13th International Workshop on OpenCL and SYCL

IWOCL 2025

One header to rule them all – evil, or not?

Alexey Sachkov, Intel

Greg Lueck, James Brodman and John Pennycook, Intel.

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

KHRONOS GROUP

- Problem statement
  - Is it really a problem?
- Possible solution
  - Challenges and questions
  - Proposal to kick off a discussion
- Technical details
  - For spec authors and implementors
- Summary

Problem statement

<sycl/sycl.hpp> hurts compile-time a lot

```
// clang++ -fsycl -fsycl-device-only -fsyntax-only file.cpp

#include <sycl/sycl.hpp>

int main() { return 0; }
```

`-fsyntax-only`: run the preprocessor, parser and semantic analysis stages

`-fsycl-device-only`: run device compilation pass only

Will take ***seconds*** to finish!

The problem is not unique to a single implementation

```
// clang++ -fsyntax-only -include bits/stdc++.h empty-file.cpp
```

vs

```
// clang++ -fsyntax-only -include path/to/sycl/sycl.hpp empty-file.cpp
```

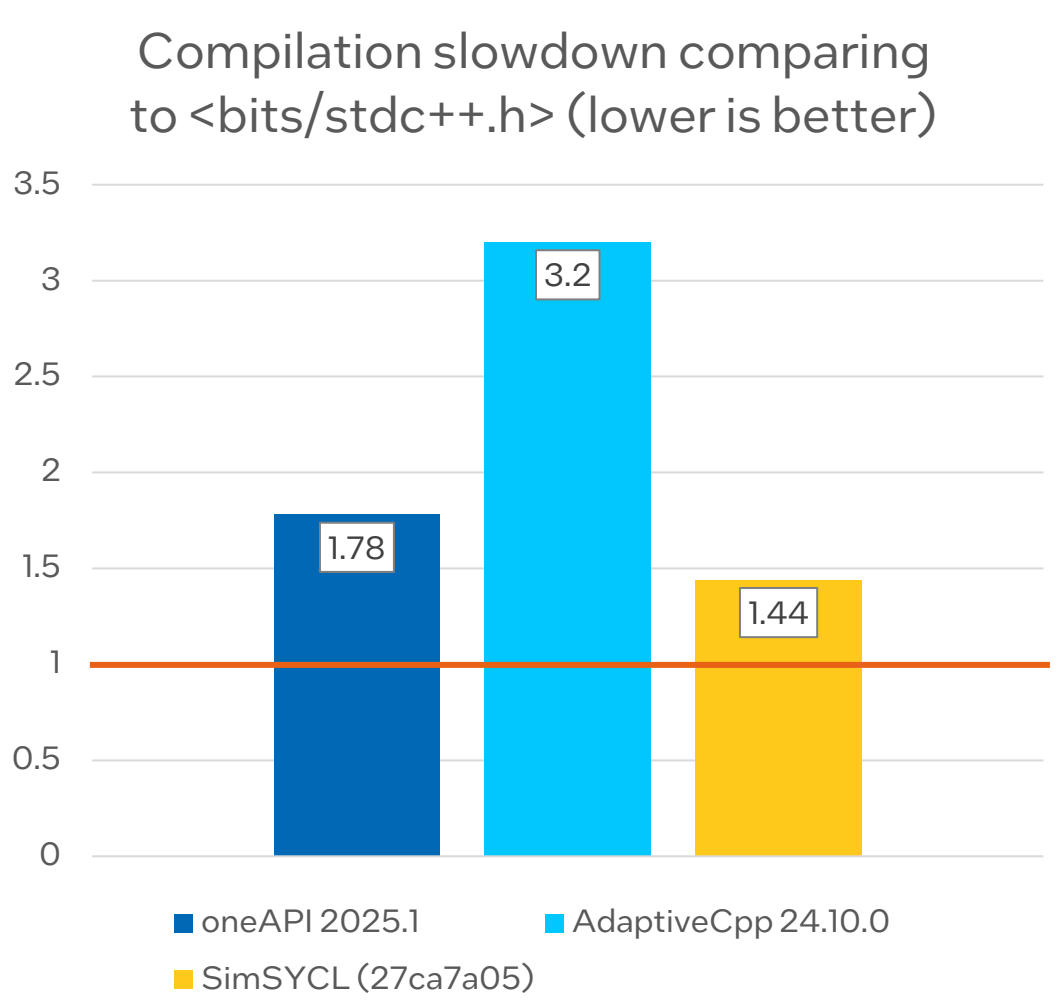
SW setup:

- Ubuntu 24.04
- clang 18.1.3

HW setup:

- Intel(R) Xeon(R) Platinum 8360Y CPU @ 2.40GHz

The problem is not unique to a single implementation



Does ***not*** represent actual compile-time of corresponding implementations.

Both oneAPI and AdaptiveCpp have plenty of extensions.

oneAPI is the only conformant implementation.

Why is it so bad?

- sycl.hpp includes plenty of things
  - Numerous core SYCL 2020 features
  - Plenty of extensions in some implementations

- SYCL uses modern C++ features
  - Templates are heavy

```
class queue {
  event prefetch(const void *Ptr, size_t Count) {
    // To process prefetch, queue::submit must be instantiated
    // even if translation unit is empty/doesn't use queue at all
    return submit([=](handler &CGH) { CGH.prefetch(Ptr, Count); });
  }
};
```

What can we do about it?

Provide a ***separate*** set of fine-grained headers for better control

What is SYCL?

SYCL 2020 rev. 9 `class` keyword, ignoring `enum`s

```
accessor
atomic
atomic_ref
backend_traits
buffer
context
context_bound
device
device_event
device_image
event
exception
exception_list
group
h_item
handler
host_accessor
host_sampled_image_accessor
host_unsampled_image_accessor
id
interop_handle
item
kernel
kernel_bundle
kernel_handler
kernel_id
local_accessor
marray
multi_ptr
nd_item
nd_range
platform
private_memory
property_list
queue
range
reducer
sampled_image
sampled_image_acessor
specialization_id
stream
sub_group
unsampled_image
unsampled_image_accessor
use_host_ptr
use_mutex
usm_allocator
vec

+half (underspecified)
+math built-ins (free functions)
+usm (free functions)
+the rest of the spec
```

How do we split that?

- Extensions can be outlined into separate headers
  - Should be easy, right?

- We could provide a separate header for ***every*** core class/feature
  - Wouldn’t be that an overkill?

- We could group core classes/features somehow
  - But how?
    - Usage frequence, common use-cases?
  - Would we provide some “core.hpp” with bare minimum functionality?

How do we split that? (cont.)

- How strict we would like to be?
  - Is it ok for <sycl/accessor.hpp> to include <sycl/buffer.hpp>?

- What about inter-dependent headers?
  - Should <sycl/half.hpp> include corresponding math built-ins?
  - Should <sycl/math.hpp> include half and corresponding math built-ins?

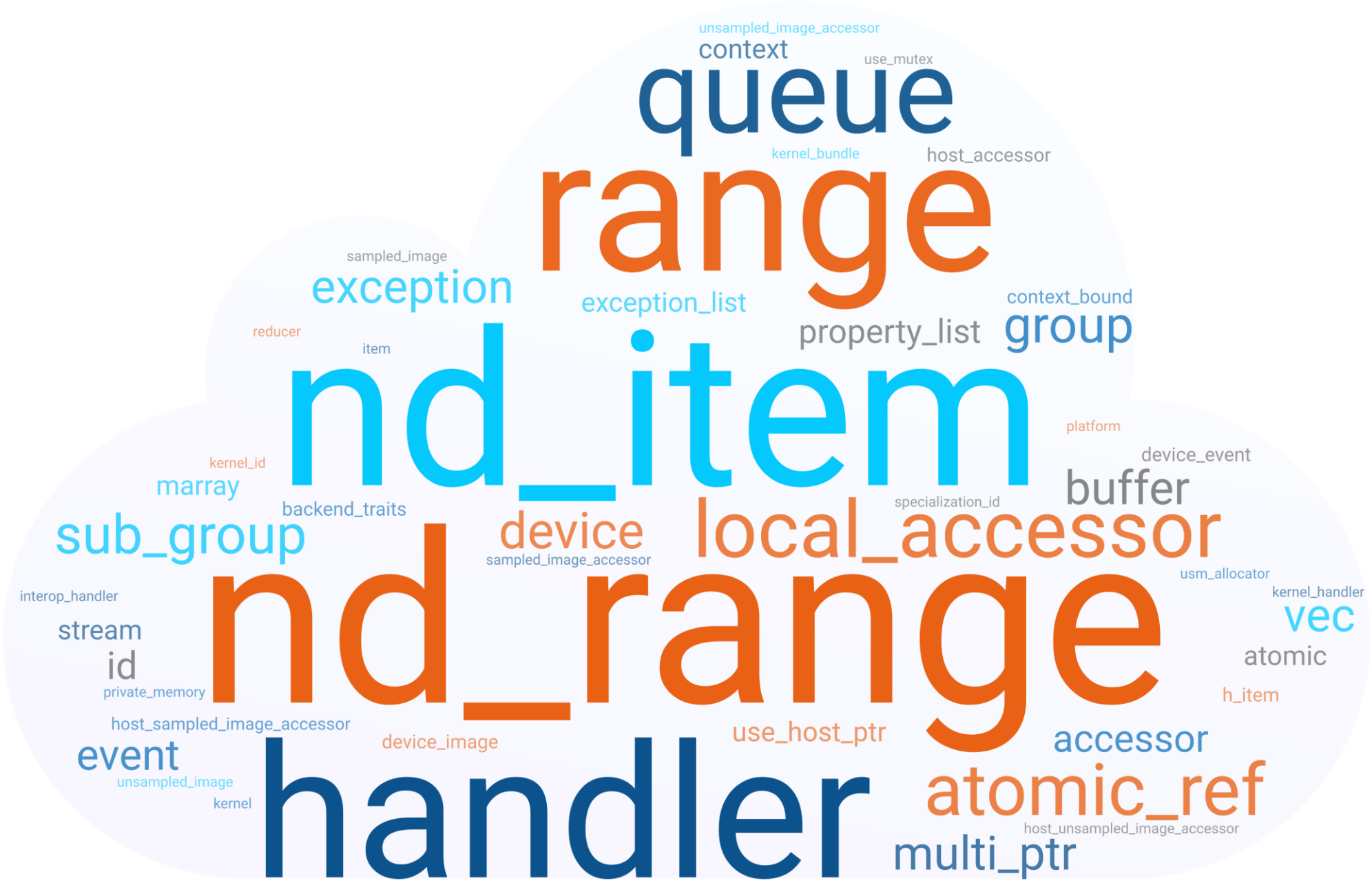
Which SYCL features are the most popular?

github.com: zjin-lcf/HeCBench

```
commit 88da95b23712706f0d4a35b2cc52cd396cc8842b
Author: Jin Z <5zj@zenith.ftpn.ornl.gov>
Date:   Mon Nov 4 13:44:40 2024 -0500

    [mpc-hip] add the HIP example; TODO: GPU kernels support a warpsize of 64
```

465 SYCL benchmarks


unsampled_image_accessor
context
use_mutex
queue
kernel_bundle
host_accessor
range
sampled_image
exception
exception_list
context_bound
reducer
property_list
group
item
nd_item
platform
device_event
kernel_id
marray
buffer
backend_traits
specialization_id
sub_group
device
local_accessor
sampled_image_accessor
usm_allocator
interop_handler
kernel_handler
nd_range
vec
stream
id
atomic
private_memory
h_item
host_sampled_image_accessor
use_host_ptr
accessor
event
device_image
handler
atomic_ref
unsampled_image
kernel
host_unsampled_image_accessor
multi_ptr

Which SYCL features are the most popular?

SYCL 2020 rev. 9 `class` keyword, ignoring `enums`

```
100% queue
 94% nd_range
 93% nd_item
 90% range
 89% handler
 33% local_accessor
 17% atomic_ref
 10% sub_group
  6% device
  6% group
  4% buffer
  4% multi_ptr
  3% exception
  3% vec
  2% event
  2% accessor
  1% id
  1% property_list
  1% atomic
  1% context
  1% exception_list
```

```
<1% marray
<1% stream
<1% use_host_ptr
<1% platform
<1% reducer

backend_traits
context_bound
device_event
device_image
h_item
host_accessor
host_sampled_image_accessor
host_unsampled_image_accessor
interop_handle
item
kernel
kernel_bundle
kernel_handler
kernel_id
private_memory
```

```
sampled_image
sampled_image_acessor
specialization_id
unsampled_image
unsampled_image_accessor
use_mutex
usm_allocator
```

Possible ways to split the header

One of many

Three headers for the most used features under the sky

- <sycl/queue.hpp>
  - Also includes handler
- <sycl/index_space.hpp>
  - id, range, nd_item, nd_range, item, group, h_item, sub_group
- <sycl/usm.hpp>
  - usm_allocator, USM free functions for allocation/deallocation

Seven for the foundation made of stone

- <sycl/buffer.hpp>
- <sycl/accessor.hpp>
- <sycl/runtime.hpp>
  - device, platform, context, event
- <sycl/atomic.hpp>
  - atomic_ref
- <sycl/vec.hpp>
- <sycl/marray.hpp>
- <sycl/builtins.hpp>
  - This will likely be a set of headers: group, common, integer, geometric, etc.

Nine for the remains doomed to be forgotten

- <sycl/stream.hpp>
- <sycl/interop.hpp>
  - interop_handle, get_native_*
- <sycl/kernel_bundle.hpp>
  - kernel_bundle, specialization_id, kernel, kernel_id, device_image
- <sycl/backend.hpp>
  - backend_traits, make_*
- <sycl/images.hpp>
  - accessors and samplers
- <sycl/half.hpp>
- <sycl/hierarchical_parallelism.hpp>
  - private_memory
- <sycl/properties.hpp>
- off by one error here

One to rule them all

- <sycl/sycl.hpp>
  - Includes everything else

Example

src/aobench-sycl/ao.cpp

from zjin-lcf/HeCBench repo

Commit c1c9af29

Uses:

- queue, gpu_selector_v, property::queue::in_order
- range, nd_range, nd_item
- malloc_device, free
- sqrt, cos, sin, fabs

```
queue q(gpu_device_selector, ...);
...  = malloc_device<...>(...);

q.memcpy(...);
q.submit([&](handler &cgh) {
  cgh.parallel_for(gws, lws), [=] (nd_item ...) {
    sin(...);
    cos(...);
  }
});

q.memcpy(...);
free(...);

// 396 lines of code in total
```

Example

```
clang++ -fsycl aobench.cpp
```

```
#include <sycl/sycl.hpp>
```

Used features:

```
sycl::queue
sycl::gpu_selector_v
sycl::range
sycl::nd_range
sycl::nd_item
sycl::malloc_device
sycl::free
sycl::sin, sycl::cos
sycl::sqrt, sycl::fabs
sycl::property::queue::in_order
```

Example

`clang++ -fsycl aobench.cpp`

Using whatever is available today (Feb 27th) at intel/llvm!

```
- #include <sycl/sycl.hpp>
```

```
+ #include <sycl/queue.hpp>
+ #include <sycl/usm.hpp>
+ #include <sycl/builtins.hpp>
+ #include <sycl/properties/queue_properties.hpp>
```

Used features:

```
sycl::queue
sycl::gpu_selector_v
sycl::range
sycl::nd_range
sycl::nd_item
sycl::malloc_device
sycl::free
sycl::sin, sycl::cos
sycl::sqrt, sycl::fabs
sycl::property::queue::in_order
```

Example

`clang++ -fsycl aobench.cpp`

Using whatever is available today (Feb 27th) at intel/llvm!

```
- #include <sycl/sycl.hpp>
```

```
+ #include <sycl/queue.hpp>
+ #include <sycl/usm.hpp>
+ #include <sycl/builtins.hpp>
+ #include <sycl/properties/queue_properties.hpp>
```

~4.2 seconds vs ~2.5 seconds

queue.hpp includes accessor and buffer “today”

There are also extensions!

An extension may define new `#include` *files under the* `"sycl"` *path. The path prefix* `"sycl/ext/<vendorstring>"` *is reserved for this purpose.*

For example, the Acme vendor could add a header file `"sycl/ext/acme/fancy.h"` *and be guaranteed that it would not conflict with other extensions or with future versions of the core SYCL specification.*

6.3.5 Include file paths

No one seems to outline extensions into separate headers

- sycl_khr_default_context #624
- sycl_khr_group_interface #638
- sycl_khr_free_function_commands #644
- sycl_khr_addrspace_cast #650
- sycl_khr_work_item_queries #682
- sycl_khr_queue_empty_query #700
- dozens of oneapi and intel extensions at intel/llvm

But why is that?

1. Lack of awareness about the compilation times problem
2. Sometimes it is not clear how it can be done

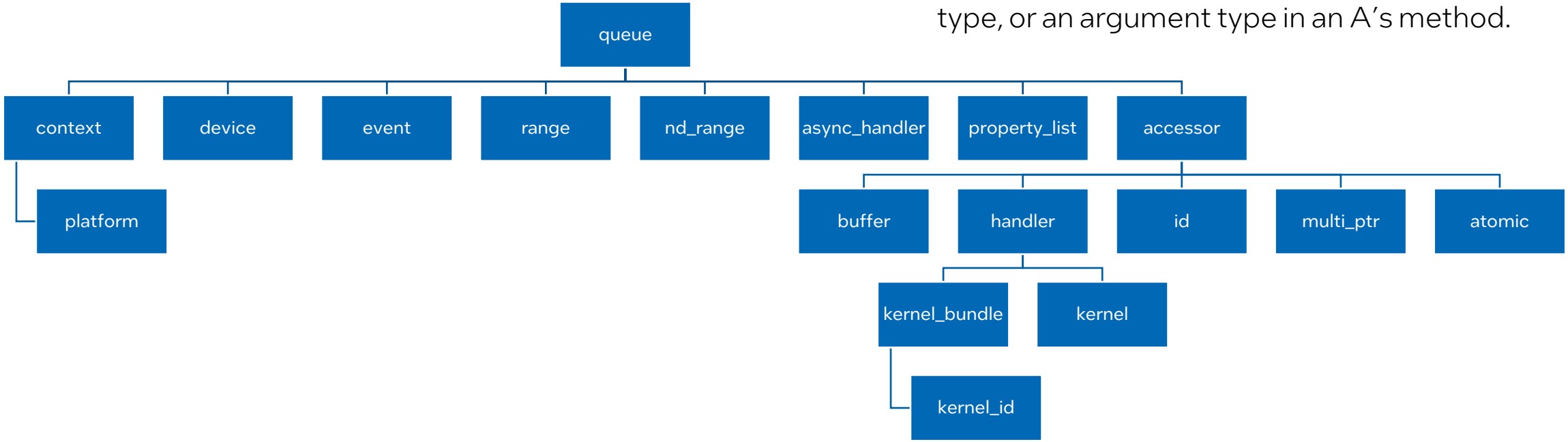
Implementation details

It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to.

SYCL classes are tightly interconnected

Read this as a tree.
Edges are directed top to bottom.

Edge A -> B means that B is either a return type, or an argument type in an A's method.



Does it mean that queue.hpp should include all other headers?

Implementation options to break dependencies

- (prerequisite) Use forward-declarations
- Move functions to other header files
  - Always possible, but can lead to bad user experience
- Move functions to library
  - Not always possible, increases ABI surface

Breaking include chains


Move to library


Move to separate header

```
// queue.hpp

class context;

class queue {
  // Implementation can be placed into a
  // library, or into context.hpp
  context get_context() const;
}
```

```
// app.cpp

#include <sycl/queue.hpp>
// #include <sycl/context.hpp> is missing!

int main() {
  queue q;

  // error: calling 'get_context' with
  // incomplete return type 'context'
  auto ctx = q.get_context();

}
```

Breaking include chains


Move to library


Move to separate header

```
// buffer.hpp

// T can be a user-defined type!
template<typename T, int Dimensions,
    typename AllocatorT>
class buffer {
public:
  // Implementation *cannot* be placed into
  // a library without using type erasure.
  // Out-of-class definition is possible
  template<access_mode Mode, target Target>
  accessor<T, Dimensions, Mode, Target>
  get_access(handler& commandGroupHandler);
};
```

```
// app.cpp

#include <sycl/buffer.hpp>
#include <sycl/queue.hpp>
// missing #include <sycl/accessor.hpp>!

int main() {
  queue q;
  buffer buf(/* ... */);
  q.submit([&](handler &cgh) {
    // error: implicit instantiation of
    // undefined template accessor<...>
    auto acc =
      buf.get_access<access_mode::write_only>(cgh);
  }
}
```

Breaking include chains


Move to library


Move to separate header

```cpp
// queue.hpp
class queue {
public:
  // Implementation can be placed into
  // a library.
  // Out-of-class definition is possible
  bool do_a_thing_from_ext() const noexcept;
};
```

```cpp
// app.cpp

#include <sycl/queue.hpp>
// #include <sycl/ext/queue.hpp> is missing!

class property;

int main() {
  queue q;
  // app.cpp:(.text+0x1d): undefined reference to
  // `bool queue::do_a_thing_from_ext() const`
  // clang++: error: linker command failed ...
  q.do_a_thing_from_ext();

  // ...
}
```

Breaking include chains

✗ Move to library

✗ Move to separate header

```
// handler.hpp

class handler {
public:
  // Implementation *cannot* be placed into
  // a library.
  // Out-of-class definition is possible
  template<auto &S>
  void set_specialization_constant(
    typename decltype(S)::value_type value);
};
```

```
// app.cpp

#include <sycl/queue.hpp>
#include <sycl/specialization_id.hpp>
// missing #include <sycl/kernel_bundle.hpp> ?

specialization_id<int> spec_id(0);

int main() {
  queue q;
  q.submit([&](handler &cgh) {
    // app.cpp:(.text+0x19): undefined reference to
    // `_ZN7handler27set_specialization...`
    // clang++: error: linker command failed ...
    cgh.set_specialization_constant<spec_id>(42);
  }
}
```

Summary

- You pay for what you don't use with existing <sycl/sycl.hpp>
  - Applicable to both extensions and core features

- This can be solved
  - Will likely require a lot of feedback that you can share in KhronosGroup/SYCL-Docs#780

- To SYCL spec writers: please consider this issue in your work!
  - Especially when working on new extensions
    - One day, this will hopefully be a part of the core spec, so be prepared

- To SYCL implementors: please consider this issue in your work!
  - Clang's "-ftime-trace" is a great tool to hunt down bottlenecks


intel