oneMKL Technical Advisory Board

Session 12

March 24, 2021

Agenda

- Welcoming remarks – 5 minutes
- Updates from last meeting – 5 minutes
- Overview of oneMKL FFT domain – Helen Parks (30 minutes)
- Wrap-up and next steps – 5 minutes

Updates from last meeting

- oneAPI Math Kernel Library (oneMKL) Interfaces Project
  - LBNL is adding support for hipRAND backend to support RNG domain on AMD GPUs
  - (Coming soon!) Adding interfaces for the LAPACK domain with support for Intel® oneAPI Math Kernel Library on CPUs and Intel GPUs
- oneAPI Specification version 1.1 - planned to be released in November
  - Extend Batch API for BLAS and LAPACK
  - Extend Sparse BLAS functionality
  - Extend Summary Statistics functionality
  - Refactor DFT
  - Add RNG device API

Overview of oneMKL FFT domain

oneMKL DPC++ FFT overview

- oneMKL DPC++ FFT API mimics Intel's Discrete Fourier Transform Interface ("DFTI interface") from oneMKL C/Fortran
  - We have FFTW3 API support for C/Fortran targeting CPU and GPU. GPU support uses the same APIs and OpenMP offload.
  - No plans for FFTW3 API in DPC++.
- Support for 1D, 2D, and 3D data
- Forward transforms of real or complex data
  - Backward domain is always complex (with conjugate-even symmetry in the case of real forward data)
- Support for SYCL buffer-based and USM data

DFTI four-step workflow

1. Create descriptor
   - Descriptor objects store information describing the FFT computation
   - A few parameters are fixed at descriptor construction
2. Configure descriptor
   - More parameters can be set through calls to descriptor::set_value (one call per parameter to set)

```
using namespace oneapi::mkl::dft;

// 1. Create descriptor
descriptor<precision::DOUBLE, domain::COMPLEX> desc(N);

// 2. Configure descriptor
desc.set_value(config_param::BACKWARD_SCALE, (double)(1.0/N));

// 3. Commit descriptor
desc.commit(queue);

// 4. Compute
compute_forward(desc, xBuffer);
```

DFTI four-step workflow

3. Commit descriptor
   - Takes a sycl::queue as input
   - Commit does one-time initialization work based on the configuration and device info deduced from the queue
4. Compute
   - Multiple calls to compute will reuse the work done at commit
   - All compute work is dispatched to the queue used at commit

```
using namespace oneapi::mkl::dft;

// 1. Create descriptor
descriptor<precision::DOUBLE, domain::COMPLEX> desc(N);

// 2. Configure descriptor
desc.set_value(config_param::BACKWARD_SCALE, (double)(1.0/N));

// 3. Commit descriptor
desc.commit(queue);

// 4. Compute
compute_forward(desc, xBuffer);
```

Spec overview

Three components: descriptor class, free functions for compute, and configuration enums

```
template <precision prec, domain dom>
class descriptor {
 public:
    descriptor(std::vector<std::int64_t> dimensions);
    descriptor(std::int64_t length);
    ~descriptor();

    void commit(cl::sycl::queue &in);

    void set_value(config_param param, ...);
    void get_value(config_param param, ...);
}
```

```
template<typename descriptor_type, typename data_type>
void compute_forward(
    descriptor_type &desc,
    cl::sycl::buffer<data_type, 1> &inout);

// More variants for forward/backward, in-place/out-of-place,
// and buffer/USM data inputs
```

```
enum class precision {
    SINGLE = DFTI_SINGLE,
    DOUBLE = DFTI_DOUBLE
};

enum class domain {
    REAL = DFTI_REAL,
    COMPLEX = DFTI_COMPLEX
};

enum class config_param {
    FORWARD_SCALE,
    NUMBER_OF_TRANSFORMS,
    COMPLEX_STORAGE,
    PLACEMENT,
    INPUT_STRIDES,
    FWD_DISTANCE,
    // and more…
};

enum class config_value {
    COMMITTED,
    UNCOMMITTED,
    COMPLEX_COMPLEX,
    REAL_COMPLEX,
    INPLACE,
    NOT_INPLACE,
    // and more…
};
```

oneMKL 1D FFT snippet, using SYCL buffer

```
#include “oneapi/mkl/dfti.hpp”
using namespace oneapi::mkl::dft;

cl::sycl::queue queue(dev, exception_handler);
double* x =
    (double*) mkl_malloc(N*2*sizeof(double), 64);
init(x, N, harmonic);
cl::sycl::buffer<double, 1>
    xBuffer(x, cl::sycl::range<1>(N*2));

descriptor<precision::DOUBLE, domain::COMPLEX> desc(N);
desc.set_value(config_param::BACKWARD_SCALE, 1.0/N);
desc.commit(queue);
compute_forward(desc, xBuffer);
```

DFT DPC++ header

DFT DPC++ namespace

Initialize input data, then use it to initialize SYCL buffer

Descriptor object is templated on data precision and forward domain data type

Descriptor constructor takes in size of FFT (scalar for 1D, vector for 2D and 3D)

The descriptor `commit` method may use device information from the input SYCL queue in its setup work

The type of the value argument in `set_value` depends on which `config_param` is being set

Compute functions dispatch work onto the committed queue

oneMKL 1D FFT snippet, buffer vs USM

SYCL buffer

```
#include “oneapi/mkl/dfti.hpp”
using namespace oneapi::mkl::dft;

cl::sycl::queue queue(dev, exception_handler);
double* x =
    (double*) mkl_malloc(N*2*sizeof(double), 64);
init(x, N, harmonic);
cl::sycl::buffer<double, 1>
    xBuffer(x, cl::sycl::range<1>(N*2));

descriptor<precision::DOUBLE, domain::COMPLEX> desc(N);
desc.commit(queue);
compute_forward(desc, xBuffer);
```

Buffer compute functions have no return value. The user can use the queue and/or data accessors to wait for oneMKL.

USM

```
#include “oneapi/mkl/dfti.hpp”
using namespace oneapi::mkl::dft;

cl::sycl::queue queue(dev, exception_handler);
double *x_usm =
    (double*) malloc_shared(N*2*sizeof(double),
                            queue.get_device(),
                            queue.get_context());
init(x_usm, N, harmonic);

descriptor<precision::DOUBLE, domain::COMPLEX> desc(N);
desc.commit(queue);
cl::sycl::event fwd = compute_forward(desc, x_usm);
fwd.wait();
```

USM compute functions return the last event, so the user can wait for oneMKL completion.

oneMKL 2D batched FFT snippet, buffer

```
#include “oneapi/mkl/dfti.hpp”
using namespace oneapi::mkl::dft;

float* in = (float*) mkl_malloc(BATCH*N2*N1*2*sizeof(float), 64);
float* out = (float*) mkl_malloc(BATCH*N2*N1*2*sizeof(float), 64);
init(in, N1, N2, BATCH, H1, H2);
cl::sycl::buffer<float, 1> inBuffer(in, cl::sycl::range<1>(BATCH*N2*N1*2));
cl::sycl::buffer<float, 1> outBuffer(out, cl::sycl::range<1>(BATCH*N2*N1*2));

descriptor<precision::SINGLE, domain::COMPLEX> desc({N2, N1});
desc.set_value(config_param::NUMBER_OF_TRANSFORMS, BATCH);
desc.set_value(config_param::FWD_DISTANCE, N1*N2);
desc.set_value(config_param::BWD_DISTANCE, N1*N2);
desc.set_value(config_param::PLACEMENT, DFTI_NOT_INPLACE);
compute_forward(desc, inBuffer, outBuffer);
```

The `FWD_DISTANCE` and `BWD_DISTANCE` parameters define the distance **in elements** between the multiple FFTs. Here, distance is N1*N2 complex elements, so 2*N1*N2 `float`s in `inBuffer`.

Overloaded `descriptor` constructor takes a vector of sizes in 2D and 3D cases

Each compute call will run `BATCH` FFTs. `BATCH` should be `std::int64_t`.

Intel oneMKL reuses enum parameter values from the C interface. **The DPC++ oneMKL spec defines `config_value` enum class (`config_value::NOT_INPLACE` here).**

Overloaded `compute_forward` takes two buffers (or two USM pointers) for out-of-place computation.

oneMKL 3D real FFT snippet

```
#include “oneapi/mkl/dfti.hpp”
using namespace oneapi::mkl::dft;

std::int64_t fwd_real_strides[4] = {0, N2*(N1/2+1)*2, (N1/2+1)*2, 1};
std::int64_t bwd_complex_strides[4] = {0, N2*(N1/2+1), (N1/2+1), 1};

descriptor<precision::SINGLE, domain::REAL> desc({N3, N2, N1});
desc.set_value(config_param::INPUT_STRIDES, fwd_real_strides);
desc.set_value(config_param::OUTPUT_STRIDES, bwd_complex_strides);
desc.commit(queue);
compute_forward(desc, xBuffer);

desc.set_value(config_param::INPUT_STRIDES, bwd_complex_strides);
desc.set_value(config_param::OUTPUT_STRIDES, fwd_real_strides);
desc.commit(queue);
compute_backward(desc, xBuffer);
```

Strides describe the data layout within a single FFT with a (d+1)-length vector.
X(k1,k2,k3) = x[s0 + k1*s1 + k2*s2 + k3*s3]

As with distances, strides are expressed in elements of the relevant real or complex domain. Hence for real-to-complex transforms, the input and output strides will almost always be different.

Input and output domains are different for the backward transform, so you must reset the strides and recommit before the backward transform.

Future directions

- There is room in the spec for managing scratch workspace on devices
  - Feature is available in NVIDIA's cuFFT.
  - Not implemented in Intel oneMKL. Spec may need to adjust with feedback from implementation.
- The current spec is based on DFTI interface, but the goal is generality and accessibility for a wide user base
  - Commit/compute workflow is similar to plan/compute workflow of popular FFT libraries (FFTW, cuFFT, clFFT)
  - Anticipate the need to expand and adjust the spec to encourage adoption by current users of non-DFTI APIs (e.g. support input data formats of popular libraries)

Questions for TAB

- Any feedback on specific expansions whose absence would deter users of FFTW, cuFFT, etc. from adopting DPC++
- Should we anticipate users running many queues of the same FFT on the same hardware?
  - Spec requires multiple descriptors and multiple commits in this case.
- Is there a strong need for device APIs that can embed in user kernels?
  - cuFFT has a preview of this feature
  - Not currently in spec, Intel oneMKL, or plans

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-1	25 March 2021
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 (soon) LAPACK domains): https://github.com/oneapi-src/oneMKL