13th International Workshop on OpenCL and SYCL

IWOCL 2025

Virtual functions in SYCL

Alexey Sachkov, Intel

Greg Lueck, James Brodman and John Pennycook, Intel.

April 7-11, 2025 | Heidelberg, Germany | iwocl.org

KHRONOS GROUP

- Motivation
- What we have already in SYCL
  - Spoiler: nothing!
- What can we do about it
  - Overview of the proposed extension
    - And why it was designed this way
  - Comparison with OpenMP & CUDA
  - Implementation details
- Summary

Motivation

- SYCL aims to be as close to C++ as possible
- Virtual functions exist in C++
- Virtual functions are supported by both CUDA & OpenMP
  - GitHub: Exawind/exw-virtuals-app

- SYCL users are already asking about them!
  - IWOCL 2024: Experimenting with Implementing Kokkos's SYCL backend

The odr-use of polymorphic classes and classes with virtual inheritance is allowed. **However, no virtual member functions are allowed to be called in a device function.**

5.4. Language restrictions for device functions

Seems to be too strict, see KhronosGroup/SYCL-Docs#565

Let’s make it work!

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {


  int foo() override { return 3; }
  int bar() override { return 4; }
};
struct Derived2 : public Base {


  int foo() override { return 5; }
  int bar() override { return 6; }
};
```

Let's make it work!

```
struct Construct {
  Base *storage; int kind;
  void operator()() const {
    kind % 2
      ? new (storage) Derived1
      : new (storage) Derived2;
  }
};
struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }



};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
single_task(q, Use{storage});
```

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {


  int foo() override { return 3; }
  int bar() override { return 4; }
};
struct Derived2 : public Base {


  int foo() override { return 5; }
  int bar() override { return 6; }
};
```

See sycl_ext_oneapi_enqueue_functions (experimental)

Let's make it work!

See sycl_ext_oneapi_kernel_properties (experimental)

```
struct Construct {
  Base *storage; int kind;
  void operator()() const {
    kind % 2
      ? new (storage) Derived1
      : new (storage) Derived2;
  }
};
struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }
  auto get(properties_tag) const {
    return properties{assume_indirect_calls};
  }
};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
single_task(q, Use{storage});
```

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { return 3; }
  int bar() override { return 4; }
};
struct Derived2 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { return 5; }
  int bar() override { return 6; }
};
```

Why properties on virtual functions?

- "`indirectly_callable`" property indicates that function should be considered as a device function

```
@_ZTV8Derived1 = { [4 x ptr] } { [4 x ptr] [
    ptr null, ptr @_ZTI8Derived1, ptr @_ZN8Derived13fooEv,
    ptr @_ZN8Derived13barEv] }
@_ZTV4Base = { [4 x ptr] } { [4 x ptr] [
    ptr null, ptr @_ZTI4Base, ptr @_ZN4Base3fooEv, ptr @_ZN4Base3barEv] }
@_ZTV8Derived2 = { [4 x ptr] } { [4 x ptr] [
    ptr null, ptr @_ZTI8Derived2, ptr @_ZN8Derived23fooEv,
    ptr @_ZN8Derived23barEv] }
```

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { return 3; }
  int bar() override { return 4; }
};
struct Derived2 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { return 5; }
  int bar() override { return 6; }
};
```

What are properties on virtual functions?

- "`indirectly_callable`" property indicates that function should be considered as a device function
  - i.e. all restrictions apply!
- Only affects a specific override
- Calling a virtual function ***not*** marked with the property is a UB

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { return 3; }
  int bar() override { return 4; }
};
struct Derived2 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { return 5; }
  int bar() override { return 6; }
};
```

What are properties on kernels and why?

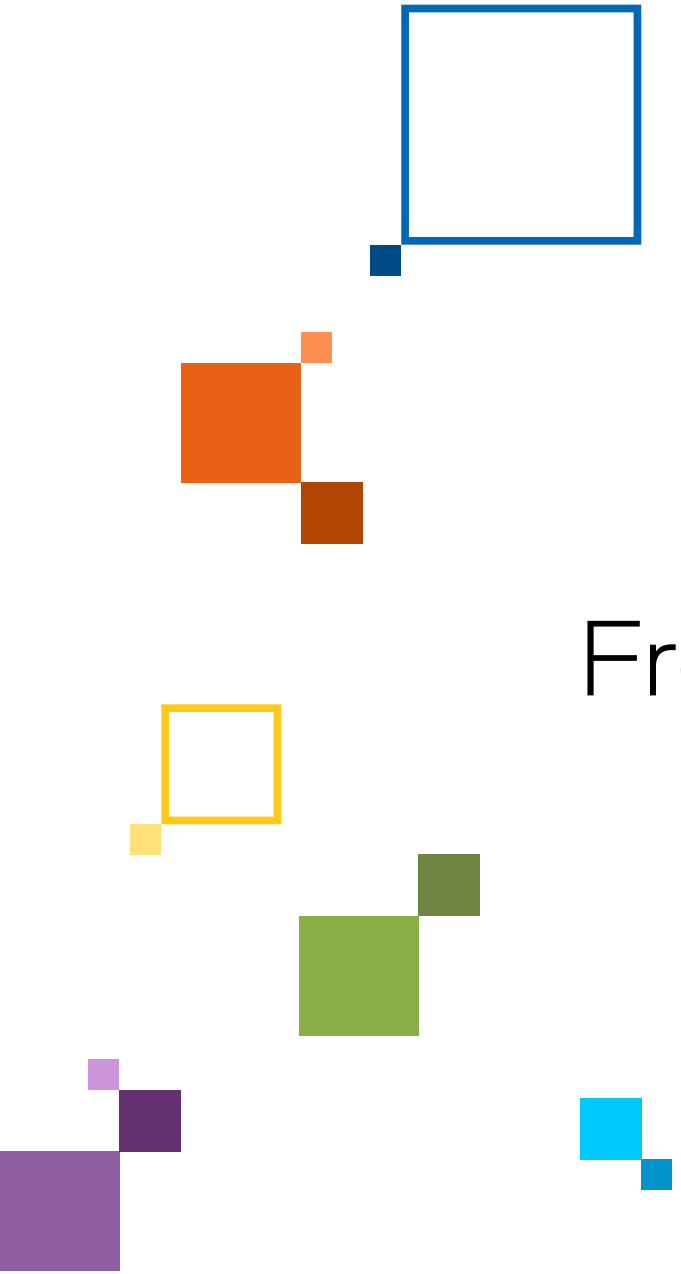
```
struct Construct {
  Base *storage; int kind;
  void operator()() const {
    kind % 2
      ? new (storage) Derived1
      : new (storage) Derived2;
  }
};
struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }
  auto get(properties_tag) const {
    return properties{assume_indirect_calls};
  }
};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
single_task(q, Use{storage});
```

- Doing a virtual call from a kernel without the property is an error
- Optional kernel features!
- Device code split needs to know what is used by a kernel
- Room for passing any extra flags to underlying runtimes

Any other rules?

- An object can only be used to do virtual function calls on the ***same*** device where it was constructed
  - Construct on device then call on host, or construct on host and call on device isn't legal either

- No RTTI still, i.e. no `dynamic_cast`, `typeid`, etc.



From here it quickly gets complicated…

Can you use `reqd_sub_group_size`?

```
struct Construct {
  Base *storage; int kind;
  void operator()() const { /* new (storage) ... */ }
};
struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }
  auto get(properties_tag) const {
    return properties{
      assume_indirect_calls, reqd_sub_group_size(16)};
  }
};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
single_task(q, Use{storage});
```

- No, that's a UB!
- Functions cannot be annotated with `reqd_sub_group_size` at SPIR-V level
  - What if someone wants to have two overloads for different sub-group sizes?
- The only legal `reqd_sub_group_size` is `primary` from sycl_ext_oneapi_named_sub_group_sizes (proposed)

How to deal with optional kernel features?

```
struct Construct {
  Base *storage; int kind;
  void operator()() const { /* new (storage) ... */ }
};
struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }
  auto get(properties_tag) const {
    return properties{assume_indirect_calls};
  }
};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
single_task(q, Use{storage});
```

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { /* use half */ }
  int bar() override { return 4; }
};
struct Derived2 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { /* use double */ }
  int bar() override { return 6; }
};
```

Which optional features Construct & Use kernels are using?

An implementation may not raise a compile time diagnostic or a run time exception merely due to speculative compilation of a kernel for a device when the application does not actually submit the kernel to that device.

5.7. Optional kernel features

`kernel_not_supported`

Exception thrown when a kernel uses an optional feature that is not supported on the device to which it is enqueued.

This exception is also thrown if a command group is bound to a kernel bundle, and the bundle does not contain the kernel invoked by the command group.

4.13.2. Exception class interface

How to deal with optional kernel features?

- We know which virtual functions are used by every kernel!
  - Thanks to the `assume_indirect_calls` property

- But there is only one property? – Not quite:
  - Virtual functions are bundled into sets:
    - `assume_indirect_calls` == `assume_indirect_calls_to<void>`
    - `indirectly_callable` == `indirectly_callable_in<void>`

- Set is a C++ typename – same rules as for SYCL kernel names

How to deal with optional kernel features?

```
struct Construct {
  Base *storage; int kind;
  void operator()() const { /* new (storage) ... */ }
};

struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }
  auto get(properties_tag) const {
    // uses both fp64 and fp16
    return properties{assume_indirect_calls};
  }
};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
single_task(q, Use{storage});
```

```
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { /* use half */ }
  int bar() override { return 4; }
};
struct Derived2 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable)
  int foo() override { /* use double */ }
  int bar() override { return 6; }
};
```

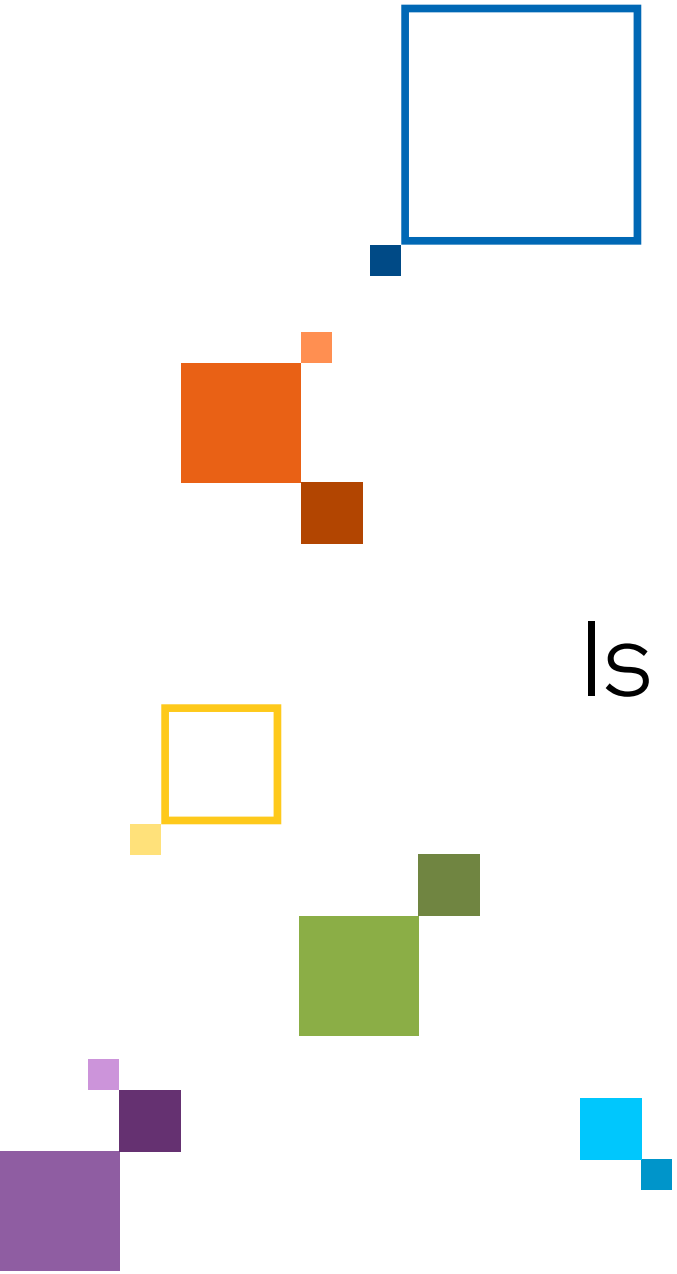
A kernel uses optional features that are in use by ***every*** function in ***all*** sets it uses

How to deal with optional kernel features?

```
struct Construct {
  Base *storage; int kind;
  void operator()() const { /* new (storage) ... */ }
};
template <typename Set>
struct Use {
  Base *storage;
  void operator()() const {
    storage->foo();
  }
  auto get(properties_tag) const {
    return properties{assume_indirect_calls_to<Set>};
  }
};

storage = malloc_device<byte>(q, 128);
single_task(q, Construct{storage, user_provided_kind});
if (q.get_device().has(aspect::fp64))
  single_task(q, Use<set_fp64>{storage});
else if (q.get_device().has(aspect::fp16))
  single_task(q, Use<set_fp16>{storage});
else ...
```

```
struct set_fp16; struct set_fp64;
struct Base {
  virtual int foo() { throw new exception("todo"); }
  virtual int bar() { return 2; }
};
struct Derived1 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable_in<set_fp16>)
  int foo() override { /* use half */ }
  int bar() override { return 4; }
};
struct Derived2 : public Base {
  SYCL_EXT_ONEAPI_FUNCTION_PROPERTY(
    indirectly_callable_in<set_fp64>)
  int foo() override { /* use double */ }
  int bar() override { return 6; }
};
```

Now the code is portable



Is it that complex in CUDA & OpenMP?

Comparison

Limitations, No is better	OpenMP	SYCL	CUDA
Virtual function markup	Yes	Yes	Yes
Kernel markup	No 👍	Yes	No 👍
All overrides must follow host/device specification	No 👍	No 👍	Yes
Object with virtual functions can ***not*** be used as a kernel argument	No 👍	No 👍	Yes

Features, Yes is better	OpenMP	SYCL	CUDA
Construct on host Use on device	Yes 👍	No	No
Construct on device Use on host	No	No	No
Construct on device Use on ***another*** device	No	No	No

Implementation notes

SPIR-V support: SPV_INTEL_function_pointers

- OpConstantFunctionPointerINTEL
  - Equivalent to C/C++ address-of operator

- OpFunctionPointerCallINTEL
  - Clone of OpFunctionCall, but for function pointers

The spec is being refreshed in intel/llvm#17857

Overall implementation design overview

- Put every virtual functions set into a separate device image
  - If any optional kernel features are in use, create a copy but make all functions empty
    - Used to avoid speculative compilation, but still provide symbols
  - Remember which other device images use those sets

- For a device image with a kernel
  - Link all device images that contain used virtual functions sets
    - Possibly choosing an alternative version with empty functions

- When targeting SPIR-V, the above happens at runtime
  - Otherwise, it happens during app compilation step

Where to look for more detail?

- intel/llvm#10540 proposes the extension specification
  - As well as implementation design document

- sycl/test-e2e/VirtualFunctions to see what works and what doesn't

- Initial support is available since oneAPI 2025.1 release
  - Support for non-SPIR-V targets is missing
  - Kernels and virtual functions they use must be in the same translation unit
  - Optional kernel features handling is not complete

What's next?

- Implementation needs to be finished
  - Resolving all aforementioned limitations
- Incorporating your feedback!
  - Share it at intel/llvm
- Support for plain function pointers!


intel