oneAPI Data Parallel C++ Library

For the DPC++ Technical Advisory Board discussion

April 2021

Agenda

1. oneAPI Data Parallel C++ Library (oneDPL) recap
2. Notable oneDPL changes
3. Experimental Range-based API
4. Experimental Asynchronous API
5. Open questions

oneAPI Data Parallel C++ Library: recap

- Complements Data Parallel C++ (DPC++) with the functionality similar to the standard C++ library
  - Extended set of STL algorithms with standard-like & special execution policies
  - “Extended subset” of standard C++ API supported in DPC++ kernels
- Goal is greater productivity and portability (comparing to “pure DPC++”)
- Specification: https://spec.oneapi.com/versions/latest/elements/oneDPL/source/index.html
- Open source implementation: https://github.com/oneapi-src/oneDPL
  - Based on LLVM PSTL project
  - Contributions are welcome!

News since the last year

Notable oneDPL changes

- The namespace was chosen: oneapi::dpl
  - Also, ::dpl is a short alias
  - Dropped the idea of namespace oneapi::std for standard compliant API because of usability problems
- Algorithms are blocking by default
  - Intel’s implementation provides a macro to allow non-blocking invocation
- Execution policy modifications
  - Class names: device_policy, fpga_policy (experimental)
  - Predefined objects: dpcpp_default, dpcpp_fpga
  - Implicit conversion to sycl::queue
- Multiple other improvements & adjustments

Notable implementation-specific additions

- <random> implementation suitable for DPC++ kernels
- Experimental range-based API for some algorithms
- Experimental asynchronous API for some algorithms

- All these APIs are not yet a part of the oneDPL specification
  - Proposals to add these to the version 1.1 will be created at the oneapi-src GitHub

<oneapi/dpl/random>

- Provides a subset of C++ <random>
- Standard compliant APIs with extensions to generate several RNs into sycl::vec
- Can be used on the host and in kernels

- Engine classes

linear_congruential_engine	discard_block_engine
subtract_with_carry_engine	default_engine

- Predefined engines

minstd_rand[_vec]	minstd_rand0[_vec]
ranlux24_base[_vec]	ranlux48_base[_vec]
ranlux24[_vec]	ranlux48[_vec]

- Distributions

uniform_int_distribution
uniform_real_distribution
normal_distribution

<oneapi/dpl/random> usage example

```
template<int VecSize>
void random_fill(float* usmptr, std::size_t n) {

  auto zero = oneapi::dpl::counting_iterator<std::size_t>(0);

  std::for_each(oneapi::dpl::execution::dpcpp_default,
                zero, zero + n/VecSize,
    [usmptr](std::size_t i){

      auto offset = i * VecSize;

      oneapi::dpl::minstd_rand_vec<VecSize> engine(seed, offset);
      oneapi::dpl::uniform_real_distribution<sycl::vec<float, VecSize>> distr;

      auto res = distr(engine);
      res.store(i, sycl::global_ptr<float>(usmptr));

    });
}
```

Experimental range-based API

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, …)
- Available as an experimental feature since oneDPL v2021.1

Ranges: programmability and performance

A pipeline of 3 kernels:

```
std::reverse(policy, begin(data), end(data));
std::transform(policy, begin(data), end(data), begin(result), [](auto i){return i*i;});
auto res = std::find_if(policy, begin(result), end(result), pred);
```

With fancy iterators (1 kernel):

```
auto res = std::find_if(policy,
    make_transform_iterator(make_reverse_iterator(end(data)), [](auto i){return i*i;}),
    make_transform_iterator(make_reverse_iterator(begin(data)), [](auto i){return i*i;}),
    pred);
```

With ranges (1 kernel):

```
using namespace oneapi::dpl::experimental::ranges;
auto res = find_if(policy,
                   data | views::reverse | views::transform([](auto i){return i*i;}),
                   pred);
```

And it can be quite faster than the original 3-kernel variant

Using ranges with DPC++ execution policies

oneAPI

- Make pipelines based on a factory
  - *iota* factory generates a sequence of indexes on the fly:
    `auto view = views::iota(0, n) | views::transform(lambda);`
- Make pipelines transforming data in USM
  - *subrange* creates a range view over two USM pointers
    `auto view = views::subrange(ptr, ptr+n) | views::transform(lambda);`
- Make pipelines transforming data in a SYCL buffer
  - *views::all* creates a range view over SYCL buffer
    `auto view = views::all(buf) | views::transform(lambda);`
  - Variations of it to specify the type of access: `all_read, all_write`
  - The data access operations may only be executed on device
- **Requirement**: ranges, views, pipelines of views must be **device copyable** to use them with DPC++ execution policies
  - How can we ensure that for the standard library implementations of GCC & LLVM?

Range-based API (as of oneDPL v2021.3)

Algorithms (34)

```
any_of
all_of
for_each
find_if
find_if_not
find
find_end
find_first_of
count
count_if
search
search_n
copy
transform
remove_if
remove
replace_if
replace
```

```
sort
stable_sort
is_sorted_until
is_sorted
equal
move
merge
min_element
max_element
minmax_element
reduce
transform_reduce
exclusive_scan
inclusive_scan
transform_exclusive_scan
transform_inclusive_scan
```

Views

```
all
all_read
all_write
subrange

iota
fill
generate

reverse
transform
rotate
take
drop
zip
permutation
```

oneAPI

Experimental asynchronous API

Blocking vs. asynchronous

- oneDPL algorithms with DPC++ execution policies are blocking
  - Standard-compliant: return when execution completes (on the device)
  - In some cases, may transfer data back to the host
- “Deferred waiting” mode can be enabled by a macro
  - Functions with no return value may submit a kernel and return immediately
  - Requires explicit waiting on the queue or buffer access
  - Non-standard, and not even in the oneDPL specification
- New: experimental explicitly asynchronous APIs
  - Functions never wait, instead returning a future-like object
  - Allows simultaneous use of multiple devices
  - Better exploits DPC++ asynchronous capabilities (hiding latencies etc.)
  - Enables composing oneDPL algorithms into static data flow graphs

Looking at the API

- Adds *_async suffix to algorithm names
  - `oneapi::dpl::experimental::for_each_async(…);`
  - **Possible alternatives:** a different namespace, a different policy class
- May take an arbitrary number of dependencies as extra arguments
  - `for_each_async(policy, first, last, lambda [,events…]);`
- Returns an unspecified future-like type

  -
    ```
    auto res = reduce_async(policy,first,last,std::plus<>{});
    auto sum = res.get();
    ```
  - Also holds internal buffers; keep track of its lifetime!
  - Interoperable with “native” backend specific types, e.g. sycl::event
  - **Possible alternatives:** a generic future type or backend specific futures

Try it: <oneapi/dpl/async>

- Algorithms supported since oneDPL v2021.2
  - copy_async, fill_async, for_each_async, reduce_async, transform_async, transform_reduce_async, sort_async
- More to be added
- Your feedback is welcome!

```
int async_example(sycl::buffer<int>& a) {
  auto& pol = dpl::execution::dpcpp_default;

  /* Build and compute a simple dependency chain:
     Fill buffer -> Transform -> Reduce */
  auto fut1 = dpl::experimental::fill_async(
                pol, dpl::begin(a), dpl::end(a), 7);

  auto fut2 = dpl::experimental::transform_async(
                pol, dpl::begin(a), dpl::end(a),
                dpl::begin(a), [](int x){return x+1;},
                fut1);

  return dpl::experimental::reduce_async(
                pol, dpl::begin(a), dpl::end(a),
                fut2).get(); // blocks to get the value
}
```

Other open questions

The minimal supported version of C++

Currently, oneDPL claims support for C++11 but:

- SYCL 2020 requires C++17
- DPC++ compiler does not support C++11 well
- Range-based API implementation requires at least C++17
  - Also the standard library of GCC 8.1 / LLVM 7 or higher
  - De-jure, ranges are a C++20 feature

- There is a strong desire to switch to C++17 (or at least C++14)
- Do you think it’s acceptable for the specification and/or the implementation?

Policy renaming/rebinding

- Sometimes one may want to name DPC++ kernels
  - For debugging, profiling, or lack of support for unnamed ones
- With oneDPL, *unique* names need to be given to each policy object

```
dpl::execution::device_policy<MyKernelName> pol;
```

- A policy can be “renamed” at algorithm invocation

```
for_each( dpl::execution::make_device_policy<NewKernelName>(pol), … );
```

  - Creates a temporary object used only onse.
  - But the syntax looks overly verbose and non-obvious
- Looking for something simpler, e.g.

```
for_each( dpl::execution::rename<NewKernelName>(pol), … );
for_each( pol.template rebind<NewKernelName>(), … );
```

- Opinions?

Closing

Call to action

- Try out the experimental API: ranges and asynchronous algorithms
  https://github.com/oneapi-src/oneDPL
- Provide your comments & feedback on the API
- Participate in review / discussions of specification update proposals
  https://github.com/oneapi-src/oneAPI-spec/discussions
- Share information with others who might be interested to contribute

- Thanks!

Backup

Materials from 2020 reviews

A usage example for ranges

- A code sample (derived from SyclParallelSTL project)

```
// compute the number characters that start a new word
int wc = std::inner_product(                // what?? Let’s figure out
    first, last,                            // a sequence of characters
    first + 1,                              // same sequence shifted by 1 character
    0,                                      // initialize the sum to 0
    std::plus<int>{},                       // combine partial sums
    [](char s1, char s2) {                  // check two consequent chars to find where a word starts
        return is_word_start(s1, s2)? 1: 0;
    });
```

- The same thing done with ranges:

```
// compute the number characters that start a new word
using namespace std::ranges::views;
int wc = std::ranges::count_if(                    // the algorithm name is descriptive
    zip( text|take(text.size()-1), text|drop(1) ), // views are easy to combine
    [](auto v) {                                   // the predicate gets a tuple created by the zip view
        return is_word_start(std::get<0>(v), std::get<1>(v)); // working with tuples is cumbersome
    });
```