Since 2020, it has been considered a name, not an acronym…

SYCL: SYstem-wide Compute Language

NPRG042: Programming in Parallel Environment

Martin Kruliš

Motivation

- Parallel programming is getting out of hand
  - Many device types (CPU, GPU, FPGA, …)
  - Even similar devices (GPUs) have different approaches (NVIDIA vs AMD)
- Holy grail of parallel computing
  - Universal parallel framework
    - Allowing code migration across devices
    - Enabling all low-level optimization
  - Raises lots of issues
    - Coding model, portability, device discovery, compilation

History

1985 – First commercial FPGA

1996 – 3Dfx Voodoo 1 (computing GPU)

2001 – GPU has programmable parts

2003 – First desktop multicore CPUs

2006 – Unified shader architecture in HW

2006 – Ageia PPU (PhysX) accelerator

2007 – NVIDIA CUDA, AMD Stream SDK

2009 – OpenCL, Direct Compute

2011 – Intel Xeon Phi (Knights Ferry)

2012 – NVIDIA Kepler Architecture (CC 3.5)

2013 – OpenMP 4.0 (accelerator support)

2014 – SYCL announced

2015 – Intel buys Altera (an FPGA co.)

2016 – ROCm (from AMD)

2018 – NVIDIA Volta architecture

2020 – Intel oneAPI launched

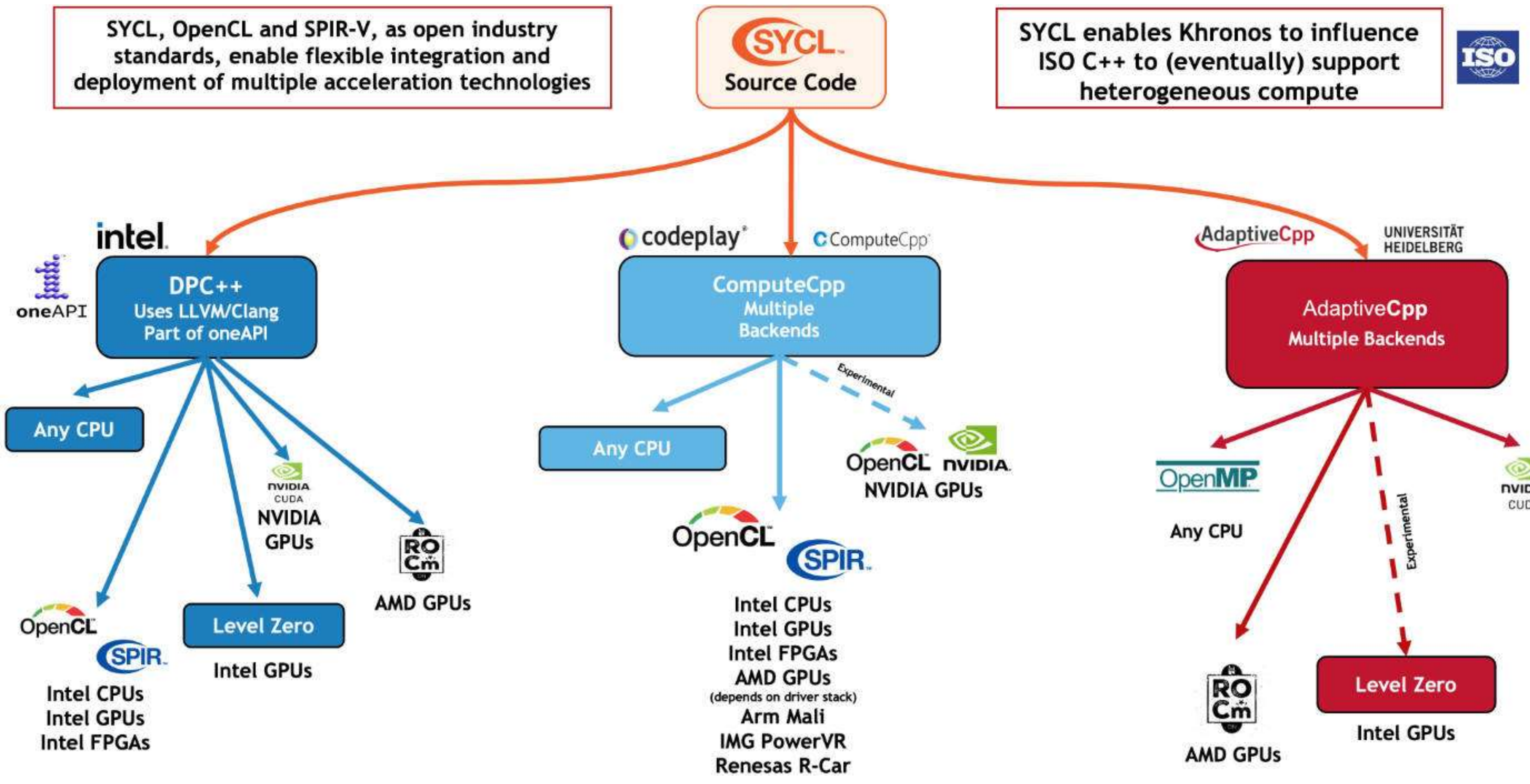
2020 – NVIDIA Ampere architecture

2022 – Latest version of SYCL

About SYCL

- High-level parallel programming model
  - Extends (modern) C++, requires special compiler
    - Platform and device detection
    - Buffer (data) management
    - Kernel (function) compilation and execution
    - Synchronization, dependency management, scheduling
  - Single source (both host control and device code)
  - It’s a standard (not a library)
    - Multiple implementations exist (we will use Intel oneAPI)
  - Target many heterogeneous platforms
    - CPU, GPU, FPGA, …

About SYCL



from https://www.khronos.org/sycl/

Computing Model

A function to be executed on the device

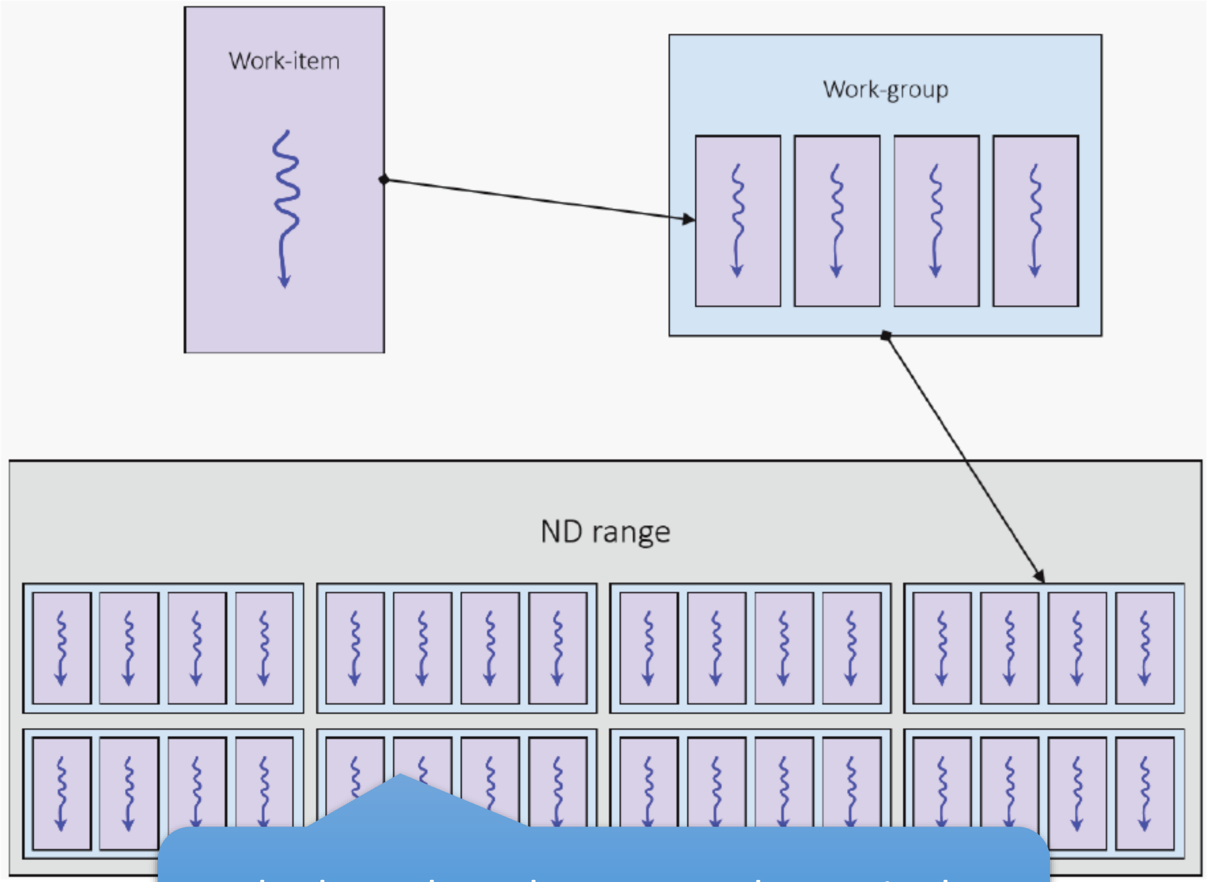
Kernel

Handles compilation and execution

SYCL

queue

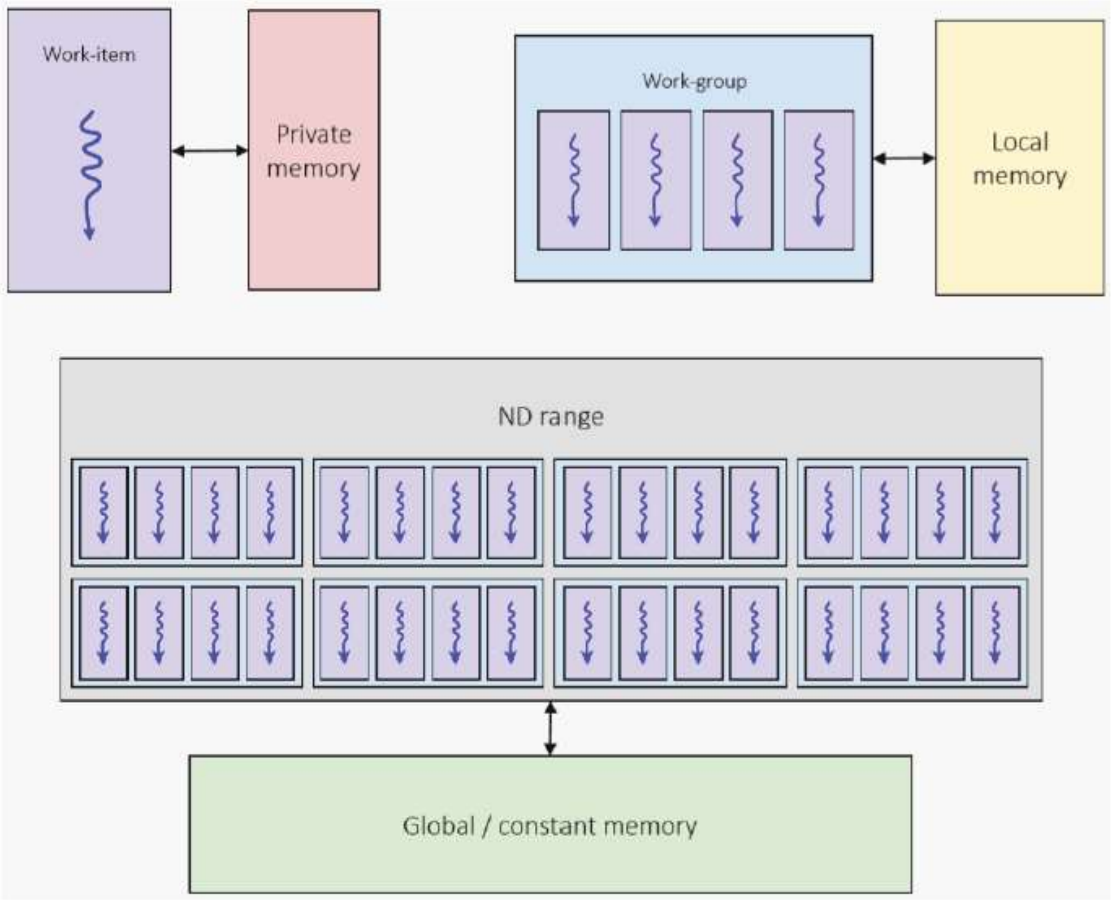
device



The kernel can be executed as a single task, but we usually use data parallelism

Memory Model

- Types of memory
  - Private memory
    - Local variables of a kernel function
  - Local memory
    - Special, shared among work-group
  - Global memory
    - Buffers shared by the whole range
    - Can be transferred to/from host
  - Constant memory
    - Global, read-only



Queues

- Queue
  - Interface for commands
  - Associated with one device
  - Out of order by default
    - **`in_order`** property for constructor
- Command groups
  - Series of commands that need to be executed in a particular order
    - Kernel execution
    - Copying data
    - Waiting for other commands

```
sycl::queue gpuQueue(
  sycl::gpu_selector_v
);
```

Easiest way, selector is a function for ranking devices

```
gpuQueue.submit(
  [&](sycl::handler &h){
     // copy data
     // execute kernel
  }
);
```

The lambda is the command group factory

Kernel Execution

- Kernel
  - Lambda or class with operator()
    - Not fnc pointer nor std::function
    - Must capture/store **by value**
    - No dynamic allocation
    - No dynamic polymorphism
    - No recursion
  - At most one kernel in CG
  - Lambdas can be named by classes

```
class name_holder {};
q.submit([&](sycl::handler &h) {
  h.single_task<name_holder>([=]() {
    // kernel code
  });
}).wait();

struct my_kernel {
  void operator()(){
    // kernel code
  }
};
...
q.submit([&](sycl::handler &h) {
  h.single_task(my_kernel{});
}).wait();
```

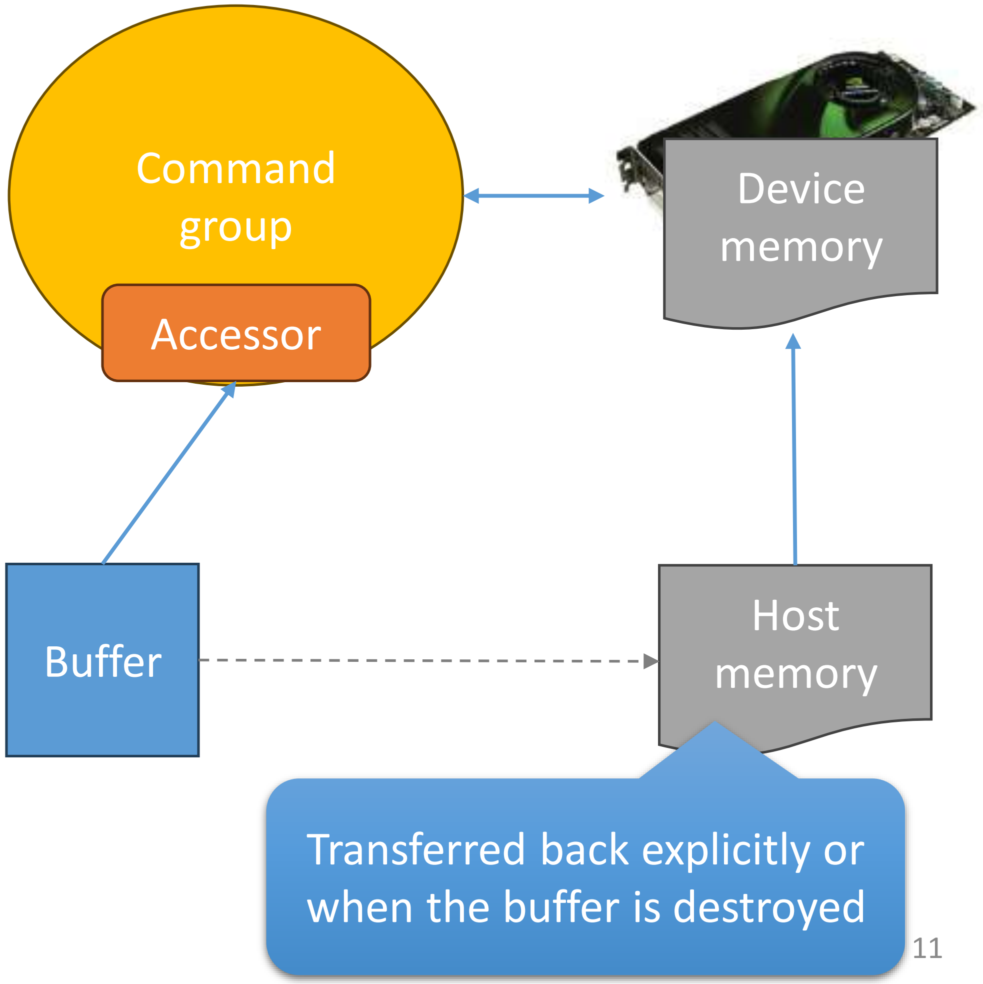
Kernel Execution

- Data parallelism
  - Executes the same kernel in many threads (work items)
  - Kernel is given an index
    - May have up to 3 dimensions
    - May be hierarchical (work-groups)
  - `sycl::id` – simple index
  - `nd_item` indexation
    - `get_global_id()`
    - `get_local_id()`
    - …

```
q.submit([&](sycl::handler &h) {
  h.parallel_for(1024, [=](auto i) {
    acc[i] = ...
  });
}).wait();

h.parallel_for(sycl::range{32, 32},
  [=](sycl::id<2> i) {
    accTo2DBuf[i] = ...
});

h.parallel_for(sycl::nd_range<1>{
  sycl::range<1>{1024}, // global
  sycl::range<1>{64},   // local
}, [=](sycl::nd_item<1> item) { ...
```

Memory Abstraction

- Buffer
  - Logically detached from physical mem.
    - Usually have 1 copy in host memory
  - May be in multiple copies
  - Transferred as needed
    - Or explicitly
- Accessor
  - Used to access buffer in kernel
  - Announcement of intention
    - So SYCL may transfer the data



Memory Abstraction

- Buffer construction
  - Tied to a host memory

```
sycl::buffer<float, 1> buf(ptr, size);
```

- Accessor
  - Mem. usage specifiers
  - Acts as a smart pointer

```
// in the command group
sycl::accessor acc(buf, handler,
  sycl::write_only, sycl::no_init);

float f = acc[i];
```

- Explicit copying

```
handler.copy(acc, hostPtr);
```

- Host accessor

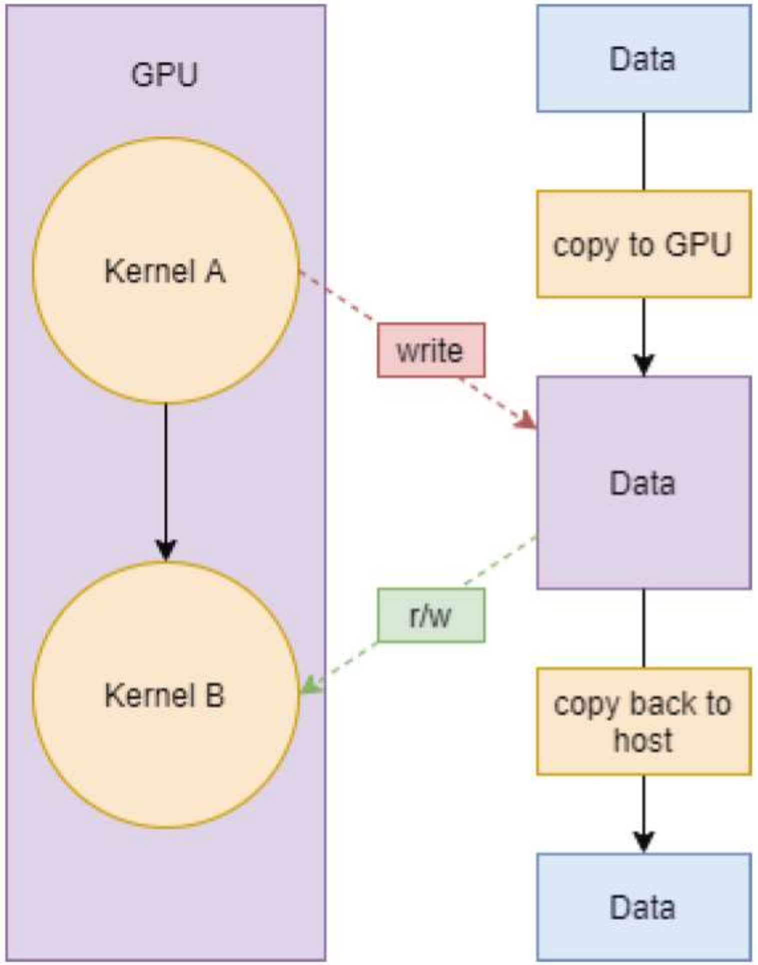
```
auto hacc = buf.get_host_access();
```

Synchronization

- Host side
  - `queue.wait();`
  - Events
    - Returned by queue operations
      ```
      auto event =
                queue.submit(…);
      ```
    - `event.wait();`
    - Events can be passed as dependencies to some operations
      `queue.submit(…, event, …)`
- In kernel
  - Barriers
    - `nd_item::barrier()`
    - Only within the work group!
  - Atomic operations
    - Similar to C++
      ```
      auto v =
      sycl::ext::oneapi::atomic_ref
      <int,order,scope,space>(a[0]);
      v += something;
      ```
    - NEVER implement a global barrier using atomics (causes DEADLOCK)!

Data dependencies

- Data dependencies tracking
  - Tracked by buffers
  - Accessors express the kernel's intent to use them
  - Scheduler executes CG (kernels) with respect to these dependencies
  - Example (on the right)
    - Read-after-write dependency between A and B
    - B is executed when A finishes



Error Handling

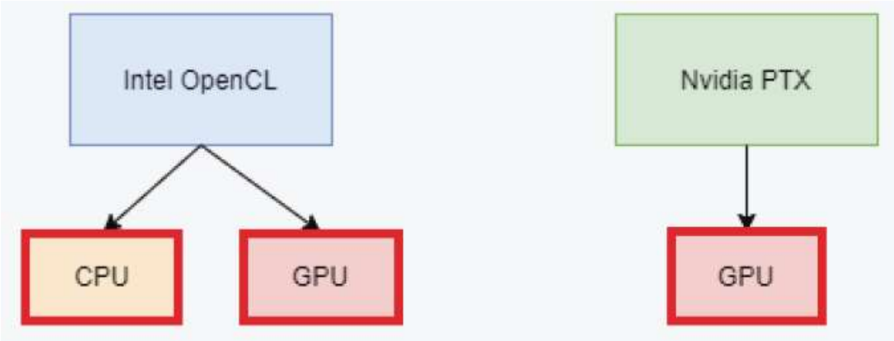
- Errors = exceptions
  - Synchronous
    - In user (main) thread
  - Asynchronous
    - Thrown by the scheduler
    - Exception handler may be passed to the queue
    - Used to re-throw exceptions in the main thread
    - Invoked by special methods of the queue

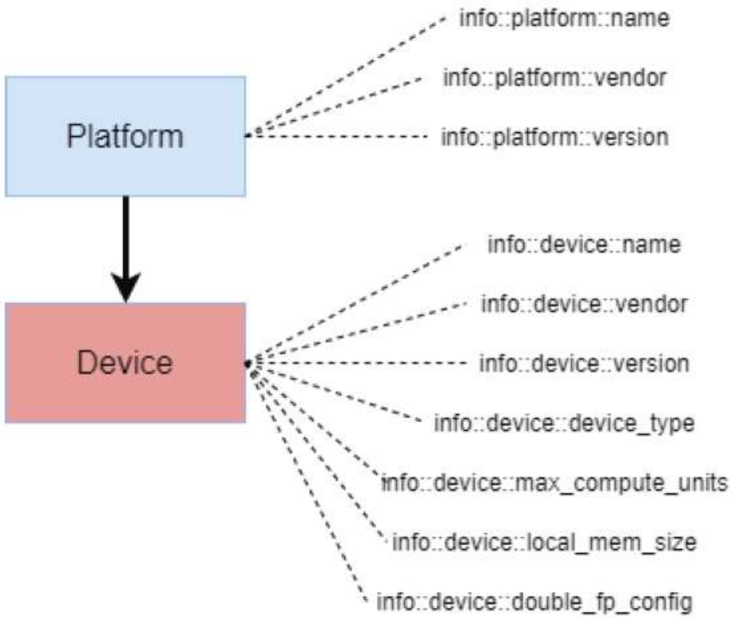
```
static auto exception_handler =
  [](sycl::exception_list l)
{
  for (std::exception_ptr const &e : l){
    try {
      std::rethrow_exception(e);
    }
    catch (std::exception const &e) {
      std::terminate();
    }
  }
};

...

sycl::queue q(sycl::default_selector_v,
  exception_handler);

q.throw_asynchronous();
q.wait_and_throw();
```

Device Detection

- Platform
  - An implementation of SYCL
    `sycl::platform::get_platforms();`
- Device
  - `myPlatform.get_devices();`
  - `sycl::device(sycl::gpu_selector_v);`
- Selectors

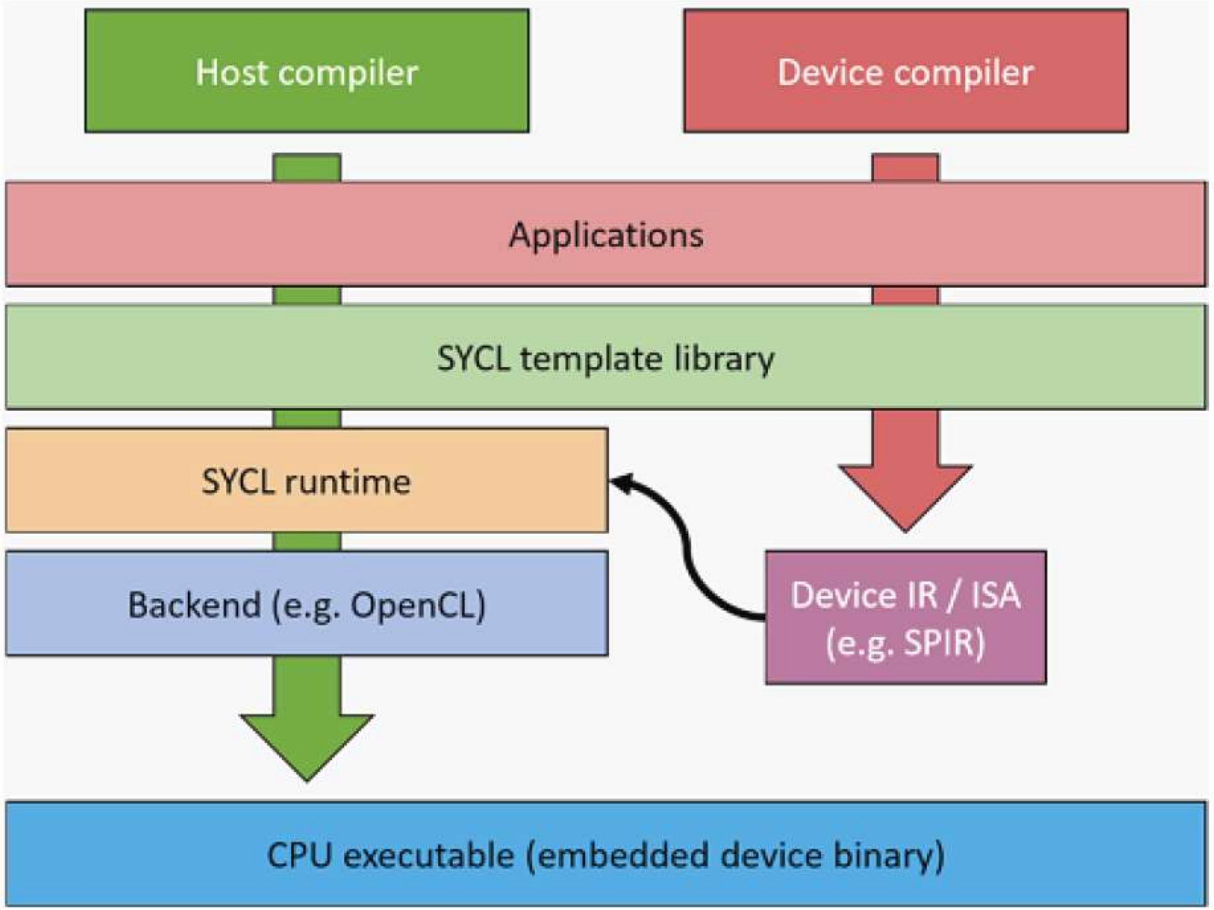
```
int fastest_gpu(const sycl::device &d) {
  if (d.is_gpu())
    return //… compute gpu speed rank
  else
    return -1;
}
```





Compilation

- Special compiler required
  - `icpx -fsycl -fsycl-targets=spir64_x86_64,nvptx64-nvidia-cuda`
  - Kernels are compiled in intermediate representation
    - Specified by target (backend)
  - SPIR
    - Standard Portable Intermediate Representation (by Khronos)
    - Used in OpenCL, Vulkan, …
    - Current version SPIR-V



References

- Khronos
  - https://www.khronos.org/sycl/
- SYCL Academy
  - https://github.com/codeplaysoftware/syclacademy
  - Most diagrams in this presentation were borrowed from them
- Intel oneAPI
  - https://www.intel.com/content/www/us/en/developer/tools/oneapi/base-toolkit.html

Discussion