oneMKL Technical Advisory Board

Session 13

May 19, 2021

Agenda

- Welcoming remarks – 5 minutes
- Updates from last meeting – 10 minutes
- Overview of device APIs for Random Number Generator domain – Alina Elizarova (30 minutes)
- Wrap-up and next steps – 5 minutes

oneMKL TAB Members

- Hartwig Anzt, Karlsruhe Institute of Technology (KIT)
- Romain Dolbeau, SiPearl
- Vincent Pascuzzi, Lawrence Berkeley National Laboratory
- *Harry Waugh, University of Bristol* - stepped down

- Mehdi Goli, Codeplay
- Mark Hoemmen, Stellar Science
- Nevin Liber, Argonne National Laboratory (ANL)
- Piotr Luszczek, Innovative Computing Laboratory (ICL) at University of Tennessee, Knoxville (UTK)
- Pat Quillen, MathWorks
- Nichols Romero, ANL
- Edward Smyth, Numerical Algorithms Group (NAG)

- Brief intro: your job; how you use math libraries

Updates from last meeting

- oneAPI Math Kernel Library (oneMKL) Interfaces Project
  - (Coming soon!) Adding interfaces for the LAPACK domain with support for Intel® oneAPI Math Kernel Library on CPUs and Intel GPUs
  - Blog post by Vincent Pascuzzi: Lawrence Berkeley National Laboratory drives heterogenous computing with oneAPI's Math Kernel Library (oneMKL) – oneMKL Random Number Generators Domain now supports Nvidia GPUs

Overview of device APIs for Random Number Generator domain

Terminology

- Host-side APIs:

DPC++ kernels are submitted inside of the library. User passes sycl::queue object to library functions to choose device/control async execution. Library responsible for dispatching, kernels submitting and parallelization.

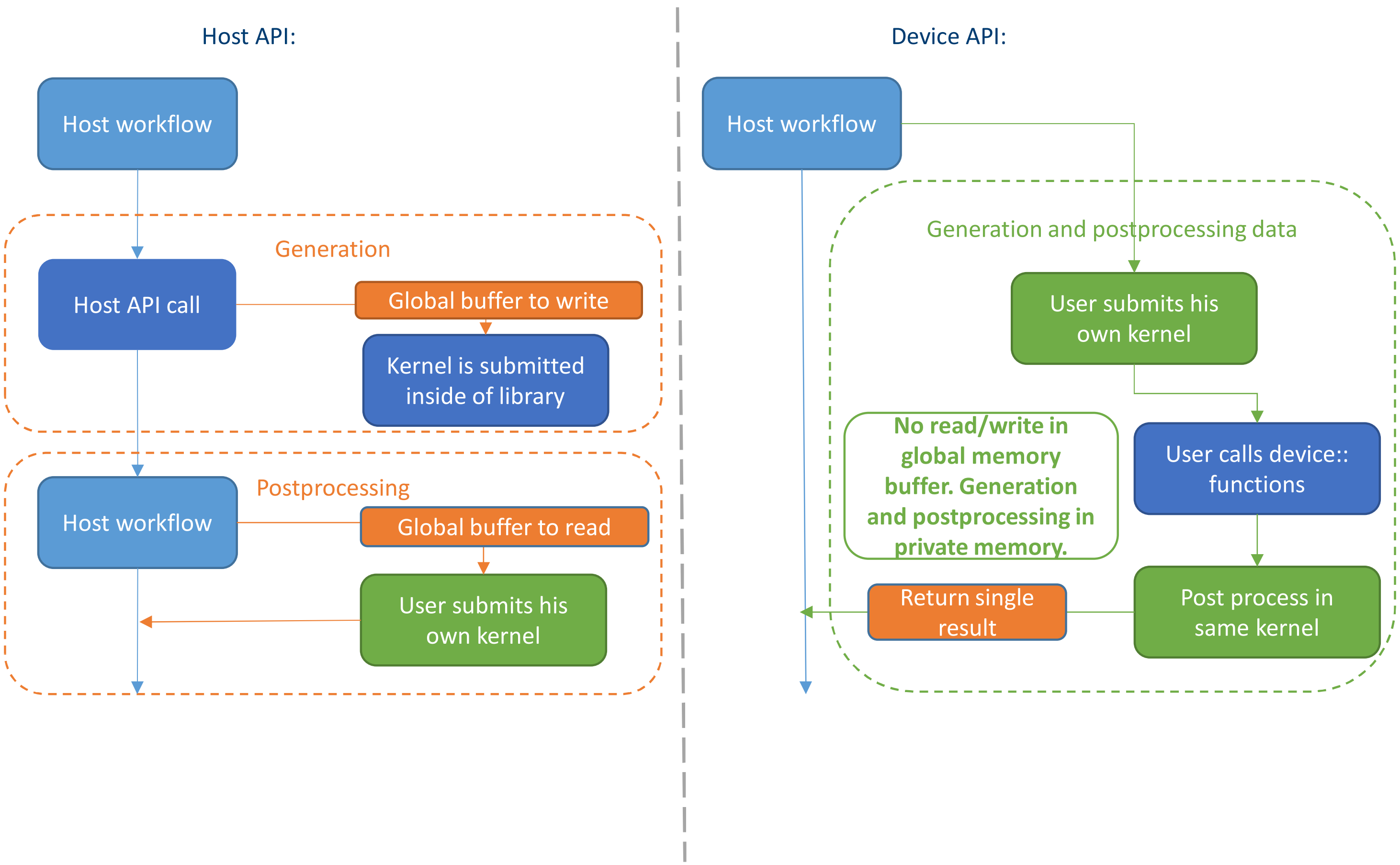
```
void oneapi::mkl::domain::function(sycl::queue& queue, …) {
    if(queue.get_device().is_gpu()) {// optimized gpu path }
    else if (queue.get_device().is_cpu()) { // optimized cpu path }
    else { throw  oneapi::mkl::device_not_supported(); }
}
```

- Device-side APIs:

APIs which can be called from user's kernels. User fully controls parallelization, data management, etc.

```
queue.submit([&](sycl::handler& cgh){
    cgh.parallel_for<class user_kernel>(sycl::range<1>{n},
    [=](sycl::item<1> item) {
        oneapi::mkl::domain::device::function(item, ...);
    });
});
```

Motivation to Have Device APIs



- The generation of random numbers is not the end goal for most applications.
- For Monte-Carlo simulations, random numbers need the following postprocessing, which in most cases includes reduction.
- For memory-bound RNG algorithms usage of global memory takes ~70% overhead.

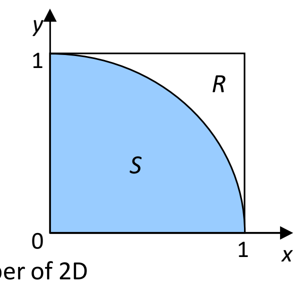
oneMKL RNG Device APIs Example

```cpp
#include "oneapi/mkl/rng/device.hpp“

constexpr int n = 1000; // Vector size
constexpr std::uint64_t seed = 777; // Seed for engine
constexpr int vec_size = 4; // Vector size for engine

int main() {
    sycl::buffer<float> r_buf(sycl::range<1>{n});
    sycl::queue{}.submit([&](sycl::handler& cgh) {
        sycl::accessor r_acc(r_buf, cgh, sycl::write_only);
        cgh.parallel_for(sycl::range<1>(n / vec_size),
            [=](sycl::item<1> item) {
            using namespace oneapi::mkl::rng::device;
            philox4x32x10<vec_size> engine(seed, item.get_id(0) * vec_size);
            uniform distr;
            sycl::vec<float, vec_size> res = generate(distr, engine);
            res.store(item.get_id(0), r_acc);
        });
    });
    //...
}
```

- RNG device engines may provide vector(sycl::vec) or scalar random output via generate function;
- RNG device APIs may be used both in host and device-side code;
- Device APIs allow generating random numbers with the following post-processing within a single kernel, without global memory data transfer.

Pi Number Evaluation Example

$$\Pr(c \in S) = \frac{Area(S)}{Area(R)} = \pi/4$$



$$\pi \cong \frac{4k}{n}$$

Where n – total number of 2D points, *k* – number of points which are fallen in S: $x^2 + y^2 \le 1$

Device API:

```
double estimate_pi(sycl::queue& queue, size_t n_points) {
    size_t n_under_curve = 0;
    {
        sycl::buffer<size_t, 1> count_buf(&n_under_curve, 1);

        queue.submit([&](sycl::handler& cgh) {
            sycl::accessor count_acc(count_buf, cgh, sycl::write_only);
            cgh.parallel_for(sycl::range<1>(n_points / (count_per_thread * vec_size / 2)),
            [=](sycl::item<1> item) {
                size_t id_global = item.get_id(0);
                sycl::atomic<size_t> atomic_counter{count_acc.get_pointer()};
                size_t count = 0;
                oneapi::mkl::rng::device::philox4x32x10<vec_size> engine(seed,
                                     id_global * vec_size * count_per_thread);
                oneapi::mkl::rng::device::uniform distr;
                for (int i = 0; i < count_per_thread; i++) {
                    auto r = oneapi::mkl::rng::device::generate(distr, engine);
                    for(int i = 0; i < vec_size / 2; i++) {
                        if (r[2*i] * r[2*i] + r[2*i + 1] * r[2*i + 1] <= 1.0f)
                            count += 1;
                    }
                }
                atomic_counter.fetch_add(count);
            });
        });
    }
    return n_under_curve / ((double)n_points) * 4.0;
}
```

Generation

Postprocess

Host API:

```
double estimate_pi(sycl::queue& queue, size_t n_points) {
    size_t n_under_curve = 0;
    sycl::buffer<float, 1>& rng_buf(n_points * 2);
    oneapi::mkl::rng::philox4x32x10 engine(queue);
    oneapi::mkl::rng::uniform distr;
    oneapi::mkl::rng::generate(distr, engine, n_points * 2, rng_buf);
    {
        sycl::buffer<size_t, 1> count_buffer(&n_under_curve, 1);

        queue.submit([&](sycl::handler& cgh) {
            sycl::accessor rng_acc(rng_buf, cgh, sycl::read_only);
            sycl::accessor count_acc(count_buf, cgh, sycl::write_only);
            cgh.parallel_for(sycl::range<1>(n_points / (count_per_thread * vec_size / 2)),
                [=](sycl::item<1> item) {
                    size_t id_global = item.get_id(0);
                    sycl::vec<float, 4> r;
                    sycl::atomic<size_t> atomic_counter{ count_acc.get_pointer() };
                    size_t count = 0;
                    for (int i = 0; i < count_per_thread; i++) {
                        r.load(i + id_global * count_per_thread, rng_acc.get_pointer());
                        for(int i = 0; i < vec_size / 2; i++) {
                            if (r[2*i] * r[2*i] + r[2*i + 1] * r[2*i + 1] <= 1.0f) {
                                count += 1;
                            }
                        }
                    }
                    atomic_counter.fetch_add(count);
                });
        });
    }
    return n_under_curve / ((double)n_points) * 4.0;
}
```

oneMKL RNG Device APIs Engines Classes Templates

```
namespace oneapi::mkl::rng::device {
template <std::int32_t VecSize = 1>
class philox4x32x10 {
public:
    static constexpr std::uint64_t default_seed = 0;
    static constexpr std::int32_t vec_size = VecSize;

    philox4x32x10();

    philox4x32x10(std::uint64_t seed, std::uint64_t offset = 0);

    philox4x32x10(std::initializer_list<std::uint64_t> seed,
                  std::uint64_t offset = 0);

    philox4x32x10(std::uint64_t seed,
                  std::initializer_list<std::uint64_t> offset);

    philox4x32x10(std::initializer_list<std::uint64_t> seed,
                  std::initializer_list<std::uint64_t> offset);
};
}
```

- All RNG device APIs are in oneapi::mkl::rng::device namespace
- Engines classes have VecSize template parameter, as sycl::vec (available sizes 1, 2, 3, 4, 8, 16).
- The offset parameter is to perform skip ahead while creating an engine for several parallelization techniques.
- Currently available philox4x32x10 and mrg32k3a engines.

oneMKL RNG Device APIs Distributions Classes Templates

```
namespace oneapi::mkl::rng::device {

namespace uniform_method {
struct standard {};
struct accurate {};
using by_default = standard;
}

template <typename Type, typename Method = uniform_method::by_default>
class uniform {
public:
    using method_type = Method;
    using result_type = Type;
    struct param_type;

    uniform();
    explicit uniform(Type a, Type b);
    explicit uniform(const param_type& pt);

    Type a() const;
    Type b() const;
    param_type param() const;
    void param(const param_type& pt);
};
}
```

- Distribution's device APIs are almost like Host APIs. But there may be a different set of distribution/different methods supported.
- Currently available distributions:
  - Uniform (standard, accurate)
  - Gaussian (box_muller2)
  - Lognormal (box_muller2)
  - Exponential (icdf, icdf_accurate)
  - Poisson (devroye)
  - bits

oneMKL RNG Device APIs Generate Routine

```
namespace oneapi::mkl::rng::device {

template <typename Distr, typename Engine>
auto generate(Distr& distr, Engine& engine) ->
    typename std::conditional<Engine::vec_size == 1,
       typename Distr::result_type,
       sycl::vec<typename Distr::result_type, Engine::vec_size>>::type;

template <typename Distr, typename Engine>
typename Distr::result_type generate_single(Distr& distr, Engine& engine);

}
```

- Generate function returns scalar or sycl::vec output depending on the engine's vec_size parameter.
- The vector output generation is vectorized wherever it's possible.
- Generate_single is used for scalar output of vector engines (can be used for 'tail' generation).
- Note: Distribution object is passed in function not as const&, unlike host APIs, as it may store some information different for each thread.

oneMKL RNG Device APIs Host-side States Allocation Example

```
sycl::buffer<oneapi::mkl::rng::device::mrg32k3a<vec_size>> engine_buf(range);

queue.submit([&](sycl::handler& cgh) { // initialize rng
    sycl::accessor engine_acc(engine_buf, cgh, sycl::write_only);
    cgh.parallel_for(range, [=](sycl::item<1> item) {
        size_t id = item.get_id(0);
        oneapi::mkl::rng::device::mrg32k3a<vec_size> engine(seed, {0, item});
        engine_acc[id] = engine;
    });
});

queue.submit([&](sycl::handler& cgh) { // generate random numbers
    sycl::accessor r_acc(r_buf, cgh, sycl::write_only);
    sycl::accessor engine_acc(engine_buf, cgh, sycl::read_write);
    cgh.parallel_for(range, [=](sycl::item<1> item) {
        size_t id = item.get_id(0);
        auto engine = engine_acc[id];
        oneapi::mkl::rng::device::uniform<Type> distr;
        auto res = oneapi::mkl::rng::device::generate(distr, engine);

        res.store(id, r_acc);
        engine_acc[id] = engine;
    });
});
```

- User can manually allocate global memory to store engines via buffers or USM to keep generators' states between kernels;
- The approach may be used in the following scenarios:
  - User has several kernels and wants to continue generation from the same state in each kernel;
  - Initialization for generators may be done at the separate kernel out of the main computational block.

oneMKL RNG Device APIs Host-side Helpers Example

```
oneapi::mkl::rng::device::engine_descriptor<oneapi::mkl::rng::device::
mrg32k3a> descr(queue, range, seed, offset);

queue.submit([&](sycl::handler& cgh) {
    auto r_acc = r_buf.template get_access<sycl::access::mode::write>(
cgh);
    auto engine_acc = descr.get_access(cgh);
    cgh.parallel_for(range, [=](sycl::item<1> item) {
        size_t id = item.get_id(0);
        auto engine = engine_acc.load(id);
        oneapi::mkl::rng::device::uniform<Type> distr;

        Type res = oneapi::mkl::rng::device::generate(distr, engine);

        r_acc[id] = res;
        engine_acc.store(engine, id);
    });
});
```

- Additional interface to avoid manual buffer creation and engine initialization.

oneMKL RNG Device APIs Host-side Helpers Classes

```
namespace oneapi::mkl::rng::device {

template <typename EngineType>
class engine_accessor {
public:
    EngineType load(size_t id) const;

    void store(EngineType engine, size_t id) const;
};

template <typename EngineType>
class engine_descriptor {
public:
    engine_descriptor(sycl::queue& queue, sycl::range<1> range,
                      std::uint64_t seed, std::uint64_t offset);

    template <typename InitEngineFunc>
    engine_descriptor(sycl::queue& queue, sycl::range<1> range,
                      InitEngineFunc func);

     engine_accessor<EngineType> get_access(sycl::handler& cgh);
};

}
```

- For simple offset case engines would be initialized in a kernel, submitted in engine_descriptor's constructor as offset * id
- For complex case user may provide InitEngineFunc functor, for example:

```
oneapi::mkl::rng::device::engine_descriptor<Engine> descr(queue, range,
[=](sycl::item<1> item) {
    return Engine(seed, {0, item.get_id(0)});
});
```

Engines would be initialized as subsequences with 2^64 offset.

Next Steps

- Add RNG device APIs into oneMKL specification 1.1.
- Extend Engines and Distributions set.
- Think of ESIMD extension support (sycl::ext::intel::experimental::esimd).

Next Steps for oneMKL TAB

- Focuses for next meeting(s):
  - Multi-tile/multi-GPU considerations
    - Quick question: Any experience with multi-GPU applications/libraries?
  - Sparse matrix * sparse matrix functionality
  - 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 (soon) LAPACK domains): https://github.com/oneapi-src/oneMKL