oneAPI Data Parallel C++ Library

For the DPC++ Technical Advisory Board discussion

May 2020

Agenda

1. Top-level namespace & include directory for oneAPI (cont.)
2. oneDPL execution policies
3. Algorithms: Synchronous vs. Asynchronous
4. Algorithms: Range-based API
5. (if time remains) Extension API overview

1. Top-level namespace for oneAPI

- Following the recommendation from the previous TAB meeting, we plan to introduce a common top-level namespace for oneAPI libraries
- Two related questions we would like to hear your feedback on:
  a) The preferred name for the namespace
  b) The preferred choice of top-level include directories

Naming options

- namespace oneapi { … }
  - Obvious correlation with oneAPI
  - Less chances of collision with other APIs
- namespace one { … }
  - Short and easy to read
  - Together with next-level (library) namespace, better matches the library short names and APIs using those (e.g. version macros)

```
oneapi::std::for_each(oneapi::dpl::execution::gpu, oneapi::dpl::begin(buf), oneapi::dpl::end(buf),
    […](…){ /*code here*/ });
```

```
one::std::for_each(one::dpl::execution::gpu, one::dpl::begin(buf), one::dpl::end(buf),
    […](…){ /*code here*/ });
```

```
using namespace oneapi; // or one
std::for_each(dpl::execution::gpu, dpl::begin(buf), dpl::end(buf),
    […](…){ /*code here*/ });
```

Top level include directory

- The include structure often mirrors the namespace structure
- **Do you see value in *identically named* top level include directories?**

```
#include “dpl/algorithm”
#include “tbb/parallel_for.h”
#include “mkl/blas/blas.hpp”
```

```
#include “onedpl/algorithm”
#include “onetbb/parallel_for.h” // backward incompatible
#include “onemkl/blas/blas.hpp”
```

```
#include “oneapi/dpl/algorithm”
#include “oneapi/tbb/parallel_for.h” // backward incompat.
#include “oneapi/mkl/blas/blas.hpp”
// or
#include “one/dpl/algorithm”
#include “one/tbb/parallel_for.h” // backward incompat.
#include “one/mkl/blas/blas.hpp”
```

```
onedpl/major.minor/
    include/
        oneapi/
            dpl/
                algorithm, …
    test/
    …
onetbb/major.minor/
    include/
        oneapi/
            tbb -> ../tbb
        tbb/
            parallel_for.h, …
    src/
    …
```

2. oneDPL execution policies

```
template <typename KernelName = /*unspecified*/>
class device_policy //: public parallel_unsequenced_policy
{
public:
    device_policy();
    explicit device_policy( sycl::queue queue );
    explicit device_policy( sycl::device device );
    template <typename OtherName>
    device_policy( const device_policy<OtherName>& policy );

    sycl::queue queue() const; // also considering implicit conversion
};
```

Examples:

```
using namespace one::dpl::execution;
device_policy<class my_kernel_name> policy{sycl::gpu_selector{}};
auto pol = make_device_policy(sycl::queue{});
```

Predefined execution policies

- Idea: create predefined policy objects for commonly used devices
  - similar to std::execution::{par, seq, unseq, par_unseq}
- **Which way of naming do you like more?**

Verbose (4 variations)

```
namespace one::dpl::execution {
    inline device_policy<> default_policy; //1
    … cpu_policy_v {sycl::cpu_selector{}}; //2
    … gpu_pol      {sycl::gpu_selector{}}; //3
    … policy_gpu   {sycl::gpu_selector{}}; //4
}
...
std::reduce(default_policy, …);         //1
std::for_each(cpu_policy_v, …);         //2
std::sort(dpl::execution::gpu_pol, …); //3
```

Concise (similar to the standard)

```
namespace one::dpl::execution {
    inline device_policy<> deflt; // maybe dev?
    … cpu {sycl::cpu_selector{}};
    … gpu {sycl::gpu_selector{}};
}

...
std::reduce(deflt, …);
std::for_each(cpu, …);
std::sort(dpl::execution::gpu, …);
```

3. Synchronous vs. asynchronous

- Now oneDPL algorithms with DPC++ execution policies vary in behavior
- Synchronous: the function waits until execution completes on the device
  - Standard-compliant; in some cases, may transfer the data back to the host
  - Used for function that return a value: reduce, find, …
- Implicitly Asynchronous: the function submits a kernel and returns
  - Non-standard; requires explicit data transfer or waiting on the queue
  - More efficient: allows SYCL to build kernel graphs/pipelines
  - Used for functions with no return value: for_each, transform, sort, …
- Long-term plans are to add explicitly asynchronous APIs
  - Thrust went through this: v1.9.4 made a breaking change for all “std-like” calls to block, and introduced asynchronous algorithms

Options for oneDPL specification v1.0

a) Describe the current behavior: specify some algorithms as blocking and some as non-blocking

b) Allow either blocking or non-blocking implementation, except where the semantics requires a certain behavior

c) Require all algorithms to block, compliant with C++ standard

- Makes the initial oneDPL release not compliant OR losing performance

d) Same as c), plus define asynchronous APIs

- Risks making it wrong due to lack of implementation/prototype expertise

Which option does seem right to you?

4. Range-based API for algorithms

- C++20 adds Ranges into the C++ standard library
  - Very powerful and expressive functional API
  - But does not yet support execution policies
- We work on adding range support for oneDPL algorithms
  - Only for a subset of the standard algorithms and views
  - Not fully standard-compliant (not based on concepts, no projections, …)

Ranges: programmability and kernel fusion

A pipeline of 3 kernels:

```
std::reverse(pol, begin(data), end(data));
std::transform(pol, begin(data), end(data), begin(result), lambda1);
auto res = std::find_if(pol, begin(result), end(result), pred);
```

With fancy iterators (1 kernel):

```
auto res = std::find_if(pol,
               make_transform_iterator(make_reverse_iterator(end(data)), lambda1),
               make_transform_iterator(make_reverse_iterator(begin(data)), lambda1),
               pred);
```

With ranges (1 kernel):

```
auto res = one::dpl::find_if(pol,
               views::all(data) | views::reverse() | views::transform(lambda1), pred);
```

Ranges: planned scope (23 algorithms)

<algorithm>

for_each
find
find_if_not
find_if
search
search_n
min_element
max_element
minmax_element

copy
transform
sort
stable_sort
partial_sort
partial_sort_copy
is_sorted_until
is_sorted

<numeric>

reduce
transform_reduce
exclusive_scan
transform_exclusive_scan
inclusive_scan
transform_inclusive_scan

The set of range views is being defined

Range-based API for algorithms (summary)

- C++20 adds Ranges into the C++ standard library
  - Very powerful and expressive functional API
  - But not yet for algorithms with execution policies
- We work on adding range support for oneDPL algorithms
  - Only for a subset of the standard algorithms and views
  - Not fully standard-compliant (not based on concepts, no projections, …)
  - **Are any important algorithms missed in the minimal subset?**
- For the oneDPL specification, the options are:
  a) Add these APIs to the oneDPL specification v1.0, or
  b) Leave it to a later version (i.e. implement as extensions in Intel's oneDPL)
  - **Which option does seem right to you?**

5. oneDPL extension APIs (as of now)

reduce_by_segment	Partial reductions on a sequence of values, by segments of equal keys
inclusive_scan_by_segment exclusive_scan_by_segment	Partial prefix scans on a sequence of values, by segments of equal keys
binary_search lower_bound upper_bound	Binary search variations for multiple values in the same sequence ("vectorized" search)
identity, maximum, minimum	functional utility classes
zip_iterator	Iterates over multiple sequences simultaneously
transform_iterator	Applies a function to each element in a sequence
permutation_iterator	Iterates over a sequence of values in the order set by a sequence of indices
counting_iterator	"Iterates" over a virtual sequence of numbers (holds a counter)
begin, end	Functions to pass a SYCL buffer to parallel algorithms (return an object holding a position in the buffer)

The principles of adding new extension APIs

- <u>Functionality:</u> the API is required to support a desired use case
- <u>Complexity</u>: desired API semantics does not allow a “trivial” mapping to existing APIs
- <u>Performance</u>: trivial mapping to existing APIs is not performant, and optimizations make it non-trivial
- <u>Convenience</u>: meaningful and commonly used API name (even with a simple mapping)
- <u>Consistency</u>: significant semantical similarity with APIs selected by other criteria
- <u>Explicit demand</u> with a reasonable justification