AOCL-Sparse

Release 5.0.1

Advanced Micro Devices, Inc

Oct 18, 2024

FUNCTIONALITY API

1 Introduction **1**

2 Naming convention **3**
2.1 Analysis Functions . . . 3
2.2 Auxiliary Functions . . . 6
2.3 Conversion Functions . . . 20
2.4 Sparse BLAS level 1, 2, and 3 functions . . . 28
2.5 Iterative Linear System Solvers . . . 67
2.6 AOCL-Sparse Types . . . 80
2.7 Storage Schemes . . . 87
2.8 Search the documentation . . . 87

Index **89**

CHAPTER ONE

INTRODUCTION

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 and optimize framework** to accelerate supported functions by a prior matrix analysis based on users' hints of expected operations.

CHAPTER

TWO

NAMING CONVENTION

API's in the library are formed by three sections: `aoclsparse` prefix, `P` data type precision, followed by an abbreviated form of the functionality. Data type precion `P` is a single letter indicating: `s` single, `d` double, `c` complex single, and `z` complex double floating point. Some illustrative examples follow.

Table 1: API naming convention examples

API	Precision P	Functionality
`aoclsparse_strsv()`	s	`TRSV` single precision linear system of equations TRiangular SolVer,
`aoclsparse_daxpyi()`	d	`AXPY` perform a variant of the operation $a\,x + y$ in double precision,
`aoclsparse_cmv()`	c	`SPMV` sparse matrix-vector product using complex single precision,
`aoclsparse_ztrsm()`	z	`TRSM` complex double precision linear system of equations TRiangular Solver with Multiple right-hand sides.

Throughout this document and where not ambiguous, if an API supports two or more data types described above, then it will be indicated by `?` (question mark) in place of the data type single-letter abbreviation. As an example, `aoclsparse_?trsv()` references all supported data types for the TRSV solver, that is, `aoclsparse_strsv()`, `aoclsparse_dtrsv()`, `aoclsparse_ctrsv()`, and `aoclsparse_ztrsv()`; while `aoclsparse_?dotci()` references only `aoclsparse_cdotci()`, and `aoclsparse_zdotci()`.

2.1 Analysis Functions

2.1.1 aoclsparse_optimize()

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

Performs analysis and possible data allocations and matrix restructuring operations related to accelerate sparse operations involving matrices.

In *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.

2.1.2 aoclsparse_set_*_hint()

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

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)

Record hints of the expected number and types of calls to optimize the input matrix for.

Any of the `aoclsparse_set_*_hint` functions may be used to indicate that a given number of calls to the same Sparse BLAS API will be performed. When *aoclsparse_optimize()* is invoked, the input matrix might be tuned to accelerate the hinted calls.

Parameters

- **mat** – **[in]** Input sparse matrix to be tuned.
- **trans** – **[in]** Matrix operation to perform during the calls.
- **descr** – **[in]** Descriptor of the sparse matrix used during the calls.
- **expected_no_of_calls** – **[in]** A rough estimate of the number of the calls.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_value** – `mat`, `trans`, `descr` or `expected_no_of_calls` is invalid. Expecting `expected_no_of_calls` > 0.
- **aoclsparse_status_invalid_pointer** – `mat` or `descr` is invalid.
- **aoclsparse_status_memory_error** – internal memory allocation failure.

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

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

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

Provides hints to optimize preconditioning matrices.

Set hints for analysis and optimization of preconditioning-related factorizations and/or accelerate the application of such preconditioner, this can also include hints for "fused" operations that accelerate two operations in a single call.

Parameters

- **mat** – **[in]** A sparse matrix
- **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]** Expected number of call to an API that uses matrix `mat`.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – indicates that `mat` is invalid.
- **aoclsparse_status_invalid_pointer** – at least one of the input pointers is invalid.
- **aoclsparse_status_internal_error** – Indicates that an internal error occurred.

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

Record a hint of the expected number of calls to *aoclsparse_strsm()* and variants to optimize the input matrix for.

aoclsparse_set_sm_hint() may be used to indicate that a given number of calls to the triangular solver *aoclsparse_strsm()* or other variant will be performed. When *aoclsparse_optimize()* is invoked, the input matrix might be tuned to accelerate the hinted calls. The hints include not only the estimated number of calls to the API solver, but also other (matrix) parameters. The hinted matrix should not be modified after the call to optimize and before the call to the solver.

Parameters

- **mat** – **[in]** Input sparse matrix to be tuned.
- **trans** – **[in]** Matrix operation to perform during the calls.
- **descr** – **[in]** Descriptor of the sparse matrix used during the calls.
- **order** – **[in]** Layout of the right-hand-side input matrix used during the calls, valid options are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **expected_no_of_calls** – **[in]** A rough estimate of the number of the calls.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_value** – `expected_no_of_calls`, `order`, `mat`, `trans` or `descr` is invalid.
- **aoclsparse_status_invalid_pointer** – `mat` or `descr` is invalid.
- **aoclsparse_status_memory_error** – internal memory allocation failure.

aoclsparse_status **aoclsparse_set_sorv_hint**(*aoclsparse_matrix* mat, const *aoclsparse_mat_descr* descr, const *aoclsparse_sor_type* type, const *aoclsparse_int* expected_no_of_calls)

Record a hint of the expected number of aoclsparse_sorv() calls to optimize the input matrix for.

`aoclsparse_set_sorv_hint` may be used to indicate that a given number of calls to the SOR preconditioner aoclsparse_sorv() will be performed. When *aoclsparse_optimize()* is invoked, the input matrix might be tuned to accelerate the hinted calls. The hints include not only the estimated number of the API calls but also their other parameters which should match the actual calls.

Parameters

- **mat** – **[in]** Input sparse matrix to be tuned.
- **descr** – **[in]** Descriptor of the sparse matrix used during the calls.
- **type** – **[in]** The operation to perform by the SOR preconditioner.
- **expected_no_of_calls** – **[in]** A rough estimate of the number of the calls.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_value** – `expected_no_of_calls`, `descr`, `type` or `mat` type is invalid.
- **aoclsparse_status_invalid_pointer** – `mat` or `descr` is NULL.
- **aoclsparse_status_memory_error** – internal memory allocation failure.

aoclsparse_status **aoclsparse_set_memory_hint**(*aoclsparse_matrix* mat, const *aoclsparse_memory_usage* policy)

Record user's attitude to the memory consumption while optimizing the input matrix for the hinted operations.

`aoclsparse_set_memory_hint` may be used to indicate how much memory can be allocated during the optimization process of the input matrix for the previously hinted operations. In particular, *aoclsparse_memory_usage_minimal* suggests that the new memory should be only of order of vectors, whereas *aoclsparse_memory_usage_unrestricted* allows even new copies of the whole matrix. The unrestricted memory policy is the default. Any change to the memory policy applies only to any new optimizations for the new hints which have not been processed by *aoclsparse_optimize()* yet. The optimizations from any previous calls are unaffected. Note that the memory policy is only an indication rather than rule.

Parameters

- **mat** – **[in]** Input sparse matrix to be tuned.
- **policy** – **[in]** Memory usage policy for future optimizations.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_value** – `policy` type is invalid.
- **aoclsparse_status_invalid_pointer** – pointer `mat` is invalid.

2.2 Auxiliary Functions

2.2.1 aoclsparse_create_mat_descr()

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.

2.2.2 aoclsparse_destroy_mat_descr()

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.

2.2.3 aoclsparse_copy_mat_descr()

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.

2.2.4 aoclsparse_create_?csr()

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)

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)

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) *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_?csr` creates *aoclsparse_matrix* and initializes it with input parameters passed. The input arrays are left unchanged by the library except for the call to *aoclsparse_order_mat()*, which performs ordering of column indices of the matrix, or *aoclsparse_sset_value()*, *aoclsparse_supdate_values()* and variants, which modify the values of a nonzero element. To avoid any changes to the input data, *aoclsparse_copy()* can be used. To convert any other format to CSR, *aoclsparse_convert_csr()* 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.

2.2.5 aoclsparse_create_?tcsr()

aoclsparse_status **aoclsparse_create_stcsr**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr_L, *aoclsparse_int* *row_ptr_U, *aoclsparse_int* *col_idx_L, *aoclsparse_int* *col_idx_U, float *val_L, float *val_U)

aoclsparse_status **aoclsparse_create_dtcsr**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr_L, *aoclsparse_int* *row_ptr_U, *aoclsparse_int* *col_idx_L, *aoclsparse_int* *col_idx_U, double *val_L, double *val_U) *aoclsparse_status* **aoclsparse_create_ctcsr**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr_L, *aoclsparse_int* *row_ptr_U, *aoclsparse_int* *col_idx_L, *aoclsparse_int* *col_idx_U, *aoclsparse_float_complex* *val_L, *aoclsparse_float_complex* *val_U)

aoclsparse_status **aoclsparse_create_ztcsr**(*aoclsparse_matrix* *mat, const *aoclsparse_index_base* base, const *aoclsparse_int* M, const *aoclsparse_int* N, const *aoclsparse_int* nnz, *aoclsparse_int* *row_ptr_L, *aoclsparse_int* *row_ptr_U, *aoclsparse_int* *col_idx_L, *aoclsparse_int* *col_idx_U, *aoclsparse_double_complex* *val_L, *aoclsparse_double_complex* *val_U)

Creates a new *aoclsparse_matrix* based on TCSR (Triangular Compressed Sparse Row) format.

`aoclsparse_create_?tcsr` creates *aoclsparse_matrix* and initializes it with input parameters passed. Array data must not be modified by the user while matrix is being used as the pointers are copied, not the data. The input arrays are not modified by the library and the matrix should be destroyed at the end using *aoclsparse_destroy()*.

TCSR matrix structure holds lower triangular (L) and upper triangular (U) part of the matrix separately with diagonal (D) elements stored in both the parts. Both triangles (L+D and D+U) are stored like CSR and assumes partial sorting (L+D and D+U order is followed, but the indices within L or U group may not be sorted)

- One array with L elements potentially unsorted, followed by D elements in the L+D part for each row of the matrix.
- Another array with D elements, followed by U elements potentially unsorted in the D+U part for each row of the matrix.
- Currently TCSR storage format supports only square matrices with full(non-zero) diagonals.

Parameters

- **mat** – **[out]** The pointer to the TCSR sparse matrix.
- **base** – **[in]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **M** – **[in]** Total number of rows in the `mat`.
- **N** – **[in]** Total number of columns in the `mat`.
- **nnz** – **[in]** Number of non-zero entries in the `mat`.
- **row_ptr_L** – **[in]** Array of lower triangular elements that point to the start of every row of the `mat` in `col_idx_L` and `val_L`.
- **row_ptr_U** – **[in]** Array of upper triangular elements that point to the start of every row of the `mat` in `col_idx_U` and `val_U`.
- **col_idx_L** – **[in]** Array of lower triangular elements containing column indices of the `mat`.
- **col_idx_U** – **[in]** Array of upper triangular elements containing column indices of the `mat`.
- **val_L** – **[in]** Array of lower triangular elements of the `mat`.
- **val_U** – **[in]** Array of upper triangular elements of the `mat`.

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** – M, N, nnz is invalid.

- **aoclsparse_status_invalid_index_value** – Index given for `mat` is out of matrix bounds depending on base given.
- **aoclsparse_status_invalid_value** – The cooridante `row_ptr` or `col_idx` is out of matrix bound or `mat` has duplicate diagonals or `mat` does not have full diagonals.
- **aoclsparse_status_unsorted_input** – The `mat` is unsorted. It supports only fully sorted and partially sorted matrix as input. The lower triangular part must not contain U elements, the upper triangular part must not contain L elements, and the position of the diagonal element must not be altered.
- **aoclsparse_status_memory_error** – Memory allocation for matrix failed.

2.2.6 aoclsparse_create_?coo()

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)

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)

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)

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_?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. The input arrays are left unchanged by the library except for the call to *aoclsparse_sset_value()*, *aoclsparse_supdate_values()* and variants, which modify the value of a nonzero element. 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.

2.2.7 aoclsparse_create_?csc()

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)

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)

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)

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_?csc` creates *aoclsparse_matrix* and initializes it with input parameters passed. The input arrays are left unchanged by the library except for the call to *aoclsparse_order_mat()*, which performs ordering of row indices of the matrix, or *aoclsparse_sset_value()*, *aoclsparse_supdate_values()* and variants, which modify the value of a nonzero element. 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.

2.2.8 aoclsparse_destroy()

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

Destroy a sparse matrix structure.

aoclsparse_destroy destroys a structure that holds matrix mat.

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.

2.2.9 aoclsparse_copy()

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 invalid. 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.

2.2.10 aoclsparse_order_mat()

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_csr()* 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 invalid.
- **aoclsparse_status_memory_error** – internal memory allocation failed.
- **aoclsparse_status_not_implemented** – matrix is not in CSR format.

2.2.11 aoclsparse_?set_value()

aoclsparse_status **aoclsparse_sset_value**(*aoclsparse_matrix* A, *aoclsparse_int* row_idx, *aoclsparse_int* col_idx, float val)

aoclsparse_status **aoclsparse_dset_value**(*aoclsparse_matrix* A, *aoclsparse_int* row_idx, *aoclsparse_int* col_idx, double val)

aoclsparse_status **aoclsparse_cset_value**(*aoclsparse_matrix* A, *aoclsparse_int* row_idx, *aoclsparse_int* col_idx, *aoclsparse_float_complex* val)

aoclsparse_status **aoclsparse_zset_value**(*aoclsparse_matrix* A, *aoclsparse_int* row_idx, *aoclsparse_int* col_idx, *aoclsparse_double_complex* val)

Set a new value to an existing nonzero in the matrix.

`aoclsparse_?set_value` modifies the value of an existing nonzero element specified by its coordinates. The row and column coordinates need to match the base (0 or 1-base) of the matrix. The change directly affects user's arrays if the matrix was created using *aoclsparse_create_scsr()*, *aoclsparse_create_scsc()*, *aoclsparse_create_scoo()* or other variants.

Note: The successful modification invalidates existing optimized data so it is desirable to call *aoclsparse_optimize()* once all modifications are performed.

Parameters

- **A** – **[inout]** The sparse matrix to be modified.
- **row_idx** – **[in]** The row index of the element to be updated.
- **col_idx** – **[in]** The column index of the element to be updated.
- **val** – **[in]** The value to be updated.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – The matrix handler `A` is invalid
- **aoclsparse_status_invalid_value** – The cooridante `row_idx` or `col_idx` is out of matrix bound
- **aoclsparse_status_wrong_type** – Matrix has different data type then the one used in API
- **aoclsparse_status_not_implemented** – Matrix format is not supported for this operation
- **aoclsparse_status_invalid_index_value** – The specified element does not exist in the matrix

2.2.12 aoclsparse_?update_values()

aoclsparse_status **aoclsparse_supdate_values**(*aoclsparse_matrix* A, *aoclsparse_int* len, float *val)

aoclsparse_status **aoclsparse_dupdate_values**(*aoclsparse_matrix* A, *aoclsparse_int* len, double *val)

aoclsparse_status **aoclsparse_cupdate_values**(*aoclsparse_matrix* A, *aoclsparse_int* len, *aoclsparse_float_complex* *val)

aoclsparse_status **aoclsparse_zupdate_values**(*aoclsparse_matrix* A, *aoclsparse_int* len, *aoclsparse_double_complex* *val)

Set new values to all existing nonzero element in the matrix.

`aoclsparse_?update_values` overwrites all existing nonzeros in the matrix with the new values provided in `val` array. The order of elements must match the order in the matrix. That would be either the order at the creation of the matrix or the sorted order if *aoclsparse_order_mat()* has been called. The change directly affects user's arrays if the matrix was created using *aoclsparse_create_scsr()*, *aoclsparse_create_scsc()*, *aoclsparse_create_scoo()* or other variants.

Note: The successful update invalidates existing optimized data so it is desirable to call *aoclsparse_optimize()* once all modifications are performed.

Parameters

- **A** – **[inout]** The sparse matrix to be modified.
- **len** – **[in]** Length of the `val` array and the number of nonzeros in the matrix.
- **val** – **[in]** Array with the values to be copied.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – The matrix `A` is invalid or `val` in NULL
- **aoclsparse_status_invalid_size** – `len` is not equal to nnz of matrix
- **aoclsparse_status_wrong_type** – Matrix has different data type then the one used in API
- **aoclsparse_status_not_implemented** – Matrix format is not supported for this operation

2.2.13 aoclsparse_export_?csr()

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)

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)

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)

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_?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.

2.2.14 aoclsparse_export_?csc()

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)

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)

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)

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_?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 invalid.
- **aoclsparse_status_invalid_value** – `mat` is not in CSC format.
- **aoclsparse_status_wrong_type** – data type of `mat` does not match the function data type.

2.2.15 aoclsparse_export_?coo()

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

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

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

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

Export a `COO` matrix.

`aoclsparse_export_?coo` exposes the components defining the `COO` 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 `COO` format, an error is obtained.

Parameters

- **mat** – **[in]** the pointer to the COO sparse matrix.
- **base** – **[out]** *aoclsparse_index_base_zero* or *aoclsparse_index_base_one*.
- **m** – **[out]** number of rows of the sparse COO matrix.
- **n** – **[out]** number of columns of the sparse COO matrix.
- **nnz** – **[out]** number of non-zero entries of the sparse CSR matrix.
- **row_ptr** – **[out]** array of `nnz` elements containing the row indices of the sparse COO matrix.
- **col_ptr** – **[out]** array of `nnz` elements containing the column indices of the sparse COO matrix.
- **val** – **[out]** array of `nnz` elements of the sparse COO 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 COO format.
- **aoclsparse_status_wrong_type** – data type of `mat` does not match the function.

2.2.16 aoclsparse_get_mat_diag_type()

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*.

2.2.17 aoclsparse_get_mat_fill_mode()

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*.

2.2.18 aoclsparse_get_mat_index_base()

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*.

2.2.19 aoclsparse_get_mat_type()

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*.

2.2.20 aoclsparse_get_version()

const char ***aoclsparse_get_version**()

Get AOCL-Sparse Library version.

Returns

AOCL-Sparse Library version number in the format "AOCL-Sparse <major>.<minor>.<patch>"

2.2.21 aoclsparse_set_mat_diag_type()

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.

2.2.22 aoclsparse_set_mat_fill_mode()

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.

2.2.23 aoclsparse_set_mat_index_base()

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.

2.2.24 aoclsparse_set_mat_type()

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.

2.3 Conversion Functions

`aoclsparse_convert.h` provides sparse format conversion functions.

2.3.1 aoclsparse_csr2ell_width()

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.

2.3.2 aoclsparse_?csr2ell()

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)

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 nnz_{ELL} is equal to `m` times `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.

2.3.3 aoclsparse_csr2dia_ndiag()

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 number of diagonals for a given CSR matrix.

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 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.

2.3.4 aoclsparse_?csr2dia()

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)

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.

2.3.5 aoclsparse_csr2bsr_nnz()

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.

2.3.6 aoclsparse_?csr2bsr()

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)

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 this function 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.

2.3.7 aoclsparse_?csr2csc()

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)

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)

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)

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. These functions 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.

2.3.8 aoclsparse_?csr2dense()

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)

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) *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)

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 size at least `nnz` nonzero elements of matrix `A`.
- **csr_row_ptr** – **[in]** CSR row pointer array of size (`m` +1).
- **csr_col_ind** – **[in]** An array of CSR column indices of at least `nnz` column indices of the nonzero elements of matrix `A`.
- **A** – **[out]** array of dimensions (`lda`, `n`)
- **ld** – **[in]** leading dimension of dense array `A`.
- **order** – **[in]** memory layout of a dense matrix `A`. It can be either *aoclsparse_order_column* or *aoclsparse_order_row*.

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`, `csr_val`, `csr_row_ptr`, or `csr_col_ind` pointers are invalid.

2.3.9 aoclsparse_convert_csr()

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 conjugated and transposed during the conversion. It should be freed by calling *aoclsparse_destroy()*. The source matrix needs to be initalized using e.g. *aoclsparse_create_scoo()*, *aoclsparse_create_scsr()*, *aoclsparse_create_scsc()* or any of their variants.

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_value** – `src_mat` contains invalid value type
- **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

2.4 Sparse BLAS level 1, 2, and 3 functions

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

2.4.1 Level 1

The sparse level 1 routines describe operations between a vector in sparse format and a vector in dense format.

This section describes all provided level 1 sparse linear algebra functions.

aoclsparse_?axpyi()

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

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

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

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 compressed sparse vector and dense vector.

`aoclsparse_?axpyi` adds a scalar multiple of compressed sparse vector to 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 the nonzero indices set for `x` of length at least `nnz` described by `indx`, then

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

Example (tests/examples/sample_axpyi.cpp)

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 at least `nnz` elements.
- **indx** – **[in]** Nonzero indices set, I_x, of `x` described by this array of length at least `nnz`. The elements in this vector are only checked for non-negativity. The caller should make sure that all indices are less than the size of `y`. Array is assumed to be in zero base.
- **y** – **[inout]** Array of at least $\max(I_{x_i}, i \in \{1, \ldots, \mathtt{nnz}\})$ 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_?dotci()

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

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` (I_x) the nonzero indices array of `x` of length at least `nnz` that is used to index into the entries of dense vector y, then these functions return

$$\mathsf{dot} = \sum_{i=0}^{\mathsf{nnz}-1} \overline{x_i} \cdot y_{I_{x_i}}.$$

Example (tests/examples/sample_dotp.cpp)

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]** Nonzero indices set, I_x, of `x` described by this array of length at least `nnz`. Each entry 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 $\max(I_{x_i}, i \in \{1, \ldots, \mathtt{nnz}\})$ complex elements.
- **dot** – **[out]** The dot product of the conjugate of x and y when $\mathtt{nnz} > 0$. If $\mathtt{nnz} \leq 0$, `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_?dotui()

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

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` (I_x) the nonzero indices array of `x` of length at least `nnz` that is used to index into the entries of dense vector y, then these functions return

$$\mathsf{dot} = \sum_{i=0}^{\mathsf{nnz}-1} x_i \cdot y_{I_{x_i}}.$$

Example (tests/examples/sample_dotp.cpp)

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]** Nonzero indices set, I_x, of `x` described by this array of length at least `nnz`. Each entry 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 $\max(I_{x_i}, i \in \{1, \ldots, \mathtt{nnz}\})$ complex elements.

- **dot** – **[out]** The dot product of x and y when nnz > 0. If nnz ≤ 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_?doti()

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

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() and *aoclsparse_ddoti()* 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 (I_x) an indices array of length at least nnz that is used to index into the entries of dense vector y, then these functions return

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

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

Parameters

- **nnz** – **[in]** The number of elements to access in vectors x and indx.
- **x** – **[in]** Array of at least nnz elements.
- **indx** – **[in]** Nonzero indices set, I_x, of x described by this array of length at least nnz. Each entry 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 $\max(I_{x_i}, i \in \{1, \ldots, \text{nnz}\})$ elements.

Return values

dot – Value of the dot product if nnz is positive, otherwise returns 0.

aoclsparse_?sctr()

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

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

aoclsparse_status **aoclsparse_csctr**(const *aoclsparse_int* nnz, const void *x, const *aoclsparse_int* *indx, void *y) *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, and x be a compressed sparse vector with I_x be its nonzero indices set of length at least `nnz` and described by the array `indx`, then

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

Example (tests/examples/sample_sctr.cpp)

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

Parameters

- **nnz** – **[in]** The number of elements to use from x and `indx`.
- **x** – **[in]** Dense array of at least size `nnz`. The first `nnz` elements are to be scattered.
- **indx** – **[in]** Nonzero index set for x of size at least `nnz`. The first `nnz` indices are used for the scattering. 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 $\max(I_{x_i}, i \in \{1, \ldots, \texttt{nnz}\})$ 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.

sparse_?sctrs()

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

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

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

$$y_{\mathsf{stride} \times i} = x_i, \quad i \in \{1, \ldots, \mathsf{nnz}\}.$$

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

Parameters

- **nnz** – **[in]** Number of nonzero elements to access in x.
- **x** – **[in]** Array of at least `nnz` elements. The first `nnz` elements are 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` × `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_?roti()

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

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

Apply Givens rotation to single or double precision real vectors.

aoclsparse_sroti() and *aoclsparse_droti()* apply the Givens rotation 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 its nonzero indices set of length at least `nnz` described by the array `indx`, then

$$x_i = \mathsf{c} * x_i + \mathsf{s} * y_{I_{x_i}},$$

$$y_{I_{x_i}} = \mathsf{c} * y_{I_{x_i}} - \mathsf{s} * x_i,$$

for $i \in 1, \ldots, \mathsf{nnz}$. The elements `c`, `s` are scalars.

Example (tests/examples/sample_roti.cpp)

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

Parameters

- **nnz** – **[in]** The number of elements to use from x and `indx`.
- **x** – **[inout]** Array x of at least `nnz` elements in compressed form. The elements of the array are updated after applying the Givens rotation.
- **indx** – **[in]** Nonzero index set of x, I_x, with at least `nnz` elements. The first `nnz` elements are used to apply the Givens rotation. The elements in this vector are only checked for non-negativity. The caller should make sure that each entry is less than the size of `y` and are all distinct.
- **y** – **[inout]** Dense array of at least $\max(I_{x_i}, \text{ for } i \in \{1, \ldots, \texttt{nnz}\})$ elements in full storage form. The elements of the array are updated after applying the 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_?gthr()

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

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

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

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

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

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

Example - Complex space (tests/examples/sample_zgthr.cpp)

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_?gthrz()

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

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

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

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 $0 < \texttt{nnz} \leq m$ described by indx, then

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

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

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_?gthrs()

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

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

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

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 $x_i = y_{\text{stride}\times i}, \quad i \in \{1, \ldots, \mathtt{nnz}\}$.

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. Note that nnz must be such that stride $\times$ nnz must be less or equal to m.
- **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. It must be such that stride $\times$ nnz is less or equal to m.

Return values

- **aoclsparse_status_success** – the operation completed successfully.
- **aoclsparse_status_invalid_size** – at least one of the parameters nnz or stride has a negative value.
- **aoclsparse_status_invalid_pointer** – at least one of the pointers y, or x is invalid.

2.4.2 Level 2

This module holds all sparse level 2 routines.

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

aoclsparse_?mv()

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)

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)

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)

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)

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

The `aoclsparse_?mv` perform sparse matrix-vector products of the form

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

where, x and y are dense vectors, α and β are scalars, and A is a sparse matrix structure. The matrix operation $op()$ is defined as:

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

Example - C++ (tests/examples/sample_spmv.cpp)

Example - C (tests/examples/sample_spmv_c.c)

Parameters

- **op** – **[in]** Matrix operation, op can be one of *aoclsparse_operation_none*, *aoclsparse_operation_conjugate_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **alpha** – **[in]** Scalar α.
- **A** – **[in]** The sparse matrix created using e.g. *aoclsparse_create_scsr()* or other variant. Matrix is considered of size m by n.
- **descr** – **[in]** Descriptor of the matrix. These functions support the following *aoclsparse_matrix_type* types: *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 with the one defined in matrix A.

- **x** – **[in]** An array of n elements if $op(A) = A$; or of m elements if $op(A) = A^T$ or $op(A) = A^H$.
- **beta** – **[in]** Scalar β.
- **y** – **[inout]** An array of m elements if $op(A) = A$; or of n elements if $op(A) = A^T$ 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_?trsv()

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

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

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)

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 function *aoclsparse_strsv()* and variants 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,

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

Notes

1. This routine supports only sparse matrices in CSR format.

2. 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.

3. 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.

4. To increase performance and if the matrix A is to be used more than once to solve for different right-hand sides `b`, 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.

5. There is a `kid` (Kernel ID) variation of TRSV, namely with a suffix of `_kid`, *aoclsparse_strsv_kid()* (and variations) where it is possible to specify the TRSV kernel to use (if possible).

Example - Real space (tests/examples/sample_dtrsv.cpp)

Example - Complex space (tests/examples/sample_ztrsv.cpp)

Parameters

- **trans** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **alpha** – **[in]** scalar α, used to multiply right-hand side vector b.
- **A** – **[inout]** matrix containing data used to represent the $m \times m$ triangular linear system to solve.
- **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*.

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_strided**(*aoclsparse_operation* trans, const float alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const float *b, const *aoclsparse_int* incb, float *x, const *aoclsparse_int* incx) *aoclsparse_status* **aoclsparse_dtrsv_strided**(*aoclsparse_operation* trans, const double alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const double *b, const *aoclsparse_int* incb, double *x, const *aoclsparse_int* incx)

aoclsparse_status **aoclsparse_ctrsv_strided**(*aoclsparse_operation* trans, const *aoclsparse_float_complex* alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const *aoclsparse_float_complex* *b, const *aoclsparse_int* incb, *aoclsparse_float_complex* *x, const *aoclsparse_int* incx)

aoclsparse_status **aoclsparse_ztrsv_strided**(*aoclsparse_operation* trans, const *aoclsparse_double_complex* alpha, *aoclsparse_matrix* A, const *aoclsparse_mat_descr* descr, const *aoclsparse_double_complex* *b, const *aoclsparse_int* incb, *aoclsparse_double_complex* *x, const *aoclsparse_int* incx)

This is a variation of TRSV, namely with a suffix of `_strided`, allows to set the stride for the dense vectors `b` and `x`.

For full details refer to `aoclsparse_?trsv()`.

Parameters

- **trans** – **[in]** matrix operation type, either *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, or *aoclsparse_operation_conjugate_transpose*.
- **alpha** – **[in]** scalar α, used to multiply right-hand side vector b.
- **A** – **[inout]** matrix containing data used to represent the $m \times m$ triangular linear system to solve.
- **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.
- **incb** – **[in]** a positive integer holding the stride value for b vector.
- **x** – **[out]** array of `m` elements, storing the solution if solver returns *aoclsparse_status_success*.
- **incx** – **[in]** a positive integer holding the stride value for x vector.

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

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

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, *aoclsparse_int* kid)

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, *aoclsparse_int* kid)

Sparse triangular solver for real/complex single and double data precisions (kernel flag variation).

For full details refer to `aoclsparse_?trsv()`.

This variation of TRSV, namely with a suffix of `_kid`, 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
Alias to `kid=2` (Kernel Template AVX 256-bit implementation)

kid=2
Kernel Template version using `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 multiply right-hand side vector b.
- **A** – **[inout]** matrix containing data used to represent the $m \times m$ triangular linear system to solve.
- **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.

aoclsparse_?dotmv()

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)

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)

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)

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

$$y := \alpha \, op(A) \, x + \beta \, y, \quad \text{with} \quad op(A) = \left\{ \begin{array}{ll} A, & \text{if op} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if op} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if op} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{array} \right.$$

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

$$\text{d} = \left\{ \begin{array}{ll} \sum_{i=0}^{\min(m,n)-1} x_i \; y_i, & \text{real case} \\ \sum_{i=0}^{\min(m,n)-1} \overline{x_i} \; y_i, & \text{complex case} \end{array} \right.$$

Example (tests/examples/sample_dotmv.cpp)

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_scsr()* or other variation.
- **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 at least 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 at least 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 index is neither *aoclsparse_index_base_zero* nor *aoclsparse_index_base_one*, or matrix base index and descr base index values 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* is *aoclsparse_matrix_type_hermitian* or, aoclsparse_matrix_format_type is not aoclsparse_csr_mat

aoclsparse_?ellmv()

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)

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)

Real single and double precision sparse matrix vector product 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

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

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 is not *aoclsparse_operation_none*, or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*.

aoclsparse_?diamv()

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)

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)

Real single and double precision sparse matrix vector product 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

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

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 matrix.
- **n** – **[in]** number of columns of the matrix.
- **nnz** – **[in]** number of non-zero entries of the matrix.
- **descr** – **[in]** descriptor of the sparse DIA matrix.
- **dia_val** – **[in]** array that contains the elements of the matrix. Padded elements should be zero.
- **dia_offset** – **[in]** array that contains the offsets of each diagonal of the matrix.
- **dia_num_diag** – **[in]** number of diagonals in the 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 is not *aoclsparse_operation_none*, or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*.

aoclsparse_?bsrmv()

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)

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)

Real single and double precision matrix vector product using BSR storage format.

aoclsparse_?bsrmv multiplies the scalar α with a sparse mb times bsr_dim by nb times 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

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

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: 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 times bsr_dim elements ($op(A) = A$) or mb times bsr_dim elements ($op(A) = A^T$ or $op(A) = A^H$).
- **beta** – **[in]** scalar β.
- **y** – **[inout]** array of mb times bsr_dim elements ($op(A) = A$) or nb times 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 is not *aoclsparse_operation_none*, or *aoclsparse_matrix_type* is not *aoclsparse_matrix_type_general*.

aoclsparse_?csrmv()

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)

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)

Real 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

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

with

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

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_?csrsv()

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)

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.

Deprecated:

This API is superseded by *aoclsparse_strsv()* and *aoclsparse_dtrsv()*.

aoclsparse_?csrsv 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

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

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: 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*.

2.4.3 Level 3

This module holds all sparse level 3 routines.

The sparse level 3 routines describe operations between matrices.

aoclsparse_?trsm()

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)

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)

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)

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

$$op(A)\;X = \alpha\;B, \text{ with solution } X = \alpha\;\left(op(A)^{-1}\right)\;B,$$

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

$$A^T\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-T}\;B, \text{ and }\; A^H\;X = \alpha\;B, \text{ with solution } X = \alpha\;A^{-H}\;B.$$

Notes

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 (ref 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 (ref aoclsparse_order_column) and thread count is greater than 1 on a parallel build.
5. There is *`aoclsparse_trsm_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 kernels avilable in *`aoclsparse_trsv_kid`* are supported.
6. This routine supports only sparse matrices in CSR format.

Example - Real space (tests/examples/sample_dtrsm.cpp)

Example - Complex space (tests/examples/sample_ztrsm.cpp)

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

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)

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)

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)

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 (kernel flag variation).

For full details refer to `aoclsparse_?trsm()`.

This variation of TRSM, namely with a suffix of `_kid`, allows to choose which underlying TRSV kernels to use (if possible). Currently, all the existing kernels supported by `aoclsparse_?trsv_kid()` are available here as well.

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 which kernel to use.

aoclsparse_sp2m()

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

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

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 - Complex space (tests/examples/sample_zsp2m.cpp)

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_spmm()

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

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

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*

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 data types do not match.
- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – Input matrices `A` or `B` is not in CSR format

aoclsparse_?csrmm()

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)

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) *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)

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_?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

$$C = \alpha\, op(A)\, B + \beta\, C, \quad \text{with} \quad op(A) = \begin{cases} A, & \text{if trans\_A} = \text{aoclsparse\_operation\_none} \\ A^T, & \text{if trans\_A} = \text{aoclsparse\_operation\_transpose} \\ A^H, & \text{if trans\_A} = \text{aoclsparse\_operation\_conjugate\_transpose} \end{cases}$$

Example (tests/examples/sample_csrmm.cpp)

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, supports *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_symmetric*, and *aoclsparse_matrix_type_hermitian* matrices. 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 $\max(1, k)$ for $op(A) = A$, or $\max(1, m)$ when $op(A) = A^T$ 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 $\max(1, m)$ for $op(A) = A$, or $\max(1, k)$ when $op(A) = A^T$ 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 values of `descr->base` and `A->base` do not coincide.
- **aoclsparse_status_not_implemented** – *aoclsparse_matrix_type* is not one of these: *aoclsparse_matrix_type_general*, *aoclsparse_matrix_type_symmetric*, *aoclsparse_matrix_type_hermitian* or input matrix `A` is not in CSR format

aoclsparse_?csr2m()

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)

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

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

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 (tests/examples/sample_csr2m.cpp)

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_?add()

aoclsparse_status **aoclsparse_sadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const float alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

aoclsparse_status **aoclsparse_dadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const double alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

aoclsparse_status **aoclsparse_cadd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_float_complex* alpha, const *aoclsparse_matrix* B, *aoclsparse_matrix* *C)

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_?add` adds two sparse matrices and returns a sparse matrix. Matrices can be either real or complex types but cannot be intermixed. It performs

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

where A is a $m \times n$ matrix and B is a $m \times n$ matrix, if `op` = *aoclsparse_operation_none*. Otherwise A is $n \times m$ 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_internal_error** – Internal Error Occured
- **aoclsparse_status_memory_error** – Memory allocation failure.
- **aoclsparse_status_not_implemented** – Matrices are not in CSR format.

aoclsparse_?spmmd()

aoclsparse_status **aoclsparse_sspmmd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_matrix* B, const *aoclsparse_order* layout, float *C, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_dspmmd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_matrix* B, const *aoclsparse_order* layout, double *C, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_cspmmd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_matrix* B, const *aoclsparse_order* layout, *aoclsparse_float_complex* *C, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_zspmmd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_matrix* B, const *aoclsparse_order* layout, *aoclsparse_double_complex* *C, const *aoclsparse_int* ldc)

Matrix multiplication of two sparse matrices stored in the CSR storage format. The output matrix is stored in a dense format.

aoclsparse_?spmmd multiplies a sparse matrix A and a sparse matrix B, both stored in the CSR storage format, and saves the result in a dense matrix C, such that

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

with

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

Parameters

- **op** – **[in]** Operation to perform on matrix A.
- **A** – **[in]** Matrix structure containing sparse matrix A of size $m \times k$.
- **B** – **[in]** Matrix structure containing sparse matrix B of size $k \times n$ if **op** is *aoclsparse_operation_none* otherwise of size $m \times n$.
- **layout** – **[in]** Ordering of the dense output matrix: valid values are *aoclsparse_order_row* and *aoclsparse_order_column*.

- `C` – **[inout]** Dense output matrix C of size $m \times n$ if `op` is *aoclsparse_operation_none*, otherwise of size $k \times n$ containing the matrix-matrix product of A and B.
- `ldc` – **[in]** Leading dimension of C, e.g., for C stored in `aoclsparse_order_row`, `ldc` must be at least $\max(1, m)$ when $op(A) = A$, or $\max(1, k)$ if $op(A) = A^T$ or $op(A) = A^H$.

Return values

- **`aoclsparse_status_success`** – The operation completed successfully.
- **`aoclsparse_status_invalid_size`** – `m`, `n`, `k`, `nnz` or `ldc` is not valid.
- **`aoclsparse_status_invalid_pointer`** – `A`, `B` or `C` pointer is not valid.
- **`aoclsparse_status_wrong_type`** – *aoclsparse_matrix_data_type* does not match the precision type.
- **`aoclsparse_status_not_implemented`** – aoclsparse_matrix_format_type is not aoclsparse_csr_mat.

aoclsparse_?sp2md()

aoclsparse_status **`aoclsparse_ssp2md`**(const *aoclsparse_operation* opA, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* A, const *aoclsparse_operation* opB, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* B, const float alpha, const float beta, float *C, const *aoclsparse_order* layout, const *aoclsparse_int* ldc)

aoclsparse_status **`aoclsparse_dsp2md`**(const *aoclsparse_operation* opA, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* A, const *aoclsparse_operation* opB, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* B, const double alpha, const double beta, double *C, const *aoclsparse_order* layout, const *aoclsparse_int* ldc)

aoclsparse_status **`aoclsparse_csp2md`**(const *aoclsparse_operation* opA, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* A, const *aoclsparse_operation* opB, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* B, *aoclsparse_float_complex* alpha, *aoclsparse_float_complex* beta, *aoclsparse_float_complex* *C, const *aoclsparse_order* layout, const *aoclsparse_int* ldc)

aoclsparse_status **`aoclsparse_zsp2md`**(const *aoclsparse_operation* opA, const *aoclsparse_mat_descr* descrA, const *aoclsparse_matrix* A, const *aoclsparse_operation* opB, const *aoclsparse_mat_descr* descrB, const *aoclsparse_matrix* B, *aoclsparse_double_complex* alpha, *aoclsparse_double_complex* beta, *aoclsparse_double_complex* *C, const *aoclsparse_order* layout, const *aoclsparse_int* ldc)

A variant of matrix multiplication of two sparse matrices stored in the CSR storage format. The output matrix is stored in a dense format. Supports operations on both sparse matrices.

`aoclsparse_?sp2md` multiplies a sparse matrix A and a sparse matrix B, both stored in the CSR storage format, and saves the result in a dense matrix C, such that

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

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

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

Parameters

- **opA** – **[in]** Operation to perform on matrix A.
- **descrA** – **[in]** Descriptor of A. Only *aoclsparse_matrix_type_general* is supported at present. As a consequence, all other parameters within the descriptor are ignored.
- **A** – **[in]** Matrix structure containing sparse matrix A of size $m \times k$.
- **opB** – **[in]** Operation to perform on matrix B.
- **descrB** – **[in]** Descriptor of B. Only *aoclsparse_matrix_type_general* is supported at present. As a consequence, all other parameters within the descriptor are ignored.
- **B** – **[in]** Matrix structure containing sparse matrix B of size $k \times n$ if `op` is *aoclsparse_operation_none* otherwise of size $m \times n$.
- **alpha** – **[in]** Value of α.
- **beta** – **[in]** Value of β.
- **C** – **[inout]** Dense output matrix C.
- **layout** – **[in]** Ordering of the dense output matrix: valid values are *aoclsparse_order_row* and *aoclsparse_order_column*.
- **ldc** – **[in]** Leading dimension of C, e.g., for C stored in `aoclsparse_order_row`, `ldc` 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** – `m`, `n`, `k`, `nnz` or `ldc` is not valid.
- **aoclsparse_status_invalid_pointer** – `A`, `B` or `C` pointer is not valid.
- **aoclsparse_status_wrong_type** – *aoclsparse_matrix_data_type* does not match the precision type.
- **aoclsparse_status_not_implemented** – aoclsparse_matrix_format_type is not aoclsparse_csr_mat.
- **aoclsparse_status_internal_error** – An internal error occurred.

aoclsparse_syrk()

aoclsparse_status **aoclsparse_syrk**(const *aoclsparse_operation* opA, const *aoclsparse_matrix* A, *aoclsparse_matrix* *C)

Multiplication of a sparse matrix and its transpose (or conjugate transpose) stored as a sparse matrix.

`aoclsparse_syrk` multiplies a sparse matrix with its transpose (or conjugate transpose) in CSR storage format. The result is stored in a newly allocated sparse matrix in CSR format, such that

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

if opA is *aoclsparse_operation_none*.

Otherwise,

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

where

$$op(A) = \begin{cases} A^T, & \text{transpose of A for real matrices} \\ A^H, & \text{conjugate transpose of A for complex matrices} \end{cases}$$

where A is a $m \times n$ matrix, opA is one of *aoclsparse_operation_none*, *aoclsparse_operation_transpose* (for real matrices) or *aoclsparse_operation_conjugate_transpose* (for complex matrices). The output matrix C is a sparse symmetric (or Hermitian) matrix stored as an upper triangular matrix in CSR format.

Example (tests/examples/sample_dsyrk.cpp)

Note: `aoclsparse_syrk` assumes that the input CSR matrix has sorted column indices in each row. If not, call *aoclsparse_order_mat()* before calling `aoclsparse_syrk`.

Note: `aoclsparse_syrk` currently does not support *aoclsparse_operation_transpose* for complex `A`.

Parameters

- **opA** – **[in]** Matrix A operation type.
- **A** – **[in]** Sorted sparse CSR matrix A.
- ***C** – **[out]** Pointer to the new sparse CSR symmetric/Hermitian matrix C. Only upper triangle of the result matrix is computed. The column indices of the output matrix in CSR format might be unsorted. The matrix should be freed by *aoclsparse_destroy()* when no longer needed.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `A`, `C` is invalid.
- **aoclsparse_status_wrong_type** – A and its operation type do not match.
- **aoclsparse_status_not_implemented** – The input matrix is not in the CSR format or `opA` is aoclsparse_operation_transpose and `A` has complex values.
- **aoclsparse_status_invalid_value** – The value of opA is invalid.
- **aoclsparse_status_unsorted_input** – Input matrices are not sorted.
- **aoclsparse_status_memory_error** – Memory allocation failure.

aoclsparse_?syrkd()

aoclsparse_status **aoclsparse_ssyrkd**(const *aoclsparse_operation* opA, const *aoclsparse_matrix* A, const float alpha, const float beta, float *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_dsyrkd**(const *aoclsparse_operation* opA, const *aoclsparse_matrix* A, const double alpha, const double beta, double *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_csyrkd**(const *aoclsparse_operation* opA, const *aoclsparse_matrix* A, const *aoclsparse_float_complex* alpha, const *aoclsparse_float_complex* beta, *aoclsparse_float_complex* *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_zsyrkd**(const *aoclsparse_operation* opA, const *aoclsparse_matrix* A, const *aoclsparse_double_complex* alpha, const *aoclsparse_double_complex* beta, *aoclsparse_double_complex* *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

Multiplication of a sparse matrix and its transpose (or conjugate transpose) for all data types.

`aoclsparse_syrkd` multiplies a sparse matrix with its transpose (or conjugate transpose) in CSR storage format. The result is stored in a dense format, such that

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

if opA is *aoclsparse_operation_none*.

Otherwise,

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

$$op(A) = \left\{ \begin{array}{ll} A^T, & \text{transpose of A for real matrices} \\ A^H, & \text{conjugate transpose of A for complex matrices} \end{array} \right.$$

where A is a $m \times n$ sparse matrix, `opA` is one of *aoclsparse_operation_none*, *aoclsparse_operation_transpose* (for real matrices) or *aoclsparse_operation_conjugate_transpose* (for complex matrices). The output matrix C is a dense symmetric (or Hermitian) matrix stored as an upper triangular matrix.

Example (tests/examples/sample_dsyrkd.cpp)

Note: `aoclsparse_syrkd` assumes that the input CSR matrix has sorted column indices in each row. If not, call *aoclsparse_order_mat()* before calling `aoclsparse_syrkd`.

Note: For complex type, only the real parts of α and β are taken into account to preserve Hermitian C.

Note: `aoclsparse_syrkd` currently does not support *aoclsparse_operation_transpose* for complex `A`.

Parameters

- **opA** – **[in]** Matrix A operation type.
- **A** – **[in]** Sorted sparse CSR matrix A.
- **alpha** – **[in]** Scalar α.
- **beta** – **[in]** Scalar β.
- **C** – **[inout]** Output dense matrix. Only upper triangular part of the matrix is processed during the computation, the strictly lower triangle is not modified.
- **orderC** – **[in]** Storage format of the output dense matrix, C. It can be *aoclsparse_order_row* or *aoclsparse_order_column*.
- **ldc** – **[in]** Leading dimension of C.

Return values

- **aoclsparse_status_success** – The operation completed successfully.
- **aoclsparse_status_invalid_pointer** – `A`, `C` is invalid.
- **aoclsparse_status_wrong_type** – `A` and its operation type do not match.
- **aoclsparse_status_not_implemented** – The input matrix is not in the CSR format or `opA` is aoclsparse_operation_transpose and `A` has complex values.
- **aoclsparse_status_invalid_value** – The value of `opA`, `orderC` or `ldc` is invalid.
- **aoclsparse_status_unsorted_input** – Input matrix is not sorted.
- **aoclsparse_status_memory_error** – Memory allocation failure.

aoclsparse_?sypr()

aoclsparse_status **aoclsparse_sypr**(*aoclsparse_operation* opA, const *aoclsparse_matrix* A, const *aoclsparse_matrix* B, const *aoclsparse_mat_descr* descrB, *aoclsparse_matrix* *C, const *aoclsparse_request* request)

Symmetric product of three sparse matrices for real and complex datatypes stored as a sparse matrix.

`aoclsparse_sypr` multiplies three sparse matrices in CSR storage format. The result is returned in a newly allocated symmetric or Hermitian sparse matrix stored as an upper triangle in CSR format.

If *opA* is *aoclsparse_operation_none*,

$$C = A \cdot B \cdot A^T,$$

or

$$C = A \cdot B \cdot A^H,$$

for real or complex input matrices, respectively, where A is a $m \times n$ general matrix , B is a $n \times n$ symmetric (for real data types) or Hermitian (for complex data types) matrix, resulting in a symmetric or Hermitian $m \times m$ matrix C.

Otherwise,

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

with

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

where A is a $m \times n$ matrix and B is a $m \times m$ symmetric (or Hermitian) matrix, resulting in a $n \times n$ symmetric (or Hermitian) matrix C.

Depending on `request`, `aoclsparse_sypr` might compute the result in a single stage (*aoclsparse_stage_full_computation*) or in two stages. Then the first stage (*aoclsparse_stage_nnz_count*) allocates memory for the new output matrix C and computes its number of non-zeros and their structure which is followed by the second stage (*aoclsparse_stage_finalize*) to compute the column indices and values of all elements. The second stage can be invoked multiple times (either after *aoclsparse_stage_full_computation* or *aoclsparse_stage_nnz_count*) to recompute the numerical values of C on assumption that the sparsity structure of the input matrices remained unchanged and only the values of the non-zero elements were modified (e.g., by a call to *aoclsparse_supdate_values()* and variants).

Example (tests/examples/sample_zsypr.cpp)

Note: `aoclsparse_sypr` supports only matrices in CSR format which have sorted column indices in each row. If the matrices are unsorted, you might want to call *aoclsparse_order_mat()*.

Note: Currently, `opA` = *aoclsparse_operation_transpose* is supported only for real data types.

Parameters

- **`opA`** – **[in]** matrix A operation type.
- **`A`** – **[in]** sorted sparse CSR matrix A.
- **`B`** – **[in]** sorted sparse CSR matrix B to be interpreted as symmetric (or Hermitian).
- **`descrB`** – **[in]** descriptor of the sparse CSR matrix B. *aoclsparse_matrix_type* must be *aoclsparse_matrix_type_symmetric* for real matrices or *aoclsparse_matrix_type_hermitian* for complex matrices. *aoclsparse_fill_mode* might be either *aoclsparse_fill_mode_upper* or *aoclsparse_fill_mode_lower* to process the upper or lower triangular matrix part, respectively.
- **`request`** – **[in]** Specifies if the computation takes place in one stage (*aoclsparse_stage_full_computation*) or in two stages (*aoclsparse_stage_nnz_count* followed by *aoclsparse_stage_finalize*).
- **`*C`** – **[inout]** Pointer to the new sparse CSR symmetric/Hermitian matrix C . Only upper triangle of the result matrix is computed. Matrix C will always have zero-based indexing, irrespective of the zero/one-based indexing of the input matrices A and B. The column indices of the output matrix in CSR format might be unsorted. If `request` is *aoclsparse_stage_finalize*, matrix C must not be modified by the user since the last call to `aoclsparse_sypr`, in the other cases is C treated as an output only. The matrix should be freed by *aoclsparse_destroy()* when no longer needed.

Return values

- **`aoclsparse_status_success`** – the operation completed successfully.
- **`aoclsparse_status_invalid_pointer`** – `descrB`, `A`, `B` or `C` is invalid.

- **aoclsparse_status_invalid_size** – Matrix dimensions do not match `A` or `B` is not square.
- **aoclsparse_status_invalid_value** – Input parameters are invalid, for example, `descrB` does not match `B` indexing or `B` is not symmetric/Hermitian, `C` has been modified between stages or `opA` or `request` is not recognized.
- **aoclsparse_status_wrong_type** – `A` and `B` matrix data types do not match.
- **aoclsparse_status_not_implemented** – Input matrix `A` or `B` is not in CSR format.
- **aoclsparse_status_unsorted_input** – Input matrices are not sorted.
- **aoclsparse_status_memory_error** – Memory allocation failure.

aoclsparse_?syprd()

aoclsparse_status **aoclsparse_ssyprd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const float *B, const *aoclsparse_order* orderB, const *aoclsparse_int* ldb, const float alpha, const float beta, float *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_dsyprd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const double *B, const *aoclsparse_order* orderB, const *aoclsparse_int* ldb, const double alpha, const double beta, double *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_csyprd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_float_complex* *B, const *aoclsparse_order* orderB, const *aoclsparse_int* ldb, const *aoclsparse_float_complex* alpha, const *aoclsparse_float_complex* beta, *aoclsparse_float_complex* *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

aoclsparse_status **aoclsparse_zsyprd**(const *aoclsparse_operation* op, const *aoclsparse_matrix* A, const *aoclsparse_double_complex* *B, const *aoclsparse_order* orderB, const *aoclsparse_int* ldb, const *aoclsparse_double_complex* alpha, const *aoclsparse_double_complex* beta, *aoclsparse_double_complex* *C, const *aoclsparse_order* orderC, const *aoclsparse_int* ldc)

Performs symmetric triple product of a sparse matrix and a dense matrix and stores the output as a dense matrix.

`aoclsparse_?syprd` performs product of a scalar α, with the symmetric triple product of a sparse $m \times k$ matrix A, defined in CSR format, with a $k \times k$ symmetric dense (or Hermitian) matrix B, and a $k \times m$ $op(A)$. Adds the resulting matrix to $m \times m$ symmetric dense (or Hermitian) matrix C that is multiplied by a scalar β, such that

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

if *op* is *aoclsparse_operation_none*.

Otherwise,

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

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

Notes

1. This routine assumes the dense matrices (B and C) are stored in full although the computations happen on the upper triangular portion of the matrices.
2. *aoclsparse_operation_transpose* is only supported for real matrices.
3. *aoclsparse_operation_conjugate_transpose* is only supported for complex matrices.
4. Complex dense matrices are assumed to be Hermitian matrices.

Parameters

- **op** – **[in]** Matrix A operation type.
- **A** – **[in]** Sparse CSR matrix A structure.
- **B** – **[in]** Array of dimension $ldb \times ldb$. Only the upper triangular matrix is used for computation.
- **orderB** – **[in]** *aoclsparse_order_row* or *aoclsparse_order_column* for dense matrix B.
- **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 $op(A) = A^H$).
- **alpha** – **[in]** Scalar α.
- **beta** – **[in]** Scalar β.
- **C** – **[inout]** Array of dimension $ldc \times ldc$. Only upper triangular part of the matrix is processed.
- **orderC** – **[in]** *aoclsparse_order_row* or *aoclsparse_order_column* for dense matrix C.
- **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_invalid_operation** – The operation is invalid if the matrix B and C has a different layout ordering.
- **aoclsparse_status_wrong_type** – The data type of the matrices are not matching or invalid.
- **aoclsparse_status_invalid_size** – The value of `m`, `k`, `nnz`, `ldb` or `ldc` is invalid.
- **aoclsparse_status_invalid_pointer** – The pointer `A`, `B`, or `C` is invalid.
- **aoclsparse_status_not_implemented** – The values of `orderB` and `orderC` are different.

2.4.4 Miscellaneous

aoclsparse_ilu_?smoother()

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) *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)

Incomplete LU factorization with zero fill-in, ILU(0).

Performs incomplete LU factorization with zero fill-in on symmetric sparse matrix `A` of size $n \times n$. It also performs a solve for `x` in

$$LUx = b, \qquad \text{where} \qquad LU \approx A.$$

Matrix `A` should be numerically of full rank. Currently single and double precision datatypes are supported.

Example (tests/examples/sample_itsol_d_gmres.cpp)

Parameters

- **op** – **[in]** matrix `A` operation type. Transpose not supported in this release.
- **A** – **[in]** sparse symmetric 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.
- **precond_csr_val** – **[out]** pointer that contains L and U factors after ILU factorization operation. `A` is not overwritten with the factors.
- **approx_inv_diag** – **[in]** Reserved for future use.
- **x** – **[out]** array of `n` elements containing the solution to solving aproximatly $Ax = b$.
- **b** – **[in]** Right-hand-side of the linear system of equations $Ax = b$.

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

2.5 Iterative Linear System Solvers

2.5.1 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.

Forward and Reverse Communication Interfaces

The suite presents two separate interfaces to all the iterative solvers, a direct one, `aoclsparse_itsol_d_solve()` (`aoclsparse_itsol_s_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.

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_s_rci_input()` (or `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()`.

Information Array

The array `rinfo[100]` is used by the solvers (e.g. `aoclsparse_itsol_s_solve()` or `aoclsparse_itsol_d_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.

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.

2.5.2 API documentation

aoclsparse_itsol_rci_job

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.

aoclsparse_itsol_?_init()

aoclsparse_status **aoclsparse_itsol_s_init**(*aoclsparse_itsol_handle* *handle)

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_itsol_destroy()

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_itsol_?_solve()

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) *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*.

Example - CG / floating point double precision (tests/examples/sample_itsol_d_cg.cpp)

Example - GMRES / floating point double precision (tests/examples/sample_itsol_d_gmres.cpp)

Example - CG / floating point single precision (tests/examples/sample_itsol_s_cg.cpp)

Example - GMRES / floating point single precision (tests/examples/sample_itsol_s_gmres.cpp)

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_itsol_option_set()

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*.

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_itsol_handle_prn_options()

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_itsol_?_rci_input()

aoclsparse_status **aoclsparse_itsol_s_rci_input**(*aoclsparse_itsol_handle* handle, *aoclsparse_int* n, const float *b)

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_itsol_?_rci_solve()

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])

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*.

Example - CG / floating point double precision (tests/examples/sample_itsol_d_cg_rci.cpp)

Example - GMRES / floating point double precision (tests/examples/sample_itsol_d_gmres.cpp)

Example - CG floating point single precision (tests/examples/sample_itsol_s_cg_rci.cpp)

Example - GMRES / floating point single precision (tests/examples/sample_itsol_s_gmres.cpp)

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.

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_?symgs()

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

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

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

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

Symmetric Gauss Seidel(SYMGS) Preconditioner for real/complex single and double data precisions.

`aoclsparse_?symgs` performs an iteration of Gauss Seidel preconditioning. Krylov methods such as CG (Conjugate Gradient) and GMRES (Generalized Minimal Residual) are used to solve large sparse linear systems of the form

$$op(A)\ x = \alpha b,$$

where A is a sparse matrix of size m, $op()$ is a linear operator, b is a dense right-hand side vector and x is the unknown dense vector, while α is a scalar. This Gauss Seidel(GS) relaxation is typically used either as a preconditioner for a Krylov solver directly, or as a smoother in a V –cycle of a multigrid preconditioner to accelerate the convergence rate. The Symmetric Gauss Seidel algorithm performs a forward sweep followed by a backward sweep to maintain symmetry of the matrix operation.

To solve a linear system $Ax = b$, Gauss Seidel(GS) iteration is based on the matrix splitting

$$A = L + D + U = -E + D - F$$

where $-E$ or L is strictly lower triangle, D is diagonal and $-F$ or D is strictly upper triangle. Gauss-Seidel is best derived as element-wise (refer Yousef Saad's book Iterative Methods for Sparse Linear Systems, Second Edition, Chapter 4.1, p. 125 onwards):

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij}\ x_j - \sum_{j=i+1}^{n} a_{ij}\ x_j \right)$$

where the first sum is lower triangle i.e., $-Ex$ and the second sum is upper triangle i.e., $-Fx$. If we iterate through the rows `i=1` to `n` and keep overwriting/reusing the new x_i, we get forward GS, expressed in matrix form as,

$$(D - E)\ x_{k+1} = F\ x_k + b$$

Iterating through the rows in reverse order from `i=n` to `1`, the upper triangle keeps using the new x_{k+1} elements and we get backward GS, expressed in matrix form as,

$$(D - F)\ x_{k+1} = E\ x_k + b$$

The above two equations can be expressed in terms of `L`, `D` and `U` as follows,

$$(L + D)\ x_1 = b - U\ x_0$$

$$(U + D)\ x = b - L\ x_1$$

So, Symmetric Gauss Seidel (SYMGS) can be computed using two `aoclsparse_?mv` and two `aoclsparse_?trsv` operations.

The sparse matrix A can be either a symmetric or a Hermitian matrix, whose fill is indicated by `fill_mode` from the matrix descriptor `descr` where either upper or lower triangular portion of the matrix 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 conjugate transposition operations. By default, no transposition is performed. The right-hand-side vector b and the solution vector x are dense and must be of the correct size, that is m. If used as fixed point iterative method, the convergence is guaranteed for strictly diagonally dominant and symmetric positive definite matrices from any starting point, `x0`. However, the API can be applied to wider types of input or as a preconditioning step. Refer Yousef Saad's Iterative Methods for Sparse Linear Systems 2nd Edition, Theorem 4.9 and related literature for mathematical theory.

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 couple of work array buffers of size m for to store intermediate results

4. If the input matrix is of triangular type, the SGS is computed using a single `aoclsparse_?trsv` operation and a quick return is made without going through the 3-step reference(described above)

Example - Real space (tests/examples/sample_dsymgs.cpp)

Example - Complex space (tests/examples/sample_zsymgs.cpp)

Note:

Parameters

- **trans** – **[in]** matrix operation to perform on A. Possible values are *aoclsparse_operation_none*, *aoclsparse_operation_transpose*, and *aoclsparse_operation_conjugate_transpose*.
- **A** – **[in]** sparse matrix A of size m.
- **descr** – **[in]** descriptor of the sparse matrix A.
- **alpha** – **[in]** scalar α.
- **b** – **[in]** dense vector, of size m.
- **x** – **[out]** solution vector x, dense vector of size m.

Return values

- **aoclsparse_status_success** – indicates that the operation completed successfully.
- **aoclsparse_status_invalid_size** – informs that either `m`, `n` or `nnz` is invalid. The error code also informs if the given sparse matrix A is not square.
- **aoclsparse_status_invalid_value** – informs that either `base`, `trans`, matrix type `descr->type` or fill mode `descr->fill_mode` is invalid. If the sparse matrix A is not of full rank, the error code is returned to indicate that the linear system cannot be solved.
- **aoclsparse_status_invalid_pointer** – informs that either `descr`, `A`, `b`, or `x` pointer is invalid.
- **aoclsparse_status_not_implemented** – this error occurs when the provided matrix's *aoclsparse_fill_mode* is *aoclsparse_diag_type_unit* or the input format is not aoclsparse_csr_mat, or when *aoclsparse_matrix_type* is *aoclsparse_matrix_type_general* and `trans` is *aoclsparse_operation_conjugate_transpose*.

Warning: doxygenfunction: Cannot find function "aoclsparse_ssymgs_mv" in doxygen xml output for project "aocl-sparse" from directory: /home/manu/AOCL/builds/gcc/docs/xml

Warning: doxygenfunction: Cannot find function "aoclsparse_dsymgs_mv" in doxygen xml output for project "aocl-sparse" from directory: /home/manu/AOCL/builds/gcc/docs/xml

Warning: doxygenfunction: Cannot find function "aoclsparse_csymgs_mv" in doxygen xml output for project "aocl-sparse" from directory: /home/manu/AOCL/builds/gcc/docs/xml

Warning: doxygenfunction: Cannot find function "aoclsparse_zsymgs_mv" in doxygen xml output for project "aocl-sparse" from directory: /home/manu/AOCL/builds/gcc/docs/xml

aoclsparse_?sorv()

aoclsparse_status **aoclsparse_ssorv**(*aoclsparse_sor_type* sor_type, const *aoclsparse_mat_descr* descr, const *aoclsparse_matrix* A, float omega, float alpha, float *x, const float *b)

aoclsparse_status **aoclsparse_dsorv**(*aoclsparse_sor_type* sor_type, const *aoclsparse_mat_descr* descr, const *aoclsparse_matrix* A, double omega, double alpha, double *x, const double *b)

Performs successive over-relaxation preconditioner operation for single and double precision datatypes to solve a linear system of equations $Ax = b$.

`aoclsparse_?sorv` performs successive over-relaxation preconditioner on a linear system of equations represented using a sparse matrix A in CSR storage format. This is an iterative technique that solves the left hand side of this expression for `x`, using an initial guess for `x`

$$(D + \omega\, L)\, x^1 = \omega\, b - (\omega\, U + (\omega - 1)\, D)\, x^0$$

where $A = L + D + U$, x^0 is an input vector `x` and x^1 is an output stored in vector `x`.

Initially

$$x^0 = \begin{cases} alpha * x^0, & \text{if } alpha \neq 0 \\ 0, & \text{if } alpha = 0 \end{cases}$$

The convergence is guaranteed for strictly diagonally dominant and positive definite matrices from any starting point, x^0. API returns the vector x after single iteration. Caller can invoke this function in a loop until their desired convergence is reached.

NOTE:

1. Input CSR matrix should have non-zero full diagonals with each diagonal occurring only once in a row.
2. API supports forward sweep on general matrix for single and double precision datatypes.

Example (tests/examples/sample_dsorv.cpp)

Parameters

- **`sor_type`** – **[in]** Selects the type of operation performed by the preconditioner. Only *aoclsparse_sor_forward* is supported at present.
- **`descr`** – **[in]** Descriptor of A. Only *aoclsparse_matrix_type_general* is supported at present. As a consequence, all other parameters within the descriptor are ignored.
- **`A`** – **[in]** Matrix structure containing a square sparse matrix A of size $m \times m$.
- **`omega`** – **[in]** Relaxation factor. For better convergence, 0 < ω < 2. If ω = 1, the preconditioner is equivalent to the Gauss-Seidel method.
- **`alpha`** – **[in]** Scalar value used to normalize or set to zero the vector `x` that holds an initial guess.
- **`x`** – **[inout]** A vector of m elements that holds an initial guess as well as the solution vector.
- **`b`** – **[in]** A vector of m elements that holds the right-hand side of the equation being solved.

Return values

- **aoclsparse_status_success** – Completed successfully.
- **aoclsparse_status_invalid_pointer** – One or more of the pointers `A`, `descr`, `x` or `b` are invalid.
- **aoclsparse_status_wrong_type** – Data type of `A` does not match the function.
- **aoclsparse_status_not_implemented** – Expecting general matrix in CSR format for single or double precision datatypes with *aoclsparse_sor_forward*.
- **aoclsparse_status_invalid_size** – Matrix is not square.
- **aoclsparse_status_invalid_value** – `M` or `N` is set to a negative value; or `A`, `descr` or `sor_type` has invalid value; or presence of zero-valued or repeated diagonal elements.

2.6 AOCL-Sparse Types

2.6.1 Numerical types

typedef int32_t **aoclsparse_int**

Specifies the size in bits of integer type to be used.

Typedef used to define the integer type this can be either 32-bit or 64-bit interger type.

This is determined at compile-time and can be specified using the CMake option **-DBUILD_ILP64=On|Off** Setting to **On** will use use 64-bit integer data type.

struct **aoclsparse_float_complex**

Default complex float type.

User can redefine to accomodate custom complex float type definition.

Note: The library expects that complex numbers real and imaginary parts are contiguous in memory.

Public Members

float **real**

Real part.

float **imag**

Imaginary part.

struct **aoclsparse_double_complex**

Default complex double type.

User can redefine to accomodate custom complex double type definition.

Note: The library expects that complex numbers real and imaginary parts are contiguous in memory.

Public Members

double **real**

Real part.

double **imag**

Imaginary part.

2.6.2 Matrix object and descriptor

typedef struct _aoclsparse_matrix ***aoclsparse_matrix**

Matrix object.

This structure holds the matrix data. It is initialized using e.g. *aoclsparse_create_scsr* (or other variants, see table bellow). The returned matrix object needs be passed to all subsequent library calls that involve the matrix. It should be destroyed at the end using *aoclsparse_destroy*.

Table 2: Initialization of matrix objects.

Storage	*Precision* P	Initialization function
Compressed Sparse Rows (CSR)	s, d, c, z	`aoclsparse_create_Pcsr`
Compressed Sparse Columns (CSC)	s, d, c, z	`aoclsparse_create_Pcsc`
Coordinate storage (COO)	s, d, c, z	`aoclsparse_create_Pcoo`
Triangular Compressed Sparse Rows (TCSR)	s, d, c, z	`aoclsparse_create_Ptcsr`

typedef struct _aoclsparse_mat_descr ***aoclsparse_mat_descr**

Matrix object descriptor.

This structure holds properties describing a matrix and how to access its data. It must be initialized using *aoclsparse_create_mat_descr* and the returned descriptor object is passed to all subsequent library calls that involve the matrix. It is destroyed by using *aoclsparse_destroy_mat_descr*.

2.6.3 Enums

Function return status

enum **aoclsparse_status**

Values returned by the library API to indicate success or failure.

This table provides a brief explanation on the reason why a function call failed. It is **strongly** encouraged during the development cycle of applications or services to check the exit status of any call.

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.

enumerator **aoclsparse_status_unsorted_input**
the input matrices are not sorted

enumerator **aoclsparse_status_invalid_kid**
user requested kernel id was not available.

Associated with `aoclsparse_matrix`

enum **`aoclsparse_matrix_data_type`**

Specify the matrix *data type*.

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.

See also:

- *`aoclsparse_index_base`*

Associated with matrix descriptor (`aoclsparse_mat_descr`)

enum **`aoclsparse_matrix_type`**

Specify the matrix type.

Specifies the type of a matrix. A matrix object descriptor describes how to interpret the type of the matrix. The data in the matrix object need not match the type in the matrix object descriptor. It can be set using *aoclsparse_set_mat_type* and retrieved using *aoclsparse_get_mat_type*.

Values:

enumerator **`aoclsparse_matrix_type_general`**

general matrix, no special pattern.

enumerator **`aoclsparse_matrix_type_symmetric`**

symmetric matrix, $A = A^T$. It stores only a single triangle specified using *aoclsparse_fill_mode*.

enumerator **`aoclsparse_matrix_type_hermitian`**

hermitian matrix, $A = A^H$. Same storage comment as for the symmetric case.

enumerator **`aoclsparse_matrix_type_triangular`**

triangular matrix, $A = \text{tril}(A)$ or $A = \text{triu}(A)$. Here too, *aoclsparse_fill_mode* specifies which triangle is available.

enum **aoclsparse_index_base**

Specify the matrix index base.

Indicate the base used on the matrix indices, either 0-base (C, C++) or 1-base (Fortran). The base is set using aoclsparse_set_mat_index_base. The current of a matrix object can be obtained by calling *aoclsparse_get_mat_index_base*.

Note: The base-indexing information is stored in two distinc locations: the matrix object *aoclsparse_matrix* and the matrix object descriptior *aoclsparse_mat_descr*, these **must** coincide, either be both zero or both one. Any function accepting both objects will fail if these do not match.

Values:

enumerator **aoclsparse_index_base_zero**

zero based indexing, C/C++ indexing.

enumerator **aoclsparse_index_base_one**

one based indexing, Fortran indexing.

enum **aoclsparse_diag_type**

Indicates how to interpret the diagonal entries of a matrix.

Used to indicate how to use the diagonal elements of a matrix. The purpose of this is to optimize certain operations inside the kernels. If the diagonal elements are not stored but should be interpreted has being all ones, then this can accelerate the operation by avoiding unnecessary memory accesses. For a given *aoclsparse_mat_descr*, the diagonal type can be set using *aoclsparse_set_mat_diag_type* and can be retrieved by calling *aoclsparse_get_mat_diag_type*.

Values:

enumerator **aoclsparse_diag_type_non_unit**

diagonal entries are present and arbitrary.

enumerator **aoclsparse_diag_type_unit**

diagonal entries are to be considered all ones. Kernels will not access the diagonal elements in the matrix data.

enumerator **aoclsparse_diag_type_zero**

ignore diagonal entries: for specifying strict lower or upper triangular matrices.

enum **aoclsparse_fill_mode**

Specify the matrix fill mode.

Indicates if the lower or the upper part of a triangular or symmetric matrix is stored. The fill mode can be set using *aoclsparse_set_mat_fill_mode*, and can be retrieved by calling *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.

enum **aoclsparse_order**

Specify the memory layout (order) used to store a dense matrix.

Values:

enumerator **aoclsparse_order_row**

Row major, (C/C++ storage).

enumerator **aoclsparse_order_column**

Column major, (Fortran storage).

Miscellaneous

enum **aoclsparse_operation**

Indicate the operation type performed on a matrix.

Values:

enumerator **aoclsparse_operation_none**

No operation is performed on the matrix.

enumerator **aoclsparse_operation_transpose**

Operate with transpose.

enumerator **aoclsparse_operation_conjugate_transpose**

Operate with conjugate transpose.

typedef struct _aoclsparse_itsol_handle ***aoclsparse_itsol_handle**

Optimization handle.

This type of handle is a container box for storing problem data and optional parameter values. it must be initialized using *aoclsparse_itsol_s_init*, and should be destroyed after using it with *aoclsparse_itsol_destroy*. For double precision data types use *aoclsparse_itsol_d_init*.

For more details, refer to Solver chapter introduction Iterative Solver Suite (itsol).

enum **aoclsparse_ilu_type**

Specify the type of Incomplete LU (ILU) factorization.

Indicates the type of factorization to perform.

Values:

enumerator **aoclsparse_ilu0**

Incomplete LU with zero fill-in, ILU(0).

enumerator **aoclsparse_ilup**

Incomplete LU with thresholding, ILU(p). Not implemented in this release.

enum **aoclsparse_request**

Request stages for API that perform sparse matrix products.

This list describes the possible request types used by matrix product kernels such as `aoclsparse_csr2m`.

Values:

enumerator **aoclsparse_stage_nnz_count**

Perform only first stage of analysis and computation. No result is returned but it is useful when optimizing for multiple calls.

enumerator **aoclsparse_stage_finalize**

Perform computation. After this stage the product result is returned. Needs to follow after a call with *aoclsparse_stage_nnz_count* request.

enumerator **aoclsparse_stage_full_computation**

Indicates to perform the entire computation in a single call.

enum **aoclsparse_sor_type**

List of successive over-relaxation types.

This is a list of supported SOR types that are supported by *aoclsparse_dsorv* (or other variants function).

Values:

enumerator **aoclsparse_sor_forward**

Forward sweep.

enumerator **aoclsparse_sor_backward**

Backward sweep.

enumerator **aoclsparse_sor_symmetric**

Symmetric preconditioner.

enum **aoclsparse_memory_usage**

List of memory utilization policy.

This is a list of supported *aoclsparse_memory_usage()* types that are used by optimization routine.

Values:

enumerator **aoclsparse_memory_usage_minimal**

Allocate memory only for auxiliary structures.

enumerator **aoclsparse_memory_usage_unrestricted**

Allocate memory upto matrix size for appropriate sparse format conversion. Default value.

2.7 Storage Schemes

This section describes the storage schemes supported by the library… etc.

2.7.1 Compressed Sparse Row (CSR) Format

2.7.2 Triangular Compressed Sparse Row (TCSR) Format

2.7.3 Compressed Sparse Column (CSC) Format

2.7.4 Coordinate (COO) storage format

2.7.5 DIAG format

2.8 Search the documentation

- genindex
- search

INDEX

A

aoclsparse_cadd (*C++ function*), 57
aoclsparse_caxpyi (*C++ function*), 28
aoclsparse_ccsr2csc (*C++ function*), 25
aoclsparse_ccsr2dense (*C++ function*), 26
aoclsparse_ccsrmm (*C++ function*), 54
aoclsparse_cdotci (*C++ function*), 29
aoclsparse_cdotmv (*C++ function*), 41
aoclsparse_cdotui (*C++ function*), 30
aoclsparse_cgthr (*C++ function*), 34
aoclsparse_cgthrs (*C++ function*), 36
aoclsparse_cgthrz (*C++ function*), 35
aoclsparse_cmv (*C++ function*), 37
aoclsparse_convert_csr (*C++ function*), 27
aoclsparse_copy (*C++ function*), 12
aoclsparse_copy_mat_descr (*C++ function*), 7
aoclsparse_create_ccoo (*C++ function*), 10
aoclsparse_create_ccsc (*C++ function*), 11
aoclsparse_create_ccsr (*C++ function*), 7
aoclsparse_create_ctcsr (*C++ function*), 8
aoclsparse_create_dcoo (*C++ function*), 10
aoclsparse_create_dcsc (*C++ function*), 11
aoclsparse_create_dcsr (*C++ function*), 7
aoclsparse_create_dtcsr (*C++ function*), 8
aoclsparse_create_mat_descr (*C++ function*), 6
aoclsparse_create_scoo (*C++ function*), 10
aoclsparse_create_scsc (*C++ function*), 11
aoclsparse_create_scsr (*C++ function*), 7
aoclsparse_create_stcsr (*C++ function*), 8
aoclsparse_create_zcoo (*C++ function*), 10
aoclsparse_create_zcsc (*C++ function*), 11
aoclsparse_create_zcsr (*C++ function*), 7
aoclsparse_create_ztcsr (*C++ function*), 9
aoclsparse_csctr (*C++ function*), 31
aoclsparse_csctrs (*C++ function*), 32
aoclsparse_cset_value (*C++ function*), 13
aoclsparse_csp2md (*C++ function*), 59
aoclsparse_cspmmd (*C++ function*), 58
aoclsparse_csr2bsr_nnz (*C++ function*), 24
aoclsparse_csr2dia_ndiag (*C++ function*), 22
aoclsparse_csr2ell_width (*C++ function*), 21
aoclsparse_csymgs (*C++ function*), 76
aoclsparse_csyprd (*C++ function*), 65
aoclsparse_csyrkd (*C++ function*), 62
aoclsparse_ctrsm (*C++ function*), 49
aoclsparse_ctrsm_kid (*C++ function*), 51
aoclsparse_ctrsv (*C++ function*), 38
aoclsparse_ctrsv_kid (*C++ function*), 40
aoclsparse_ctrsv_strided (*C++ function*), 40
aoclsparse_cupdate_values (*C++ function*), 14
aoclsparse_dadd (*C++ function*), 57
aoclsparse_daxpyi (*C++ function*), 28
aoclsparse_dbsrmv (*C++ function*), 45
aoclsparse_dcsr2bsr (*C++ function*), 24
aoclsparse_dcsr2csc (*C++ function*), 25
aoclsparse_dcsr2dense (*C++ function*), 26
aoclsparse_dcsr2dia (*C++ function*), 23
aoclsparse_dcsr2ell (*C++ function*), 21
aoclsparse_dcsr2m (*C++ function*), 56
aoclsparse_dcsrmm (*C++ function*), 54
aoclsparse_dcsrmv (*C++ function*), 46
aoclsparse_dcsrsv (*C++ function*), 47
aoclsparse_ddiamv (*C++ function*), 44
aoclsparse_ddoti (*C++ function*), 31
aoclsparse_ddotmv (*C++ function*), 41
aoclsparse_dellmv (*C++ function*), 43
aoclsparse_destroy (*C++ function*), 12
aoclsparse_destroy_mat_descr (*C++ function*), 7
aoclsparse_dgthr (*C++ function*), 34
aoclsparse_dgthrs (*C++ function*), 36
aoclsparse_dgthrz (*C++ function*), 35
aoclsparse_diag_type (*C++ enum*), 84
aoclsparse_diag_type::aoclsparse_diag_type_non_unit (*C++ enumerator*), 84
aoclsparse_diag_type::aoclsparse_diag_type_unit (*C++ enumerator*), 84
aoclsparse_diag_type::aoclsparse_diag_type_zero (*C++ enumerator*), 84
aoclsparse_dilu_smoother (*C++ function*), 66
aoclsparse_dmv (*C++ function*), 37
aoclsparse_double_complex (*C++ struct*), 80
aoclsparse_double_complex::imag (*C++ member*), 81
aoclsparse_double_complex::real (*C++ member*),

81
aoclsparse_droti (*C++ function*), 33
aoclsparse_dsctr (*C++ function*), 31
aoclsparse_dsctrs (*C++ function*), 32
aoclsparse_dset_value (*C++ function*), 13
aoclsparse_dsorv (*C++ function*), 79
aoclsparse_dsp2md (*C++ function*), 59
aoclsparse_dspmmd (*C++ function*), 58
aoclsparse_dsymgs (*C++ function*), 76
aoclsparse_dsyprd (*C++ function*), 65
aoclsparse_dsyrkd (*C++ function*), 62
aoclsparse_dtrsm (*C++ function*), 49
aoclsparse_dtrsm_kid (*C++ function*), 51
aoclsparse_dtrsv (*C++ function*), 38
aoclsparse_dtrsv_kid (*C++ function*), 40
aoclsparse_dtrsv_strided (*C++ function*), 39
aoclsparse_dupdate_values (*C++ function*), 14
aoclsparse_export_ccoo (*C++ function*), 17
aoclsparse_export_ccsc (*C++ function*), 16
aoclsparse_export_ccsr (*C++ function*), 15
aoclsparse_export_dcoo (*C++ function*), 17
aoclsparse_export_dcsc (*C++ function*), 16
aoclsparse_export_dcsr (*C++ function*), 15
aoclsparse_export_scoo (*C++ function*), 17
aoclsparse_export_scsc (*C++ function*), 16
aoclsparse_export_scsr (*C++ function*), 15
aoclsparse_export_zcoo (*C++ function*), 17
aoclsparse_export_zcsc (*C++ function*), 16
aoclsparse_export_zcsr (*C++ function*), 15
aoclsparse_fill_mode (*C++ enum*), 84
aoclsparse_fill_mode::aoclsparse_fill_mode_lower (*C++ enumerator*), 84
aoclsparse_fill_mode::aoclsparse_fill_mode_upper (*C++ enumerator*), 84
aoclsparse_float_complex (*C++ struct*), 80
aoclsparse_float_complex::imag (*C++ member*), 80
aoclsparse_float_complex::real (*C++ member*), 80
aoclsparse_get_mat_diag_type (*C++ function*), 18
aoclsparse_get_mat_fill_mode (*C++ function*), 18
aoclsparse_get_mat_index_base (*C++ function*), 18
aoclsparse_get_mat_type (*C++ function*), 18
aoclsparse_get_version (*C++ function*), 19
aoclsparse_ilu_type (*C++ enum*), 85
aoclsparse_ilu_type::aoclsparse_ilu0 (*C++ enumerator*), 85
aoclsparse_ilu_type::aoclsparse_ilup (*C++ enumerator*), 85
aoclsparse_index_base (*C++ enum*), 83
aoclsparse_index_base::aoclsparse_index_base_one (*C++ enumerator*), 84
aoclsparse_index_base::aoclsparse_index_base_zero (*C++ enumerator*), 84
aoclsparse_int (*C++ type*), 80
aoclsparse_itsol_d_init (*C++ function*), 70
aoclsparse_itsol_d_rci_input (*C++ function*), 74
aoclsparse_itsol_d_rci_solve (*C++ function*), 75
aoclsparse_itsol_d_solve (*C++ function*), 70
aoclsparse_itsol_destroy (*C++ function*), 70
aoclsparse_itsol_handle (*C++ type*), 85
aoclsparse_itsol_handle_prn_options (*C++ function*), 74
aoclsparse_itsol_option_set (*C++ function*), 72
aoclsparse_itsol_rci_job (*C++ enum*), 69
aoclsparse_itsol_rci_job::aoclsparse_rci_interrupt (*C++ enumerator*), 69
aoclsparse_itsol_rci_job::aoclsparse_rci_mv (*C++ enumerator*), 69
aoclsparse_itsol_rci_job::aoclsparse_rci_precond (*C++ enumerator*), 69
aoclsparse_itsol_rci_job::aoclsparse_rci_start (*C++ enumerator*), 69
aoclsparse_itsol_rci_job::aoclsparse_rci_stop (*C++ enumerator*), 69
aoclsparse_itsol_rci_job::aoclsparse_rci_stopping_criterio (*C++ enumerator*), 69
aoclsparse_itsol_s_init (*C++ function*), 70
aoclsparse_itsol_s_rci_input (*C++ function*), 74
aoclsparse_itsol_s_rci_solve (*C++ function*), 75
aoclsparse_itsol_s_solve (*C++ function*), 70
aoclsparse_mat_descr (*C++ type*), 81
aoclsparse_matrix (*C++ type*), 81
aoclsparse_matrix_data_type (*C++ enum*), 83
aoclsparse_matrix_data_type::aoclsparse_cmat (*C++ enumerator*), 83
aoclsparse_matrix_data_type::aoclsparse_dmat (*C++ enumerator*), 83
aoclsparse_matrix_data_type::aoclsparse_smat (*C++ enumerator*), 83
aoclsparse_matrix_data_type::aoclsparse_zmat (*C++ enumerator*), 83
aoclsparse_matrix_type (*C++ enum*), 83
aoclsparse_matrix_type::aoclsparse_matrix_type_general (*C++ enumerator*), 83
aoclsparse_matrix_type::aoclsparse_matrix_type_hermitian (*C++ enumerator*), 83
aoclsparse_matrix_type::aoclsparse_matrix_type_symmetric (*C++ enumerator*), 83
aoclsparse_matrix_type::aoclsparse_matrix_type_triangular (*C++ enumerator*), 83
aoclsparse_memory_usage (*C++ enum*), 86
aoclsparse_memory_usage::aoclsparse_memory_usage_minimal (*C++ enumerator*), 86
aoclsparse_memory_usage::aoclsparse_memory_usage_unrestric (*C++ enumerator*), 86
aoclsparse_operation (*C++ enum*), 85
aoclsparse_operation::aoclsparse_operation_conjugate_trans

(C++ *enumerator*), 85
aoclsparse_operation::aoclsparse_operation_none (C++ *enumerator*), 85
aoclsparse_operation::aoclsparse_operation_transpose (C++ *enumerator*), 85
aoclsparse_optimize (C++ *function*), 3
aoclsparse_order (C++ *enum*), 85
aoclsparse_order::aoclsparse_order_column (C++ *enumerator*), 85
aoclsparse_order::aoclsparse_order_row (C++ *enumerator*), 85
aoclsparse_order_mat (C++ *function*), 13
aoclsparse_request (C++ *enum*), 86
aoclsparse_request::aoclsparse_stage_finalize (C++ *enumerator*), 86
aoclsparse_request::aoclsparse_stage_full_computation (C++ *enumerator*), 86
aoclsparse_request::aoclsparse_stage_nnz_count (C++ *enumerator*), 86
aoclsparse_sadd (C++ *function*), 57
aoclsparse_saxpyi (C++ *function*), 28
aoclsparse_sbsrmv (C++ *function*), 45
aoclsparse_scsr2bsr (C++ *function*), 24
aoclsparse_scsr2csc (C++ *function*), 25
aoclsparse_scsr2dense (C++ *function*), 26
aoclsparse_scsr2dia (C++ *function*), 23
aoclsparse_scsr2ell (C++ *function*), 21
aoclsparse_scsr2m (C++ *function*), 56
aoclsparse_scsrmm (C++ *function*), 54
aoclsparse_scsrmv (C++ *function*), 46
aoclsparse_scsrsv (C++ *function*), 47
aoclsparse_sdiamv (C++ *function*), 44
aoclsparse_sdoti (C++ *function*), 31
aoclsparse_sdotmv (C++ *function*), 41
aoclsparse_sellmv (C++ *function*), 43
aoclsparse_set_2m_hint (C++ *function*), 4
aoclsparse_set_dotmv_hint (C++ *function*), 4
aoclsparse_set_lu_smoother_hint (C++ *function*), 4
aoclsparse_set_mat_diag_type (C++ *function*), 19
aoclsparse_set_mat_fill_mode (C++ *function*), 19
aoclsparse_set_mat_index_base (C++ *function*), 20
aoclsparse_set_mat_type (C++ *function*), 20
aoclsparse_set_memory_hint (C++ *function*), 6
aoclsparse_set_mm_hint (C++ *function*), 4
aoclsparse_set_mv_hint (C++ *function*), 4
aoclsparse_set_sm_hint (C++ *function*), 5
aoclsparse_set_sorv_hint (C++ *function*), 5
aoclsparse_set_sv_hint (C++ *function*), 4
aoclsparse_set_symgs_hint (C++ *function*), 4
aoclsparse_sgthr (C++ *function*), 34
aoclsparse_sgthrs (C++ *function*), 36
aoclsparse_sgthrz (C++ *function*), 35
aoclsparse_silu_smoother (C++ *function*), 66
aoclsparse_smv (C++ *function*), 37
aoclsparse_sor_type (C++ *enum*), 86
aoclsparse_sor_type::aoclsparse_sor_backward (C++ *enumerator*), 86
aoclsparse_sor_type::aoclsparse_sor_forward (C++ *enumerator*), 86
aoclsparse_sor_type::aoclsparse_sor_symmetric (C++ *enumerator*), 86
aoclsparse_sp2m (C++ *function*), 52
aoclsparse_spmm (C++ *function*), 54
aoclsparse_sroti (C++ *function*), 33
aoclsparse_ssctr (C++ *function*), 31
aoclsparse_ssctrs (C++ *function*), 32
aoclsparse_sset_value (C++ *function*), 13
aoclsparse_ssorv (C++ *function*), 79
aoclsparse_ssp2md (C++ *function*), 59
aoclsparse_sspmmd (C++ *function*), 58
aoclsparse_ssymgs (C++ *function*), 76
aoclsparse_ssyprd (C++ *function*), 65
aoclsparse_ssyrkd (C++ *function*), 62
aoclsparse_status (C++ *enum*), 81
aoclsparse_status::aoclsparse_status_internal_error (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_invalid_index_value (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_invalid_kid (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_invalid_operation (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_invalid_pointer (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_invalid_size (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_invalid_value (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_maxit (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_memory_error (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_not_implemented (C++ *enumerator*), 81
aoclsparse_status::aoclsparse_status_numerical_error (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_success (C++ *enumerator*), 81
aoclsparse_status::aoclsparse_status_unsorted_input (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_user_stop (C++ *enumerator*), 82
aoclsparse_status::aoclsparse_status_wrong_type (C++ *enumerator*), 82
aoclsparse_strsm (C++ *function*), 49
aoclsparse_strsm_kid (C++ *function*), 51
aoclsparse_strsv (C++ *function*), 38

aoclsparse_strsv_kid (*C++ function*), 40
aoclsparse_strsv_strided (*C++ function*), 39
aoclsparse_supdate_values (*C++ function*), 14
aoclsparse_sypr (*C++ function*), 63
aoclsparse_syrk (*C++ function*), 60
aoclsparse_zadd (*C++ function*), 57
aoclsparse_zaxpyi (*C++ function*), 28
aoclsparse_zcsr2csc (*C++ function*), 25
aoclsparse_zcsr2dense (*C++ function*), 27
aoclsparse_zcsrmm (*C++ function*), 55
aoclsparse_zdotci (*C++ function*), 29
aoclsparse_zdotmv (*C++ function*), 41
aoclsparse_zdotui (*C++ function*), 30
aoclsparse_zgthr (*C++ function*), 34
aoclsparse_zgthrs (*C++ function*), 36
aoclsparse_zgthrz (*C++ function*), 35
aoclsparse_zmv (*C++ function*), 37
aoclsparse_zsctr (*C++ function*), 31
aoclsparse_zsctrs (*C++ function*), 32
aoclsparse_zset_value (*C++ function*), 13
aoclsparse_zsp2md (*C++ function*), 59
aoclsparse_zspmmd (*C++ function*), 58
aoclsparse_zsymgs (*C++ function*), 76
aoclsparse_zsyprd (*C++ function*), 65
aoclsparse_zsyrkd (*C++ function*), 62
aoclsparse_ztrsm (*C++ function*), 49
aoclsparse_ztrsm_kid (*C++ function*), 51
aoclsparse_ztrsv (*C++ function*), 38
aoclsparse_ztrsv_kid (*C++ function*), 40
aoclsparse_ztrsv_strided (*C++ function*), 40
aoclsparse_zupdate_values (*C++ function*), 14