; A^H\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-H}\;B.$$

1. If the matrix descriptor descr specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **order** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **B** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **n** – **[in]** n, number of columns of the dense matrix B.
- **ldb** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.
- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_dtrsm_kid**(const *aoclsparse_operation* trans, const double alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const double *B, *aoclsparse_int* n, *aoclsparse_int* ldb, double *X, *aoclsparse_int* ldx, const *aoclsparse_int* kid)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\;X = \alpha\;B, \text{ with solution } X = \alpha\;(op(A)^{-1})\;B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none}, \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose}, \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose}. \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-T}\;B, \text{ and }\;A^H\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-H}\;B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **order** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **B** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **n** – **[in]** n, number of columns of the dense matrix B.
- **ldb** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.

- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_ctrsm_kid**(*aoclsparse_operation* trans, const aoclsparse_float_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const aoclsparse_float_complex *B, *aoclsparse_int* n, *aoclsparse_int* ldb, aoclsparse_float_complex *X, *aoclsparse_int* ldx, const *aoclsparse_int* kid)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropriate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\;X = \alpha\;B, \text{ with solution } X = \alpha\;(op(A)^{-1})\;B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none}, \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose}, \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose}. \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-T}\;B, \text{ and }\; A^H\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-H}\;B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **`trans`** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **`alpha`** – **[in]** scalar α.
- **`A`** – **[in]** sparse matrix A of size m.
- **`descr`** – **[in]** descriptor of the sparse matrix A.
- **`order`** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **`B`** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **`n`** – **[in]** n, number of columns of the dense matrix B.
- **`ldb`** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **`X`** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **`ldx`** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **`kid`** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **`aoclsparse_status_success`** – indicates that the operation completed successfully.
- **`aoclsparse_status_invalid_size`** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.
- **`aoclsparse_status_invalid_pointer`** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.

- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_ztrsm_kid**(*aoclsparse_operation* trans, const aoclsparse_double_complex alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const aoclsparse_double_complex *B, *aoclsparse_int* n, *aoclsparse_int* ldb, aoclsparse_double_complex *X, *aoclsparse_int* ldx, const *aoclsparse_int* kid)

Solve sparse triangular linear system of equations with multiple right hand sides for real/complex single and double data precisions.

`aoclsparse_?trsm` solves a sparse triangular linear system of equations with multiple right hand sides, of the form

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

where A is a sparse matrix of size m, $op()$ is a linear operator, X and B are rectangular dense matrices of appropiate size, while α is a scalar. The sparse matrix A can be interpreted either as a lower triangular or upper triangular. This is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix is only referenced. The matrix can also be of class symmetric in which case only the selected triangular part is used. Matrix A must be of full rank, that is, the matrix must be invertible. The linear operator $op()$ can define the transposition or Hermitian transposition operations. By default, no transposition is performed. The right-hand-side matrix B and the solution matrix X are dense and must be of the correct size, that is m by n, see `ldb` and `ldx` input parameters for further details.

Explicitly, this kernel solves

$$op(A)\;X = \alpha\;B, \text{ with solution } X = \alpha\;(op(A)^{-1})\;B,$$

where

$$op(A) = \begin{cases} A, & \text{if trans} = \text{aoclsparse\_operation\_none}, \\ A^T, & \text{if trans} = \text{aoclsparse\_operation\_transpose}, \\ A^H, & \text{if trans} = \text{aoclsparse\_operation\_conjugate\_transpose}. \end{cases}$$

If a linear operator is applied then, the possible problems solved are

$$A^T\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-T}\;B, \text{ and }\;A^H\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-H}\;B.$$

1. If the matrix descriptor `descr` specifies that the matrix A is to be regarded as having a unitary diagonal, then the main diagonal entries of matrix A are not accessed and are considered to all be ones.

2. If the matrix A is described as upper triangular, then only the upper triangular portion of the matrix is referenced. Conversely, if the matrix A is described lower triangular, then only the lower triangular portion of the matrix is used.

3. This set of APIs allocates work array of size m for each case where the matrices B or X are stored in row-major format (*aoclsparse_order_row*).

4. A subset of kernels are parallel (on parallel builds) and can be expected potential acceleration in the solve. These kernels are available when both dense matrices X and B are stored in column-major format (*aoclsparse_order_column*) and thread count is greater than 1 on a parallel build.

5. There is `_kid` (Kernel ID) variation of TRSM, namely with a suffix of `_kid`, this solver allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing `aoclsparse_?trsm` kernels are supported.

6. This routine supports only sparse matrices in CSR format.

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and aoclsparse_operation_conjugate_transpose.
- **alpha** – **[in]** scalar α.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **order** – **[in]** storage order of dense matrices B and X. Possible options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **B** – **[in]** dense matrix, potentially rectangular, of size $m \times n$.
- **n** – **[in]** n, number of columns of the dense matrix B.
- **ldb** – **[in]** leading dimension of B. Eventhough the matrix B is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldb` by $N > n$) in which only the submatrix B is of interest. In this case, this parameter provides means to access the correct elements of B within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldb` with `ldb` $\geq n$
aoclsparse_order_column	`ldb` with `ldb` $\geq m$	n

- **X** – **[out]** solution matrix X, dense and potentially rectangular matrix of size $m \times n$.
- **ldx** – **[in]** leading dimension of X. Eventhough the matrix X is considered of size $m \times n$, its memory layout may correspond to a larger matrix (`ldx` by $N > n$) in which only the submatrix X is of interest. In this case, this parameter provides means to access the correct elements of X within the larger layout.

matrix layout	row count	column count
aoclsparse_order_row	m	`ldx` with `ldx` $\geq n$
aoclsparse_order_column	`ldx` with `ldx` $\geq m$	n

- **kid** – **[in]** kernel ID, hints a request on which kernel to use (see notes).

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n`, `nnz`, `ldb` or `ldx` is invalid.
- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `alpha`, `A`, `B`, or `X` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* or *aoclsparse_matrix_type_hermitian* or when matrix `A` is not in CSR format.

aoclsparse_status **aoclsparse_sp2m**(*aoclsparse_operation* opA, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* A, *aoclsparse_operation* opB, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* B, const *aoclsparse_request* request, *aoclsparse_matrix* *C)

Sparse matrix Sparse matrix multiplication for real and complex datatypes.

`aoclsparse_sp2m` multiplies two sparse matrices in CSR storage format. The result is stored in a newly allocated sparse matrix in CSR format, such that

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

with

$$op(A) = \begin{cases} A, & \text{if opA} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if opA} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if opA} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

and

$$op(B) = \begin{cases} B, & \text{if opB} = \text{aoclsparse\_operation\_none} \\ B^T, & \text{if opB} = \text{aoclsparse\_operation\_transpose} \\ B^H, & \text{if opB} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

where A is a $m \times k$ matrix , B is a $k \times n$ matrix, resulting in $m \times n$ matrix C, for opA and opB = *aoclsparse_operation_none*. A is a $k \times m$ matrix when opA = *aoclsparse_operation_transpose* or *aoclsparse_operation_conjugate_transpose* and B is a $n \times k$ matrix when opB = *aoclsparse_operation_transpose* or *aoclsparse_operation_conjugate_transpose*

aoclsparse_sp2m can be run in single-stage or two-stage. The single-stage algorithm allocates and computes the entire output matrix in a single stage *aoclsparse_stage_full_computation*. Whereas, in two-stage algorithm, the first stage *aoclsparse_stage_nnz_count* allocates memory for the output matrix and computes the number of entries of the matrix. The second stage *aoclsparse_stage_finalize* computes column indices of non-zero elements and values of the output matrix. The second stage has to be invoked only after the first stage. But, it can be also be invoked multiple times consecutively when the sparsity structure of input matrices remains unchanged, with only the values getting updated.

Example

Shows multiplication of 2 sparse matrices to give a newly allocated sparse matrix

```
aoclsparse_matrix  A;
aoclsparse_create_dcsr(&A, base, M, K, nnz_A, csr_row_ptr_A.data(), csr_col_ind_
↪A.data(), csr_val_A.data());
aoclsparse_matrix  B;
aoclsparse_create_dcsr(&B, base, K, N, nnz_B, csr_row_ptr_B.data(), csr_col_ind_
↪B.data(), csr_val_B.data());

aoclsparse_matrix  C = NULL;
aoclsparse_int *csr_row_ptr_C = NULL;
aoclsparse_int *csr_col_ind_C = NULL;
double             *csr_val_C = NULL;
aoclsparse_int C_M, C_N;
aoclsparse_status status;
request =  aoclsparse_stage_full_computation;
status = aoclsparse_sp2m(opA,
        descrA,
```

(continues on next page)

(continued from previous page)

```
        A,
        opB,
        descrB,
        B,
        request,
        &C);

aoclsparse_export_dcsr(C, &base, &C_M, &C_N, &nnz_C, &csr_row_ptr_C, &csr_col_
↪ind_C, (void **)&csr_val_C);
```

Parameters

- **opA** – **[in]** matrix A operation type.
- **descrA** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported.
- **A** – **[in]** sparse CSR matrix A .
- **opB** – **[in]** matrix B operation type.
- **descrB** – **[in]** descriptor of the sparse CSR matrix B. Currently, only *aoclsparse_matrix_type_general* is supported.
- **B** – **[in]** sparse CSR matrix B .
- **request** – **[in]** Specifies full computation or two-stage algorithm *aoclsparse_stage_nnz_count* , Only rowIndex array of the CSR matrix is computed internally. The output sparse CSR matrix can be extracted to measure the memory required for full operation. *aoclsparse_stage_finalize* . Finalize computation of remaining output arrays (column indices and values of output matrix entries) . Has to be called only after aoclsparse_sp2m call with aoclsparse_stage_nnz_count parameter. *aoclsparse_stage_full_computation* . Perform the entire computation in a single step.
- ***C** – **[out]** Pointer to sparse CSR matrix C . Matrix C arrays will always have zero-based indexing, irrespective of matrix A or matrix B being one-based or zero-based indexing. The column indices of the output matrix in CSR format can appear unsorted.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – descrA, descrB, A, B, C is invalid.
- **aoclsparse_status_invalid_size** – input size parameters contain an invalid value.
- **aoclsparse_status_invalid_value** – input parameters contain an invalid value.
- **aoclsparse_status_wrong_type** – A and B matrix datatypes dont match.
- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrices A or B is not in CSR format

aoclsparse_status **aoclsparse_spmm**(*aoclsparse_operation* opA, const *aoclsparse_matrix* A, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

Sparse matrix Sparse matrix multiplication for real and complex datatypes.

`aoclsparse_spmm` multiplies two sparse matrices in CSR storage format. The result is stored in a newly allocated sparse matrix in CSR format, such that

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

with

$$op(A) = \begin{cases} A, & \text{if opA} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if opA} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if opA} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

where A is a $m \times k$ matrix , B is a $k \times n$ matrix, resulting in $m \times n$ matrix C, for opA = *aoclsparse_operation_none*. A is a $k \times m$ matrix when opA = *aoclsparse_operation_transpose* or *aoclsparse_operation_conjugate_transpose*

Example

Shows multiplication of 2 sparse matrices to give a newly allocated sparse matrix

```
aoclsparse_matrix  A;
aoclsparse_create_dcsr(&A, base, M, K, nnz_A, csr_row_ptr_A.data(), csr_col_ind_
↪A.data(), csr_val_A.data());
aoclsparse_matrix  B;
aoclsparse_create_dcsr(&B, base, K, N, nnz_B, csr_row_ptr_B.data(), csr_col_ind_
↪B.data(), csr_val_B.data());

aoclsparse_matrix  C = NULL;
aoclsparse_int *csr_row_ptr_C = NULL;
aoclsparse_int *csr_col_ind_C = NULL;
double              *csr_val_C = NULL;
aoclsparse_int C_M, C_N;
aoclsparse_status status;
status = aoclsparse_spmm(opA,
        A,
        B,
        &C);

aoclsparse_export_dcsr(C, &base, &C_M, &C_N, &nnz_C, &csr_row_ptr_C, &csr_col_
↪ind_C, (void **)&csr_val_C);
```

Parameters

- **opA** – **[in]** matrix A operation type.
- **A** – **[in]** sparse CSR matrix A.
- **B** – **[in]** sparse CSR matrix B.
- ***C** – **[out]** Pointer to sparse CSR matrix C . Matrix C arrays will always have zero-based indexing, irrespective of matrix A or matrix B being one-based or zero-based indexing. The column indices of the output matrix in CSR format can appear unsorted.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_pointer** – A, B, C is invalid.
- **aoclsparse_status_invalid_size** – input size parameters contain an invalid value.

- **aoclsparse_status_invalid_value** – input parameters contain an invalid value.
- **aoclsparse_status_wrong_type** – A and B matrix datatypes dont match.
- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – Input matrices `A` or \B is not in CSR format

aoclsparse_status **aoclsparse_scsrmm**(*aoclsparse_operation* op, const float alpha, const *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const float *B, *aoclsparse_int* n, *aoclsparse_int* ldb, const float beta, float *C, *aoclsparse_int* ldc)

Sparse matrix dense matrix multiplication using CSR storage format.

`aoclsparse_`(s/d/c/z)csrmm multiplies a scalar α with a sparse $m \times k$ matrix A, defined in CSR storage format, and a dense $k \times n$ matrix B and adds the result to the dense $m \times n$ matrix C that is multiplied by a scalar β, such that

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

with

$$op(A) = \begin{cases} A, & \text{if trans\_A} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

```
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];
        }
    }
}
```

Parameters

- **Op** – **[in]** Matrix A operation type.
- **Alpha** – **[in]** Scalar α.
- **A** – **[in]** Sparse CSR matrix A structure.
- **descr** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported. Both, base-zero and base-one input arrays of CSR matrix are supported
- **Order** – **[in]** Aoclsparse_order_row/aoclsparse_order_column for dense matrix
- **B** – **[in]** Array of dimension $ldb \times n$ or $ldb \times k$.
- **N** – **[in]** Number of columns of the dense matrix B and C.
- **Ldb** – **[in]** Leading dimension of B, must be at least $\max(1, k)$ ($op(A) = A$) or $\max(1, m)$ ($op(A) = A^T$ or $op(A) = A^H$).

- **Beta** – **[in]** Scalar β.
- **C** – **[inout]** Array of dimension $ldc \times n$.
- **Ldc** – **[in]** Leading dimension of C, must be at least $\max(1, m)$ ($op(A) = A$) or $\max(1, k)$ ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of `m`, `n`, `k`, `nnz`, `ldb` or `ldc` is invalid.
- **aoclsparse_status_invalid_pointer** – The pointer `descr`, `A`, `B`, or `C` is invalid.
- **aoclsparse_status_invalid_value** – The value of `descr->base`, `A->base` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrix `A` is not in CSR format

aoclsparse_status **aoclsparse_dcsrmm**(*aoclsparse_operation* op, const double alpha, const *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const double *B, *aoclsparse_int* n, *aoclsparse_int* ldb, const double beta, double *C, *aoclsparse_int* ldc)

Sparse matrix dense matrix multiplication using CSR storage format.

`aoclsparse_`(s/d/c/z)csrmm multiplies a scalar α with a sparse $m \times k$ matrix A, defined in CSR storage format, and a dense $k \times n$ matrix B and adds the result to the dense $m \times n$ matrix C that is multiplied by a scalar β, such that

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

with

$$op(A) = \begin{cases} A, & \text{if trans\_A = aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

```
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];
        }
    }
}
```

Parameters

- **Op** – **[in]** Matrix A operation type.
- **Alpha** – **[in]** Scalar α.
- **A** – **[in]** Sparse CSR matrix A structure.

- **descr** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported. Both, base-zero and base-one input arrays of CSR matrix are supported
- **Order** – **[in]** Aoclsparse_order_row/aoclsparse_order_column for dense matrix
- **B** – **[in]** Array of dimension $ldb \times n$ or $ldb \times k$.
- **N** – **[in]** Number of columns of the dense matrix B and C.
- **Ldb** – **[in]** Leading dimension of B, must be at least $\max(1, k)$ ($op(A) = A$) or $\max(1, m)$ ($op(A) = A^T$ or $op(A) = A^H$).
- **Beta** – **[in]** Scalar β.
- **C** – **[inout]** Array of dimension $ldc \times n$.
- **Ldc** – **[in]** Leading dimension of C, must be at least $\max(1, m)$ ($op(A) = A$) or $\max(1, k)$ ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of `m`, `n`, `k`, `nnz`, `ldb` or `ldc` is invalid.
- **aoclsparse_status_invalid_pointer** – The pointer `descr`, `A`, `B`, or `C` is invalid.
- **aoclsparse_status_invalid_value** – The value of `descr->base`, `A->base` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrix `A` is not in CSR format

aoclsparse_status **aoclsparse_ccsrmm**(*aoclsparse_operation* op, const aoclsparse_float_complex alpha, const *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const aoclsparse_float_complex *B, *aoclsparse_int* n, *aoclsparse_int* ldb, const aoclsparse_float_complex beta, aoclsparse_float_complex *C, *aoclsparse_int* ldc)

Sparse matrix dense matrix multiplication using CSR storage format.

`aoclsparse_`(s/d/c/z)csrmm multiplies a scalar α with a sparse $m \times k$ matrix A, defined in CSR storage format, and a dense $k \times n$ matrix B and adds the result to the dense $m \times n$ matrix C that is multiplied by a scalar β, such that

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

with

$$op(A) = \begin{cases} A, & \text{if trans\_A = aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

```
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)
        {
```

(continues on next page)

(continued from previous page)

```
            C[i][j] += alpha * csr_val[k] * B[csr_col_ind[k]][j];
        }
    }
}
```

Parameters

- **Op** – **[in]** Matrix A operation type.
- **Alpha** – **[in]** Scalar α.
- **A** – **[in]** Sparse CSR matrix A structure.
- **descr** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported. Both, base-zero and base-one input arrays of CSR matrix are supported
- **Order** – **[in]** Aoclsparse_order_row/aoclsparse_order_column for dense matrix
- **B** – **[in]** Array of dimension $ldb \times n$ or $ldb \times k$.
- **N** – **[in]** Number of columns of the dense matrix B and C.
- **Ldb** – **[in]** Leading dimension of B, must be at least $\max(1, k)$ ($op(A) = A$) or $\max(1, m)$ ($op(A) = A^T$ or $op(A) = A^H$).
- **Beta** – **[in]** Scalar β.
- **C** – **[inout]** Array of dimension $ldc \times n$.
- **Ldc** – **[in]** Leading dimension of C, must be at least $\max(1, m)$ ($op(A) = A$) or $\max(1, k)$ ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_size** – The value of `m`, `n`, `k`, `nnz`, `ldb` or `ldc` is invalid.
- **aoclsparse_status_invalid_pointer** – The pointer `descr`, `A`, `B`, or `C` is invalid.
- **aoclsparse_status_invalid_value** – The value of `descr->base`, `A->base` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrix `A` is not in CSR format

aoclsparse_status **aoclsparse_zcsrmm**(*aoclsparse_operation* op, const aoclsparse_double_complex alpha, const *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, *aoclsparse_order* order, const aoclsparse_double_complex *B, *aoclsparse_int* n, *aoclsparse_int* ldb, const aoclsparse_double_complex beta, aoclsparse_double_complex *C, *aoclsparse_int* ldc)

Sparse matrix dense matrix multiplication using CSR storage format.

`aoclsparse_`(s/d/c/z)csrmm multiplies a scalar α with a sparse $m \times k$ matrix A, defined in CSR storage format, and a dense $k \times n$ matrix B and adds the result to the dense $m \times n$ matrix C that is multiplied by a scalar β, such that

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

with

$$op(A) = \begin{cases} A, & \text{if trans\_A} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

```
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];
        }
    }
}
```

Parameters

- **`Op`** – **[in]** Matrix A operation type.
- **`Alpha`** – **[in]** Scalar α.
- **`A`** – **[in]** Sparse CSR matrix A structure.
- **`descr`** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported. Both, base-zero and base-one input arrays of CSR matrix are supported
- **`Order`** – **[in]** Aoclsparse_order_row/aoclsparse_order_column for dense matrix
- **`B`** – **[in]** Array of dimension $ldb \times n$ or $ldb \times k$.
- **`N`** – **[in]** Number of columns of the dense matrix B and C.
- **`Ldb`** – **[in]** Leading dimension of B, must be at least $\max(1, k)$ ($op(A) = A$) or $\max(1, m)$ ($op(A) = A^T$ or $op(A) = A^H$).
- **`Beta`** – **[in]** Scalar β.
- **`C`** – **[inout]** Array of dimension $ldc \times n$.
- **`Ldc`** – **[in]** Leading dimension of C, must be at least $\max(1, m)$ ($op(A) = A$) or $\max(1, k)$ ($op(A) = A^T$ or $op(A) = A^H$).

Return values

- **`aoclsparse_status_success`** – The operation completed successfully.
- **`aoclsparse_status_invalid_size`** – The value of `m`, `n`, `k`, `nnz`, `ldb` or `ldc` is invalid.
- **`aoclsparse_status_invalid_pointer`** – The pointer `descr`, `A`, `B`, or `C` is invalid.
- **`aoclsparse_status_invalid_value`** – The value of `descr->base`, `A->base` is invalid.
- **`aoclsparse_status_not_implemented`** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrix `A` is not in CSR format

aoclsparse_status **aoclsparse_dcsr2m**(*aoclsparse_operation* trans_A, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* csrA, *aoclsparse_operation* trans_B, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* csrB, const *aoclsparse_request* request, *aoclsparse_matrix* *csrC)

Sparse matrix Sparse matrix multiplication using CSR storage format for single and double precision datatypes.

`aoclsparse_csr2m` multiplies a 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 stores the result to the sparse $m \times n$ matrix C, such that

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

with

$$op(A) = \begin{cases} A, & \text{if trans\_A} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

and

$$op(B) = \begin{cases} B, & \text{if trans\_B} = \text{aoclsparse\_operation\_none} \\ B^T, & \text{if trans\_B} = \text{aoclsparse\_operation\_transpose} \\ B^H, & \text{if trans\_B} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Example

Shows multiplication of 2 sparse matrices to give a newly allocated sparse matrix

```
    aoclsparse_matrix  csrA;
    aoclsparse_create_dcsr(&csrA, base, M, K, nnz_A, csr_row_ptr_A.data(), csr_
↪col_ind_A.data(), csr_val_A.data());
    aoclsparse_matrix  csrB;
    aoclsparse_create_dcsr(&csrB, base, K, N, nnz_B, csr_row_ptr_B.data(), csr_
↪col_ind_B.data(), csr_val_B.data());

aoclsparse_matrix  csrC = NULL;
aoclsparse_int *csr_row_ptr_C = NULL;
aoclsparse_int *csr_col_ind_C = NULL;
double             *csr_val_C = NULL;
aoclsparse_int C_M, C_N;
request =  aoclsparse_stage_nnz_count;
CHECK_AOCLSPARSE_ERROR(aoclsparse_dcsr2m(transA,
    descrA,
    csrA,
    transB,
    descrB,
    csrB,
    request,
    &csrC));

request =  aoclsparse_stage_finalize;
CHECK_AOCLSPARSE_ERROR(aoclsparse_dcsr2m(transA,
    descrA,
    csrA,
    transB,
    descrB,
```

(continues on next page)

(continued from previous page)

```
    csrB,
    request,
    &csrC));
aoclsparse_export_mat_csr(csrC, &base, &C_M, &C_N, &nnz_C, &csr_row_ptr_C, &csr_
↪col_ind_C, (void **)&csr_val_C);
```

Parameters

- **trans_A** – **[in]** matrix A operation type.
- **descrA** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported.
- **csrA** – **[in]** sparse CSR matrix A structure.
- **trans_B** – **[in]** matrix B operation type.
- **descrB** – **[in]** descriptor of the sparse CSR matrix B. Currently, only *aoclsparse_matrix_type_general* is supported.
- **csrB** – **[in]** sparse CSR matrix B structure.
- **request** – **[in]** Specifies full computation or two-stage algorithm *aoclsparse_stage_nnz_count* , Only rowIndex array of the CSR matrix is computed internally. The output sparse CSR matrix can be extracted to measure the memory required for full operation. *aoclsparse_stage_finalize* . Finalize computation of remaining output arrays (column indices and values of output matrix entries) . Has to be called only after aoclsparse_dcsr2m call with aoclsparse_stage_nnz_count parameter. *aoclsparse_stage_full_computation* . Perform the entire computation in a single step.
- ***csrC** – **[out]** Pointer to sparse CSR matrix C structure.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – input parameters contain an invalid value.
- **aoclsparse_status_invalid_pointer** – `descrA`, `csr`, `descrB`, `csrB`, `csrC` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrices `A` or `B` is not in CSR format

aoclsparse_status **aoclsparse_scsr2m**(*aoclsparse_operation* trans_A, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* csrA, *aoclsparse_operation* trans_B, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* csrB, const *aoclsparse_request* request, *aoclsparse_matrix* *csrC)

Sparse matrix Sparse matrix multiplication using CSR storage format for single and double precision datatypes.

`aoclsparse_csr2m` multiplies a 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 stores the result to the sparse $m \times n$ matrix C, such that

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

with

$$op(A) = \begin{cases} A, & \text{if trans\_A = aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A = aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A = aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

and

$$op(B) = \begin{cases} B, & \text{if trans\_B} = \text{aoclsparse\_operation\_none} \\ B^T, & \text{if trans\_B} = \text{aoclsparse\_operation\_transpose} \\ B^H, & \text{if trans\_B} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Example

Shows multiplication of 2 sparse matrices to give a newly allocated sparse matrix

```
    aoclsparse_matrix  csrA;
    aoclsparse_create_dcsr(&csrA, base, M, K, nnz_A, csr_row_ptr_A.data(), csr_
↪col_ind_A.data(), csr_val_A.data());
    aoclsparse_matrix  csrB;
    aoclsparse_create_dcsr(&csrB, base, K, N, nnz_B, csr_row_ptr_B.data(), csr_
↪col_ind_B.data(), csr_val_B.data());

aoclsparse_matrix  csrC = NULL;
aoclsparse_int *csr_row_ptr_C = NULL;
aoclsparse_int *csr_col_ind_C = NULL;
double             *csr_val_C = NULL;
aoclsparse_int C_M, C_N;
request =  aoclsparse_stage_nnz_count;
CHECK_AOCLSPARSE_ERROR(aoclsparse_dcsr2m(transA,
    descrA,
    csrA,
    transB,
    descrB,
    csrB,
    request,
    &csrC));

request =  aoclsparse_stage_finalize;
CHECK_AOCLSPARSE_ERROR(aoclsparse_dcsr2m(transA,
    descrA,
    csrA,
    transB,
    descrB,
    csrB,
    request,
    &csrC));
aoclsparse_export_mat_csr(csrC, &base, &C_M, &C_N, &nnz_C, &csr_row_ptr_C, &csr_
↪col_ind_C, (void **)&csr_val_C);
```

Parameters

- **trans_A** – **[in]** matrix A operation type.
- **descrA** – **[in]** descriptor of the sparse CSR matrix A. Currently, only *aoclsparse_matrix_type_general* is supported.
- **csrA** – **[in]** sparse CSR matrix A structure.
- **trans_B** – **[in]** matrix B operation type.
- **descrB** – **[in]** descriptor of the sparse CSR matrix B. Currently, only *aoclsparse_matrix_type_general* is supported.

- **csrB** – **[in]** sparse CSR matrix B structure.
- **request** – **[in]** Specifies full computation or two-stage algorithm *aoclsparse_stage_nnz_count* , Only rowIndex array of the CSR matrix is computed internally. The output sparse CSR matrix can be extracted to measure the memory required for full operation. *aoclsparse_stage_finalize* . Finalize computation of remaining output arrays (column indices and values of output matrix entries) . Has to be called only after aoclsparse_dcsr2m call with aoclsparse_stage_nnz_count parameter. *aoclsparse_stage_full_computation* . Perform the entire computation in a single step.
- ***csrC** – **[out]** Pointer to sparse CSR matrix C structure.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – input parameters contain an invalid value.
- **aoclsparse_status_invalid_pointer** – `descrA`, `csr`, `descrB`, `csrB`, `csrC` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general* or input matrices `A` or `B` is not in CSR format

aoclsparse_status **aoclsparse_sadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const float alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

Addition of two sparse matrices.

`aoclsparse_(s/d/c/z)add` sums two sparse matrices and returns the result as a newly allocated sparse matrix for real and complex types, respectively. It performs the following operation:

$$C = \alpha * op(A) + B$$

with

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

where A is a $m \times n$ matrix and B is a $m \times n$ matrix if op = *aoclsparse_operation_none* and $n \times m$ otherwise and the result matrix C has the same dimension as B.

Note: Only matrices in CSR format are supported in this release.

Parameters

- **op** – **[in]** matrix A operation type.
- **alpha** – **[in]** scalar with same precision as A and B matrix
- **A** – **[in]** source sparse matrix A
- **B** – **[in]** source sparse matrix B
- ***C** – **[out]** pointer to the sparse output matrix C

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `A` or `B` or `C` are invalid
- **aoclsparse_status_invalid_size** – The dimensions of `A` and `B` are not compatible.

- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – Matrices are not in CSR format.

aoclsparse_status **aoclsparse_dadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const double alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

Addition of two sparse matrices.

`aoclsparse_(s/d/c/z)add` sums two sparse matrices and returns the result as a newly allocated sparse matrix for real and complex types, respectively. It performs the following operation:

$$C = \alpha * op(A) + B$$

with

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

where A is a $m \times n$ matrix and B is a $m \times n$ matrix if op = *aoclsparse_operation_none* and $n \times m$ otherwise and the result matrix C has the same dimension as B.

Note: Only matrices in CSR format are supported in this release.

Parameters

- **op** – **[in]** matrix A operation type.
- **alpha** – **[in]** scalar with same precision as A and B matrix
- **A** – **[in]** source sparse matrix A
- **B** – **[in]** source sparse matrix B
- ***C** – **[out]** pointer to the sparse output matrix C

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `A` or `B` or `C` are invalid
- **aoclsparse_status_invalid_size** – The dimensions of `A` and `B` are not compatible.
- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – Matrices are not in CSR format.

aoclsparse_status **aoclsparse_cadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const aoclsparse_float_complex alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

Addition of two sparse matrices.

`aoclsparse_(s/d/c/z)add` sums two sparse matrices and returns the result as a newly allocated sparse matrix for real and complex types, respectively. It performs the following operation:

$$C = \alpha * op(A) + B$$

with

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

where A is a $m \times n$ matrix and B is a $m \times n$ matrix if op = *aoclsparse_operation_none* and $n \times m$ otherwise and the result matrix C has the same dimension as B.

Note: Only matrices in CSR format are supported in this release.

Parameters

- **op** – **[in]** matrix A operation type.
- **alpha** – **[in]** scalar with same precision as A and B matrix
- **A** – **[in]** source sparse matrix A
- **B** – **[in]** source sparse matrix B
- ***C** – **[out]** pointer to the sparse output matrix C

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `A` or `B` or `C` are invalid
- **aoclsparse_status_invalid_size** – The dimensions of `A` and `B` are not compatible.
- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – Matrices are not in CSR format.

aoclsparse_status **aoclsparse_zadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const aoclsparse_double_complex alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

Addition of two sparse matrices.

`aoclsparse_(s/d/c/z)add` sums two sparse matrices and returns the result as a newly allocated sparse matrix for real and complex types, respectively. It performs the following operation:

$$C = \alpha * op(A) + B$$

with

$$op(A) = \begin{cases} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

where A is a $m \times n$ matrix and B is a $m \times n$ matrix if op = *aoclsparse_operation_none* and $n \times m$ otherwise and the result matrix C has the same dimension as B.

Note: Only matrices in CSR format are supported in this release.

Parameters

- **op** – **[in]** matrix A operation type.

- `alpha` – **[in]** scalar with same precision as A and B matrix
- `A` – **[in]** source sparse matrix A
- `B` – **[in]** source sparse matrix B
- `*C` – **[out]** pointer to the sparse output matrix C

Return values

- `aoclsparse_status_success` – The operation completed successfully.
- `aoclsparse_status_invalid_pointer` – `A` or `B` or `C` are invalid
- `aoclsparse_status_invalid_size` – The dimensions of `A` and `B` are not compatible.
- `aoclsparse_status_memory_error` – Memory allocation failure.
- `aoclsparse_status_not_implemented` – Matrices are not in CSR format.

4.4 Miscellaneous

aoclsparse_status **aoclsparse_dilu_smoother**(*aoclsparse_operation* op, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, double **precond_csr_val, const double *approx_inv_diag, double *x, const double *b)

Sparse Iterative solver algorithms for single and double precision datatypes.

`aoclsparse_ilu_smoother` performs Incomplete LU factorization on the sparse matrix `A`, defined in CSR storage format and also does an iterative LU solve to find an approximate `x`

For a usage example, see the ILU example in `tests/include` folder.

Parameters

- `op` – **[in]** matrix `A` operation type. Transpose not yet supported.
- `A` – **[in]** sparse matrix handle. Currently ILU functionality is supported only for CSR matrix format.
- `descr` – **[in]** descriptor of the sparse matrix handle `A`. Currently, only *aoclsparse_matrix_type_symmetric* is supported. Both, base-zero and base-one input arrays of CSR matrix are supported
- `precond_csr_val` – **[out]** output pointer that contains L and U factors after ILU operation. The original value buffer of matrix `A` is not overwritten with the factors.
- `approx_inv_diag` – **[in]** It is unused as of now.
- `x` – **[out]** array of `n` element vector found using the known values of CSR matrix `A` and resultant vector product `b` in $Ax = b$. Every call to the API gives an iterative update of `x`, whcih is used to find norm during LU solve phase. Norm and Relative Error % decides the convergence of `x` with respect to `x_ref`
- `b` – **[in]** array of `m` elements which is the result of `A` and `x` in $Ax = b$. `b` is calculated using a known reference `x` vector, which is then used to find the norm for iterative `x` during LU solve phase. Norm and Relative Error percentage decides the convergence

Return values

- `aoclsparse_status_success` – the operation completed successfully.

- **aoclsparse_status_invalid_size** – input parameters contain an invalid value.
- **aoclsparse_status_invalid_pointer** – `descr`, `A` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_symmetric* or input matrix `A` is not in CSR format

aoclsparse_status **aoclsparse_silu_smoother**(*aoclsparse_operation* op, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, float **precond_csr_val, const float *approx_inv_diag, float *x, const float *b)

Sparse Iterative solver algorithms for single and double precision datatypes.

`aoclsparse_ilu_smoother` performs Incomplete LU factorization on the sparse matrix `A`, defined in CSR storage format and also does an iterative LU solve to find an approximate `x`

For a usage example, see the ILU example in `tests/include` folder.

Parameters

- **op** – **[in]** matrix `A` operation type. Transpose not yet supported.
- **A** – **[in]** sparse matrix handle. Currently ILU functionality is supported only for CSR matrix format.
- **descr** – **[in]** descriptor of the sparse matrix handle `A`. Currently, only *aoclsparse_matrix_type_symmetric* is supported. Both, base-zero and base-one input arrays of CSR matrix are supported
- **precond_csr_val** – **[out]** output pointer that contains L and U factors after ILU operation. The original value buffer of matrix `A` is not overwritten with the factors.
- **approx_inv_diag** – **[in]** It is unused as of now.
- **x** – **[out]** array of `n` element vector found using the known values of CSR matrix `A` and resultant vector product `b` in $Ax = b$. Every call to the API gives an iterative update of `x`, whcih is used to find norm during LU solve phase. Norm and Relative Error % decides the convergence of `x` with respect to `x_ref`
- **b** – **[in]** array of `m` elements which is the result of `A` and `x` in $Ax = b$. `b` is calculated using a known reference `x` vector, which is then used to find the norm for iterative `x` during LU solve phase. Norm and Relative Error percentage decides the convergence

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – input parameters contain an invalid value.
- **aoclsparse_status_invalid_pointer** – `descr`, `A` is invalid.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_symmetric* or input matrix `A` is not in CSR format

CHAPTER

FIVE

AOCL-SPARSE ITERATIVE LINEAR SYSTEM SOLVERS

5.1 Introduction of Iterative Solver Suite (itsol)

AOCL-Sparse Iterative Solver Suite (itsol) is an iterative framework for solving large-scale sparse linear systems of equations of the form

$$Ax = b,$$

where A is a sparse full-rank square matrix of size n by n, b is a dense n-vector, and x is the vector of unknowns also of size n. The framework solves the previous problem using either the Conjugate Gradient method or GMRES. It supports a variety of preconditioners (*accelerators*) such as Symmetric Gauss-Seidel or Incomplete LU factorization, ILU(0).

Iterative solvers at each step (iteration) find a better approximation to the solution of the linear system of equations in the sense that it reduces an error metric. In contrast, direct solvers only provide a solution once the full algorithm as been executed. A great advantage of iterative solvers is that they can be interrupted once an approximate solution is deemed acceptable.

5.1.1 Forward and Reverse Communication Interfaces

The suite presents two separate interfaces to all the iterative solvers, a direct one, `aoclsparse_itsol_d_rci_solve()` (`aoclsparse_itsol_s_rci_solve()`), and a reverse communication (RCI) one `aoclsparse_itsol_d_rci_solve()` (`aoclsparse_itsol_s_rci_solve()`). While the underlying algorithms are exactly the same, the difference lies in how data is communicated to the solvers.

The direct communication interface expects to have explicit access to the coefficient matrix A. On the other hand, the reverse communication interface makes no assumption on the matrix storage. Thus when the solver requires some matrix operation such as a matrix-vector product, it returns control to the user and asks the user perform the operation and provide the results by calling again the RCI solver.

5.1.2 Recommended Workflow

For solving a linear system of equations, the following workflow is recommended:

- Call `aoclsparse_itsol_s_init()` or `aoclsparse_itsol_d_init()` to initialize aoclsparse_itsol_handle.
- Choose the solver and adjust its behaviour by setting optional parameters with `aoclsparse_itsol_option_set()`, see there all options available.
- If the reverse communication interface is desired, define the system's input with `aoclsparse_itsol_d_rci_input()`.

- Solve the system with either using direct interface *aoclsparse_itsol_s_solve()* (or *aoclsparse_itsol_d_solve()*) or reverse communication interface *aoclsparse_itsol_s_rci_solve()* (or *aoclsparse_itsol_d_rci_solve()*)
- Free the memory with *aoclsparse_itsol_destroy()*.

5.1.3 Information Array

The array `rinfo[100]` is used by the solvers (e.g. *aoclsparse_itsol_d_solve()* or *aoclsparse_itsol_s_rci_solve()*) to report back useful convergence metrics and other solver statistics. The user callback `monit` is also equipped with this array and can be used to view or monitor the state of the solver. The solver will populate the following entries with the most recent iteration data

Index	Description
0	Absolute residual norm, $r_{\text{abs}} = \|Ax - b\|_2$.
1	Norm of the right-hand side vector b, $\|b\|_2$.
2-29	Reserved for future use.
30	Iteration counter.
31-99	Reserved for future use.

5.1.4 Examples

Each iterative solver in the itsol suite is provided with an illustrative example on its usage. The source file for the examples can be found under the `tests/examples/` folder.

Solver	Precision	Filename	Description
itsol forward communication interface	double	`sample_itsol_d_cg.cpp`	Solves a linear system of equations using the Conjugate Gradient method.
	single	`sample_itsol_s_cg.cpp`	
itsol reverse communication interface	double	`sample_itsol_d_cg_rci.cpp`	Solves a linear system of equations using the Conjugate Gradient method.
	single	`sample_itsol_s_cg_rci.cpp`	

5.1.5 References

- Collaborative. Acceleration methods. *Encyclopedia of Mathematics*, 2023 (retrieved in). https://encyclopediaofmath.org/index.php?title=Acceleration_methods&oldid=52131.
- Collaborative. Conjugate gradients, method of. *Encyclopedia of Mathematics*, 2023 (retrieved in). https://encyclopediaofmath.org/index.php?title=Conjugate_gradients,_method_of&oldid=46470.
- Yousef Saad. *Iterative Methods for Sparse Linear Systems*. 2nd edition, 2003.

5.2 API documentation

typedef enum *aoclsparse_itsol_rci_job_* **aoclsparse_itsol_rci_job**

Values of `ircomm` used by the iterative solver reverse communication interface (RCI) *aoclsparse_itsol_d_rci_solve* and *aoclsparse_itsol_s_rci_solve* to communicate back to the user which operation is required.

enum **aoclsparse_itsol_rci_job_**

Values of `ircomm` used by the iterative solver reverse communication interface (RCI) *aoclsparse_itsol_d_rci_solve* and *aoclsparse_itsol_s_rci_solve* to communicate back to the user which operation is required.

Values:

enumerator **aoclsparse_rci_interrupt**

if set by the user, signals the solver to terminate. This is never set by the solver. Terminate.

enumerator **aoclsparse_rci_stop**

found a solution within specified tolerance (see options "cg rel tolerance", "cg abs tolerance", "gmres rel tolerance", and "gmres abs tolerance" in *Options*). Terminate, vector `x` contains the solution.

enumerator **aoclsparse_rci_start**

initial value of the `ircomm` flag, no action required. Call solver.

enumerator **aoclsparse_rci_mv**

perform the matrix-vector product $v = Au$. Return control to solver.

enumerator **aoclsparse_rci_precond**

perform a preconditioning step on the vector u and store in v. If the preconditioner M has explicit matrix form, then applying the preconditioner would result in the operations $v = Mu$ or $v = M^{-1}u$. The latter would be performed by solving the linear system of equations $Mv = u$. Return control to solver.

enumerator **aoclsparse_rci_stopping_criterion**

perform a monitoring step and check for custom stopping criteria. If using a positive tolerance value for the convergence options (see *aoclsparse_rci_stop*), then this step can be ignored and control can be returned to solver.

void **aoclsparse_itsol_handle_prn_options**(aoclsparse_itsol_handle handle)

Print options stored in a problem handle.

This function prints to the standard output a list of available options stored in a problem handle and their current value. For available options, see Options in *aoclsparse_itsol_option_set*.

Parameters

handle – **[in]** pointer to the iterative solvers' data structure.

aoclsparse_status **aoclsparse_itsol_option_set**(aoclsparse_itsol_handle handle, const char *option, const char *value)

Option Setter.

This function sets the value to a given option inside the provided problem handle. Handle options can be printed using *aoclsparse_itsol_handle_prn_options*. Available options are listed in *Options*.

5.2.1 Options

The iterative solver framework has the following options.

Option name	Type	Default	Description	Constraints
cg iteration limit	integer	$i = 500$	Set CG iteration limit	$1 \leq i$.
gmres iteration limit	integer	$i = 150$	Set GMRES iteration limit	$1 \leq i$.
gmres restart iterations	integer	$i = 20$	Set GMRES restart iterations	$1 \leq i$.
cg rel tolerance	real	$r = 1.08735e - 06$	Set relative convergence tolerance for cg method	$0 \leq r$.
cg abs tolerance	real	$r = 0$	Set absolute convergence tolerance for cg method	$0 \leq r$.
gmres rel tolerance	real	$r = 1.08735e - 06$	Set relative convergence tolerance for gmres method	$0 \leq r$.
gmres abs tolerance	real	$r = 1e - 06$	Set absolute convergence tolerance for gmres method	$0 \leq r$.
iterative method	string	$s = cg$	Choose solver to use	$s = cg$, *gm res*, *gmres*, or *pcg*.
cg preconditioner	string	$s = none$	Choose preconditioner to use with cg method	$s = gs$, *none*, *sgs*, *symgs*, or *user*.
gmres preconditioner	string	$s = none$	Choose preconditioner to use with gmres method	$s = $ *ilu0*, *none*, or *user*.

Note: It is worth noting that only some options apply to each specific solver, e.g. name of options that begin with "cg" affect the behaviour of the CG solver.

Parameters

- `handle` – **[inout]** pointer to the iterative solvers' data structure.
- `option` – **[in]** string specifying the name of the option to set.
- `value` – **[in]** string providing the value to set the option to.

Return values

- `aoclsparse_status_success` – the operation completed successfully.
- `aoclsparse_status_invalid_value` – either the option name was not found or the provided option value is out of the valid range.
- `aoclsparse_status_invalid_pointer` – the pointer to the problem handle is invalid.
- `aoclsparse_status_internal_error` – an unexpected error occurred.

aoclsparse_status **aoclsparse_itsol_d_init**(aoclsparse_itsol_handle *handle)

Initialize a problem `handle` (`aoclsparse_itsol_handle`) for the iterative solvers suite of the library.

aoclsparse_itsol_s_init and aoclsparse_itsol_d_init initialize a data structure referred to as problem `handle`. This `handle` is used by iterative solvers (itsol) suite to setup options, define which solver to use, etc.

Note: Once the `handle` is no longer needed, it can be destroyed and the memory released by calling *aoclsparse_itsol_destroy*.

Parameters
`handle` – **[inout]** the pointer to the problem handle data structure.

Return values

- **`aoclsparse_status_success`** – the operation completed successfully.
- **`aoclsparse_status_memory_error`** – internal memory allocation error.
- **`aoclsparse_status_invalid_pointer`** – the pointer to the problem handle is invalid.
- **`aoclsparse_status_internal_error`** – an unexpected error occurred.

aoclsparse_status **aoclsparse_itsol_s_init**(aoclsparse_itsol_handle *handle)

Initialize a problem `handle` (`aoclsparse_itsol_handle`) for the iterative solvers suite of the library.

aoclsparse_itsol_s_init and aoclsparse_itsol_d_init initialize a data structure referred to as problem `handle`. This `handle` is used by iterative solvers (itsol) suite to setup options, define which solver to use, etc.

Note: Once the `handle` is no longer needed, it can be destroyed and the memory released by calling *aoclsparse_itsol_destroy*.

Parameters
`handle` – **[inout]** the pointer to the problem handle data structure.

Return values

- **`aoclsparse_status_success`** – the operation completed successfully.
- **`aoclsparse_status_memory_error`** – internal memory allocation error.
- **`aoclsparse_status_invalid_pointer`** – the pointer to the problem handle is invalid.
- **`aoclsparse_status_internal_error`** – an unexpected error occurred.

void **aoclsparse_itsol_destroy**(aoclsparse_itsol_handle *handle)

Free the memory reserved in a problem `handle` previously initialized by *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init*.

Once the problem handle is no longer needed, calling this function to deallocate the memory is advisable to avoid memory leaks.

Note: Passing a `handle` that has not been initialized by *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* may have unpredictable results.

Parameters

handle – **[inout]** pointer to a problem handle.

aoclsparse_status **aoclsparse_itsol_d_rci_input**(aoclsparse_itsol_handle handle, *aoclsparse_int* n, const double *b)

Store partial data of the linear system of equations into the problem `handle`.

This function needs to be called before the reverse communication interface iterative solver is called. It registers the linear system's dimension `n`, and stores the right-hand side vector `b`.

Note: This function does not need to be called if the forward communication interface is used.

Parameters

- **handle** – **[inout]** problem `handle`. Needs to be initialized by calling *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init*.
- **n** – **[in]** the number of columns of the (square) linear system matrix.
- **b** – **[in]** the right hand side of the linear system. Must be a vector of size `n`.

Return values

- **aoclsparse_status_success** – initialization completed successfully.
- **aoclsparse_status_invalid_pointer** – one or more of the pointers `handle`, and `b` are invalid.
- **aoclsparse_status_wrong_type** – `handle` was initialized with a different floating point precision than requested here, e.g. *aoclsparse_itsol_d_init* (double precision) was used to initialize `handle` but *aoclsparse_itsol_s_rci_input* (single precision) is being called instead of the correct double precision one, *aoclsparse_itsol_d_rci_input*.
- **aoclsparse_status_invalid_value** – `n` was set to a negative value.
- **aoclsparse_status_memory_error** – internal memory allocation error.

aoclsparse_status **aoclsparse_itsol_s_rci_input**(aoclsparse_itsol_handle handle, *aoclsparse_int* n, const float *b)

Store partial data of the linear system of equations into the problem `handle`.

This function needs to be called before the reverse communication interface iterative solver is called. It registers the linear system's dimension `n`, and stores the right-hand side vector `b`.

Note: This function does not need to be called if the forward communication interface is used.

Parameters

- **handle** – **[inout]** problem `handle`. Needs to be initialized by calling *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init*.
- **n** – **[in]** the number of columns of the (square) linear system matrix.
- **b** – **[in]** the right hand side of the linear system. Must be a vector of size `n`.

Return values

- **aoclsparse_status_success** – initialization completed successfully.

- **aoclsparse_status_invalid_pointer** – one or more of the pointers `handle`, and `b` are invalid.
- **aoclsparse_status_wrong_type** – `handle` was initialized with a different floating point precision than requested here, e.g. *aoclsparse_itsol_d_init* (double precision) was used to initialize `handle` but *aoclsparse_itsol_s_rci_input* (single precision) is being called instead of the correct double precision one, *aoclsparse_itsol_d_rci_input*.
- **aoclsparse_status_invalid_value** – `n` was set to a negative value.
- **aoclsparse_status_memory_error** – internal memory allocation error.

aoclsparse_status **aoclsparse_itsol_d_rci_solve**(aoclsparse_itsol_handle handle, *aoclsparse_itsol_rci_job* *ircomm, double **u, double **v, double *x, double rinfo[100])

Reverse Communication Interface (RCI) to the iterative solvers (itsol) suite.

This function solves the linear system of equations

$$Ax = b,$$

where the matrix of coefficients A is not required to be provided explicitly. The right hand-side is the dense vector `b` and the vector of unknowns is `x`. If A is symmetric and positive definite then set the option "iterative method" to "cg" to solve the problem using the Conjugate Gradient method, alternatively set the option to "gmres" to solve using GMRes. See the *Options* for a list of available options to modify the behaviour of each solver.

The reverse communication interface (RCI), also know as _matrix-free_ interface does not require the user to explicitly provide the matrix A. During the solve process whenever the algorithm requires a matrix operation (matrix-vector or transposed matrix-vector products), it returns control to the user with a flag `ircomm` indicating what operation is requested. Once the user performs the requested task it must call this function again to resume the solve.

The expected workflow is as follows:

a. Call *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* to initialize the problem `handle` (aoclsparse_itsol_handle)
b. Choose the solver and adjust its behaviour by setting optional parameters with *aoclsparse_itsol_option_set*, see also *Options*.
c. Define the problem size and right-hand side vector b with *aoclsparse_itsol_d_rci_input*.
d. Solve the system with either *aoclsparse_itsol_s_rci_solve* or *aoclsparse_itsol_d_rci_solve*.
e. If there is another linear system of equations to solve with the same matrix but a different right-hand side b, then repeat from step 3.
f. If solver terminated successfully then vector `x` contains the solution.
g. Free the memory with *aoclsparse_itsol_destroy*.

These reverse communication interfaces complement the _forward communication_ interfaces *aoclsparse_itsol_d_rci_solve* and *aoclsparse_itsol_s_rci_solve*.

Note: This function returns control back to the user under certain circumstances. The table in *aoclsparse_itsol_rci_job_* indicates what actions are required to be performed by the user.

Note: For an illustrative example see Examples.

Parameters

- **handle** – **[inout]** problem handle. Needs to be previously initialized by *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* and then populated using either *aoclsparse_itsol_s_rci_input* or *aoclsparse_itsol_d_rci_input*, as appropriate.
- **ircomm** – **[inout]** pointer to the reverse communication instruction flag and defined in *aoclsparse_itsol_rci_job_*.
- **u** – **[inout]** pointer to a generic vector of data. The solver will point to the data on which the operation defined by `ircomm` needs to be applied.
- **v** – **[inout]** pointer to a generic vector of data. The solver will ask that the result of the operation defined by `ircomm` be stored in `v`.
- **x** – **[inout]** dense vector of unknowns. On input, it should contain the initial guess from which to start the iterative process. If there is no good initial estimate guess then any arbitrary but finite values can be used. On output, it contains an estimate to the solution of the linear system of equations up to the requested tolerance, e.g. see "cg rel tolerance" or "cg abs tolerance" in *Options*.
- **rinfo** – **[out]** vector containing information and stats related to the iterative solve, see Information Array. This parameter can be used to monitor progress and define a custom stopping criterion when the solver returns control to user with `ircomm = ` *aoclsparse_rci_stopping_criterion*.

aoclsparse_status **aoclsparse_itsol_s_rci_solve**(aoclsparse_itsol_handle handle, *aoclsparse_itsol_rci_job* *ircomm, float **u, float **v, float *x, float rinfo[100])

Reverse Communication Interface (RCI) to the iterative solvers (itsol) suite.

This function solves the linear system of equations

$$Ax = b,$$

where the matrix of coefficients A is not required to be provided explicitly. The right hand-side is the dense vector `b` and the vector of unknowns is `x`. If A is symmetric and positive definite then set the option "iterative method" to "cg" to solve the problem using the Conjugate Gradient method, alternatively set the option to "gmres" to solve using GMRes. See the *Options* for a list of available options to modify the behaviour of each solver.

The reverse communication interface (RCI), also know as _matrix-free_ interface does not require the user to explicitly provide the matrix A. During the solve process whenever the algorithm requires a matrix operation (matrix-vector or transposed matrix-vector products), it returns control to the user with a flag `ircomm` indicating what operation is requested. Once the user performs the requested task it must call this function again to resume the solve.

The expected workflow is as follows:

a. Call *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* to initialize the problem `handle` (aoclsparse_itsol_handle)
b. Choose the solver and adjust its behaviour by setting optional parameters with *aoclsparse_itsol_option_set*, see also *Options*.
c. Define the problem size and right-hand side vector b with *aoclsparse_itsol_d_rci_input*.
d. Solve the system with either *aoclsparse_itsol_s_rci_solve* or *aoclsparse_itsol_d_rci_solve*.
e. If there is another linear system of equations to solve with the same matrix but a different right-hand side b, then repeat from step 3.
f. If solver terminated successfully then vector `x` contains the solution.

g. Free the memory with *aoclsparse_itsol_destroy*.

These reverse communication interfaces complement the _forward communication_ interfaces *aoclsparse_itsol_d_rci_solve* and *aoclsparse_itsol_s_rci_solve*.

Note: This function returns control back to the user under certain circumstances. The table in *aoclsparse_itsol_rci_job_* indicates what actions are required to be performed by the user.

Note: For an illustrative example see Examples.

Parameters

- **handle** – **[inout]** problem `handle`. Needs to be previously initialized by *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* and then populated using either *aoclsparse_itsol_s_rci_input* or *aoclsparse_itsol_d_rci_input*, as appropriate.
- **ircomm** – **[inout]** pointer to the reverse communication instruction flag and defined in *aoclsparse_itsol_rci_job_*.
- **u** – **[inout]** pointer to a generic vector of data. The solver will point to the data on which the operation defined by `ircomm` needs to be applied.
- **v** – **[inout]** pointer to a generic vector of data. The solver will ask that the result of the operation defined by `ircomm` be stored in `v`.
- **x** – **[inout]** dense vector of unknowns. On input, it should contain the initial guess from which to start the iterative process. If there is no good initial estimate guess then any arbitrary but finite values can be used. On output, it contains an estimate to the solution of the linear system of equations up to the requested tolerance, e.g. see "cg rel tolerance" or "cg abs tolerance" in *Options*.
- **rinfo** – **[out]** vector containing information and stats related to the iterative solve, see Information Array. This parameter can be used to monitor progress and define a custom stopping criterion when the solver returns control to user with `ircomm = aoclsparse_rci_stopping_criterion`.

aoclsparse_status **aoclsparse_itsol_d_solve**(aoclsparse_itsol_handle handle, *aoclsparse_int* n, *aoclsparse_matrix* mat, const *aoclsparse_mat_descr* descr, const double *b, double *x, double rinfo[100], *aoclsparse_int* precond(*aoclsparse_int* flag, *aoclsparse_int* n, const double *u, double *v, void *udata), *aoclsparse_int* monit(*aoclsparse_int* n, const double *x, const double *r, double rinfo[100], void *udata), void *udata)

Forward communication interface to the iterative solvers suite of the library.

This function solves the linear system of equations

$$Ax = b,$$

where the matrix of coefficients A is defined by `mat`. The right hand-side is the dense vector `b` and the vector of unknowns is `x`. If A is symmetric and positive definite then set the option "iterative method" to "cg" to solve the problem using the Conjugate Gradient method, alternatively set the option to "gmres" to solve using GMRes. See the *Options* for a list of available options to modify the behaviour of each solver.

The expected workflow is as follows:

a. Call *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* to initialize the problem `handle` (aoclsparse_itsol_handle).

b. Choose the solver and adjust its behaviour by setting optional parameters with *aoclsparse_itsol_option_set*, see also *Options*.

c. Solve the system by calling *aoclsparse_itsol_s_solve* or *aoclsparse_itsol_d_solve*.

d. If there is another linear system of equations to solve with the same matrix but a different right-hand side b, then repeat from step 3.

e. If solver terminated successfully then vector `x` contains the solution.

f. Free the memory with *aoclsparse_itsol_destroy*.

This interface requires to explicitly provide the matrix A and its descriptor `descr`, this kind of interface is also known as _forward communication_ which contrasts with *reverse communication* in which case the matrix A and its descriptor `descr` need not be explicitly available. For more details on the latter, see *aoclsparse_itsol_d_rci_solve* or *aoclsparse_itsol_s_rci_solve*.

Note: For an illustrative example see Examples.

Parameters

- **`handle`** – **[inout]** a valid problem handle, previously initialized by calling *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init*.
- **`n`** – **[in]** the size of the square matrix `mat`.
- **`mat`** – **[inout]** coefficient matrix A.
- **`descr`** – **[inout]** matrix descriptor for `mat`.
- **`b`** – **[in]** right-hand side dense vector b.
- **`x`** – **[inout]** dense vector of unknowns. On input, it should contain the initial guess from which to start the iterative process. If there is no good initial estimate guess then any arbitrary but finite values can be used. On output, it contains an estimate to the solution of the linear system of equations up to the requested tolerance, e.g. see "cg rel tolerance" or "cg abs tolerance" in *Options*.
- **`rinfo`** – **[out]** vector containing information and stats related to the iterative solve, see Information Array.
- **`precond`** – **[in]** (optional, can be nullptr) function pointer to a user routine that applies the preconditioning step

  $$v = Mu \text{or } v = M^{-1}u,$$

  where v is the resulting vector of applying a preconditioning step on the vector u and M refers to the user specified preconditioner in matrix form and need not be explicitly available. The void pointer udata, is a convenience pointer that can be used by the user to point to user data and is not used by the itsol framework. If the user requests to use a predefined preconditioner already available in the suite (refer to e.g. "cg preconditioner" or "gmres preconditioner" in *Options*), then this parameter need not be provided.
- **`monit`** – **[in]** (optional, can be nullptr) function pointer to a user monitoring routine. If provided, then at each iteration, the routine is called and can be used to define a custom stopping criteria or to oversee the convergence process. In general, this function need not be provided. If provided then the solver provides `n` the problem size, `x` the current iterate, `r`

the current residual vector ($r = Ax - b$), `rinfo` the current solver's stats, see Information Array, and `udata` a convenience pointer that can be used by the user to point to arbitrary user data and is not used by the itsol framework.

- **udata** – **[inout]** (optional, can be nullptr) user convenience pointer, it can be used by the user to pass a pointer to user data. It is not modified by the solver.

aoclsparse_status **aoclsparse_itsol_s_solve**(aoclsparse_itsol_handle handle, *aoclsparse_int* n, *aoclsparse_matrix* mat, const *aoclsparse_mat_descr* descr, const float *b, float *x, float rinfo[100], *aoclsparse_int* precond(*aoclsparse_int* flag, *aoclsparse_int* n, const float *u, float *v, void *udata), *aoclsparse_int* monit(*aoclsparse_int* n, const float *x, const float *r, float rinfo[100], void *udata), void *udata)

Forward communication interface to the iterative solvers suite of the library.

This function solves the linear system of equations

$$Ax = b,$$

where the matrix of coefficients A is defined by `mat`. The right hand-side is the dense vector `b` and the vector of unknowns is `x`. If A is symmetric and positive definite then set the option "iterative method" to "cg" to solve the problem using the Conjugate Gradient method, alternatively set the option to "gmres" to solve using GMRes. See the *Options* for a list of available options to modify the behaviour of each solver.

The expected workflow is as follows:

a. Call *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init* to initialize the problem `handle` (aoclsparse_itsol_handle).
b. Choose the solver and adjust its behaviour by setting optional parameters with *aoclsparse_itsol_option_set*, see also *Options*.
c. Solve the system by calling *aoclsparse_itsol_s_solve* or *aoclsparse_itsol_d_solve*.
d. If there is another linear system of equations to solve with the same matrix but a different right-hand side b, then repeat from step 3.
e. If solver terminated successfully then vector `x` contains the solution.
f. Free the memory with *aoclsparse_itsol_destroy*.

This interface requires to explicitly provide the matrix A and its descriptor `descr`, this kind of interface is also known as _forward communication_ which contrasts with *reverse communication* in which case the matrix A and its descriptor `descr` need not be explicitly available. For more details on the latter, see *aoclsparse_itsol_d_rci_solve* or *aoclsparse_itsol_s_rci_solve*.

Note: For an illustrative example see Examples.

Parameters

- **handle** – **[inout]** a valid problem handle, previously initialized by calling *aoclsparse_itsol_s_init* or *aoclsparse_itsol_d_init*.
- **n** – **[in]** the size of the square matrix `mat`.
- **mat** – **[inout]** coefficient matrix A.
- **descr** – **[inout]** matrix descriptor for `mat`.
- **b** – **[in]** right-hand side dense vector b.

- `x` – **[inout]** dense vector of unknowns. On input, it should contain the initial guess from which to start the iterative process. If there is no good initial estimate guess then any arbitrary but finite values can be used. On output, it contains an estimate to the solution of the linear system of equations up to the requested tolerance, e.g. see "cg rel tolerance" or "cg abs tolerance" in *Options*.
- `rinfo` – **[out]** vector containing information and stats related to the iterative solve, see Information Array.
- `precond` – **[in]** (optional, can be nullptr) function pointer to a user routine that applies the preconditioning step

  $$v = Mu \text{or } v = M^{-1}u,$$

  where v is the resulting vector of applying a preconditioning step on the vector u and M refers to the user specified preconditioner in matrix form and need not be explicitly available. The void pointer udata, is a convenience pointer that can be used by the user to point to user data and is not used by the itsol framework. If the user requests to use a predefined preconditioner already available in the suite (refer to e.g. "cg preconditioner" or "gmres preconditioner" in *Options*), then this parameter need not be provided.
- `monit` – **[in]** (optional, can be nullptr) function pointer to a user monitoring routine. If provided, then at each iteration, the routine is called and can be used to define a custom stopping criteria or to oversee the convergence process. In general, this function need not be provided. If provided then the solver provides `n` the problem size, `x` the current iterate, `r` the current residual vector ($r = Ax - b$), `rinfo` the current solver's stats, see Information Array, and `udata` a convenience pointer that can be used by the user to point to arbitrary user data and is not used by the itsol framework.
- `udata` – **[inout]** (optional, can be nullptr) user convenience pointer, it can be used by the user to pass a pointer to user data. It is not modified by the solver.

CHAPTER

SIX

AOCL-SPARSE TYPES

6.1 Numerical Types

typedef int32_t **aoclsparse_int**

Specifies whether int32 or int64 is used.

struct alignas aoclsparse_float_complex

struct alignas aoclsparse_double_complex

6.2 Other Types

typedef struct _aoclsparse_matrix ***aoclsparse_matrix**

AOCL sparse matrix.

The aoclsparse matrix structure holds the all matrix storage format supported. It must be initialized using `aoclsparse_create_`(s/d/c/z)(csr/csc/coo) and the returned matrix must be passed to all subsequent library calls that involve the matrix. It should be destroyed at the end using *aoclsparse_destroy()*.

typedef struct _aoclsparse_mat_descr ***aoclsparse_mat_descr**

Descriptor of the matrix.

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

6.3 Enums

typedef enum *aoclsparse_operation_* **aoclsparse_operation**

Specify whether the matrix is to be transposed or not.

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

enum **aoclsparse_operation_**

Specify whether the matrix is to be transposed or not.

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

Values:

enumerator **aoclsparse_operation_none**

Operate with matrix.

enumerator **aoclsparse_operation_transpose**

Operate with transpose.

enumerator **aoclsparse_operation_conjugate_transpose**

Operate with conj. transpose.

typedef enum *aoclsparse_index_base_* **aoclsparse_index_base**

Specify the matrix index base.

The *aoclsparse_index_base* indicates the index base of the indices. For a given *aoclsparse_mat_descr*, the *aoclsparse_index_base* can be set using *aoclsparse_set_mat_index_base()*. The current *aoclsparse_index_base* of a matrix can be obtained by *aoclsparse_get_mat_index_base()*.

enum **aoclsparse_index_base_**

Specify the matrix index base.

The *aoclsparse_index_base* indicates the index base of the indices. For a given *aoclsparse_mat_descr*, the *aoclsparse_index_base* can be set using *aoclsparse_set_mat_index_base()*. The current *aoclsparse_index_base* of a matrix can be obtained by *aoclsparse_get_mat_index_base()*.

Values:

enumerator **aoclsparse_index_base_zero**

zero based indexing.

enumerator **aoclsparse_index_base_one**

one based indexing.

typedef enum *aoclsparse_matrix_type_* **aoclsparse_matrix_type**

Specify the matrix type.

The *aoclsparse_matrix_type* indices the type of a matrix. For a given *aoclsparse_mat_descr*, the *aoclsparse_matrix_type* can be set using *aoclsparse_set_mat_type()*. The current *aoclsparse_matrix_type* of a matrix can be obtained by *aoclsparse_get_mat_type()*.

enum **aoclsparse_matrix_type_**

Specify the matrix type.

The *aoclsparse_matrix_type* indices the type of a matrix. For a given *aoclsparse_mat_descr*, the *aoclsparse_matrix_type* can be set using *aoclsparse_set_mat_type()*. The current *aoclsparse_matrix_type* of a matrix can be obtained by *aoclsparse_get_mat_type()*.

Values:

enumerator **aoclsparse_matrix_type_general**

general matrix type.

enumerator **aoclsparse_matrix_type_symmetric**

symmetric matrix type.

enumerator **aoclsparse_matrix_type_hermitian**

hermitian matrix type.

enumerator **aoclsparse_matrix_type_triangular**

triangular matrix type.

typedef enum *aoclsparse_matrix_data_type_* **aoclsparse_matrix_data_type**

Specify the matrix data type.

The *aoclsparse_matrix_data_type* indices the data-type of a matrix.

enum **aoclsparse_matrix_data_type_**

Specify the matrix data type.

The *aoclsparse_matrix_data_type* indices the data-type of a matrix.

Values:

enumerator **aoclsparse_dmat**

double precision data.

enumerator **aoclsparse_smat**

single precision data.

enumerator **aoclsparse_cmat**

single precision complex data.

enumerator **aoclsparse_zmat**

double precision complex data.

typedef enum *aoclsparse_ilu_type_* **aoclsparse_ilu_type**

Specify the type of ILU factorization.

The *aoclsparse_ilu_type* indicates the type of ILU factorization like ILU0, ILU(p) etc.

enum **aoclsparse_ilu_type_**

Specify the type of ILU factorization.

The *aoclsparse_ilu_type* indicates the type of ILU factorization like ILU0, ILU(p) etc.

Values:

enumerator **aoclsparse_ilu0**

ILU0.

enumerator **aoclsparse_ilup**

ILU(p).

typedef enum *aoclsparse_matrix_format_type_* **aoclsparse_matrix_format_type**

Specify the matrix storage format type.

The *aoclsparse_matrix_format_type* indices the storage format of a sparse matrix.

enum **aoclsparse_matrix_format_type_**

Specify the matrix storage format type.

The *aoclsparse_matrix_format_type* indices the storage format of a sparse matrix.

Values:

enumerator **aoclsparse_csr_mat**

CSR format.

enumerator **aoclsparse_ell_mat**

ELLPACK format.

enumerator **aoclsparse_ellt_mat**

ELLPACK format stored as transpose format.

enumerator **aoclsparse_ellt_csr_hyb_mat**

ELLPACK transpose + CSR hybrid format.

enumerator **aoclsparse_ell_csr_hyb_mat**

ELLPACK + CSR hybrid format.

enumerator **aoclsparse_dia_mat**

diag format.

enumerator **aoclsparse_csr_mat_br4**

Modified CSR format for AVX2 double.

enumerator **aoclsparse_csc_mat**

CSC format.

enumerator **aoclsparse_coo_mat**

COO format.

typedef enum *aoclsparse_diag_type_* **aoclsparse_diag_type**

Indicates if the diagonal entries are unity.

The *aoclsparse_diag_type* indicates whether the diagonal entries of a matrix are unity or not. If *aoclsparse_diag_type_unit* is specified, all present diagonal values will be ignored. For a given *aoclsparse_mat_descr*, the *aoclsparse_diag_type* can be set using *aoclsparse_set_mat_diag_type()*. The current *aoclsparse_diag_type* of a matrix can be obtained by *aoclsparse_get_mat_diag_type()*.

enum **aoclsparse_diag_type_**

Indicates if the diagonal entries are unity.

The *aoclsparse_diag_type* indicates whether the diagonal entries of a matrix are unity or not. If *aoclsparse_diag_type_unit* is specified, all present diagonal values will be ignored. For a given *aoclsparse_mat_descr*, the *aoclsparse_diag_type* can be set using *aoclsparse_set_mat_diag_type()*. The current *aoclsparse_diag_type* of a matrix can be obtained by *aoclsparse_get_mat_diag_type()*.

Values:

enumerator **aoclsparse_diag_type_non_unit**

diagonal entries are non-unity.

enumerator **aoclsparse_diag_type_unit**

diagonal entries are unity

enumerator **aoclsparse_diag_type_zero**

ignore diagonal entries: for strict L/U matrices

typedef enum *aoclsparse_fill_mode_* **aoclsparse_fill_mode**

Specify the matrix fill mode.

The *aoclsparse_fill_mode* indicates whether the lower or the upper part is stored in a sparse triangular matrix. For a given *aoclsparse_mat_descr*, the *aoclsparse_fill_mode* can be set using *aoclsparse_set_mat_fill_mode()*. The current *aoclsparse_fill_mode* of a matrix can be obtained by *aoclsparse_get_mat_fill_mode()*.

enum **aoclsparse_fill_mode_**

Specify the matrix fill mode.

The *aoclsparse_fill_mode* indicates whether the lower or the upper part is stored in a sparse triangular matrix. For a given *aoclsparse_mat_descr*, the *aoclsparse_fill_mode* can be set using *aoclsparse_set_mat_fill_mode()*. The current *aoclsparse_fill_mode* of a matrix can be obtained by *aoclsparse_get_mat_fill_mode()*.

Values:

enumerator **aoclsparse_fill_mode_lower**

lower triangular part is stored.

enumerator **aoclsparse_fill_mode_upper**

upper triangular part is stored.

typedef enum *aoclsparse_order_* **aoclsparse_order**

List of dense matrix ordering.

This is a list of supported *aoclsparse_order* types that are used to describe the memory layout of a dense matrix

enum **aoclsparse_order_**

List of dense matrix ordering.

This is a list of supported *aoclsparse_order* types that are used to describe the memory layout of a dense matrix

Values:

enumerator **aoclsparse_order_row**

Row major.

enumerator **aoclsparse_order_column**

Column major.

typedef enum *aoclsparse_status_* **aoclsparse_status**

List of aoclsparse status codes definition.

List of *aoclsparse_status* values returned by the functions in the library.

enum **aoclsparse_status_**

List of aoclsparse status codes definition.

List of *aoclsparse_status* values returned by the functions in the library.

Values:

enumerator **aoclsparse_status_success**

success.

enumerator **aoclsparse_status_not_implemented**

functionality is not implemented.

enumerator **aoclsparse_status_invalid_pointer**

invalid pointer parameter.

enumerator **aoclsparse_status_invalid_size**

invalid size parameter.

enumerator **aoclsparse_status_internal_error**

internal library failure.

enumerator **aoclsparse_status_invalid_value**

invalid parameter value.

enumerator **aoclsparse_status_invalid_index_value**

invalid index value.

enumerator **aoclsparse_status_maxit**

function stopped after reaching number of iteration limit.

enumerator **aoclsparse_status_user_stop**

user requested termination.

enumerator **aoclsparse_status_wrong_type**

function called on the wrong type (double/float).

enumerator **aoclsparse_status_memory_error**

memory allocation failure.

enumerator **aoclsparse_status_numerical_error**

numerical error, e.g., matrix is not positive definite, divide-by-zero error

enumerator **aoclsparse_status_invalid_operation**

cannot proceed with the request at this point.

typedef enum *aoclsparse_request_* **aoclsparse_request**

List of request stages for sparse matrix * sparse matrix.

This is a list of the *aoclsparse_request* types that are used by the aoclsparse_csr2m funtion.

enum **aoclsparse_request_**

List of request stages for sparse matrix * sparse matrix.

This is a list of the *aoclsparse_request* types that are used by the aoclsparse_csr2m funtion.

Values:

enumerator **aoclsparse_stage_nnz_count**

Only rowIndex array of the CSR matrix is computed internally.

enumerator **aoclsparse_stage_finalize**

Finalize computation. Has to be called only after csr2m call with aoclsparse_stage_nnz_count parameter.

enumerator **aoclsparse_stage_full_computation**

Perform the entire computation in a single step.

CHAPTER

SEVEN

SEARCH THE DOCUMENTATION

- genindex
- search