oneAPI Data Parallel C++ Library

For the DPC++ Technical Advisory Board discussion

April 2020

Agenda

- oneAPI Data Parallel C++ library (oneDPL) recap
- The required version of C++
- Top-level namespace for oneDPL
- Supporting the C++ standard library
- Range-based API for parallel algorithms
- Guiding principles to add extension APIs

The DPC++ library concept: recap

- Goal: Extend DPC++ language making it applicable to a broader set of problems
- The standard C++ library API
  - A subset to be used in kernels, based on capabilities of device/accelerator and on anticipated needs
- Parallel STL to run standard algorithms on devices
  - Based on C++17, but with a special backend and extensions (e.g. non-standard execution policies)
- Non-standard but useful API extensions
  - Practical usability and functional parity with similar APIs

oneDPL

- The C++ std library
- Parallel STL
- API extensions

General topics

The required version of C++

- DPC++ will require C++17
  - Primarily to simplify its API usage with CTAD
- **Is the same requirement OK for oneDPL?**

- Pro: allows to rely on CTAD for ease of use
- Pro: allows implementers to simplify the code (generic lambdas, if constexpr)
- Con: limits supported host-side environments
  - Min: GCC 7, Clang/LLVM 6; default is C++14 even for latest host compilers

Top-level namespace for oneDPL

- DPC++ programs will use APIs from multiple namespaces
  - sycl::, sycl::intel:: for DPC++ API; std:: for C++ standard library API
  - oneXXX for oneAPI components, e.g. onemkl, onednn, …
- oneDPL adds its own namespace to the zoo
  - Cannot use the above for non-standard extensions
- The currently used variant is dpstd::
  - Follows the same naming principle as DPC++, referring to std::
  - **Does dpstd:: seem right to standardize in oneDPL specification?**
  - Other possible options: onedpl::, onestd::, sycl::std:: (?), …
- What others are doing
  - NVidia: cuda::, <u>cuda::std::</u>, thrust::, cub::
  - AMD: hc::, thrust::, hipcub::, rocprim::

Support of the C++ standard library

The goals

- Allow to use a selected subset of Standard C++ Library templates, classes and functions in DPC++ kernels, as well as for passing data between the host program and the kernels
- Support a variety of host environments
- Comply to the requirements of both C++ and SYCL
- Have a path to converge with the C++ standard in the future

The basic issues

- Some standard library classes cannot be supported (fully or partially)
  - use of exceptions, dynamic memory allocation, not trivially copyable, etc.
- At least 3 *different* implementations of the standard library to support or interoperate with
  - Multiple versions of libstdc++, libc++, and MSVC++ STL
- Different implementations cannot be intermixed within a program
  - Name conflicts / ODR violation

The options considered

- Allow use of “white-listed” functions and classes defined in the host standard library
  - Implement the necessary support in the DPC++ compiler & device libraries
- Provide a separate “freestanding” standard library implementation
- “Duplicate” standard library functionality in SYCL
  - The specification would define whether to reimplement or to alias host C++ library definitions

- Each one has advantages as well as problems

The options: major pros and cons

	Whitelisting	Freestanding	Duplicate in SYCL
Advantages	Usage looks no different from regular C++; Single instance of the std library;	Unambiguous compile-time resolution of what's supported; Allows to overcome limitations and adjust std API if necessary;	Clear specification of what's supported where, including interoperability with the host; Allows to adjust the std API
Problems	Some APIs hardly can be supported this way Difficult to understand what works and what does not	Enforces a namespace other than std:: for everything; Harder to mix/interop with host-side std library; Functions may need different implementations for host vs. kernels – ODR violation?	Namespace other than std:: (except when allowed); "Forking" a significant portion of C++ seems a bad idea; Significant work for Khronos & other implementers

Proposed for oneDPL: combined approach

- APIs that need substantial adjustments are defined in SYCL spec
  - atomic_ref, simd - in sycl::
- Whitelisting for APIs that “just work” or if support for existing implementations is feasible
  - Functional classes, type traits, complex, … - in std::
- A separate, “freestanding” implementation for the rest
  - tuple, … - in dpstd:: <dpstd/tuple>
- oneDPL may alias or wrap the APIs from all three parts
  - So dpstd:: contains everything that we want

Combined approach: pros & cons

- Pro: avoids or overcomes the major problems of “pure” approaches
- Pro: aligns with existing practice of the SYCL spec
- Pro: seems able to satisfy varying expectations

- Con: understanding of what’s supported is somewhat complicated
  - Esp. for APIs that are modified or not supported in full
- Path to C++ standardization not well understood yet
- Anything we missed?

Mapping onto the desired subset

Header	Approach
<cassert>	Whitelisted*
<cmath>	Whitelisted*
<complex>	Whitelisted*
<cstddef>	Whitelisted
<functional> (partial)	Whitelisted
<random>	oneMKL
<tuple>	Custom impl.
<type_traits>	Whitelisted
<utility>	Whitelisted (except std::pair - custom)
<atomic>	SYCL (atomic_ref)
<cstdint>, <cfloat>, <climits>	Whitelisted

Header	Approach
<initializer_list>	Whitelisted
<limits>	Whitelisted
<new> (partial)	Whitelisted
<algorithm> (partial)	Whitelisted
<array>	? (may throw)
<chrono>	?
<compare>	Whitelisted
<numeric> (partial)	?
<optional>	?
<ratio>	?
<variant>	?
<iterator>	?

* Requires special support in DPC++ runtime/device libraries

Parallel algorithms and extension API

Range-based API for algorithms (1)

- 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
  - Not fully standard-compliant (e.g. not based on concepts)
  - Likely only a subset of the standard algorithms and views

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 = dpstd::find_if(pol,
               views::all(data) | views::reverse() | views::transform(lambda1), pred);
```

Ranges: minimal scope (23 algorithms)

oneAPI

<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 (2)

- 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
  - Not fully standard-compliant (e.g. not based on concepts)
  - Likely only a subset of the standard algorithms and views
  - **Are any important algorithms missing in the minimal subset?**
- For the oneDPL specification, the options are:
  - Add these APIs to the oneDPL specification v1.0, or
  - Leave as extensions for now, add to a later specification version
  - **Which option does seem right to you?**

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	Partial prefix scans on a sequence of values, by segments of equal keys
exclusive_scan_by_segment	
binary_search	Binary search variations for multiple values in the same sequence ("vectorized" search)
lower_bound	
upper_bound	
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 extension APIs

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