oneMKL Technical Advisory Board

Session 15

July 14, 2021

Agenda

- Welcoming remarks – 5 minutes
- Updates from last meeting – 5 minutes
- (Finish from last time) Overview of sparse matrix * sparse matrix API – Spencer Patty (10 minutes)
- Multi-tile/multi-GPU discussion - Maria Kraynyuk (30 minutes)
- Wrap-up and next steps – 5 minutes

Updates from last meeting

- Intel oneMKL 2021.3 was released

Overview of sparse matrix * sparse matrix API

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.

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, should be done before
compute() stage
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
oneapi::mkl::sparse::release_matmat_descr(&descr);

queue.wait();
```

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,  should be done before
compute() stage
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.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.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_1 = 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({ev3_1});
    cgh.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);
```

Matmat s*s->s Computation Workflow (buffer user allocation only for C, not temporary buffers)

```
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
// should be done before call to compute()
oneapi::mkl::sparse::set_csr_data(C, c_nrows, c_ncols, ind,
c_rowptr, nullptr, nullptr);

(!tempBuffer) { throw std::bad_alloc(); }
}

// Step 1  do work_estimation
req = oneapi::mkl::sparse::matmul_request::work_estimation;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr,
nullptr, nullptr);

// Step 2 do compute
req = oneapi::mkl::sparse::matmul_request::compute;
oneapi::mkl::sparse::matmat(queue, A, B, C, req, descr,
nullptr, nullptr);
```

```
// 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
oneapi::mkl::sparse::release_matmat_descr(&descr);

queue.wait();
```

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

```
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
// and should be done before compute()
oneapi::mkl::sparse::set_csr_data(C, c_nrows, c_ncols, ind,
c_rowptr, nullptr, nullptr);

// Step 1  do work_estimation
req = oneapi::mkl::sparse::matmul_request::work_estimation;
sycl::event ev1 = oneapi::mkl::sparse::matmat(queue, A, B, C, req,
descr, nullptr, nullptr, {});

// Step 2 do compute
req = oneapi::mkl::sparse::matmul_request::compute;
sycl::event ev2 = oneapi::mkl::sparse::matmat(queue, A, B, C, req,
descr, nullptr, nullptr, {ev1});
```

```
// 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_1 = oneapi::mkl::sparse::matmat(queue, A, B, C, req,
descr, c_nnzBuf, nullptr, {ev2});

std::int64_t c_nnz = 0;
sycl::event ev3_2 = queue.submit([&](sycl::handler &cgh) {
    cgh.depends_on({ev3_1});
    cgh.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});

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

queue.wait();
```

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, …

Multi-tile/Multi-GPU

Motivation

Current oneMKL API is targeted for single SYCL queue (device) only.

Need to define how oneMKL API can support more complex configurations:

- multi-tile devices;
- multi-GPUs.

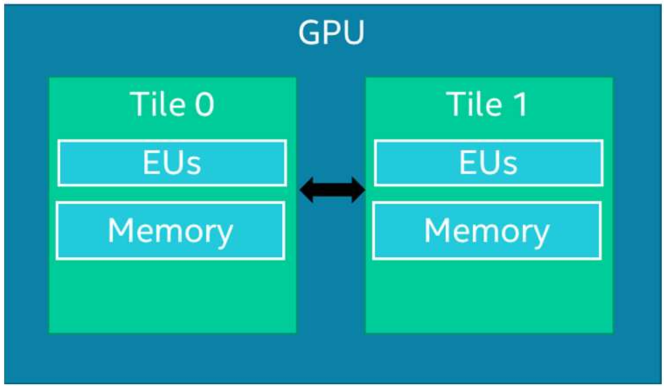
Outline

1. Multi-tile device usage modes overview
2. oneMKL API for multi-tile devices
3. Future consideration: oneMKL API extensions for multi-GPUs

Multi-tile Device Usage Modes

Implicit scaling mode: multi-tile device is treated as one SYCL device

- One context, memory allocations are automatically distributed across tiles.
- One queue, computations are automatically distributed across tiles



Explicit scaling mode: multi-tile device is partitioned into multiple SYCL devices for each tile with `create_sub_devices()`

- If one context, memory allocation is automatically distributed across tiles. If context per sub-device, memory allocations are pinned to each sub-device, no cross-tile communications.
- Queue per sub-device. Computations are pinned to each sub-device.

oneMKL API for Multi-tile Devices

Current oneMKL DPC++ API relies on input queue/memory and can be used for both multi-tile scaling modes from high-level application/plugin.

Implicit scaling

```
sycl::device dev = sycl::device(sycl::gpu_selector());
sycl::queue queue(dev);
fp *x = (fp *)malloc_shared(sizex * sizeof(fp), queue);
...
auto done = oneapi::mkl::blas::axpy(queue, n, alpha, x, incx, y, incy);
done.wait();
free(x, queue);
...
```

Explicit scaling

```
sycl::device dev = sycl::device(sycl::gpu_selector());
std::vector<sycl::device> subdev = {};
subdev = dev.create_sub_devices<partition_property::partition_by_affinity_domain>(
                              partition_affinity_domain::numa);
sycl::context cxt0({subdev[0]});
sycl::context cxt1({subdev[1]});
sycl::queue queue0(cxt0, subdev[0]);
sycl::queue queue1(cxt1, subdev[1]);

fp *x0 = (fp *)malloc_shared((sizex / 2) * sizeof(fp), queue0);
fp *x1 = (fp *)malloc_shared((sizex / 2) * sizeof(fp), queue1);
...
auto done0 = oneapi::mkl::blas::axpy(queue0, n/2, alpha, x0, incx, y0, incy);
auto done1 = oneapi::mkl::blas::axpy(queue1, n/2, alpha, x1, incx, y1, incy);
done0.wait();
done1.wait();
free(x0, queue0);
...
```

Future Consideration: oneMKL API Extensions for multi-GPUs

- Multi-GPU support can be implemented via oneMKL API for single queue with most control on the higher-level application/plugin.
- General API extension types for multi-GPUs to consider:

	Host Low-level API	Host High-level API
API modifications	Multiple SYCL queues as an input	No SYCL queue as an input, some options to control internal logic for devices/data management
Devices management	On the user side	On the library side
Data management	On the user side or runtime side if shared between devices	On the library side or runtime side if shared between devices
Pros	Doesn't require internal state	Easy to start with
Cons	• All devices/data management complexity is on the user side, almost like single device API • Not clear how library can handle data locations on different GPUs (different vendors)	• Requires internal state • Complex synchronization with the rest of application • Doesn't guarantee most optimal configuration for each user case
Example of API	clBLAS for AMD GPU	NVIDIA cuBLASXt

Future Consideration:
oneMKL API Extensions for multi-GPUs

Host Low-level API Example:

clBLAS for AMD GPU

```
clblasStatus clblasSgemm (
  clblasOrder order,
  clblasTranspose transA, clblasTranspose transB,
  size_t M, size_t N, size_t K,
  cl_float alpha,
  const cl_mem A, size_t offA, size_t lda,
  const cl_mem B, size_t offB, size_t ldb,
  cl_float beta,
  cl_mem C, size_t offC, size_t ldc,
  cl_uint numCommandQueues,
  cl_command_queue *commandQueues,
  cl_uint numEventsInWaitList,
  const cl_event *eventWaitList,
  cl_event *events
)
```

Host High-level API Example:

NVIDIA cuBLASXt

```
cublasStatus_t cublasXtSgemm (
  cublasXtHandle_t handle,
  cublasOperation_t transa, cublasOperation_t transb,
  size_t m, size_t n, size_t k,
  const float *alpha,
  const float *A, int lda,
  const float *B, int ldb,
  const float *beta,
  float *C, int ldc
)
// Additional control options
cublasXtDeviceSelect(cublasXtHandle_t handle ...)
cublasXtSetCpuRatio(cublasXtHandle_t handle ...)
cublasXtSetPinningMemMode(cublasXtHandle_t handle ...)
...
```

Future Consideration: oneMKL API Extensions for multi-GPUs

Summary and the next steps:

- In general, implementation/adoption for both API extension types is not clear
- We need to start with vendor specific API extension to see if such API can be generalized for oneMKL specification.

Wrap-up

Next Steps

- Focuses for next meeting(s):
  - 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