oneAPI Technical Advisory Board Meeting:

DPC++ Enhanced property_list

July 28, 2021

James Brodman, Jessica Davies, Joe Garvey, Mike Kinsner,
Greg Lueck, John Pennycook, Roland Schulz, Jason Sewall

Motivation

- SYCL 1.2.1 and 2020 define `sycl::property_list`
  - Runtime (not compile-time) property information for construction of runtime classes
  - Property types (which properties) and their values are known only at runtime

```
template <typename AllocatorT, typename TagT>
accessor(buffer<dataT, dimensions, AllocatorT> &bufferRef, TagT tag,
         const property_list &propList = {});
```

SYCL 2020 runtime properties

```
accessor result{ buf, h, write_only, no_init };
```

- Problem: Some property information is important at compile-time
  - Semantic modifiers and/or optimization opportunities

Motivation: Challenges with C++ attributes in SYCL

- Attributes are an unsafe way to change semantics
  - (Host) C++ compilers are free to ignore unrecognized attributes
  - Library-only implementations can't see custom attributes at all
- Attributes can't be used for overloading, templates, etc
  - Compiling the same kernel with multiple attributes requires ugly hacks
  - Users cannot inspect kernel properties at compile-time
- Handling attributes differently between host/device is confusing

Current + proposed SYCL use cases for attributes

1. **Request special behavior from the SYCL runtime**
   e.g. work-group size

2. **Request special behavior from the device compiler**
   e.g. sub-group size, forward progress guarantees

3. **Encode requirements into a kernel**
   e.g. kernel provided by library must use specific work-group size

4. **Encode requirements into a function**
   e.g. function provided by library is optimized for sub-group size

Direction: SYCL attributes ⇒ properties

- Example: Kernel properties

```
myQueue.parallel_for( range<2>(16,16),
      [=] (id<2> i) [[sycl::reqd_work_group_size(8, 8)]] [[sycl::reqd_sub_group_size(8)]] {
        //[kernel code]
      });
});
```

```
sycl::ext::oneapi::property_list properties{sycl::ext::oneapi::work_group_size_v<8, 8>,
                                            sycl::ext::oneapi::sub_group_size_v<8>};

myQueue.parallel_for(range<2>{16, 16}, properties, [=](id<2> i) {
  //[kernel code]
});
```

oneAPI

Functor example

```
struct KernelFunctor {

  KernelFunctor(accessor<int, 2> a,
                accessor<int, 2> b,
                accessor<int, 2> c) : a(a), b(b), c(c)
  {}

  void operator()(id<2> i) const {
    a[i] = b[i] + c[i];
  }

  static constexpr auto properties =
    sycl::ext::oneapi::property_list{sycl::ext::oneapi::work_group_size_v<8, 8>,
                                     sycl::ext::oneapi::sub_group_size_v<8>};

  sycl::accessor<int, 2> a;
  sycl::accessor<int, 2> b;
  sycl::accessor<int, 2> c;

};
...
q.parallel_for(range<2>{16, 16}, KernelFunctor(a, b, c)).wait();
```

New property mechanism

Example: Extended memory semantics

```
using namespace sycl::ext::oneapi;
double *base = …;
double d0 = load(base, property_list{temporality_hint<nontemporal>});
store(base+4, d0, property_list{L1_hint<nontemporal>});

prefetch(base+8, 16, property_list{temporality_hint<temporal>});

multi_pointer<double, property_list{L3_hint<nontemporal>}> noL3_base(base);
noL3_base[12] = d0;
```

Compile-time properties change behavior of load/store to have different temporal behavior, at various granularities

An annotated pointer would allow for coarse-grained control over behavior with no RT overhead

Work in progress to be enabled by new property_lists

Goals

- Extend the SYCL properties mechanism:
    1. Make the type of all properties (but not their values) visible at compile-time
    2. Enable compile-time values for some properties
    3. Enable runtime values for other properties

- Create general framework for properties – used heavily in extensions

- Provide better alternative to C++ attributes which currently act as modifiers in the SYCL spec and its extensions

Extension detailed in following slides

- SYCL_EXT_ONEAPI_PROPERTY_LIST
  - Adds new `sycl::ext::oneapi::property_list` class
    - Variant of existing `sycl::property_list`
  - Supports storage + manipulation of compile-time-constant properties in addition to runtime properties

- Status:
  - Extensive discussion and iteration within Intel
  - Spec draft on GitHub and pushing forward with open source implementation
  - Looking for as much feedback as possible
  - Expected to be an important proposal for the next major version of SYCL

Property styles to enable

1. Property with **no associated value**
   - e.g.: `sycl::no_init` accessor property

2. Property with value(s) known only at **runtime**
   - e.g.: `sycl::buffer<int, 1>{p, r, sycl::property::context_bound{myContext}}`

New!

3. Property with value(s) known at **compile time**
   - e.g.:

```
sycl::ext::oneapi::property_list properties{sycl::ext::oneapi::work_group_size_v<8, 8>,
                                            sycl::ext::oneapi::sub_group_size_v<8>};

q.parallel_for(range<2>{16, 16}, properties, [=](id<2> i) {
  a[i] = b[i] + c[i];
});
```

New approach: Definition of properties

- Runtime + compile-time constant properties represented as classes
  - Convention: Declare in root sycl namespace (no longer semantically nested)

`sycl::property::buffer::use_mutex` → `sycl::ext::oneapi::use_mutex`

- Runtime properties recommended to have constructors which take property value(s) and public member variables that store those values
  - Runtime properties can represent/contain only non-type values
- Compile-time constant properties with params must define a type alias `value_t`
  - `value_t` is templated on the params, and is an alias to an unspecified instantiation of the `property_value` class which holds values of compile-time params
  - Compile-time-constant properties can represent/contain type or non-type values

Example: Runtime property

```
namespace sycl {
namespace ext {
namespace oneapi {

  // This is a runtime property with one integer parameter
  struct foo {
    foo(int);
    int value;
  };

} // namespace oneapi
} // namespace ext
} // namespace sycl
```

Example: Compile-time-constant property

```
namespace sycl {
namespace ext {
namespace oneapi {

  // This property has no parameters.
  struct bar {
    using value_t = property_value<bar>;
  };

  // This property has one integer non-type parameter.
  struct baz {
    template<int K>
    using value_t = property_value<baz, integral_constant<int, K>>;
  };

  // This property has an arbitrary number of type parameters.
  struct boo {
    template<typename...Ts>
    using value_t = property_value<boo, Ts...>;
  };
} // namespace oneapi
} // namespace ext
} // namespace sycl
```

Property traits

- Just as with SYCL 2020 properties, all runtime and compile-time-constant properties must:
  1. Specialize `sycl::is_property` - inherit from `std::true_type`
  2. Specialize `sycl::is_property_of` - inherit from `std::true_type` for each SYCL runtime class that the property can be applied to

```
namespace sycl {

  template<> struct is_property<ext::oneapi::foo> : std::true_type {};

  // Can be applied to any SYCL object
  template<typename syclObjectT>
  struct is_property_of<ext::oneapi::foo, syclObjectT> : std::true_type {};

} // namespace sycl
```

property_value class

- Compile-time-constant properties must alias `value_t` to an instantiation of `property_value`
- The property_value class has implementation-defined template parameters:
  - If a property has a single parameter, it provides a member variable named value and a type alias named `value_t` to retrieve the param value and type
  - If a property has more than one parameter, `property_value` should provide semantically meaningful ways to retrieve values and types of the parameters

property_list

- New template class (`sycl::ext::oneapi::property_list`)
  - Can contain compile-time constant as well as runtime props
  - Its properties influence its type

  - Two `property_list` objects have same type iff constructed with the same set of compile-time property values, and the same set of runtime properties
    - Runtime properties contained in the property list affect the type of `property_list`, but their property values do not

  - `has_property`: Determine at compile time if property_list has particular property
  - `get_property`: Determine value of property
    - If compile-time constant prop, returns object of prop value class (e.g. foo::value_t)
    - If runtime prop or prop without value, returns *copy* of property instance

Passing to kernels

- All instances of **compile-time constant** properties are device copyable
- A property_list with a non-device-copyable runtime prop can’t be a kernel arg
- Compile-time constant props enable kernel optimization, reduce multi-versioning

```
static_assert(sycl::is_device_copyable_v<decltype(foo_v<1>)>);
static_assert(sycl::is_device_copyable_v<bar>);

property_list P1{foo_v<1>, bar{}};

// All properties in P1 are device copyable, so P1 is device copyable
static_assert(sycl::is_device_copyable_v<decltype(P1)>);

h.single_task([=] {
  auto a = P1.has_property<foo>(); // OK
  auto b = P1.get_property<foo>(); // OK
  auto c = P1.has_property<bar>(); // OK
  auto d = P1.get_property<bar>(); // OK
});
```

property_list is invariant to property ordering

```
using P1 = property_list_t< bar_v<1>, foo_v >;
using P2 = property_list_t< foo_v, bar_v<1> >;

static_assert(std::is_same<P1, P2>::value); // Succeeds - order doesn’t matter
static_assert(P1.get_property<bar>().value == 1);
```

- Internal implementation-defined sorting or similar mechanism to make type agnostic to property ordering

Recommended style – prefer runtime values

- Prefer runtime property values unless there is sufficient benefit to compile-time exposure (of the value)
  - Semantic or “significant” optimization impact

- Compile-time properties become part of the property type
  - Can’t be passed to non-template functions - complicate the interface
  - Should only incur this cost when there is a benefit, not simply because the value can be known at compile time

Example 1

```
property_list P5{foo_v<1>};
property_list P6{foo_v<2>};
property_list P7{foo_v<1>, bar_v};

static_assert(P6.has_property<foo>()); // No need to specify the value of the property's parameter

static_assert(!std::is_same_v<decltype(P5), decltype(P6)>); // parameter vals of foo are different

auto f1 = P5.get_property<foo>(); // f1 is a copy of global variable foo_v < 1 >
auto f2 = P6.get_property<foo>(); // f2 is a copy of global variable foo_v < 2 >

static_assert(f1 != f2); // Not equal since the property values are different, i.e., 1 vs. 2

auto f3 = P7.get_property<foo>();

static_assert(f3 == f1); // Equal because the property values are the same, i.e., equal to 1
```

Example 2

```
property_list P8{foo_types_v<float, int, bool>()};

using f = decltype(P8.get_property<foo_types>());
using t1 = f::first_t;
using t2 = f::second_t;
using t3 = f::third_t;

static_assert(std::is_same_v<t1, float);

static_assert(std::is_same_v<t2, int>);

static_assert(std::is_same_v<t3, bool>);
```

Example 3

```
template<typename propertyListT>
std::enable_if_t<is_property_list_v<propertyListT>>
  my_func1(propertyListT p);

template<typename propertyListT>
std::enable_if_t<is_property_list_v<propertyListT> &&
  propertyListT::template has_property<foo>()>
  my_func2(propertyListT p);

template<typename propertyListT>
std::enable_if_t<is_property_list_v<propertyListT> &&
  (propertyListT::template get_property<bar>().value == 2)>
  my_func3(propertyListT p);

my_func1(property_list{foo_v}); // Legal.  my_func1 accepts any properties
my_func2(property_list{foo_v}); // Legal.  my_func2 requires foo
my_func2(property_list{bar_v}); // Illegal.  my_func2 requires foo
my_func2(property_list{foo_v, bar_v}); // Legal.  Other properties can also be specified.
my_func3(property_list{bar_v<2>); // Legal. my_func3 requires bar with value 2
my_func3(property_list{bar_v<1>); // Illegal. my_func3 requires bar with value 2
```

Interaction with existing SYCL runtime classes

- Will add new SYCL runtime class constructor overloads taking the new `property_list`
  - ALL classes will take properties
- No conversion operator between new/old property_list forms
- Will decide on case-by-case basis which SYCL runtime classes will have type impacted by properties and which properties in that case
  - Will only make SYCL runtime classes into templated class and bake a property into the type when there is strong justification

Conclusion

- Add new `property_list` class that supports compile-time constant properties in addition to runtime properties (which are in SYCL 2020)
  - `sycl::ext::oneapi::property_list`
- Existence of property in `property_list` is visible at compile-time
  - The value of a property may not be known at compile-time (if a runtime prop)
- Creates framework for properties to replace attributes and other SYCL mechanisms
  - Intended to be proposed for the next major version of SYCL

Rules of the Road

- DO NOT share any confidential information or trade secrets with the group
- DO keep the discussion at a High Level
  - Focus on the specific Agenda topics
  - We are asking for feedback on features for the oneAPI specification (e.g. requirements for functionality and performance)
  - We are NOT asking for feedback on any implementation details
- Please submit any implementation feedback in writing on Github in accordance with the Contribution Guidelines at spec.oneapi.com. This will allow Intel to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification.

Notices and Disclaimers

The content of this oneAPI Specification is licensed under the Creative Commons Attribution 4.0 International License . Unless stated otherwise, the sample code examples in this document are released to you under the MIT license.

This specification is a continuation of Intel's decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to The Khronos Group or other standard bodies under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, *you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback in its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.* For complete contribution policies and guidelines, see Contribution Guidelines on www.spec.oneapi.com.

Property value instances

- For each compile-time constant property a value variable whose name has the suffix "_v" is defined
  - e.g.: If a property is named `foo`, the pre-defined property value instance is named `foo_v`
  - Variable type encodes the property value

```
// bar is only available if p has the foo property with value 3
template<typename property_listT>
std::enable_if_t<property_listT::template getProperty<foo>() == foo_v<3>> bar(property_listT p) {
  ...
}
```