oneMKL Technical Advisory Board

Session 14

June 16, 2021

Agenda

- Welcoming remarks – 5 minutes
- Updates from last meeting – 5 minutes
- Overview of sparse matrix * sparse matrix API – Spencer Patty (40 minutes)
- Wrap-up and next steps – 5 minutes

Updates from last meeting

- oneAPI Math Kernel Library (oneMKL) Interfaces Project
  - Interest from community to be able to support rocBLAS
  - Added interfaces for the LAPACK domain with support for Intel® oneAPI Math Kernel Library on CPUs and Intel GPUs
- oneAPI Developer Summit at ISC, June 22-23
  - Schedule, register: https://www.oneapi.com/events/devcon2021isc/
  - Keynote from Nevin Liber: oneAPI, SYCL and Standard C++: Where Do We Need To Go From Here?
  - Tech Talk from Hartwig Anzt: Ginkgo - An Open Source Math Library in the oneAPI Ecosystem

Overview of sparse matrix * sparse matrix API

Outline

1. Existing Library APIs
   - Comparison of existing sparse * sparse APIs
2. Proposal of Sparse Matrix * Sparse Matrix API
   - Mathematical Description of Operation
   - Allocation Scenarios for the Output C matrix
   - Matmat API proposal for user allocated C matrix memory
   - Supporting multiple stages of computation
   - Simplifications to Matmat API arguments
   - Examples of Matmat API usage
3. Possible Future Extensions
   - Other API extensions

Existing Libraries with sparse*sparse APIs

Comparison of existing sparse * sparse APIs

Library	API name(s)	Math Operation	Count of Stages	Tracking Stages	Structure only/ Structure + values	Memory Allocation of C is done by	User passes temporary buffers	Current Matrix Formats Supported
cuSPARSE Generic APIs	cusparseSpGEMM_workEstimation cusparseSpGEMM_compute cusparseSpGEMM_copy	C <- alpha * op(A) * op(B) + beta * C ‡	3	Multiple APIs	Structure + values only •	User	Yes	CSR
rocSPARSE Generic APIs	rocsparse_spgemm	C = alpha * op(A) * op(B) + beta * D †	2	Enum argument	Structure + values only •	User	Yes	CSR°
hipSPARSE Generic APIs	hipsparseSpGEMM_workEstimation hipsparseSpGEMM_compute hipsparseSpGEMM_copy	C <- alpha * op(A) * op(B) + beta * C ‡	3	Multiple APIs	Structure + values only •	User	Yes	CSR°
MKL IE Sparse BLAS	mkl_sparse_sp2m	C = op(A) * op(B)	2	Enum argument	Both	Library	No	CSR, CSC, BSR
ViennaCL	viennacl::linalg::prod	C = A * B	1	N/A	Structure + values only	Library	No	CSR
Ginkgo	A->apply(B,C) A->apply(alpha,B,beta, C)	C = A*B C = alpha*A*B + beta*C	1	N/A	Both	Library	No	CSR, Dense, COO, Hybrid, ELL, SELL-P
Magma	magma_?_spmm	C = alpha * A * B	1	N/A	Structure + values only	Library	No	CSR, ELL, SELL-P
oneMKL BLAS GEMM	oneapi::mkl::blas::gemm	C <- alpha op(A) * op(B) + beta*C	1	N/A	Structure + values only	User	No	Dense

‡ Sparsity pattern of matrix product must not change sparsity pattern of C.
† If D == C, then matrix product must not change the sparsity pattern of C input.
• It is not clear from documentation or API whether “only sparsity pattern” is supported
° It is not clear from documentation if more sparse matrix formats are supported

Proposal of Sparse * Sparse DPC++ API

Sparse Matrix-Matrix Product Mathematical Description

We propose to add support for the sparse matrix * sparse matrix product with sparse matrix (s*s->s) output:

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

where A,B and C are real general sparse matrices and op is one of

$$\mathrm{op}(A) = \begin{cases} A \\ A^T. \end{cases}$$

The first sparse matrix format to be supported will be the Compressed Sparse Row (CSR) matrix format.

Sidenote: A possible future extension can overload the proposed API with 4 matrix arguments, fusing the sparse matrix add:

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

where D can be C with the caveat that the matrix product does not change the sparsity pattern of C. See “Possible Future API Extensions” slide at the end of the deck for other possible extensions.

Allocation of Memory for output matrix C

Background:

A priori, the size of the output sparse matrix, C, is not known, so memory allocation for the final C matrix must be a part of the algorithm.

There are two scenarios for handling the allocation of memory for the sparse matrix-matrix product
Scenario 1: User allocates all memory arrays for the C output matrix (and possibly for any temporary workspaces as well)
Scenario 2: Library allocates all memory arrays for the C output matrix (and then provides a way for the user to get access)

Scenario 1: User Allocation of memory <- this is main part of this proposal

1. Users will allocate arrays for C matrix and provide them to the library to fill:
   1. User needs a way of allocating and providing output matrix C matrix format arrays to be filled by us
   2. We need a way of providing users (asynchronously) the size of any temporary buffers and size of output matrix C matrix format arrays (C_nnz) so they can allocate the arrays themselves.
   3. User needs a way of providing us any temporary buffers (and it's size) as workspaces for computation

2. API is compatible with backend implementations for multiple stage (S * S -> S) algorithms that support "**user allocated memory for C matrix**"
   1. cuSPARSE (3 stages: workEstimation, compute, copy)
   2. rocSPARSE (2 stages: nnz, compute)

Scenario 2: Library Allocation of memory (See "Possible Future Extensions" Section)

1. Add a separate API to support **"library allocated memory for C matrix"**
   1. MKL IE SPARSE BLAS: (2 stages NNZ_COUNT, FINALIZE_MULT)
   2. GINKGO (1 stage: A->apply(B,C) which computes C = A * B)
   3. viennaCL (1 stage: viennacl::compressed_matrix<T> C = viennacl::linalg::prod(A,B))

2. We will need to add a separate API to give access to library allocated C matrix types
   1. Library should maintain ownership of these arrays for proper cleanup, but provide access for use.
   2. Users should agree not to modify the data directly, can make their own copy if this becomes necessary.

Proposed Matmat API with user allocated C memory
(C = op(A) * op(B))

USM api:

```
namespace sparse {

sycl::event matmat(
        sycl::queue &queue,
        sparse::matrix_handle_t A,
        sparse::matrix_handle_t B,
        sparse::matrix_handle_t C,
        sparse::matmat_request req,
        sparse::matmat_descr_t descr,
        std::int64_t *sizeTempBuffer,
        void *tempBuffer,
        const sycl::vector_class<sycl::event> &dependencies);
}
```

sycl::buffer API:

```
namespace sparse {

void matmat (
        sycl::queue &queue,
        sparse::matrix_handle_t A,
        sparse::matrix_handle_t B,
        sparse::matrix_handle_t C,
        sparse::matmat_request req,
        sparse::matmat_descr_t descr,
        sycl::buffer<std::int64_t,1> *sizeTempBuffer,
        sycl::buffer<std::byte,1> *tempBuffer);
}
```

1. (Design): We introduce “sparse::matmat_request” enum type to support multiple stages of work in the algorithm. See “Sparse * sparse multi-stage computation model” slide.
2. (Design Goal): We want to simplify API arguments as much as possible, so we introduce “sparse::matmat_descr_t” opaque pointer to house the matrix product description. See “Simplifying the matmat API arguments” slide
3. (Design): We choose to always use std::int64_t for Sparse BLAS “sizes” in DPC++ APIs since matrix sizes and/or number of elements may exceed 4 billion boundary
4. (Design Goal): We want the matmat API to be asynchronous as much as is possible. Memory allocation puts a wrinkle in this as this is currently a synchronization point. Make the temporary sizes be “runtime aware” containers.
5. (Design): We use sycl::buffer<std::byte,1> * to represent temporary workspaces in buffer APIs (a replacement for void *), internally can be reinterpreted to appropriate datatype types for use.

Comparison of existing sparse * sparse APIs

Library	API name(s)	Math Operation	Count of Stages	Tracking Stages	Structure only/ Structure + values	Memory Allocation of C is done by	User passes temporary buffers	Current Matrix Formats Supported
cuSPARSE Generic APIs	cusparseSpGEMM_workEstimation cusparseSpGEMM_compute cusparseSpGEMM_copy	C <- alpha * op(A) * op(B) + beta * C ‡	3	Multiple APIs	Structure + values only •	User	Yes	CSR
rocSPARSE Generic APIs	rocsparse_spgemm	C = alpha * op(A) * op(B) + beta * D †	2	Enum argument	Structure + values only •	User	Yes	CSR°
hipSPARSE Generic APIs	hipsparseSpGEMM_workEstimation hipsparseSpGEMM_compute hipsparseSpGEMM_copy	C <- alpha * op(A) * op(B) + beta * C ‡	3	Multiple APIs	Structure + values only •	User	Yes	CSR°
MKL IE Sparse BLAS	mkl_sparse_sp2m	C = op(A) * op(B)	2	Enum argument	Both	Library	No	CSR, CSC, BSR
ViennaCL	viennacl::linalg::prod	C = A * B	1	N/A	Structure + values only	Library	No	CSR
Ginkgo	A->apply(B,C) A->apply(alpha,B,beta, C)	C = A*B C = alpha*A*B + beta*C	1	N/A	Both	Library	No	CSR, Dense, COO, Hybrid, ELL, SELL-P
Magma	magma_?_spmm	C = alpha * A * B	1	N/A	Structure + values only	Library	No	CSR, ELL, SELL-P
oneMKL BLAS GEMM	oneapi::mkl::blas::gemm	C <- alpha op(A) * op(B) + beta*C	1	N/A	Structure + values only	User	No	Dense
oneMKL Sparse BLAS	oneapi::mkl::sparse::matmat	C = op(A) * op(B)	3	Enum argument	Both	User	Yes	CSR

‡ Sparsity pattern of matrix product must not change sparsity pattern of C.
† If D == C, then matrix product must not change the sparsity pattern of C input.
• It is not clear from documentation or API whether “only sparsity pattern” is supported
° It is not clear from documentation if more sparse matrix formats are supported

Sparse * sparse multi-stage computation model:

Background:
There are common steps to a matrix-matrix multiplication workflow that each reveal new information about the final result and require different temporary workspaces, to support user allocation, we must break up the computation into these stages.

Additionally, the user may want different configurations of the output C:

1. User may want to solve for only the **sparsity pattern** of output matrix C
2. User may want to solve for the **full matrix** C (sparsity pattern and values)
3. User may want to solve for **sparsity pattern**, then later update to fill the **full matrix** values

Solution:

- We introduce a "3-stage" computation approach governed by "enum class matmat_request"
  - **work_estimation** – do initial estimation of work and load balancing (upper bound estimate of nnz)
  - **compute** – do internal products for computing C matrix (calculate nnz and do product in internal representations)
  - **finalize** – do any remaining internal product and transfer into final C matrix arrays (accumulate and convert from internal representation)
- To support user allocation of all memory workspaces, we also add corresponding buffer size requests for each stage:
  - **get_work_estimation_buf_size** – return size of temporary buffer for work_estimation stage
  - **get_compute_buf_size** – return size of temporary buffer for compute stage
  - **get_nnz** – return number of non-zeros in C matrix for user to allocate col_ind and values arrays and store in C matrix_handle_t
- Note that there are an alternative second two stages for only doing symbolic computations, ie no values:
  - **get_compute_structure_buf_size** – return size of temporary buffer for compute_structure stage
  - **compute_structure** – do internal products for computing sparsity pattern of C (rowptr and col_ind arrays)
  - **finalize_structure** – (follows get_nnz) do any remaining internal products and transfer into final CSR sparsity pattern arrays

```
• namespace sparse {
    enum class matmat_request {
        get_work_estimation_buf_size,
        work_estimation,

        get_compute_buf_size,
        compute,
        get_nnz,
        finalize,

        get_compute_structure_buf_size,
        compute_structure,
        //(get_nnz)
        finalize_structure

    };
• }
```

Compute full matrix

Compute sparsity pattern of matrix

Simplifying the matmat API arguments

Introduce **"sparse::matrix_view_descr"** enum to describe the way the matrix data is to be viewed in the matmat operations

- The {ge, sy, tr, he, …} view types are encoded in other sparse blas apis where there is a single sparse matrix present (gemv, trsv, etc)
- Uplo::{lower,upper,full} values and diag::{nonunit,unit} enums are arguments to the other apis as appropriate for the view type.
- We combine the applicable cases to a single matrix view per matrix for the matmat API

```
namespace sparse {
enum class matrix_view_descr : std::int32_t {
    general,
    symmetric_lower,
    symmetric_lower_unit,
    symmetric_upper,
    symmetric_upper_unit,
    symmetric_full,
    symmetric_full_unit,
    triangular_lower,
    triangular_lower_unit,
    triangular_upper,
    triangular_upper_unit

// other configurations to be added as appropriate
extensions are made
};
}
```

We introduce an opaque pointer handle **"sparse::matmat_descr_t"** to house the mathematical description of the matrix-matrix product to be performed.

- It stores matrix_views for each matrix and the transpose operations to be applied.
- At some future point if the matrix product is extended (for instance to C = alpha op(A) * op(B) + beta D) would also house the alpha and beta coefficients.
- C-style init/release and standalone set/get accessors match existing sparse Blas and oneMKL design

```
struct matmat_descr;
typedef matmat_descr *matmat_descr_t;

void init_matmat_descr( matmat_descr_t *desc);

void release_matmat_descr( matmul_descr_t *desc);

void set_matmat_data(matmat_descr_t descr,
                     matrix_view_descr viewA,
                     transpose opA,
                     matrix_view_descr viewB,
                     transpose opB,
                     matrix_view_descr viewC);

void get_matmat_data(matmat_descr_t descr,
                     matrix_view_descr &viewA,
                     transpose   &opA,
                     matrix_view_descr &viewB,
                     transpose   &opB,
                     matrix_view_descr &viewC );
```

Matmat s*s->s Computation Workflow (buffer user allocation)

```
oneapi::mkl::sparse::request req;
oneapi::mkl::transpose opA = oneapi::mkl::transpose::nontrans;
oneapi::mkl::transpose opB = oneapi::mkl::transpose::nontrans;
oneapi::mkl::sparse::matrix_view_descr viewA =
oneapi::mkl::sparse::matrix_view_descr::general;
oneapi::mkl::sparse::matrix_view_descr viewB =
oneapi::mkl::sparse::matrix_view_descr::general;
oneapi::mkl::sparse::matrix_view_descr viewC =
oneapi::mkl::sparse::matrix_view_descr::general;

oneapi::mkl::sparse::matmat_descr_t descr;
oneapi::mkl::sparse::init_matmat_descr(&descr);
oneapi::mkl::sparse::set_matmat_data(descr, viewA, opA, viewB, opB, viewC);

// provide rowptr for C already as it's size is known a priori
oneapi::mkl::sparse::set_csr_data(C, c_nrows, c_ncols, ind, c_rowptr, nullptr, nullptr);

// Step 1.1  query for size of work_estimation temp buffer
sizeTempBufferBuf = new sycl::buffer<std::int64_t,1>(1);
if (!sizeTempBufferBuf) { throw std::bad_alloc(); }
req = oneapi::mkl::sparse::matmul_request::get_work_estimation_buf_size;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf, nullptr);

// Step 1.2 allocate temp buffer for work_estimation
{
    auto sizeTempBuffer_acc = sizeTempBufferBuf->template
get_access<sycl::access::mode::read>();
    size_t sizeTempBuffer = sizeTempBuffer_acc[0];
    tempBuffer = new sycl::buffer<std::byte,1>(sizeTempBuffer);
    if (!tempBuffer) { throw std::bad_alloc(); }
}

// Step 1.3  do work_estimation
req = oneapi::mkl::sparse::matmul_request::work_estimation;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf, tempBuffer);


// Step 2.1 query size of compute temp buffer
sizeTempBufferBuf2 = new sycl::buffer<std::int64_t,1>(1);
if (!sizeTempBufferBuf2) { throw std::bad_alloc(); }
req = oneapi::mkl::sparse::matmul_request::get_compute_buf_size;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf2, nullptr);

// Step 2.2 allocate temp buffer for compute
{
    auto sizeTempBuffer2_acc = sizeTempBufferBuf2->template
get_access<sycl::access::mode::read>();
    size_t sizeTempBuffer2 = sizeTempBuffer2_acc[0];
    tempBuffer2 = new sycl::buffer<std::byte,1>(sizeTempBuffer2);
    if (!tempBuffer2) { throw std::bad_alloc(); }
}
// Step 2.3 do compute
req = oneapi::mkl::sparse::matmul_request::compute;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf2,
tempBuffer2);
```

```
// Step 3.1  get nnz
c_nnzBuf = new sycl::buffer<std::int64_t,1>(1);
if (!c_nnzBuf) { throw std::bad_alloc(); }
req = oneapi::mkl::sparse::matmul_request::get_nnz;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, c_nnzBuf, nullptr);

// Step 3.2:  allocate c_col_ind and c_vals
{
    auto c_nnz_acc = c_nnzBuf->template get_access<sycl::access::mode::read>();
    std::int64_t c_nnz = c_nnz_acc[0];
    log_info("c_nnz = %ld", c_nnz);

    c_col_ind = new sycl::buffer<INT_TYPE,1>(c_nnz);
    c_vals = new sycl::buffer<DATA_TYPE,1>(c_nnz);
    if ( !c_col_ind || !c_vals) { throw std::bad_alloc(); }

    oneapi::mkl::sparse::set_csr_data(C, c_nrows, c_ncols,ind, *c_rowptr, *c_col_ind,
*c_vals);
}

// Step 3.3  finalize into C matrix
req = oneapi::mkl::sparse::matmul_request::finalize;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, nullptr, nullptr);


// Finish: clean up all arrays and objects
queue.wait();
oneapi::mkl::sparse::release_matmat_descr(&descr);
```

Matmat s*s->s Computation Workflow (USM user allocation)

Note: we must add event.waits() before USM allocations until USM is host thread safe, and we can add allocations into a host_task

```
oneapi::mkl::transpose opA = oneapi::mkl::transpose::nontrans;
oneapi::mkl::transpose opB = oneapi::mkl::transpose::nontrans;
oneapi::mkl::sparse::matrix_view_descr viewA =
oneapi::mkl::sparse::matrix_view_descr::general;
oneapi::mkl::sparse::matrix_view_descr viewB =
oneapi::mkl::sparse::matrix_view_descr::general;
oneapi::mkl::sparse::matrix_view_descr viewC =
oneapi::mkl::sparse::matrix_view_descr::general;

oneapi::mkl::sparse::request req;
oneapi::mkl::sparse::matmat_descr_t descr;
oneapi::mkl::sparse::init_matmat_descr(&descr);
oneapi::mkl::sparse::set_matmat_data(descr, viewA, opA, viewB, opB, viewC);
// provide rowptr for C already as it's size is known a priori
oneapi::mkl::sparse::set_csr_data(C, c_nrows, c_ncols, ind, c_rowptr, nullptr, nullptr);

// Step 1.1 query for size of work_estimation temp buffer
std::int64_t *sizeTempBufferBuf = static_cast<std::int64_t *>(malloc_shared(
1*sizeof(std::int64_t), dev, ctxt));
if (!sizeTempBufferBuf) { throw std::bad_alloc(); }
req = oneapi::mkl::sparse::matmul_request::get_work_estimation_buf_size;
sycl::event ev1 = oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf,
nullptr, {});

// Step 1.2 allocate temp buffer for work_estimation
std::int64_t sizeTempBuffer = 0;
sycl::event ev1_2 = queue.submit([&](sycl::handler &cgh) {
    cgh.depends_on({ev1});
    cgh.codeplay_host_task([&](){
        sizeTempBuffer = sizeTempBufferBuf[0];
    });
});
ev1_2.wait();
void *tempBuffer = malloc_shared( sizeTempBuffer*sizeof(std::byte), dev, ctxt);
if (!tempBuffer) { throw std::bad_alloc(); }

// Step 1.3  do work_estimation
req = oneapi::mkl::sparse::matmul_request::work_estimation;
sycl::event ev1_3 = oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr,
sizeTempBufferBuf, tempBuffer, {ev1_2});

// Step 2.1 query size of compute temp buffer
std::int64_t *sizeTempBufferBuf2 = static_cast<std::int64_t *>(malloc_shared(
1*sizeof(std::int64_t), dev, ctxt));
if (!sizeTempBufferBuf2) { throw std::bad_alloc(); }

req = oneapi::mkl::sparse::matmul_request::get_compute_buf_size;
sycl::event ev2 = oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf2,
nullptr, {ev1_3});
```

```
// Step 2.2 allocate temp buffer for compute
std::int64_t sizeTempBuffer2 = 0;
sycl::event ev2_2 = queue.submit([&](sycl::handler &cgh) {
    cgh.depends_on({ev2});
    cgh.codeplay_host_task([&](){
        sizeTempBuffer2 = sizeTempBufferBuf2[0];
    });
});
ev2_2.wait();
void *tempBuffer2 = malloc_shared( sizeTempBuffer2*sizeof(std::byte), dev, ctxt);
if (!tempBuffer2) { throw std::bad_alloc(); }

// Step 2.3 do compute
req = oneapi::mkl::sparse::matmul_request::compute;
sycl::event ev2_3 = oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, sizeTempBufferBuf2,
tempBuffer2, {ev2_2});

// Step 3.1  get nnz
std::int64_t *c_nnzBuf = static_cast<std::int64_t *>(malloc_shared( 1*sizeof(std::int64_t), dev,
ctxt));
if (!c_nnzBuf) { throw std::bad_alloc(); }
req = oneapi::mkl::sparse::matmul_request::get_nnz;
sycl::event ev3 = oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, c_nnzBuf, nullptr,
{ev2_3});

std::int64_t c_nnz = 0;
sycl::event ev3_2 = queue.submit([&](sycl::handler &cgh) {
    cgh.depends_on({ev5});
    cgh.codeplay_host_task([&](){
        c_nnz = c_nnzBuf[0];
        log_info("c_nnz = %ld", static_cast<std::int64_t>(c_nnz));
    });
});
ev3_2.wait();

// Step 3.2:  allocate c_col_ind and c_vals
INT_TYPE *c_col_ind = static_cast<INT_TYPE *>(malloc_shared((c_nnz)*sizeof(INT_TYPE), dev,
ctxt));
DATA_TYPE *c_vals = static_cast<DATA_TYPE *>(malloc_shared((c_nnz)*sizeof(DATA_TYPE), dev,
ctxt));
if ( !c_col_ind || !c_vals) { throw std::bad_alloc(); }
oneapi::mkl::sparse::set_csr_data(C, c_nrows, c_ncols, ind, c_rowptr, c_col_ind, c_vals);

// Step 3.3  finalize into C matrix
req = oneapi::mkl::sparse::matmul_request::finalize;
sycl::event ev3_3 = oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr, nullptr, nullptr,
{ev3_2});
// clean up afterwards, including release_matmat_descr(&descr);
```

Possible Future Extensions

Possible Future API Extensions

- Extend "Matmat functionality" to support library allocated C matrix memory

- Extend with a fused sparse matrix add:
- C = alpha * op(A) * op(B) + beta * D
- where D can be C (with caveat that the product does not change the sparsity pattern of C)

- Add full support for complex data and operations
- Add op(A) = A^H (conjugate transpose)
- Add op(A) = conj(A)   (conjugate)

- Add support for mixed precisions
- Data output is governed by precision of C
- Add a template argument to matmat API for the accumulation data type
- Example: matmat<double>(queue, A, B, C, req, descr, nullptr, nullptr); would do accumulation of products in double precision, and then static_cast results to C data type.

- Extend sparse::matmat to be a generic matrix-matrix multiplication API:
- Extend sparse::matrix_handle_t to cover both sparse and dense matrix formats
- Can cover any matrix * matrix -> output matrix multiplication that contains at least one sparse matrix
    - s*s->s : current proposal
- Possible extensions of matmat products (s = sparse, d = dense) :
    - s*d->d
    - d*s->d
    - s*s->d

- Extend to support other Sparse Matrix Formats: CSC, BSR, COO, …

Wrap-up

Next Steps

- Focuses for next meeting(s):
  - Multi-tile/multi-GPU considerations
  - Any topics from oneMKL TAB members?

- If anyone has content that they would like posted on oneAPI.com, please let us know

Version of oneAPI Specification	Date
1.1-provisional-rev-2	24 June 2021
1.1-provisional-rev-3	21 September 2021
1.1-rev-1	12 November 2021

Resources

- oneAPI Main Page: https://www.oneapi.com/
- Latest release of oneMKL Spec (currently v. 1.0): https://spec.oneapi.com/versions/latest/elements/oneMKL/source/index.html
- GitHub for oneAPI Spec: https://github.com/oneapi-src/oneAPI-spec
- GitHub for oneAPI TAB: https://github.com/oneapi-src/oneAPI-tab

- GitHub for open source oneMKL interfaces (currently BLAS, RNG, and LAPACK domains): https://github.com/oneapi-src/oneMKL