IWOCL 2024

The 12th International Workshop on OpenCL and SYCL

SYCL properties with compile-time information

James Brodman, Intel

Ben Ashbaugh, Michael Kinsner, Steffen Larsen, Greg Lueck, John Pennycook, Roland Schulz – Intel

Gordon Brown – Codeplay

SYCL 2020 property lists

```
sycl::property_list Properties{sycl::property::queue::in_order{}};
sycl::queue Queue{Properties};

if (!Queue.has_property<sycl::property::queue::in_order>()) {
  // Do something here. The compiler will always evaluate this
  // but it will never execute.                                    ✓
}

if constexpr (!Queue.has_property<sycl::property::queue::in_order>()) {
  // Queue.has_property cannot evaluate at compile-time, so this
  // if-statement is ill-formed.                                     ✗
}
```

Properties extension

- New property list type: `properties`
- These property lists can contain two kinds of properties:
  1. Runtime properties
     i.e. properties containing a value specified at runtime.
  2. Compile-time constant properties
     i.e. properties containing only values evaluated at compile-time.
- All properties have a “value” type and a “key” type.
  - For runtime properties, the key type is an alias of the value type.

Example runtime property foo

- Consider a runtime property foo with a float member variable x.

```
namespace sycl::ext::oneapi::experimental {

struct foo {
  foo(float X) : x{X} {}
  float x;
};

using foo_key = foo;

} // namespace experimental::oneapi::ext::sycl
```

Example compile-time constant property **bar**

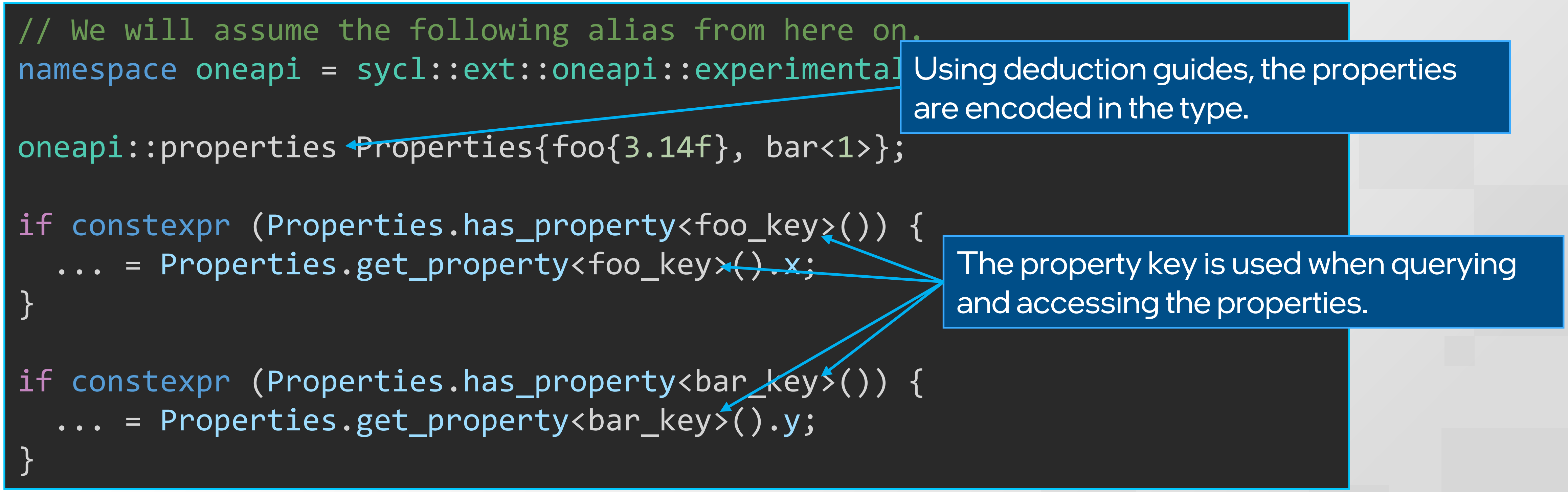
- Consider a compile-time constant property **bar** with an `int` template argument with `constexpr` member variable **y**.

```
namespace sycl::ext::oneapi::experimental {
struct bar_key {
  template<int Y>
  using value_t = property_value<bar_key, std::integral_constant<int, Y>>;
};

template <int Y>
struct property_value<bar_key, std::integral_constant<int, Y>> {
  static constexpr int y = Y;
};

template<int Y> inline constexpr bar_key::value_t<Y> bar;
} // namespace experimental::oneapi::ext::sycl
```

Specialization of `property_value` is not strictly needed if there is only a single template parameter.

Example property list

- Using **foo** and **bar**, we can create a properties containing them.



```
// We will assume the following alias from here on.
namespace oneapi = sycl::ext::oneapi::experimental

oneapi::properties Properties{foo{3.14f}, bar<1>};

if constexpr (Properties.has_property<foo_key>()) {
  ... = Properties.get_property<foo_key>().x;
}

if constexpr (Properties.has_property<bar_key>()) {
  ... = Properties.get_property<bar_key>().y;
}
```

Using deduction guides, the properties are encoded in the type.

The property key is used when querying and accessing the properties.

Properties extension – An example

- Consequently, since the presence of properties in a `properties` object can be queried at compile-time, the following applies:

```
oneapi::properties Properties{};

// Both of the following will fail to compile as Properties do not have
// the properties.
... = Properties.get_property<foo_key>().x;
... = Properties.get_property<bar_key>().y;
```

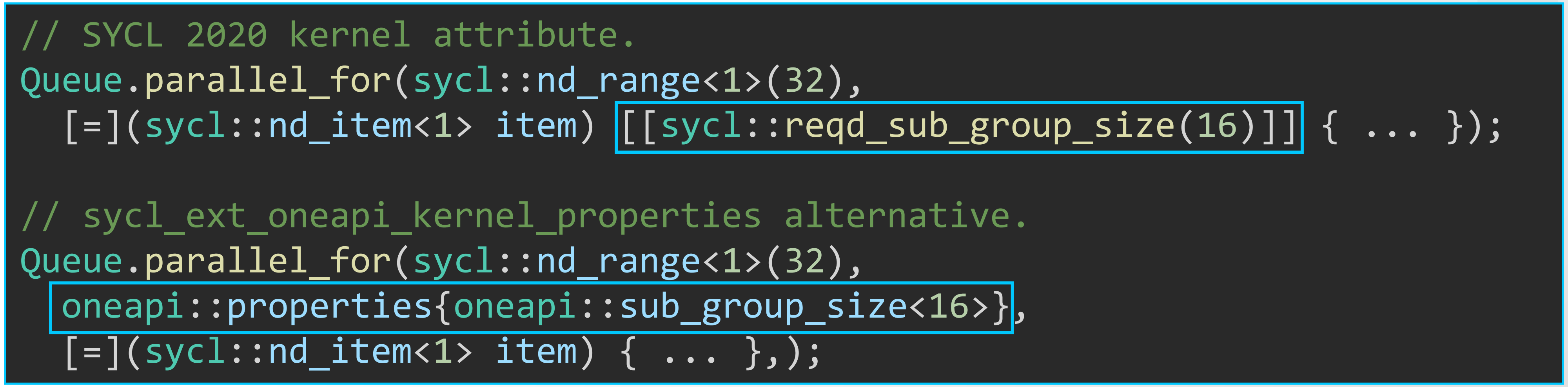
Properties in action – An alternative to attributes

- SYCL 2020 uses C++ attributes to give compile-time information about kernels.
- Attributes require compiler support, making the use of non-SYCL host-compilers difficult.

Compile-time constant properties can help!

Properties in action – An alternative to attributes

- DPC++ supports the sycl_ext_oneapi_kernel_properties experimental extension.



```
// SYCL 2020 kernel attribute.
Queue.parallel_for(sycl::nd_range<1>(32),
  [=](sycl::nd_item<1> item) [[sycl::reqd_sub_group_size(16)]] { ... });

// sycl_ext_oneapi_kernel_properties alternative.
Queue.parallel_for(sycl::nd_range<1>(32),
  oneapi::properties{oneapi::sub_group_size<16>},
  [=](sycl::nd_item<1> item) { ... },);
```

Properties in action – An alternative to attributes

- Properties can also apply to functor classes.

```
// SYCL 2020 kernel attribute.
class SYCL2020AttributeFunctor {
public:
  [[sycl::reqd_sub_group_size(16)]]
  void operator()(sycl::nd_item<1> Item) const { ... }
};

// sycl_ext_oneapi_kernel_properties alternative.
class ExtPropertiesFunctor {
  void operator()(sycl::nd_item<1> Item) const { ... }

  auto get(oneapi::properties_tag) {
    return oneapi::properties{oneapi::sub_group_size<16>};
  }
};
```

Properties in action – Annotating kernel arguments

- Compile-time constant properties can also help with extensions to SYCL that need to communicate information to the compiler.
- sycl_ext_oneapi_kernel_arg_properties* is an example of such extension, building on sycl_ext_oneapi_annotated_ptr and sycl_ext_oneapi_annotated_arg.

```
oneapi::properties RestrictProp{oneapi::restrict{}};
oneapi::annotated_arg RestrictArg{Ptr, RestrictProp};
Queue.single_task([=]() {
  // The capture of RestrictArg tells the compiler that the argument
  // does not alias any other kernel arguments.
  DoSomething(RestrictArg);
});
```

* The extension `restrict` property has not yet been implemented in DPC++.

Properties in action – Device-side prefetching

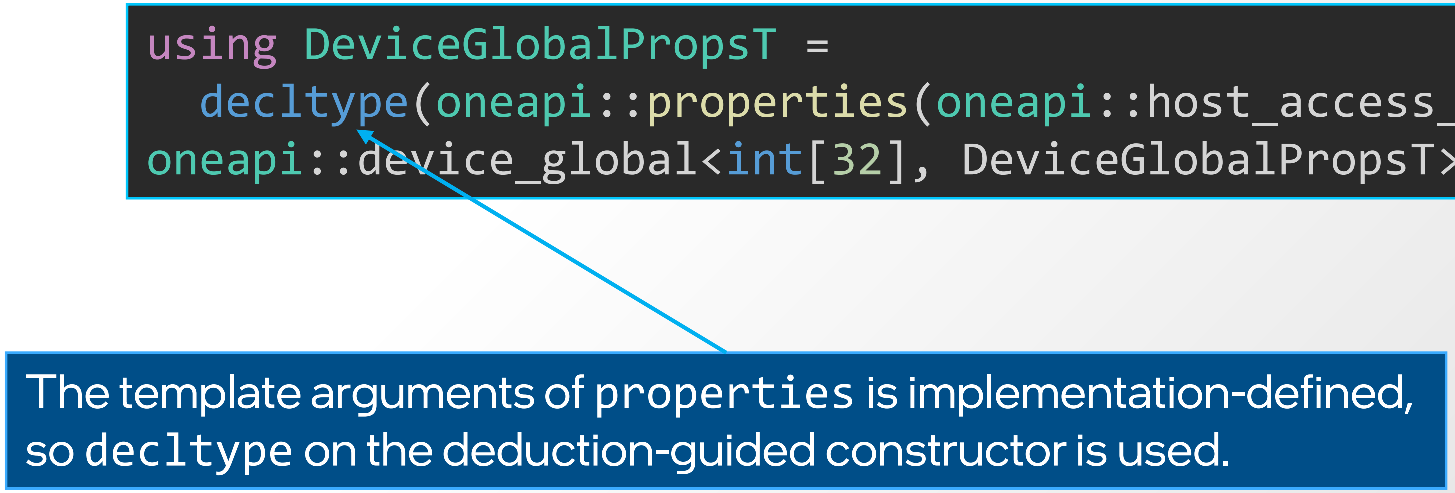
- sycl_ext_oneapi_prefetch adds the ability to prefetch memory into specific cache levels.
- Though prefetch is called at runtime, the compiler needs to know which cache level it will fetch to.

```
float *Ptr = sycl::malloc_shared<float>(N * 32);
Queue.parallel_for(N, [=](sycl::item<1> It) {
  // Fetch 32 elements into the L1 cache.
  oneapi::prefetch(Ptr + It * 32, 32, oneapi::prefetch_hint_L1);
  // Use the prefetched elements.
  for (size_t I = 0; I < 32; ++I)
    DoSomething(Ptr[It + I]);
});
```

`oneapi::prefetch_hint_L1` is a shorthand for `oneapi::prefetch_hint<oneapi::cache_level::L1, void>`. This is common practice for properties with very limited valid values.

Properties in action – Device-global variables

- Compile-time property information is part of the properties type, so some interfaces can rely solely on the `properties` type.
- For example, sycl_ext_oneapi_device_global uses this for global variables accessible on device.



```
using DeviceGlobalPropsT =
  decltype(oneapi::properties(oneapi::host_access_read_write));
oneapi::device_global<int[32], DeviceGlobalPropsT> DeviceGlobal;
```

The template arguments of `properties` is implementation-defined, so `decltype` on the deduction-guided constructor is used.

oneAPI SYCL extension references

- sycl_ext_oneapi_properties (experimental)
- sycl_ext_oneapi_kernel_properties (experimental)
- sycl_ext_oneapi_kernel_arg_properties (experimental)
- sycl_ext_oneapi_annotated_arg (experimental)
- sycl_ext_oneapi_annotated_ptr (experimental)
- sycl_ext_oneapi_prefetch (experimental)
- sycl_ext_oneapi_device_global (experimental)
- Other oneAPI extensions using properties:
  - sycl_ext_intel_grf_size (experimental)
  - sycl_ext_intel_fpga_kernel_interface_properties (proposed)
  - ... And more!

Summary

- Communicating compile-time information in SYCL 2020 is limited:
  - Property lists are insufficient for compile-time diagnostics.
  - Property lists cannot be used for extensions that need information about user-provided properties at compile-time.
  - Attributes may not work with non-SYCL host compilers.
- The properties with compile-time information extension solves all these limits by encoding information about properties into its type.

intel® software