Intel® oneMKL for PETSc Developers Conference 2023

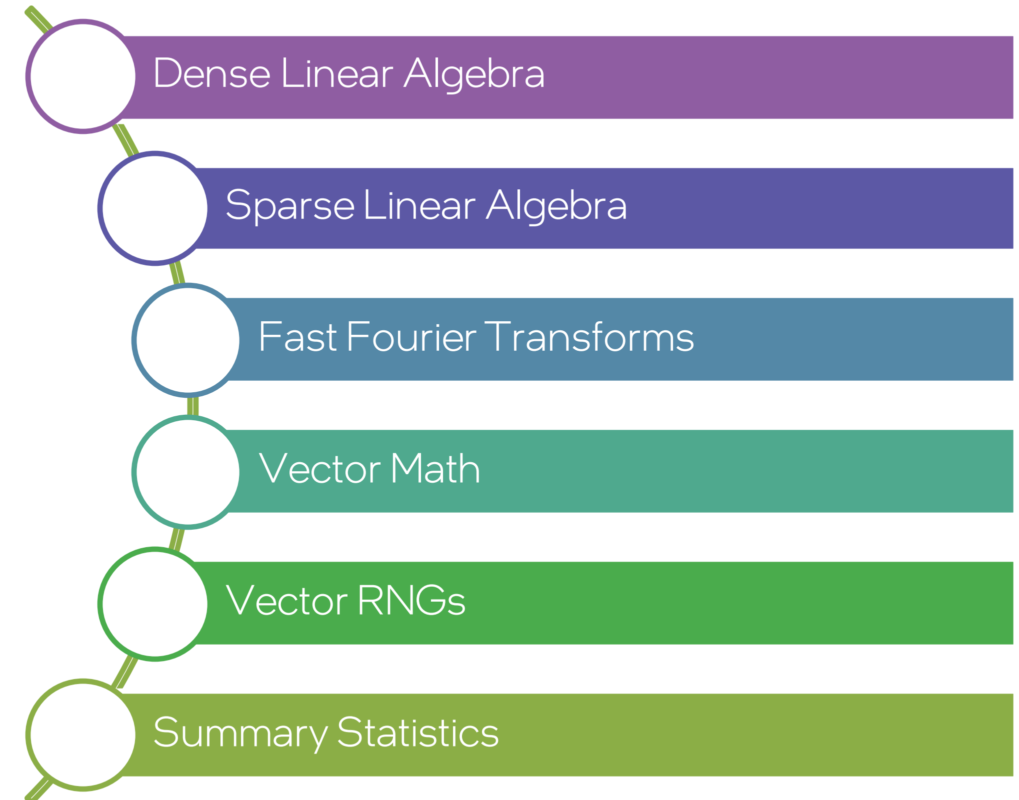
Spencer Patty

06/06/2023

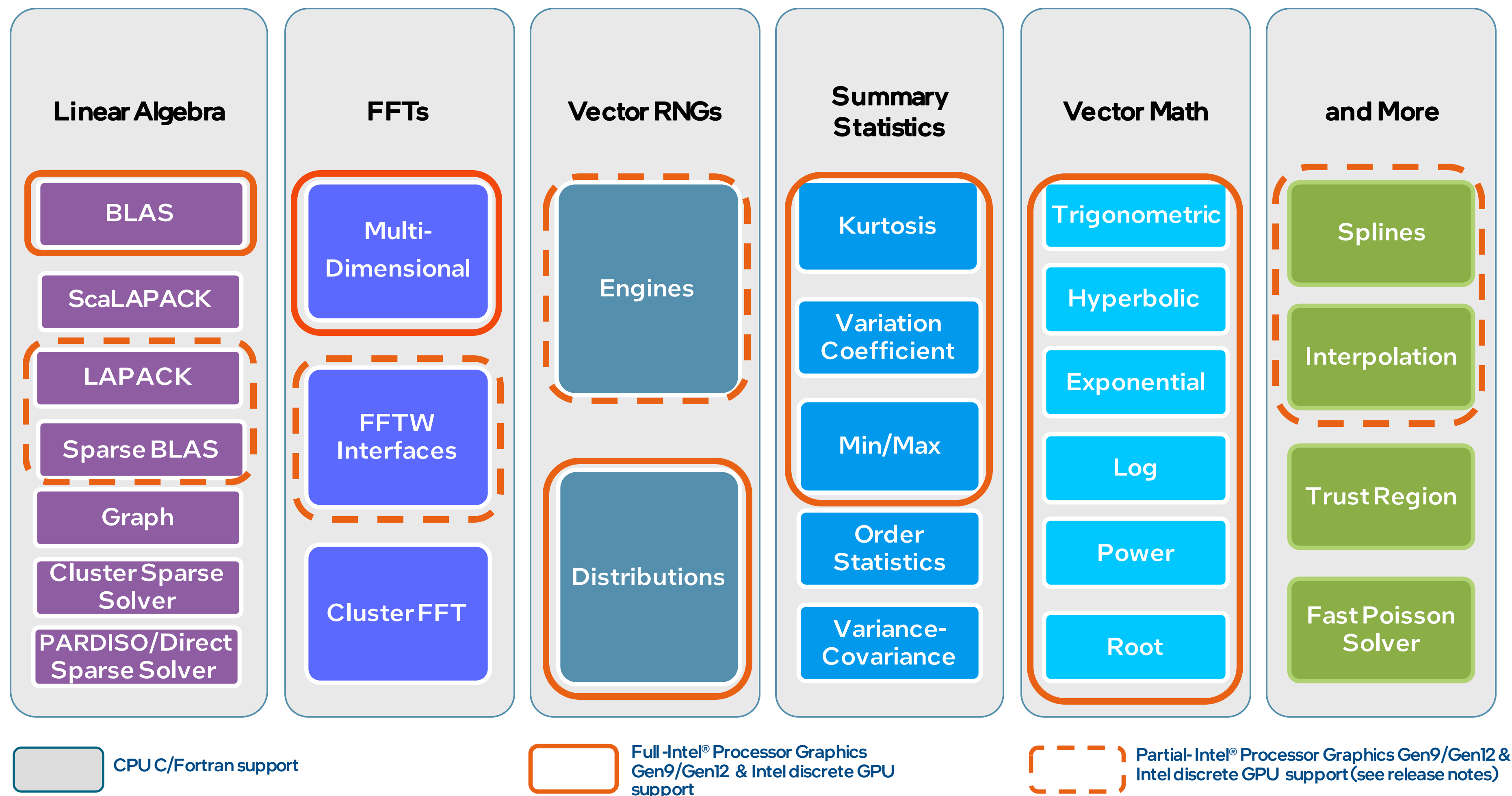
Intel® oneAPI Math Kernel Library (oneMKL)

- Language support for SYCL and Intel® C and Fortran compilers
- Available at no cost and royalty-free
- Great performance with minimal effort
- Full support for CPUs and select support for Intel processor and discrete graphics
- Speeds computations for scientific, engineering, and financial applications by providing highly optimized, threaded, and vectorized math functions
- Provides key functionality for dense and sparse linear algebra (BLAS, LAPACK, sparse direct solvers), FFTs, vector math, summary statistics, splines, and more
- Dispatches optimized code for each processor automatically without the need to branch code
- Optimized for single-core vectorization and cache utilization
- Automatic parallelism for multicore CPUs, GPUs and scales from core to clusters

Intel® oneAPI Math Kernel Library offers



What's Inside Intel® oneAPI Math Kernel Library (oneMKL)



What's New for Intel® oneAPI Math Kernel Library (oneMKL) 2023.1

Better GPU Performance + Intel® Data Center GPU Max Series & 4th Gen Intel® Xeon® Scalable processors Support

oneMKL has optimized support for Intel's upcoming portfolio of CPU and GPU architectures

4th Gen Intel® Xeon® Scalable Processors with Intel® Advanced Matrix Extensions, Quick assist Technology, Intel® AVX-512, bfloat16, and more built-in accelerators

Intel® Xeon® Max Series CPUs with high-bandwidth memory

Intel® Data Center GPUs, including Flex Series with hardware AV1 encode and Max with datatype flexibility, Intel® Xe Matrix Extensions, vector engine, XE-Link, and other features

Intel® Math Kernel Library (MKL) changed it's name to Intel® oneAPI Math Kernel Library (oneMKL) in April 2020 with Initial release oneMKL 2021.1.

Since then, we have introduced support for fundamental math operations on Intel® GPUs and continued expanding support for them on latest Intel® CPUs.

The latest release is oneMKL 2023.1 (released in early April 2023) which has optimized support for latest Intel products, with more to come in future releases.

Release Notes 2021.x (2021.1 - 2021.4)
Release Notes 2022.x (2022.0 – 2022.2)
Release Notes 2023.x (2023.0 – latest)

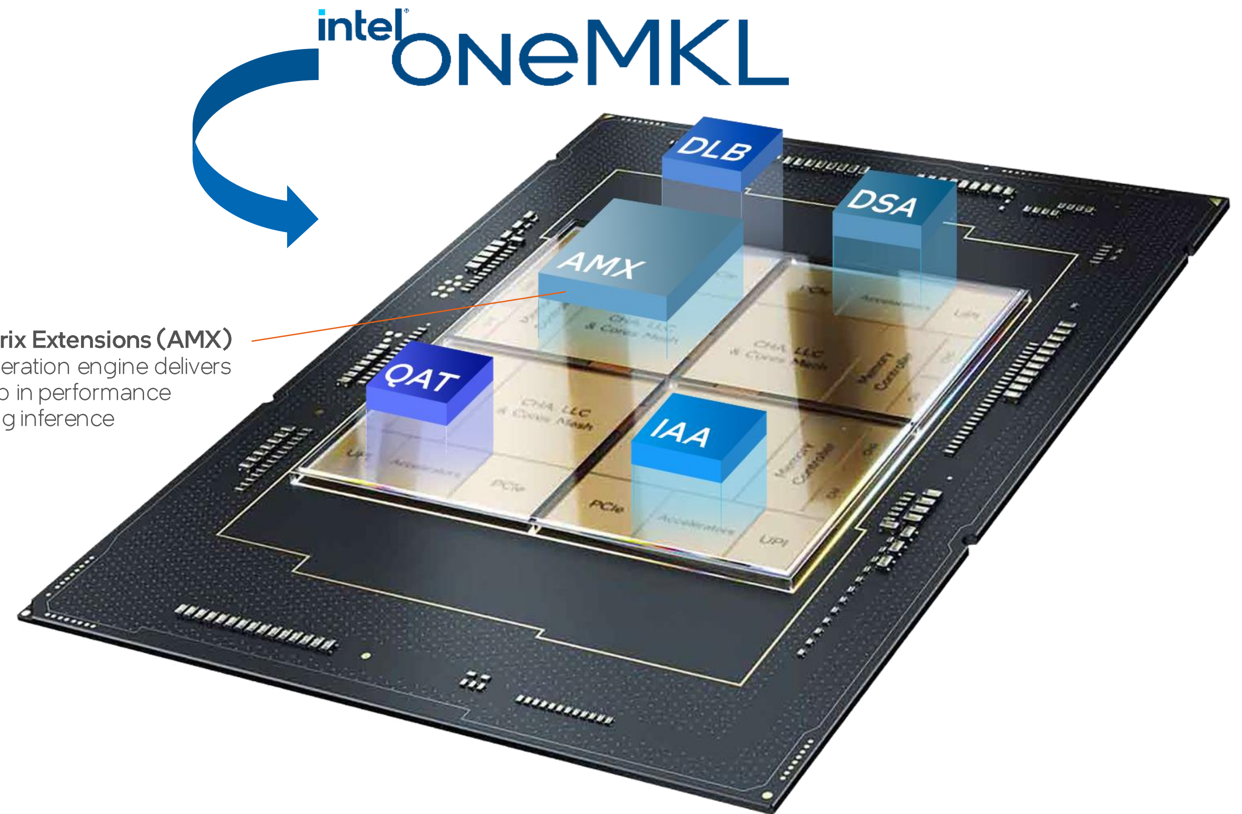
oneMKL on 4th Gen Intel® Xeon® Scalable processors

Maximize performance with oneMKL, unleashing the power of built-in accelerators

- The Intel® oneAPI Math Kernel Library (oneMKL) leverages Intel® AMX-Advanced Matrix eXtensions to optimize matrix computations for the BF16 and INT8 data types.

- oneMKL also leverages Intel® AVX-512-Advanced Vector Extensions for the FP16 data type on 4th Gen Intel® Xeon® Scalable processors.

- Most oneMKL memory-bound dense and sparse linear algebra (BLAS, LAPACK, sparse direct solvers), FFT, vector math, vector RNG, summary statistics, or spline computations, directly benefit from the onboard High Bandwidth Memory (HBM).



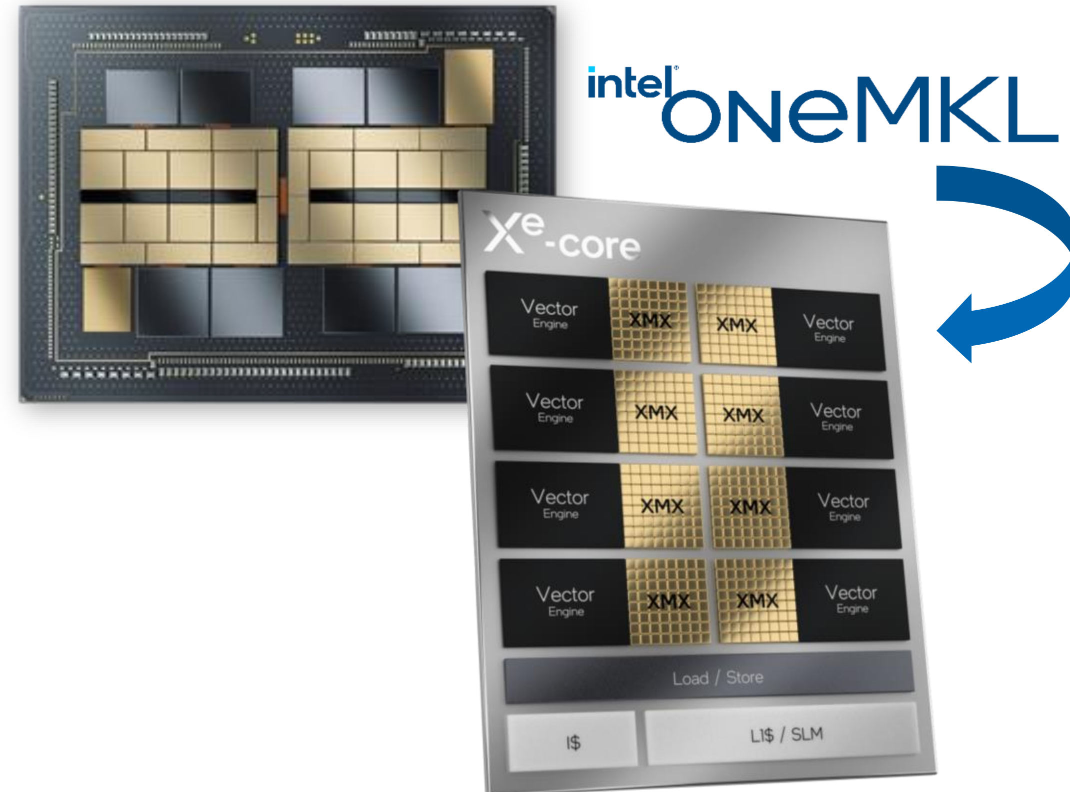
Advanced Matrix Extensions (AMX)
Built-in AI acceleration engine delivers a significant leap in performance for deep learning inference and training.

4th Gen Intel® Xeon® Scalable Processors with Intel® Advanced Matrix Extensions, Quick assist Technology, Intel® AVX-512, bfloat16, and more built-in accelerators

oneMKL on Intel® Data Center GPU Max Series

Breakthrough Performance for HPC and AI

- The Intel® oneAPI Math Kernel Library (oneMKL) leverages Intel® Xe Matrix Extensions (Intel® XMX) to optimize matrix computations for TF32, FP16, BF16 and INT8 data types on Intel® Data Center GPU Max Series (codenamed Ponte Vecchio).
- oneMKL provides a variety of dense and sparse linear algebra (BLAS, LAPACK, sparse BLAS), FFT, vector math, vector RNG, summary statistics, and spline interfaces both for the SYCL and C/Fortran OpenMP* offload programming models to enable applications targeting Intel® Data Center GPUs.



Intel® Data Center GPUs with hardware AV1 encode and Max with datatype flexibility, Intel® Xe Matrix Extensions, vector engine, XE-Link, and other features

How to take advantage of Intel® GPUs through oneMKL?

oneMKL library provides two methods for enqueuing work on GPUs

1. **SYCL C++ APIs** – new set of APIs with SYCL C++ for heterogenous data parallelism on CPUs or GPUs
2. **C/Fortran OpenMP Offload APIs** – extensions of several existing C/Fortran APIs in oneMKL that offloads computations to GPU through OpenMP 5.0 or OpenMP 5.1 preprocessing directives

For PETSc, our main focus has been performance through C OpenMP offload APIs:

- **mkl_sparse_?_mv()** – sparse matrix – dense vector multiplication, $y = A * x$
- **mkl_sparse_sp2m()** – sparse matrix – sparse matrix multiplication, $C = A * B$

Questions for Discussion:

- Who has used the Intel® oneAPI Math Kernel Library (oneMKL)?
- What would you like to hear about?
- What do you think would be most useful to you in oneMKL?

Some possibilities for continued discussion topic:

- SYCL C++ language examples
- C OpenMP Offload examples
- oneMKL Inspector-Executor Sparse BLAS routines in C (with OpenMP Offload)
- oneMKL Sparse BLAS routines in SYCL C++

Resources

- Intel(R) oneAPI Math Kernel Library Developer References (Documentation)
  - SYCL C++: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-mkl-dpcpp-developer-reference/top.html
  - C: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html
  - Fortran: https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-fortran/top.html
- oneAPI Main Page: https://www.oneapi.com/
- oneAPI Samples: https://github.com/oneapi-src/oneAPI-samples
- Latest release of oneMKL Spec (currently oneAPI-v1.2-rev-1): https://spec.oneapi.com/versions/latest/elements/oneMKL/source/index.html
- GitHub for oneAPI Specification: https://github.com/oneapi-src/oneAPI-spec
- GitHub for oneAPI Math Special Interest Group: https://github.com/oneapi-src/oneAPI-tab
- GitHub for open source oneMKL interfaces: https://github.com/oneapi-src/oneMKL

Backup

Backup – Overview of C/Fortran Inspector-Executor Sparse BLAS APIs for CPU

+ C Openmp Offload extensions to GPU

Inspector-Executor Sparse BLAS API for CPU (C/Fortran)

Motivation:

Dividing sparse function calls into two steps (optimization and execution), Inspector-Executor Sparse BLAS (IE Sparse BLAS) enables **best performance for many repeated calls** to Sparse BLAS operations.

Design Details:

- An opaque matrix handle (type *sparse_matrix_t*) is introduced to house the various arrays and scalars that compose the sparse matrix format.
  - Sparse matrix as an operator is abstracted from the specific matrix format used to represent it.
  - Gives room for library implementation to create and store optimized data that has same lifetime as sparse matrix object and whose creation can be amortized over several subsequent execution calls.
- Users set hints about sparse matrix operations for which to be optimized.
- One call to the optimization routine will:
  - Use heuristics to determine best internal format for user matrix based on it's sparsity pattern and the operations desired.
  - The degree of optimization is chosen based on "expected number of calls" to function provided in hints.
  - As needed, library may convert matrix to one of the high-performance internal formats.
  - Use strategies to collect balancing information.
  - Assign the best suitable computational kernel based on matrix portrait (sparsity pattern).
- Goal: User can omit the optimization routine and get on par or better performance than the Sparse BLAS NIST-Style APIs (currently deprecated since oneMKL 2019.0).

Inspector-Executor Sparse BLAS APIs for CPU

$$\text{op}(A) = \begin{cases} A \\ A^T \\ A^H \end{cases}$$

- Inspector: (An example process for CSR SpMV)
  - mkl_sparse_?_create_csr(sparse_matrix_t *A, …)
    - supported input matrix types: coo, csr, csc, bsr
  - mkl_sparse_set_mv_hint(A, operation, descr, expected_calls)
    - hints available for: mv, sv, mm, sm, dotmv
    - operation (op) = non-transpose, transpose, conjugate transpose
    - descr = matrix sparsity type (Gen, Sym, Tri, Diagonal…) and other details about the matrix being used
    - expected_calls = estimate of number of calls to executor API, will affect choice of internal matrix representation
  - mkl_sparse_memory_hint(A, memory_policy)
    - choose allowed internal memory usage for optimization routines: aggressive(default, can change internal matrix storage format for optimizations) or none(can only use auxiliary structures like workload balancing for optimizations)
  - mkl_sparse_optimize(A)
    - Takes all provided hints, converts matrix A internally to optimal storage format based on hints, and selects optimal kernels for desired operations
- Other:
  - mkl_sparse_order(A)
    - Reorders the internal column/row indices (and values) subordinate to the rows_start/rows_end or cols_start/cols_end as appropriate by increasing index
  - mkl_sparse_?_export_<format>(A,…)
    - Where <format> = csr, bsr, etc. Allows users to get access to the internal matrix format arrays that may have been allocated internal to MKL library calls (Level 3 APIs). Any such arrays created by mkl library will be deallocated on call to mkl_sparse_destroy().

- Executor: (α, β, d scalars, x, y dense vectors, X, Y dense matrices, A, B, C sparse matrices)

Executor API*	Math Operation
mkl_sparse_?_mv(…)	$y = \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y$
mkl_sparse_?_mm(…)	$Y = \alpha \cdot \text{op}(A) \cdot X + \beta \cdot Y$
mkl_sparse_?_dotmv(…)	$y = \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y,$ $d = \text{dot}(x, y)$
mkl_sparse_?_trsv(…)	Solve y: $\text{op}(A) \cdot y = \alpha \cdot x$
mkl_sparse_?_trsm(…)	Solve Y: $\text{op}(A) \cdot Y = \alpha \cdot X$
mkl_sparse_?_add(…)	$C = \alpha \cdot \text{op}(A) + B$
mkl_sparse_spmm(…)**	$C = \text{op}(A) \cdot B$
mkl_sparse_sp2m(…)**	$C = \text{op}^1(A) * \text{op}^2(B)$
mkl_sparse_syrk(…)**	$C = \text{op}(A) \cdot \text{op}(A)^H \quad (= A \cdot \text{op}(A) \text{ or } \text{op}(A) \cdot A)$
mkl_sparse_sypr(…)**	$C = \text{op}(A) \cdot B \cdot \text{op}(A)^H \quad (= A \cdot B \cdot \text{op}(A) \text{ or } \text{op}(A) \cdot B \cdot A)$ where B is a Sym/Herm matrix

* IE Sparse BLAS also includes some inspector/executor APIs for some preconditioners like Symmetric Gauss-Seidel (mkl_sparse_?_symgs), an LU smoother (mkl_sparse_?_lu_smoother) and SOR (mkl_sparse_?_sorv) along with their corresponding hints.

** Can also get output as dense matrix C, adding "d" to end of API names

Inspector-Executor Sparse BLAS Preconditioner APIs for CPU

(α, ω scalars, x, y, r, b dense vectors, X, Y dense matrices, $A = L + D + U$ sparse matrices)

$$\text{op}(A) = \begin{cases} A \\ A^T \\ A^H \end{cases}$$

Executor API*	Description	Math Operation
mkl_sparse_?_trsv(...)	Solves lower or upper sparse triangular system	Solve y: $\text{op}(T) \cdot y = \alpha \cdot x$ where T is triangular ($L + D$ or $D + U$)
mkl_sparse_?_trsm(...)	Solves lower or upper sparse triangular system with multiple right-hand sides	Solve Y: $\text{op}(T) \cdot Y = \alpha \cdot X$ where T is triangular ($L + D$ or $D + U$)
mkl_sparse_?_symgs(...)	Applies single step of Symmetric Gauss-Seidel preconditioner (applies $M = (D + L) \cdot D^{-1} \cdot (D + U)$ preconditioner)	Solve x^1: $(L + D) \cdot x^1 = b - \alpha \cdot U \cdot x^0$ Solve x: $(U + D) \cdot x = b - L \cdot x^1$
mkl_sparse_?_symgs_mv(...)	Applies single step of symmetric Gauss-seidel preconditioner fused with an extra sparse MV operation (you get back both x and $A \cdot x$)	Solve x^1: $(L + D) \cdot x^1 = b - \alpha \cdot U \cdot x^0$ Solve x: $(U + D) \cdot x = b - L \cdot x^1$ Update: $y = A \cdot x$
mkl_sparse_?_lu_smoother(...)	Applies Symmetric Gauss-Seidel like preconditioner corresponding to the approximate matrix decomposition $A \sim (L + D) \cdot E \cdot (U + D)$ for system $A \cdot x = b$ where E is an approximate inverse of diagonal (using exact inverse reduces to Gauss-Seidel). (This is most useful for BSR matrix format where D^{-1} is not just a reciprocal)	Compute: $r = b - A \cdot x$ Solve dx: $(L + D) \cdot E \cdot (U + D) \cdot dx = r$ Update: $x \leftarrow x + dx$
mkl_sparse_?_sorv()	Applies single step of (forward/backward or symmetric) Successive-Over-relaxation (SOR) preconditioner with single right-hand side for system $A \cdot x = b$ and with $0 < \omega < 2$. (Note symmetric with $\omega = 1$ is the same as Gauss-Seidel preconditioner)	FWD solve x^1: $(\omega L + D) \cdot x^1 = ((1 - \omega)D - \omega U) \cdot x^0 + \omega \cdot b$ BWD solve x^1: $(\omega U + D) \cdot x^1 = ((1 - \omega)D - \omega L) \cdot x^0 + \omega \cdot b$ SYM: Applies $M = \frac{\omega}{2-\omega}\left(\frac{1}{\omega}D + L\right)D^{-1}\left(\frac{1}{\omega}D + L\right)^T$ precondtioner
mkl_sparse_?_sorm()**	Applies single step of (forward/backward or symmetric) successive over-relaxation (SOR) preconditioner with multiple right-hand sides for system $A \cdot X = Y$.	Same as SORV but with dense matrix Y instead of b and dense matrix X instead of x.

** API may be added in future, TBD

Inspector-Executor Sparse BLAS Multi-stage Algorithms for CPU

APIs that support multi-stage algorithms:

- mkl_sparse_sp2m(…, sparse_request_t request, …)
  - supported input matrix types: csr, csc, bsr
- mkl_sparse_sypr(…, sparse_request_t request, …)
  - supported input matrix types: csr, bsr

sparse_request_t	Description
SPARSE_STAGE_ NNZ_COUNT	Allocates and computes on **rows_start/rows_end** or **cols_start/cols_end** as appropriate. After this stage, by calling "*mkl_sparse_?_export_<format>*", you can obtain the number of non-zeros in the output matrix and calculate the amount of memory required for output matrix
SPARSE_STAGE_ FINALIZE_MULT_NO_VAL	Allocates (if needed) and fills row/column indices (ie matrix structure) but not values, when called after SPARSE_STAGE_NNZ_COUNT stage
SPARSE_STAGE_ FINALIZE_MULT	Allocates (if needed) and fills row/column indices and values array. If row/columns indices were previously filled, it only does the values.
SPARSE_STAGE_ FULL_MULT_NO_VAL	Single-step: allocates and computes the matrix structure, but not the values
SPARSE_STAGE_ FULL_MULT	Single-step: allocates and computes the entire output (matrix structure and values)

Example: symbolic/numeric multi-stage

```
// First Stage: estimate nnz count
sparse_matrix_t csrC = NULL;
status = mkl_sparse_sp2m(opA, descrA, csrA, opB, descrB,
csrB, SPARSE_STAGE_NNZ_COUNT, &csrC);

// optional calculation of nnz in the output
// matrix for getting a memory estimate
status = mkl_sparse_?_export_csr(csrC, &indexing, &nrows,
&ncols, &rows_start, &rows_end, &col_indx, &values);

MKL_INT nnz = rows_end[nrows-1] - rows_start[0];

// Second Stage: fill columns/values
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB,
csrB, SPARSE_STAGE_FINALIZE_MULT, &csrC);

// get access to C matrix arrays if desired
status = mkl_sparse_?_export_csr(csrC, &indexing, &nrows,
&ncols, &rows_start, &rows_end, &col_indx, &values);
```

Example: Single-stage

```
// status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB,
csrB, SPARSE_STAGE_FULL_MULT, &csrC);

// get access to C matrix arrays if desired
status = mkl_sparse_?_export_csr (csrC, &indexing, &nrows,
&ncols, &rows_start, &rows_end, &col_indx, &values);
```

Sparse BLAS C OpenMP Offload APIs for GPU

Currently supported precisions

$$? = \begin{cases} s, & \text{real float} \\ d, & \text{real double} \end{cases}$$

Uses oneMKL Inspector-Executor Sparse BLAS C APIs for GPU offload via OpenMP:

Inspector APIs with GPU offload:

- mkl_sparse_?_create_csr() – create sparse matrix handle
- mkl_sparse_?_export_csr() – access sparse arrays in handle
- mkl_sparse_destroy() – destroy sparse matrix handle
- mkl_sparse_set_mv_hint() – add hint to optimize SpMV
- mkl_sparse_set_sv_hint() – add hint to optimize SpTRSV
- mkl_sparse_optimize() – perform optimizations based on hints

Executor APIs with GPU offload:

- mkl_sparse_?_mv() – sparse * dense vector multiplication
- mkl_sparse_?_mm() – sparse * dense matrix multiplication
- mkl_sparse_?_trsv() – sparse triangular solve
- mkl_sparse_sp2m() – sparse * sparse matrix multiplication

Other APIs with GPU offload:

- mkl_sparse_order() – sort sparse matrix

```
// Example of OpenMP 5.1 dispatch construct

#pragma omp declare variant (MKL_SPBLAS_VARIANT_NAME(s_mv)) \
    match(construct={dispatch}, device={arch(gen)}) \
    append_args(interop(prefer_type("sycl", "level_zero"),targetsync)) \
    adjust_args(need_device_ptr:x,y)
    sparse_status_t mkl_sparse_s_mv ( const sparse_operation_t  operation,
                                      const float               alpha,
                                      const sparse_matrix_t     A,
                                      const struct matrix_descr descr,
                                      const float               *x,
                                      const float               beta,
                                      float                     *y );

// Example of OpenMP 5.0 dispatch construct

 #pragma omp declare variant (MKL_SPBLAS_VARIANT_NAME(s_trsv))\
    match(construct={target variant dispatch}, device={arch(gen)})
    sparse_status_t mkl_sparse_d_trsv ( const sparse_operation_t  operation,
                                        const double              alpha,
                                        const sparse_matrix_t     A,
                                        const struct matrix_descr descr,
                                        const double              *x,
                                        double                    *y );
```

Limitations:

- For Sparse BLAS computations, a sequence of calls "*create a CSR matrix handle* -> *compute* -> *destroy the CSR matrix handle*" should be done in a single "target data" region so that the data is available throughout the computations on the offload device.
  - Alternative is to use the *omp_target_alloc_device* or *omp_target_alloc_shared* or *omp_target_alloc_host* directly for creating gpu-aware array allocations directly.
- Currently, only the execute (compute functionality) APIs can be used asynchronously.

Backup – Overview of SYCL C++ Sparse BLAS APIs for CPU/GPU

SYCL C++ Sparse BLAS strategy with an analysis-execution design (for CPU/GPU)

Motivation:

- Sparse BLAS operations are often used in algorithms where the same kernel operation is applied multiple times until some stopping criteria. Example: iterative solvers like conjugate gradient, preconditioners, power method for eigen-solvers, etc…
- An opaque matrix handle is introduced (type matrix_handle_t) to house the components of the sparse matrix format as well as provide a place to store any library generated data related to that matrix.
- Changing the internal format of data and/or analyzing the sparsity structure helps to exploit better optimized kernels and, in some cases like TRSV, enable a greater level of parallelism.

Analysis-Execution Strategy:

- **Initialization Stage** – create the opaque matrix handle and provide it user data
- **Analysis Stage** (Preprocessing step) – prepare internal structures, data for given matrices and operations. Only called once per operation to be optimized. Can be skipped at cost of possibly worse performance in execution stage.
- **Execution Stage** – performs the operation, can be called once or many times
- **Release Stage** – clean up matrix handle and release internally allocated data

Sparse operations supported by SYCL C++ Sparse BLAS APIs

There are three main Sparse BLAS operations supported, and some other helpers:

1) Sparse matrix – dense vector/dense matrix multiplication:

   a) **(ge/sy/tr)mv, gemvdot,** sparse matrix-dense vector product (+an optional dot product). A is a sparse matrix, x, y are dense vectors, $\alpha, \beta, \mathrm{d}$ are scalars:

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

   b) **gemm**, general sparse matrix-dense matrix multiplication: A is a sparse matrix, X, Y are dense matrices(row-major or column-major format), α, β are scalars:

$$Y = \alpha \cdot \mathrm{op}(A) \cdot X + \beta \cdot Y$$

2) Sparse matrix triangular solve:

   a) **trsv**, A is a sparse triangular matrix, x, y are dense vectors. Solve for y:

$$\mathrm{op}(A) \cdot y = x$$

   b) **(to be added) trsm**, A is a sparse triangular matrix, X, Y are dense matrices (row-major or column-major format). Solve for Y:

$$\mathrm{op}(A) \cdot Y = X$$

3) Sparse matrix – sparse matrix multiplication: **matmat**, $A, \mathrm{B}, \mathrm{C}$ are sparse matrices:

$$C = \mathrm{op}(A) \cdot \mathrm{op}(B)$$

matrix type	full name
ge	General structure
sy	Symmetric structure
he	Hermitian structure
tr	Triangular structure

$$\text{where } \mathrm{op}(A) = \begin{cases} A \\ A^T \\ A^H \end{cases}$$

4) Other sparse helper operations:

   a) **omatcopy**, general sparse matrix copy or sparse matrix transpose to a new handle: A, B are sparse matrices:

$$B = \mathrm{op}(A)$$

   b) **sort_matrix**, general sparse matrix sort.

   c) **update_diagonal_data,** change out diagonal values in the matrix handle and all internal optimizations

Sparse BLAS Level 2 and Level 3 SYCL C++ APIs

State-Management Routines

- sparse::init_matrix_handle
- sparse::release_matrix_handle
- sparse::set_csr_data
- sparse::set_matrix_property

- sparse::init_matmat_descr
- sparse::set_matmat_data
- sparse::get_matmat_data
- sparse::release_matmat_descr

Analysis Routines

- sparse::optimize_gemv
- sparse::optimize_trmv
- sparse::optimize_trsv

Helper Routines

- sparse::omatcopy
- sparse::sort_matrix
- sparse::update_diagonal_data

Execution Routines

- sparse::gemv
- sparse::trmv*
- sparse::symv*
- sparse::gemvdot*
- sparse::trsv*
- sparse::gemm*
- sparse::matmat

*Execution and Helper APIs support all options for the $op(A)$, but for these APIs we only have implementations for "non-transpose" currently, so a oneMKL runtime exception "not implemented" is thrown for other cases.

Sparse BLAS GEMV SYCL C++ API

GEMV: A a general sparse matrix, x, y dense vectors, α, β scalars:

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

```
namespace oneapi::mkl::sparse {

sycl::event optimize_gemv(sycl::queue &queue,
                          oneapi::mkl::transpose transpose_flag,
                          oneapi::mkl::sparse::matrix_handle_t handle,
                          const std::vector<sycl::event> &dependencies = {});

void gemv(sycl::queue &queue,
          oneapi::mkl::transpose transpose_flag,
          const float alpha,
          oneapi::mkl::sparse::matrix_handle_t handle,
          sycl::buffer<float, 1> &x,
          const float beta,
          sycl::buffer<float, 1> &y);


sycl::event gemv(sycl::queue &queue,
                 oneapi::mkl::transpose transpose_flag,
                 const double alpha,
                 oneapi::mkl::sparse::matrix_handle_t handle,
                 double *x,
                 const double beta,
                 double *y,
                 const std::vector<sycl::event> &dependencies = {});

} // namespace oneapi::mkl::sparse
```

API that works for both sycl::buffer or USM arrays in matrix_handle_t handle

Example of sycl::buffer APIs

Example of USM APIs

Sparse BLAS TRSV SYCL C++ API

TRSV: A a sparse triangular matrix, x, y dense vectors:

Solve for y:

$$\text{op}(A) \cdot y = x$$

```
namespace oneapi::mkl::sparse {

sycl::event optimize_trsv(sycl::queue &queue,
                          oneapi::mkl::uplo uplo_flag,
                          oneapi::mkl::transpose transpose_flag,
                          oneapi::mkl::diag diag_val,
                          oneapi::mkl::sparse::matrix_handle_t handle,
                          const std::vector<sycl::event> &dependencies = {});

 void trsv(sycl::queue &queue,
           oneapi::mkl::uplo uplo_flag,
           oneapi::mkl::transpose transpose_flag,
           oneapi::mkl::diag diag_val,
           oneapi::mkl::sparse::matrix_handle_t handle,
           sycl::buffer<float, 1> &x,
           sycl::buffer<float, 1> &y);

sycl::event trsv(sycl::queue &queue,
                 oneapi::mkl::uplo uplo_flag,
                 oneapi::mkl::transpose transpose_flag,
                 oneapi::mkl::diag diag_flag,
                 oneapi::mkl::sparse::matrix_handle_t handle,
                 float *x,
                 float *y,
                 const std::vector<sycl::event> &dependencies = {});

} // namespace oneapi::mkl::sparse
```

API works for both sycl::buffer and USM

Example of sycl::buffer APIs

Example of USM API

Sparse BLAS GEMM SYCL C++ API

GEMM: A a general sparse matrix, B, C dense matrices, row-major or column-major format, α, β scalars:

$$C = \alpha \cdot \mathrm{op}(A) \cdot \mathrm{op}(B) + \beta \cdot C$$

```
namespace oneapi::mkl::sparse {

void gemm(sycl::queue &queue,
          oneapi::mkl::layout dense_matrix_layout,
          oneapi::mkl::transpose transpose_A,
          oneapi::mkl::transpose transpose_B,
          const float alpha,
          oneapi::mkl::sparse::matrix_handle_t handle,
          sycl::buffer<float, 1> &b,
          const std::int64_t columns,
          const std::int64_t ldb,
          const float beta,
          sycl::buffer<float, 1> &c,
          const std::int64_t ldc);

sycl::event gemm(sycl::queue &queue,
                 oneapi::mkl::layout dense_matrix_layout,
                 oneapi::mkl::transpose transpose_A,
                 oneapi::mkl::transpose transpose_B,
                 const double alpha,
                 oneapi::mkl::sparse::matrix_handle_t handle,
                 double *b,
                 const std::int64_t columns,
                 const std::int64_t ldb,
                 const double beta,
                 double *c,
                 const std::int64_t ldc,
                 const std::vector<sycl::event> &dependencies = {});
} // namespace oneapi::mkl::sparse
```

Example of sycl::buffer APIs

Example of USM APIs

Backup – Intel oneMKL SYCL C++ Examples

oneMKL SYCL Usage Constructs

```
sycl::queue Q{sycl::cpu_selector_v};
sycl::queue Q{sycl::gpu_selector_v};
sycl::queue Q{device};
```

Create device queue attached to a given device or device type.

All device execution goes through a queue object.

```
void *mem = sycl::malloc_shared(bytes, Q);
void *mem = sycl::malloc_device(bytes, Q);
```

Allocate device-accessible memory. `malloc_shared` memory is also accessible from the host.

```
sycl::buffer<T,1> mem(elements);
sycl::buffer<T,1> mem(elements, hostptr);
```

Smart buffer object. Migrates memory automatically and tracks data dependencies.

Can be attached to host memory (synchronized at creation and destruction).

oneMKL Traditional C API: GEMM Example

$$C \leftarrow \alpha AB + \beta C$$

```
int main() {

    int64_t m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    int64_t sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;


    // Allocate matrices
    double *A = (double *) mkl_malloc(sizeof(double) * sizea);
    double *B = (double *) mkl_malloc(sizeof(double) * sizeb);
    double *C = (double *) mkl_malloc(sizeof(double) * sizec);

    // Initialize matrices here
    …
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                alpha, A, lda, B, ldb, beta, C, ldc);
    …
}
```

oneMKL SYCL API: GEMM Example

```
int main() {
    using namespace oneapi::mkl;

    int64_t m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    int64_t sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;

    sycl::queue Q{sycl::gpu_selector_v};

    // Allocate matrices
    double *A = malloc_shared<double>(sizea, Q);
    double *B = malloc_shared<double>(sizeb, Q);
    double *C = malloc_shared<double>(sizec, Q);

    // Initialize matrices here
    …
    auto e = blas::gemm(Q, transpose::N, transpose::N, m, n, k,
                        alpha, A, lda, B, ldb, beta, C, ldc);
    …
}
```

$$C \leftarrow \alpha AB + \beta C$$

Set up GPU queue

Allocate CPU/GPU-accessible shared memory

New oneMKL SYCL API

Computation is performed on given queue

Output **e** is a sycl::event object representing command completion

Call `e.wait()` to wait for completion

oneMKL SYCL Matrix Multiply Example (1 of 2)

$$C \leftarrow A \cdot B$$

Buffer API

```
using namespace oneapi::mkl;
int64_t n = 32;

sycl::device dev({host, cpu, gpu}_selector_v);
sycl::queue Q(dev);


sycl::buffer<double, 1> A_buf{n * n},
                        B_buf{n * n},
                        C_buf{n * n};

// Initialize data here

blas::gemm(Q, transpose::N, transpose::N,
           n, n, n, 1.0, A_buf, n, B_buf, n,
           0.0, C_buf, n);
```

- device setup
- prepare matrices
- C = A * B

USM API

```
using namespace oneapi::mkl;
int64_t n = 32;

sycl::device dev({host, cpu, gpu}_selector_v);
sycl::queue Q(dev);


double *A = sycl::malloc_shared<double>(n * n, Q);
double *B = sycl::malloc_shared<double>(…);
double *C = sycl::malloc_shared<double>(…);

// Initialize data here

blas::gemm(Q, transpose::N, transpose::N,
           n, n, n, 1.0, A, n, B, n,
           0.0, C, n);

Q.wait_and_throw();
```

oneMKL SYCL Matrix Multiply Example (2 of 2)

$D \leftarrow A \cdot B \cdot A$

Buffer API

```
using namespace oneapi::mkl;
int64_t n = 32;

sycl::device dev({host, cpu, gpu}_selector_v);
sycl::queue Q(dev);


sycl::buffer<double, 1> A_buf{n * n},
                        B_buf{n * n},
                        C_buf{n * n},
                        D_buf{n * n};
// Initialize data here

blas::gemm(Q, transpose::N, transpose::N,
           n, n, n, 1.0, A_buf, n, B_buf, n,
           0.0, C_buf, n);

blas::gemm(Q, transpose::N, transpose::N,
           n, n, n, 1.0, C_buf, n, A_buf, n,
           0.0, D_buf, n);
```

device setup

prepare matrices

C = A * B

D = C * A

USM API

```
using namespace oneapi::mkl;
int64_t n = 32;

sycl::device dev({host, cpu, gpu}_selector_v);
sycl::queue Q(dev);


double *A = sycl::malloc_shared<double>(n * n, Q);
double *B = sycl::malloc_shared<double>(…);
double *C = sycl::malloc_shared<double>(…);
double *D = sycl::malloc_shared<double>(…);
// Initialize data here

event e = blas::gemm(Q, transpose::N, transpose::N,
                     n, n, n, 1.0, A, n, B, n,
                     0.0, C, n);

blas::gemm(Q, transpose::N, transpose::N,
           n, n, n, 1.0, C, n, A, n,
           0.0, D, n, {e});
```

oneMKL OpenMP Offload Usage Directives (C)

```
#pragma omp target data
```

Map host-side variables to device variables inside this block.

```
#pragma omp target enter data
#pragma omp target exit data
```

Map/unmap host-side variables to device variables: the two halves of #pragma omp target data.

```
#pragma omp target
```

Offload execution of block to the GPU.

```
#pragma omp target variant dispatch
#pragma omp dispatch (OpenMP 5.1) - recommended
```

Offload supported oneMKL calls inside this block to the GPU.

oneMKL C OpenMP Offload: GEMM

```
int main() {
    long m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    long sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;

    // Allocate matrices
    double *A = (double *) mkl_malloc(sizeof(double) * sizea, 64);
    double *B = (double *) mkl_malloc(sizeof(double) * sizeb, 64);
    double *C = (double *) mkl_malloc(sizeof(double) * sizec, 64);

    // Initialize matrices here
    …
#pragma omp target data map(to:A[0:sizea],B[0:sizeb]) map(tofrom:C[0:sizec])
    {
#pragma omp dispatch nowait
        {
            // Compute C = A * B on GPU
            cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                        alpha, A, lda, B, ldb, beta, C, ldc);
        }
    }
    …
}
```

$C \leftarrow \alpha AB + \beta C$

Use `target data map` to send matrices to the device

Use `omp dispatch` to request GPU execution for cblas_dgemm

Optional `nowait` clause for asynchronous execution
Use `#pragma omp taskwait` for synchronization

oneMKL Fortran OpenMP Offload: GEMM

```
include “mkl_omp_offload.f90”

program main
use onemkl_blas_omp_offload_ilp64

integer                :: m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10
integer                :: sizea = lda * k, sizeb = ldb * n, sizec = ldc * n
double                 :: alpha = 1.0, beta = 0.0
double, allocatable :: A(:), B(:), C(:)

// Allocate matrices here
allocate(A(sizea))
…

// Initialize matrices here
…
!$omp target data map(to:A(1:sizea), B(1:sizeb)) map(tofrom:C(1:sizec))
!$omp dispatch nowait

! Compute C = A * B on GPU
call dgemm(‘N’, ‘N’, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)

!$omp end target data
…
end program
```

Module for Fortran OpenMP offload for MKL_ILP64 (64-bit integers)

Use `target data map` to send matrices to the device
Use `dispatch` to request GPU execution for `dgemm`

Optional `nowait` clause for asynchronous execution
Use `!$omp taskwait` for synchronization

What is it?

Who needs this product?

Why is this important?

Intel® oneAPI Base & HPC Toolkits

Direct Programming	API-Based Programming	Analysis & debug Tools
Intel® C++ Compiler Classic	Intel® MPI Library	Intel® Inspector
Intel® Fortran Compiler Classic	Intel® oneAPI DPC++ Library oneDPL	Intel® Trace Analyzer & Collector
Intel® Fortran Compiler	Intel® oneAPI Math Kernel Library - oneMKL	Intel® Cluster Checker
Intel® oneAPI DPC++/C++ Compiler	Intel® oneAPI Data Analytics Library - oneDAL	Intel® VTune™ Profiler
Intel® DPC++ Compatibility Tool	Intel® oneAPI Threading Building Blocks - oneTBB	Intel® Advisor
Intel® Distribution for Python	Intel® oneAPI Video Processing Library - oneVPL	Intel® Distribution for GDB
Intel® FPGA Add-on for oneAPI Base Toolkit	Intel® oneAPI Collective Communications Library oneCCL	
	Intel® oneAPI Deep Neural Network Library - oneDNN	
	Intel® Integrated Performance Primitives – Intel® IPP	

Intel® oneAPI **HPC** Toolkit +
Intel® oneAPI **Base** Toolkit

Learn More: intel.com/oneAPI-HPCKit

Building and Linking – OpenMP Offload

- Headers

```
#include “omp.h”
#include “mkl_omp_offload.h”
```

- Build flags (new flags highlighted)

```
icx -c -DMKL_ILP64 -m64 -fiopenmp -fopenmp-targets=spir64 –mllvm
-vpo-paropt-use-raw-dev-ptr -I${MKLROOT}/include source.c -o source.o
```

- Link flags (example: dynamic link; sequential threading)

```
icx source.o  -fiopenmp -fopenmp-targets=spir64 -mllvm -vpo-paropt-use-raw-dev-ptr -
L${MKLROOT}/lib/intel64 -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core  -lOpenCL -lpthread -
ldl -lm -o source.out
```

See https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html for latest recommendations on build/link lines for oneMKL.

Building and Linking – SYCL

- Headers

```
#include <sycl.hpp>
#include "mkl.h" // for C APIs
#include "oneapi/mkl.hpp" // for SYCL C++ APIs
```

- Build flags (new flags highlighted)

```
icpx -fsycl -DMKL_ILP64 -I${MKLROOT}/include source.c -o source.o
```

- Link flags (example: dynamic link; sequential threading)

```
icpx -fsycl source.o -L${MKLROOT}/lib/intel64 -lmkl_sycl –lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -lsycl -lOpenCL -lpthread -ldl -lm -o source.out
```

See https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-link-line-advisor.html for latest recommendations on build/link lines for oneMKL.

Intel® oneAPI Math Kernel Library Resources

software.intel.com/oneAPI/mkl

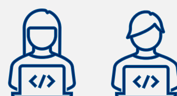
Get Started

- software.intel.com/oneAPI/mkl
- oneMKL - Get Started Guide
- oneMKL code samples
- oneMKL how-to's
- Migrating the MonteCarloMultiGPU from CUDA* to SYCL*

OneMKL Developer References & Guides

Developer References:
C | Fortran
Developer Guides:
Windows* | Linux* | macOS*

Learn



- Training: Webinars & courses
- oneMKL Essentials Training
- Base toolkit on Intel® DevCloud
- oneMKL documentation
- Intel® oneMKL Link Line Advisor

Ecosystem & Support

Rich active developer ecosystem eases adoption

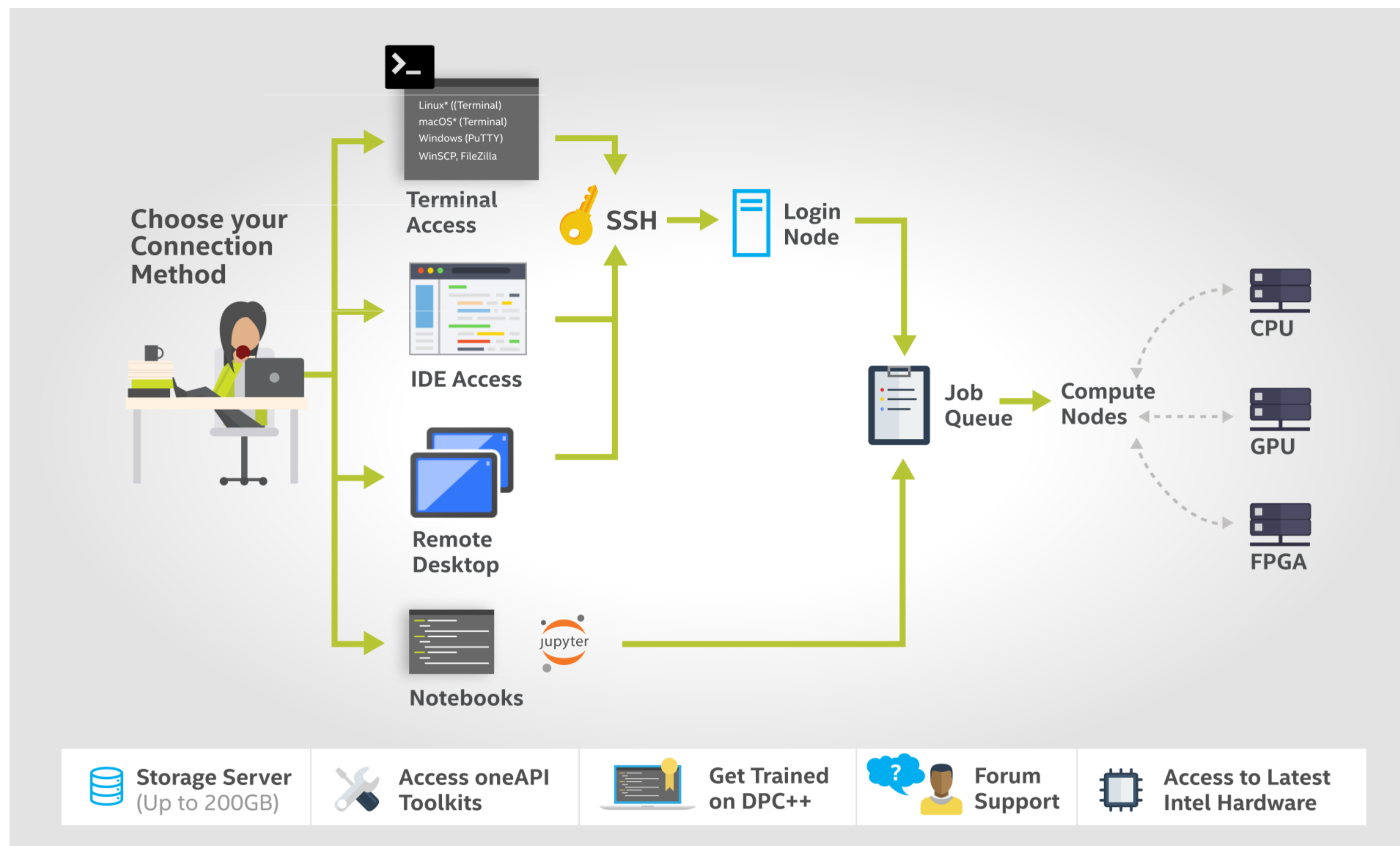
- oneMKL Community Forum
- Intel® DevMesh Innovator oneMKL Projects
- oneMKL Academic Programs: oneAPI Centers of Excellence: research, enabling code, curriculum, teaching
- Online Service Center (paid support)

Intel® oneAPI Math Kernel Library (oneMKL) available on Intel® DevCloud

Implement and test your applications on Intel® DevCloud today

intel. DevCloud

- 1 Minute to Code
- No Hardware Acquisition
- No Download, Install or Configuration
- Easy Access to Samples & Tutorials
- Support for Jupyter Notebooks, Visual Studio Code

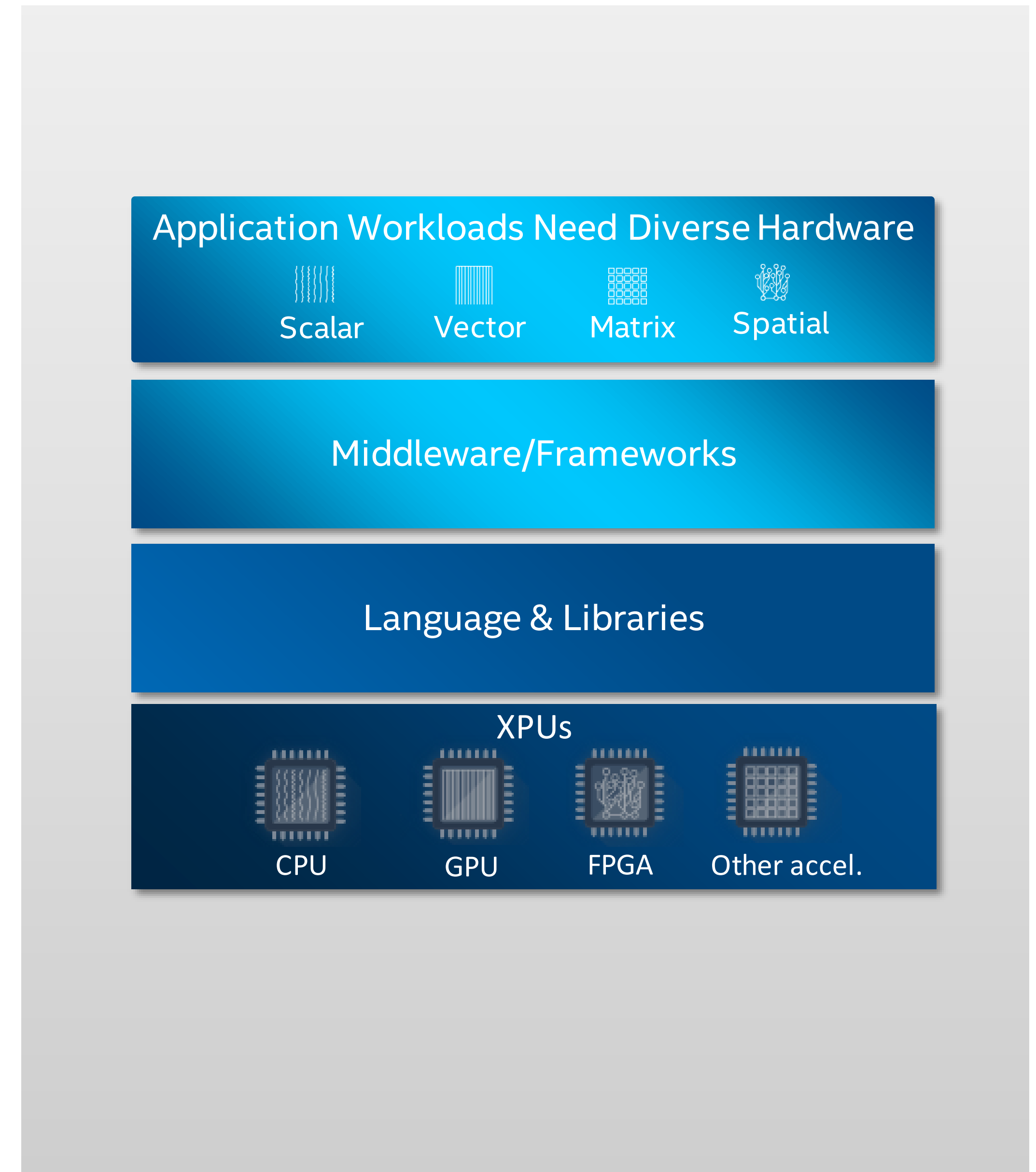


software.intel.com/devcloud/oneapi

Backup – Parallel Languages and Architecture

Programming Challenges for Multiple Architectures

- Growth in specialized workloads
- No common programming language or APIs
- Inconsistent tool support across platforms
- Each platform requires unique software investment
- Diverse set of data-centric hardware required



SYCL: Standards-Based, Cross-Architecture Language

Productivity and performance for CPUs and accelerators

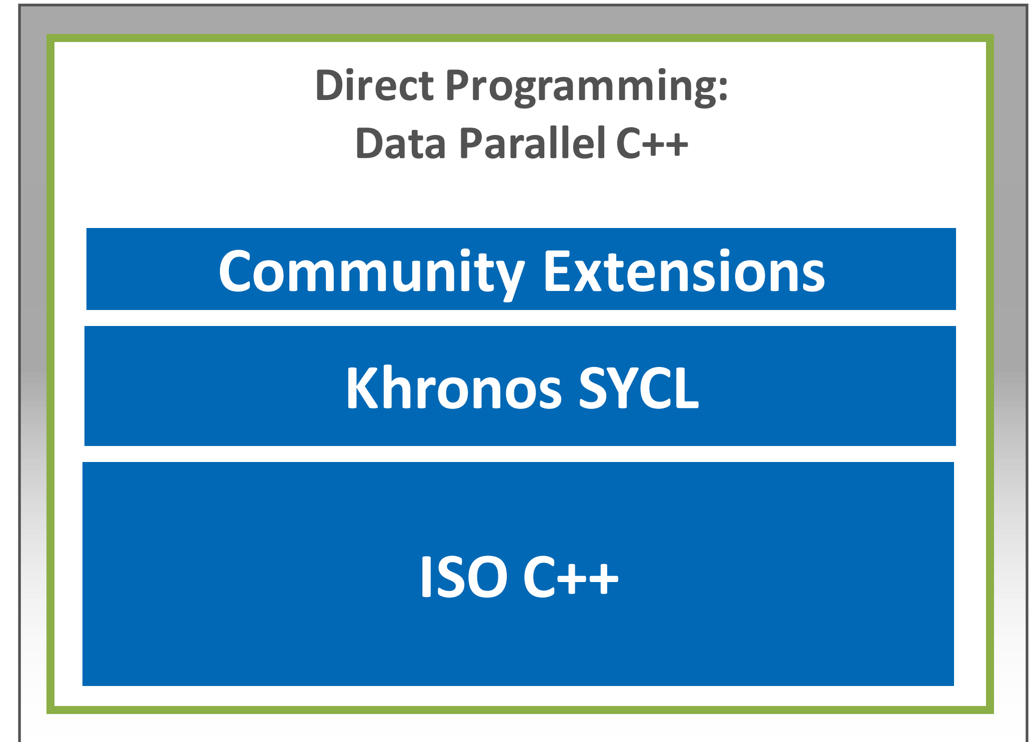
- Allows code reuse across hardware targets while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single-architecture proprietary language

Based on ISO C++ and Khronos SYCL

- Delivers C++ productivity benefits, using common and familiar C and C++ constructs
- Incorporates SYCL from the Khronos Group to support data parallelism and heterogeneous programming

Community project to drive language enhancements

- Extensions to simplify data parallel programming
- Open and cooperative development for continued evolution



The open source and Intel DPC++ compiler currently support hardware including Intel CPUs, GPUs, and FPGAs.

Codeplay announced a DPC++ compiler that targets Nvidia GPUs.

In the Beginning, There Was C++11

```
std::vector<float> A(n,2);
std::vector<float> B(n,0);

for(int i=0; i<n; i++)
{
    B[i] += A[i] * A[i];
}
```

Loops Can Be Rewritten Using Lambdas

```
auto range = ranges::view::iota(0,n);
auto begin = std::begin(range);
auto end   = std::end(range);
std::for_each(begin, end, [&] (auto i)
{
    B[i] += A[i] * A[i];
});
```

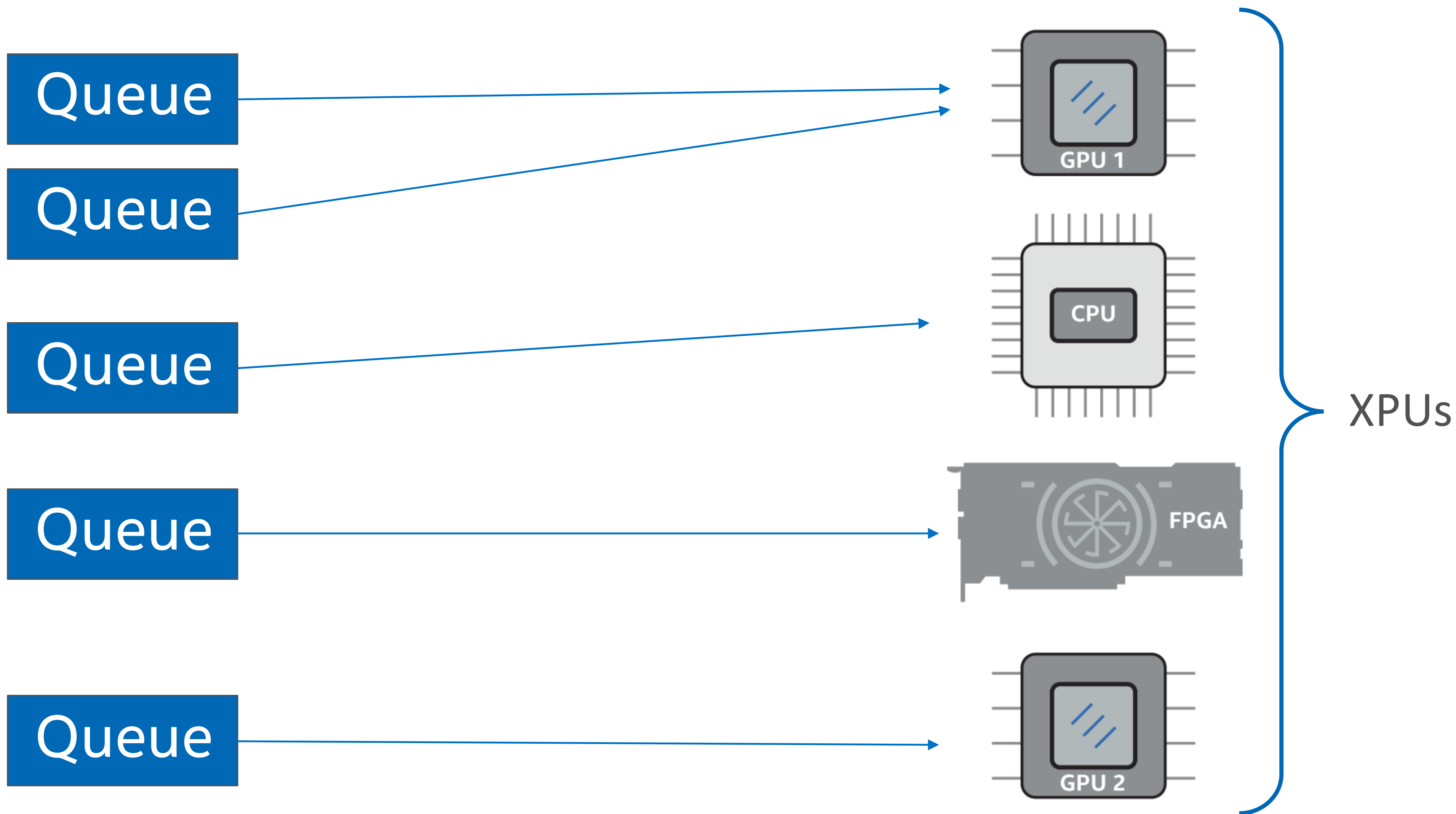
These Loops Can Be Parallelized

```
// TBB
tbb::parallel_for_each(begin, end, [&](auto i)
{
    B[i] += A[i] * A[i];
});
// Parallel STL
std::for_each(exec::par_unseq, begin, end, [&](auto i)
{
    B[i] += A[i] * A[i];
});
```

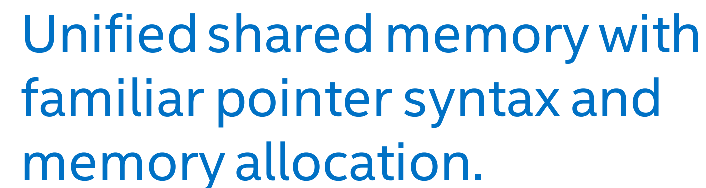
SYCL Allows Us To Run This Code on a Device

```
sycl::buffer<float> bA(A.data(), A.size());
sycl::buffer<float> bB(B.data(), B.size());
sycl::queue q(sycl::default_selector{});
q.submit([&](sycl::handler& h) {
    auto pA = bA.get_access<sycl::access::mode::read>(h);
    auto pB = bB.get_access<sycl::access::mode::read_write>(h);
    h.parallel_for<class aKernel>(sycl::range<1>{n}, [=](sycl::id<1> it)
    {
        pB[it] += pA[it] * pA[it];
    });
});
q.wait();
```

Device Discovery and Managing Work



SYCL Adds New Features




```
sycl::queue q(gpu_selector{});
float * A = sycl::malloc_shared(n * sizeof(float), q);
float * B = sycl::malloc_shared(n * sizeof(float), q);
q.submit([&](sycl::handler& h)
{
    h.parallel_for<class aKernel>(sycl::range<1>{n}, [=](sycl::id<1> it)
    {
        B[it] += A[it] * A[it];
    });
});
q.wait();
```

SYCL Adds New Features

```
sycl::queue q(gpu_selector{});
float * A = sycl::malloc_shared<float>(n, q);
float * B = sycl::malloc_shared<float>(n, q);
q.submit([&](sycl::handler& h)
{
    h.parallel_for<class aKernel>(sycl::range<1>{n}, [=](sycl::id<1> it)
    {
        B[it] += A[it] * A[it];
    });
});
q.wait();
```

SYCL Adds New Features

```
sycl::queue q(gpu_selector{});

float * A = sycl::malloc_shared<float>(n, q);
float * B = sycl::malloc_shared<float>(n, q);

q.parallel_for(sycl::range{n}, [=](sycl::id<1> it)
{
    B[it] += A[it] * A[it];
}).wait();
```

Backup – Visual Sparse BLAS Operations

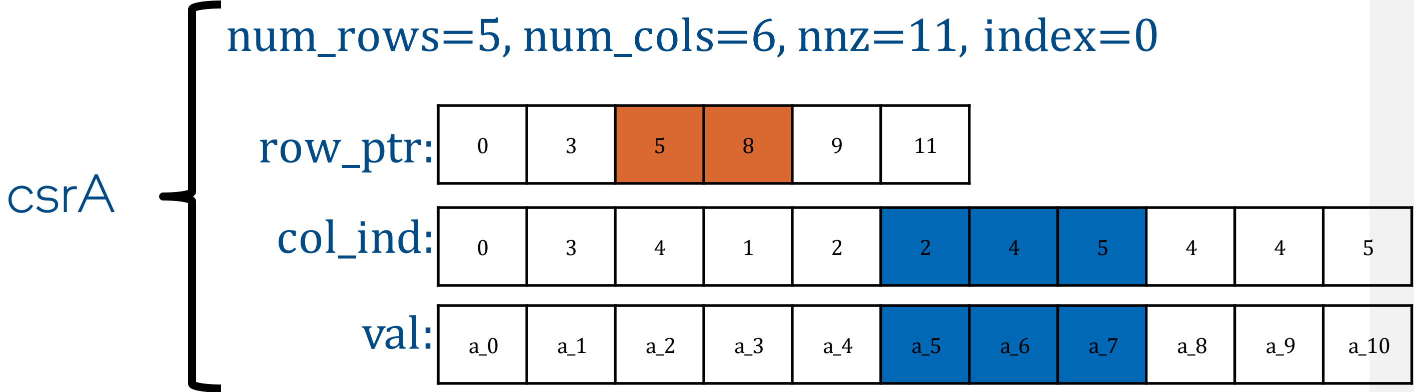
Sparse CSR 3-array matrix format

intType : std::int32_t, std::int64_t
dataType : float, double, std::complex<float>, std::complex<double>

- A in compressed sparse row matrix format:
  - **num_rows** – (intType) number of rows in A
  - **num_cols** – (intType) number of columns in A
  - **nnz** – (intType) number of non-zeros in A (= **row_ptr**[num_rows])
  - **index** – (intType) 0 (C/C++ style) or 1 (Fortran style) based indices in **row_ptr**, **col_ind** arrays
  - **row_ptr** – (intType[num_rows+1]) array of offsets for each row k in **col_ind** /**val** arrays.
    I.e. **row_ptr**[k] is the start of row k in **col_ind** and **val** arrays.
  - **col_ind** – (intType[nnz]) array of column indices
  - **val** – (dataType[nnz]) array of values

A:

a_0			a_1	a_2	
	a_3	a_4			
		a_5		a_6	a_7
				a_8	
				a_9	a_10



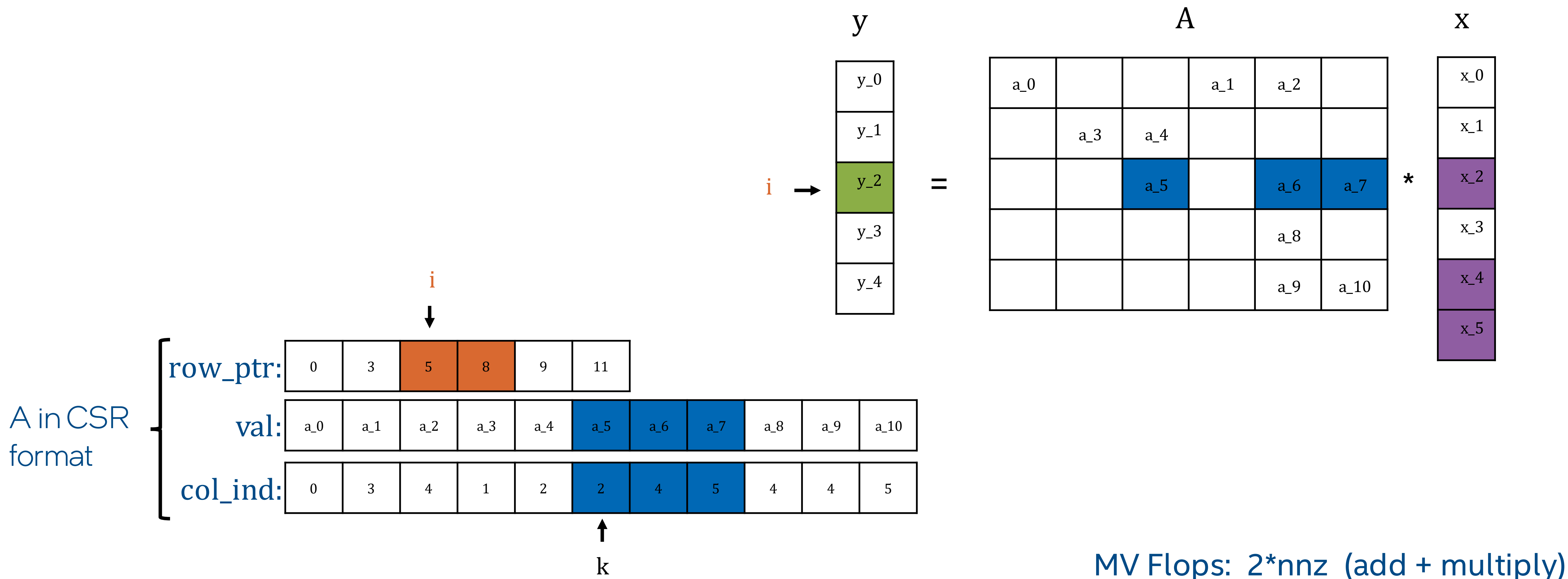
- Memory footprint of csrA is
  - sizeof(csrA) = [sizeof(intType) + sizeof(dataType)]*nnz + sizeof(intType)*(num_rows+1)

Sparse GEMV Operation

$$y \leftarrow \alpha \cdot A \cdot x + \beta \cdot y$$

with alpha = 1, beta = 0, simplifies to

$$y = A \cdot x$$



MV Flops: 2*nnz (add + multiply)

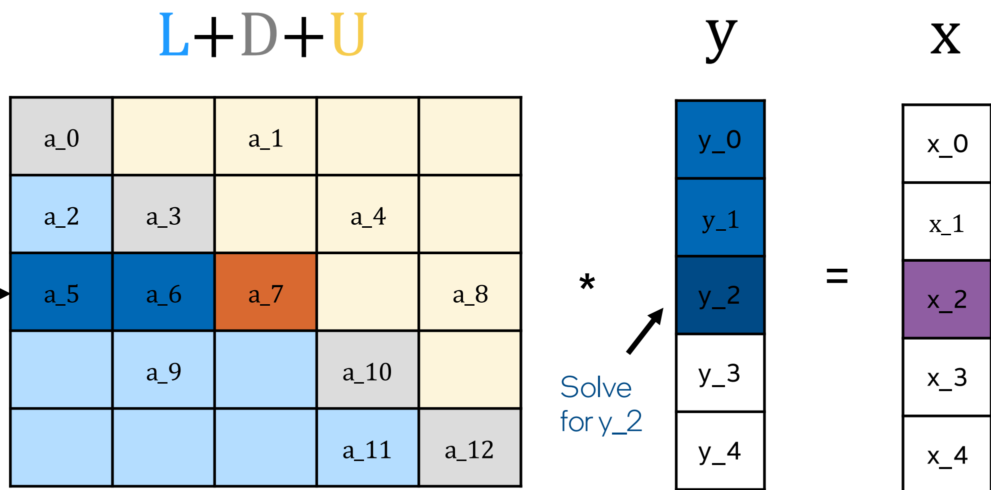
Compute y_i:

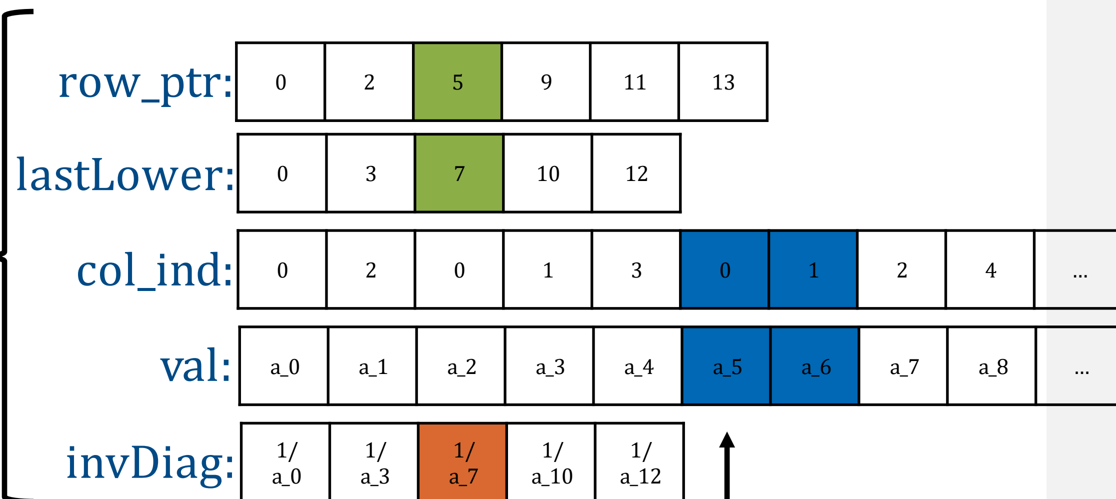
y[i] = sum_{ k in [row_ptr[i], row_ptr[i+1]) } (val[k] * x[col_ind[k]])

Sparse TRSV Operation

Forward Solve: $(L + D) \cdot y = x$

A = L+D+U – Sparse CSR(nrows x nrows) format, is partially sorted by lower/diag/upper on each row with an additional lastLower pointer array for where lower parts ends and an inverse diagonal values array. Performance can be better if A is L+D or just L.





y[0] = (x[0]) *invDiag[0]

y[1] = (x[1] – val[2]*y[0]) * invDiag[1]

y[2] = (x[2] – val[5]*y[0] – val[6]*y[1]) * invDiag[2]

...

y[i] = (x[i] - sum_{ k in [row_ptr[i], lastLower[i]) } (val[k] * y[col_ind[k]])) * invDiag[i]

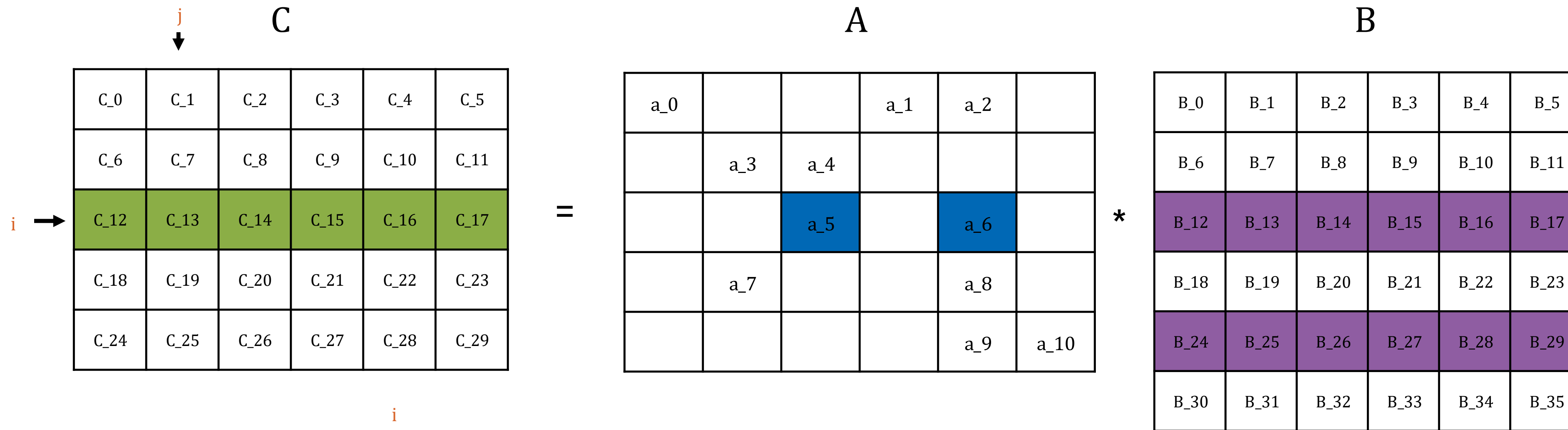
TRSV Flops: 2*nnz(L) + 2*nrow

Note 1: this sum is just GEMV (L*y) on strictly lower portion (L) of A

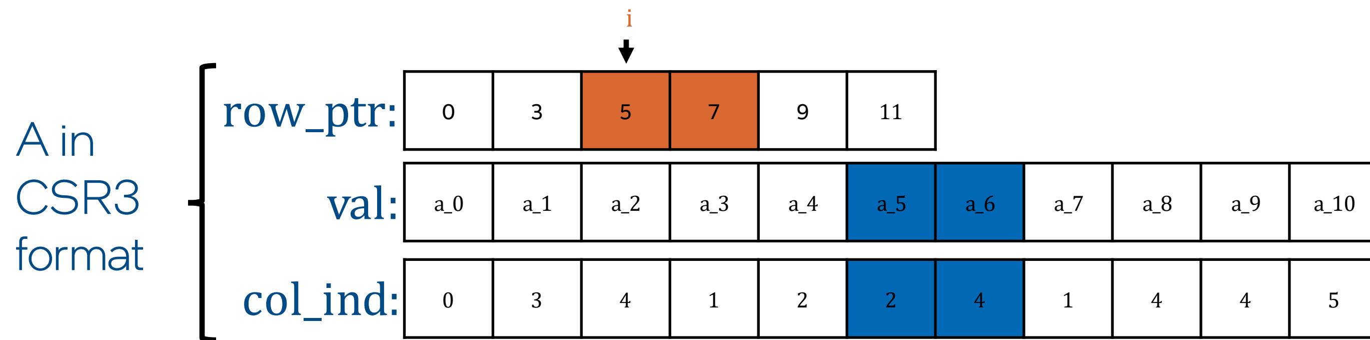
Note 2: y is being updated incrementally and the GEMV-like sum part only uses previously updated parts of y.

Sparse GEMM Operation (col-major)

C = A * B
(i.e. alpha=1, beta = 0)
A - sparse CSR (nrows x ncols)
B - dense col-major (ncols x nrhs)
C - dense col-major (nrows x nrhs)



A in CSR3 format

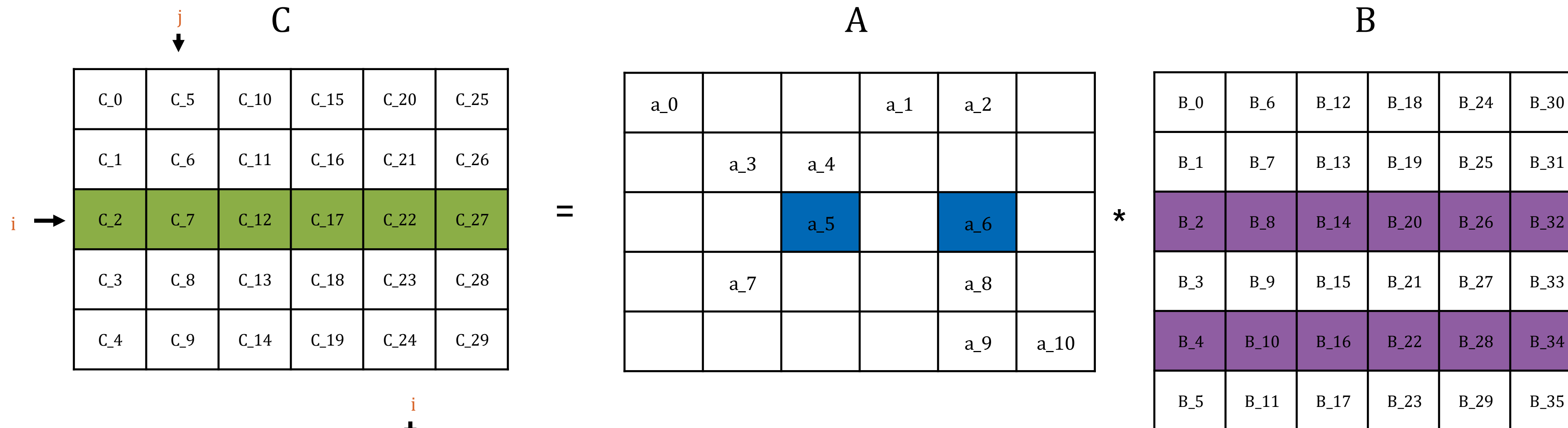


Compute C_ij value:

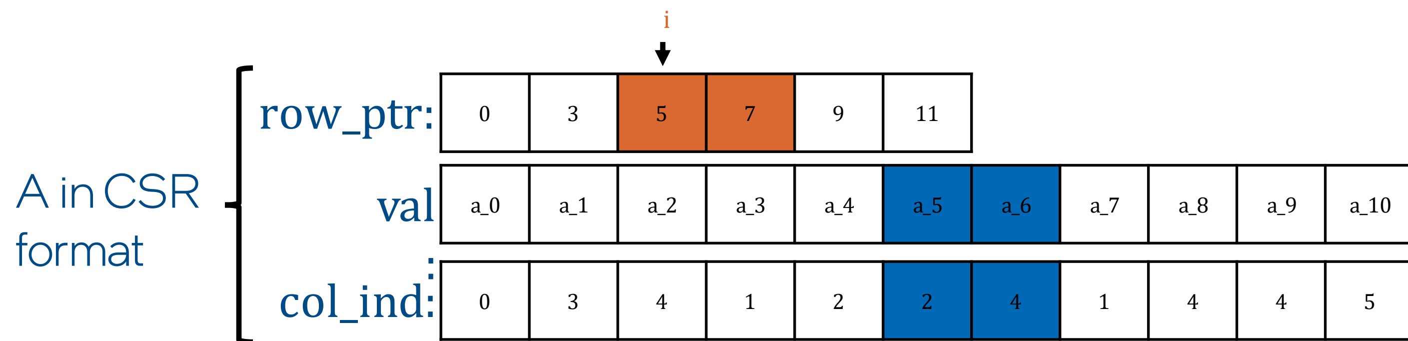
C[i * nrhs + j] = sum_{ k in [row_ptr[i], row_ptr[i+1]) } (val[k] * B[col_ind[k] * nrhs + j])

Sparse GEMM Operation (row-major)

C = A * B
(i.e. alpha=1, beta = 0)
A - sparse CSR (nrows x ncols)
B - dense row-major (ncols x nrhs)
C - dense row-major (nrows x nrhs)



A in CSR format



Compute C_ij value:

C[j*nrows + i] = sum_{ k in [row_ptr[i], row_ptr[i+1]) } (val[k] * B[j*ncols + col_ind[k]])

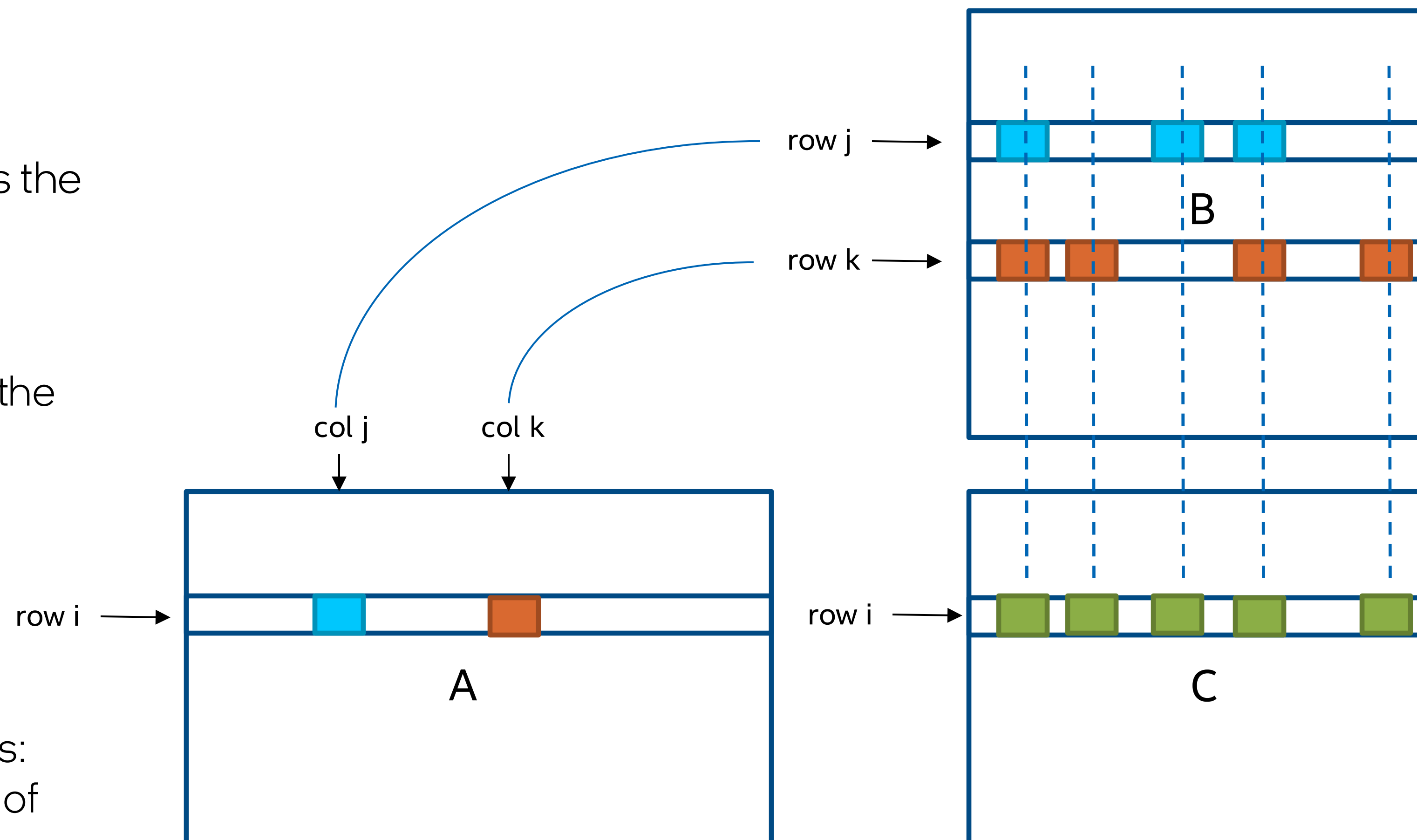
Sparse * Sparse Matrix Multiplication

$$C_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

The most common CSR-CSR algorithm is the **Gustavson algorithm** which is a row contraction: $C(i,:) = A(i,:) \cdot B$

The algorithm requires two passes through the sparse structures:

- **First pass** determines size (number of non-zeros) of each row of C (using a k-way merge of rows of B based on non-zero column ids in row of A),
- **Allocate memory for C**
- **Second pass** fills in the indices and values:
    - accumulate each row of C as a sum of A-value-scaled rows of B into some intermediate structure often called a **sparse accumulator.**
    - Sparse accumulator possibilities: dense array of length B_ncols, several passes through dense array of length N << B_ncols, hash table of indices/values, heap structure (sorted balanced min-tree of indices/values), etc



intel