The SYCL standard

and the ComputeCpp implementation

Lectures on Modern Scientific Programming 2016

Dániel Berényi

Wigner GPU Lab

What is SYCL?

- SYCL is a single-source parallel programming standard created by Khronos

What is SYCL?

- SYCL is a single-source parallel programming standard created by Khronos

- Yet another GPU API? Why should we care?

What is Khronos?

- The Khronos Group is a not for profit industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices.

KHRONOS™ GROUP

CONNECTING SOFTWARE TO SILICON

www.khronos.org

What is Khronos?

Standards managed by Khronos:

- OpenGL - graphics API (and spinoffs: WebGL, OpenGL ES)
- Vulkan – low level graphics API
- OpenCL – GPGPU API (separate dynamically loaded kernels written in C)

- Many more
  (3D asset management, Neural Network format, accelerated audio, media steaming, real-time vision and image processing, vector graphics acceleration)

What is Khronos?

What is Khronos?

- The standards are developed in close partnership with industry leader companies, hardware, and software vendors, game development studios and universities/research institutes.

- However, input from anyone is welcome and encouraged!

Why SYCL?

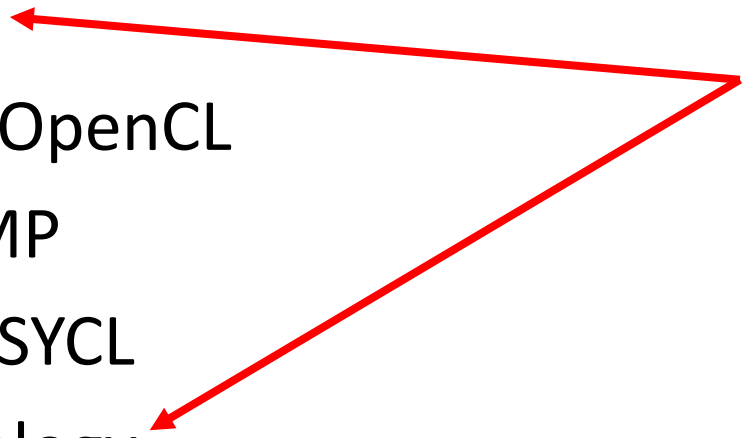
Current GPGPU technologies:

- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology

Why SYCL?

Current GPGPU technologies:

- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology

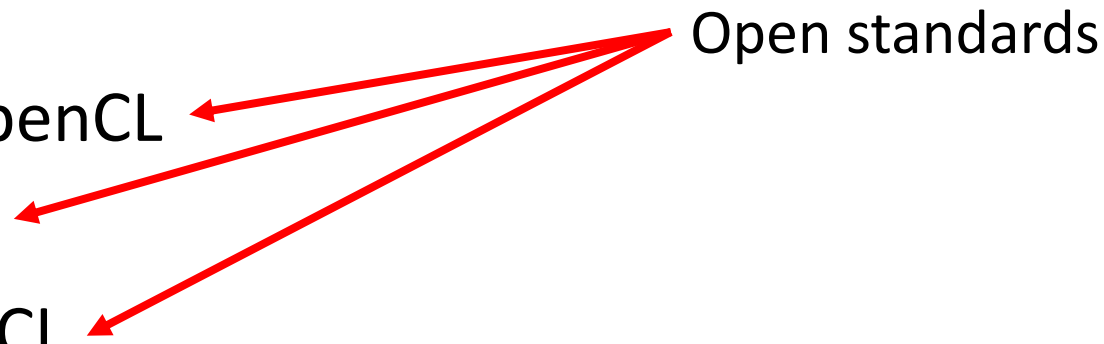


Vendor hardware specific technologies

Why SYCL?

Current GPGPU technologies:

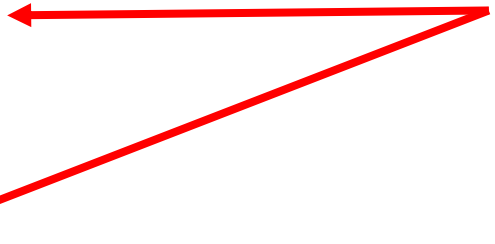
- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology



Why SYCL?

Current GPGPU technologies:

- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology



Open standards developed with industry wide support

Why SYCL?

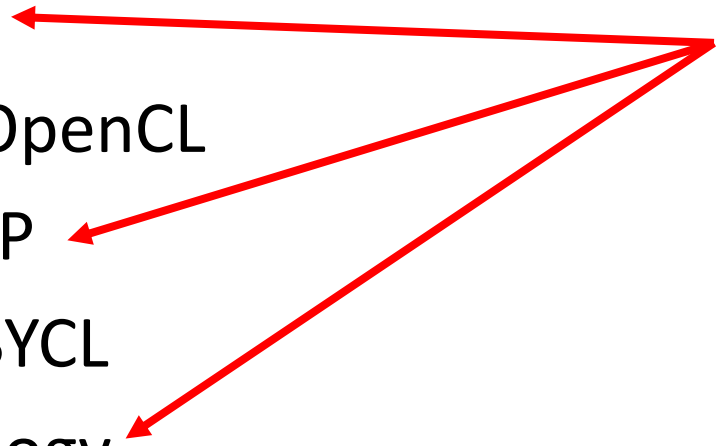
Current GPGPU technologies:

- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology

Why SYCL?

Current GPGPU technologies:

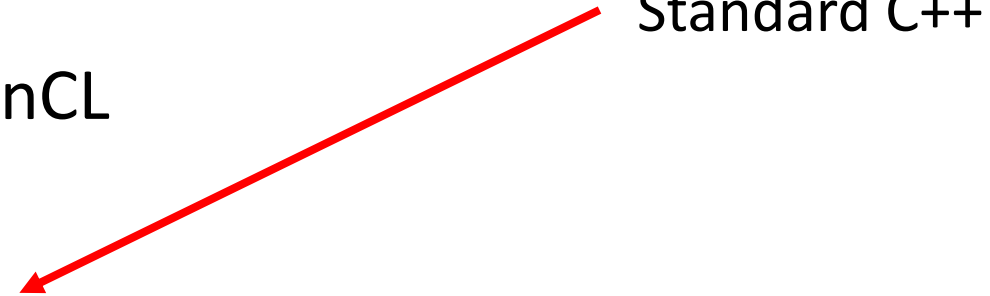
- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology



Why SYCL?

Current GPGPU technologies:

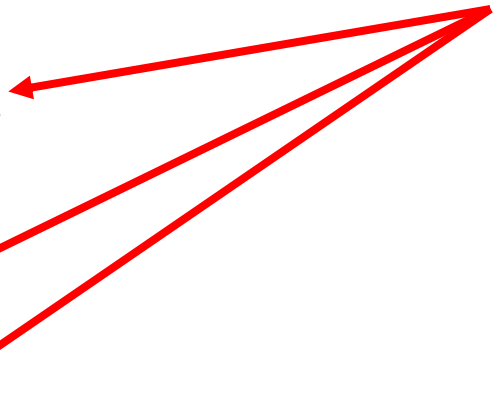
- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology



Why SYCL?

Current GPGPU technologies:

- NVIDIA's CUDA
- Khronos standard OpenCL
- Microsoft's C++AMP
- Khronos standard SYCL
- AMD's HCC technology



Heterogeneous computing (CPU, GPU, + others)

Why SYCL?

Current GPGPU technologies:

SYCL is the only technology, that is

- vendor neutral (and thus portable)
- standard C++11 without language extensions
- Supports heterogeneous parallelism (CPU, GPU, other...)

Why SYCL?

Current GPGPU technologies:

SYCL is the only technology, that is

- vendor neutral (and thus portable)
- standard C++11 without language extensions
- Supports heterogeneous parallelism (CPU, GPU, other...)

This means you can take complex C++ libraries and use them within GPU kernels out of the box!
(some GPU hw restrictions still apply)

Why SYCL?

SYCL is abstracting OpenCL!

OpenCL 1.2 technology is abstracted by SYCL 1.2

OpenCL 2.0+ technologies will be abstracted by SYCL 2.2 (provisional spec is released)

SYCL Implementations

SYCL 1.2 is a standard. What implementations are available?

- CPU reference implementation (now also for SYCL 2.2): triSYCL

- First GPU implementation by Codeplay: ComputeCpp
  - Based on clang and the SPIR intermediate

  Currently available as beta, for linux only
  For research and education it will be free later too.

ComputeCpp

ComputeCpp is using clang to compile code:

- full C++11 support
- in contrast to NVIDIA's CUDA, where they have a home solution for C++ compilation, that still lacking C++11 conformance

ComputeCpp is using clang to generate the OpenCL binaries in SPIR format

- Only those OpenCL implementations can work with it, that support SPIR loading (NVIDIA does not!)

ComputeCpp

Sample codes are available from Github:

https://github.com/codeplaysoftware/computecpp-sdk

ComputeCpp

Lets write a SYCL Application:

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

using namespace cl::sycl;
```

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

Construct the queue on the default device where commands will be executed on.

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

All commands are lambdas, taking a special parameter the `handler` as argument

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

All execution types are accessed through the handler

now we just launch a single task

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

The body of the task is a lambda

The template argument name is necessary to generate a unique name for the generated OpenCL kernel

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

Now we are creating a pre-sized SYCL stream to store output

And write a string to it.

Capture the stream on the task

ComputeCpp

ComputeCpp™

Lets write a SYCL Application:

```
int main()
{
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        stream os(1024, 80, cgh);
        cgh.single_task<class hello_world>(
            [=](){ os << "Hello, World!“ << endl; } );
    });
    return 0;
}
```

All SYCL handles (like myQueue) are released at the end of scope (RAII again!)

ComputeCpp

ComputeCpp™

Building a ComputeCpp application the recommended way: use cmake

```
project(computecpp_helloworld)
cmake_minimum_required(VERSION 3.2.2)
set(CMAKE_MODULE_PATH ./)
include(FindOpenCL)
include(FindComputeCpp)

include_directories(SYSTEM ${COMPUTECPP_INCLUDE_DIRECTORY})
include_directories(SYSTEM ${OpenCL_INCLUDE_DIR})

add_executable(hello_world.out ${CMAKE_CURRENT_SOURCE_DIR}/hello_world.cpp)
target_compile_options(hello_world.out PUBLIC -std=c++11 -Wall)
add_sycl_to_target(hello_world.out ${CMAKE_CURRENT_SOURCE_DIR}/hello_world.cpp
                                   ${CMAKE_CURRENT_BINARY_DIR})
```

ComputeCpp

Building using cmake:

```
> cmake .
> make
> ./hello_world.out
Hello, World!
```

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;
```

The name of our kernel

It is templated!
Different instances get generated for different template arguments

```
template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Our function will take a function f and two `std::arrays`, and applies the function elementwise, writing to the first array.

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Create a default queue

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Create a range representing the extent of the arrays, and later the number of threads to launch

ComputeCpp

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Create two buffers from the arrays that represent the arrays on the GPU side.

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Inside the command we ask access to the GPU buffers

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

Launch multiple GPU threads with parallel_for

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Specify the kernel name

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

N is the number of threads

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

Capture all the accessors AND the function f!!!

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

The kernels will receive their thread index via the lambda argument

```
template <typename T> class Map;

template <typename F, typename T, size_t n>
void map_kernel(F f, std::array<T, n>& arrB, std::array<T, n> const& arrA){
  queue myQueue; range<1> N{n};
  buffer<T, 1> bA(arrA.data(), N), bB(arrB.data(), N);

  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

ComputeCpp

ComputeCpp™

Now a more realistic example! Simple map (transform):

The kernels will receive their thread index via the lambda argument that we can use to index the accessors!

```
  myQueue.submit([&](handler& cgh){
    auto src = bA.template get_access< access::mode::read >(cgh);
    auto dst = bB.template get_access< access::mode::write >(cgh);
    cgh.parallel_for<class Map<T>>(N, [=](id<1> idx){ dst[idx] = f(src[idx]); });
  });
}
```

ComputeCpp

ComputeCpp™

Using our map:

```
int main()
{
    std::array<float, 4> A = {{1, 2, 3, 4}};
    std::array<float, 4> B;
    float c = 5.0f;
    map_kernel([=](float x){ return c*x; }, B, A);
    return 0;
}
```

ComputeCpp

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```

ComputeCpp

ComputeCpp™

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```

Type of elements that are stored

ComputeCpp

ComputeCpp™

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```

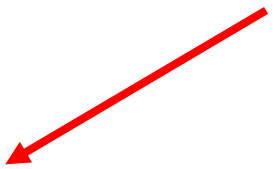
Dimensions of data (can be 1, 2, 3)

ComputeCpp

ComputeCpp™

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```



ComputeCpp

ComputeCpp™

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```

Target type,
now local memory

ComputeCpp

ComputeCpp™

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```

Number of elements of type T

ComputeCpp

ComputeCpp™

Accessing local memory in SYCL:

```
accessor<T,
         1,
         access::mode::read_write,
         access::target::local> scratch(range<1>(n), cgh);
```

Command handler instance

ComputeCpp

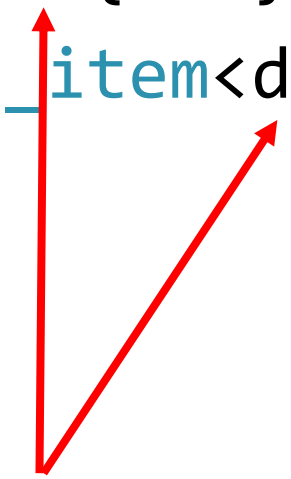
ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{



});
```



We are now in d dimensions!

ComputeCpp

ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{
  auto gid = id.get_global();
  auto gid_x = id.get_global(0);
  auto gid_y = id.get_global(1);
});
```

Get all d global indices

ComputeCpp

ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{
  auto gid = id.get_global();
  auto gid_x = id.get_global(0);
  auto gid_y = id.get_global(1);
});
```

Get global x index

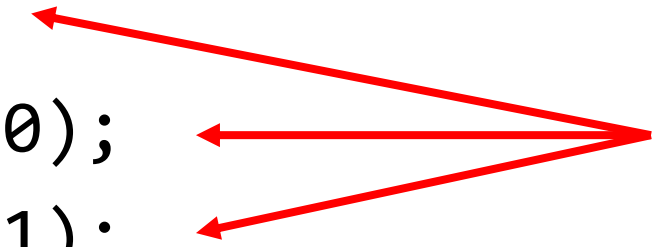
Get global y index

ComputeCpp

ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{
  auto lid = id.get_local();
  auto lid_x = id.get_local(0);
  auto lid_y = id.get_local(1);
});
```



Same for local indices

Also there are similar methods for groups

ComputeCpp

ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{
  access::fence_space flag = access::global_and_local;
  id.barrier(flag);
});
```

ComputeCpp

ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{
  access::fence_space flag = access::global_and_local;
  id.barrier(flag);
});
```

Execute a barrier on work-group threads

ComputeCpp

ComputeCpp™

Accessing thread related data and operations:

```
cgh.parallel_for<class myKernel>(range<d>{...},
                                 [=](nd_item<d> id)
{
  access::fence_space flag = access::global_and_local;
  id.barrier(flag);
});
```

Memory fence is specified by a `fence_space` value

ComputeCpp

With the preceding knowledge a reduction can be implemented in SYCL.

We avoid repeating the full code that is available in the SDK.

On a similar note, block grouped matrix multiplication can be also written.

ComputeCpp

Selecting the SYCL devices

When constructing a queue, a selector can be specified to select a subset of available devices:

```
queue myQueue(selector);
```

ComputeCpp

ComputeCpp™

Selecting the SYCL devices

When constructing a queue, a selector can be specified to select a subset of available devices:

```
default_selector selector;
queue myQueue(selector);
```

Some suitable default selector (implementation defined)

ComputeCpp

ComputeCpp™

Selecting the SYCL devices

When constructing a queue, a selector can be specified to select a subset of available devices:

```
gpu_selector selector;
queue myQueue(selector);
```

Select a GPU device

ComputeCpp

ComputeCpp™

Selecting the SYCL devices

When constructing a queue, a selector can be specified to select a subset of available devices:

```
cpu_selector selector;
queue myQueue(selector);
```

Select a CPU device

ComputeCpp

ComputeCpp™

Selecting the SYCL devices

When constructing a queue, a selector can be specified to select a subset of available devices:

```
my_device_selector selector;
queue myQueue(selector);
```

Custom device selector class

```
class my_device_selector :
          public device_selector
{
   public:
   my_device_selector() :
             device_selector() {}

  int operator()(const device&)
                    const override;
};
```

Example code

ComputeCpp

Using *multiple* SYCL devices...

In the earlier examples the OpenCL context was implicit. The sycl::context constructor can take multiple devices.

But how to get the devices?

You can use the OpenCL platform handle to create a sycl::platform and query the devices from it, and pass them to the sycl::context.

Now you can create a different queues for all the devices

ComputeCpp

OpenCL interoperability:

- The `sycl::platform, device, context`, … Objects can be constructed from native OpenCL handles
- Native handles can be extracted from `sycl::platform`… Objects
- See also this example.

OpenGL interoperability:

- Follows from the above☺

ComputeCpp

ComputeCpp™

Using images:

```
std::vector<float4> src_data(w*h);

image<2> myImage(src_data,
                 image_format::channel_order::RGBA,
                 image_format::channel_type::FLOAT,
                 range<2>(w, h));
```

ComputeCpp

ComputeCpp™

Using images:

```
std::vector<float4> src_data(w*h);

image<2> myImage(src_data,
                 image_format::channel_order::RGBA,
                 image_format::channel_type::FLOAT,
                 range<2>(w, h));
```

Order of color channels in the memory

ComputeCpp

ComputeCpp™

Using images:

```
std::vector<float4> src_data(w*h);

image<2> myImage(src_data,
                 image_format::channel_order::RGBA,
                 image_format::channel_type::FLOAT,
                 range<2>(w, h));
```

Representation of the color channels

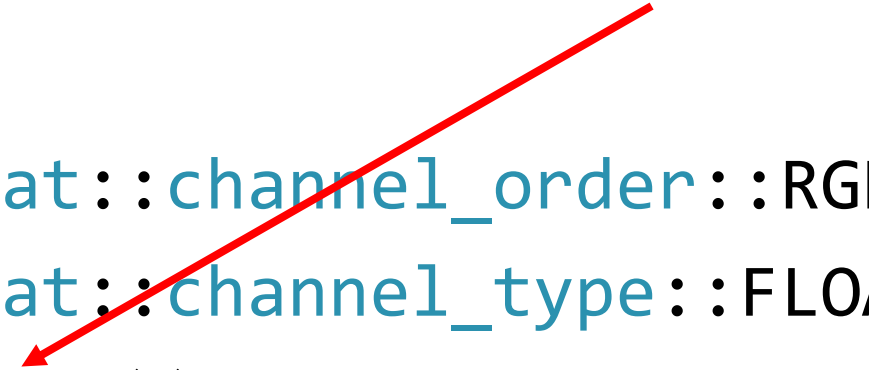
ComputeCpp

ComputeCpp™

Using images:

```
std::vector<float4> src_data(w*h);

image<2> myImage(src_data,
                 image_format::channel_order::RGBA,
                 image_format::channel_type::FLOAT,
                 range<2>(w, h));
```



ComputeCpp

ComputeCpp™

Using images:

```
myQueue.submit([&](handler& cgh)
{
    accessor<float4, 2,
             access::mode::read,
             access::target::image> acc(myImage, cgh);

    sampler smpl(false,
                 sampler_addressing_mode::clamp,
                 sampler_filter_mode::nearest);
});
```

ComputeCpp

ComputeCpp™

Using images:

```
myQueue.submit([&](handler& cgh)
{
    accessor<float4, 2,
             access::mode::read,
             access::target::image> acc(myImage, cgh);

    sampler smpl(false,
                 sampler_addressing_mode::clamp,
                 sampler_filter_mode::nearest);
});
```

Access is requested just like for buffers or local memory

ComputeCpp

ComputeCpp™

Using images:

```
myQueue.submit([&](handler& cgh)
{
    accessor<float4, 2,
             access::mode::read,
             access::target::image> acc(myImage, cgh);

    sampler smpl(false,
                 sampler_addressing_mode::clamp,
                 sampler_filter_mode::nearest);
});
```

To read from an image you need a sampler

(to specify interpolation and out of bound access handling)

ComputeCpp

ComputeCpp™

Using images:

```
myQueue.submit([&](handler& cgh)
{
    accessor<float4, 2,
             access::mode::read
             access::target::image> acc(myImage, cgh);

    sampler smpl(false,
                 sampler_addressing_mode::clamp,
                 sampler_filter_mode::nearest);
});
```

Normalized or
Unnormalized coordinates

ComputeCpp

Using images:

```
cgh.parallel_for<class kernel_name>(range<2>(w, h),
                                    [=](item<2> item)
{
      auto coords = int2(item.get(0), item.get(1));
      float4 pixel = myImage(smpl)[coords];
      myImage_out[coords] = pixel;
   });
});
```

ComputeCpp

ComputeCpp™

Using images:

```
cgh.parallel_for<class kernel_name>(range<2>(w, h),
                                    [=](item<2> item)
{
      auto coords = int2(item.get(0), item.get(1));
      float4 pixel = myImage(smpl)[coords];
      myImage_out[coords] = pixel;
   });
});
```

Reading is performed through the sampler!

ComputeCpp

Hierarchical parallelism:

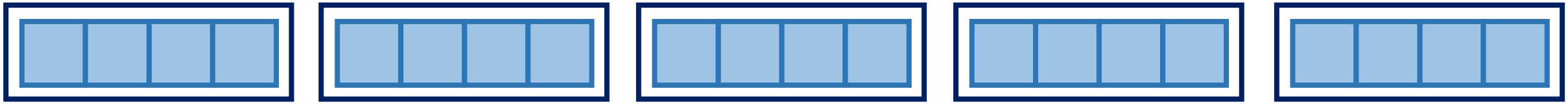
The same invocation formalism can be used to distribute a task over workgroups and workitems!

ComputeCpp

ComputeCpp™

First: OpenCL work group indexing and nomenclature:

This is 5 groups of 20 work items launched, each group has 4 items:



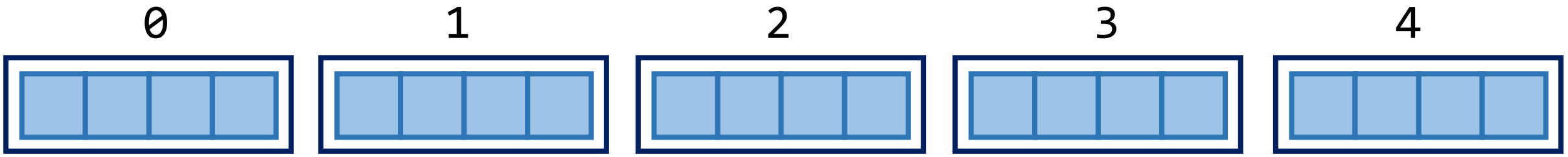
ComputeCpp

ComputeCpp™

First: OpenCL work group indexing and nomenclature:

This is 5 groups of 20 work items launched, each group has 4 items:

Group indices:



Global work item indices:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Local work item indices:

0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
   cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                             range<1>(16),
                                             [=](group<1> g)
      {
         int myLocal;
         private_memory<int> pr(g);

         parallel_for_work_item(g,
                                [=](item<1> i){ pr(i) = 0; });
      });
}
```

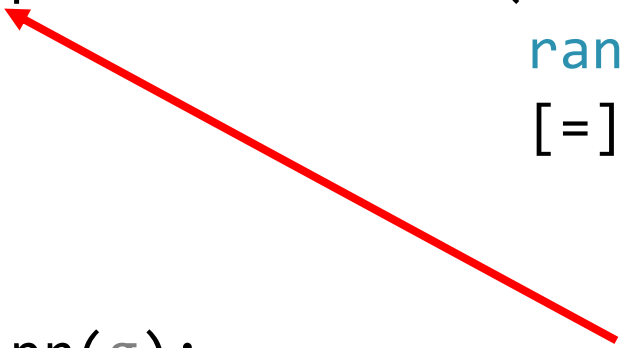
ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
    cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                              range<1>(16),
                                              [=](group<1> g)
        {
            int myLocal;
            private_memory<int> pr(g);

            parallel_for_work_item(g,
                                   [=](item<1> i){ pr(i) = 0; });
        });
}
```



Here we're launching a lambda over *workgroups*.

ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
    cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                              range<1>(16),
                                              [=](group<1> g)
        {
            int myLocal;
            private_memory<int> pr(g);

            parallel_for_work_item(g,
                                   [=](item<1> i){ pr(i) = 0; });
        });
}
```

We start 256 groups and each of them will have 16 workitems!

ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
    cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                              range<1>(16),
                                              [=](group<1> g)
    {
        int myLocal;
        private_memory<int> pr(g);

        parallel_for_work_item(g,
                               [=](item<1> i){ pr(i) = 0; });
    });
}
```

Now, all data inside the function body is automatically *local*!

ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
    cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                              range<1>(16),
                                              [=](group<1> g)
    {
        int myLocal;
        private_memory<int> pr(g);

        parallel_for_work_item(g,
                               [=](item<1> i){ pr(i) = 0; });
    });
}
```

We need to specifically request to denote thread private memory (registers)

ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
    cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                              range<1>(16),
                                              [=](group<1> g)
    {
        int myLocal;
        private_memory<int> pr(g);

        parallel_for_work_item(g,
                               [=](item<1> i){ pr(i) = 0; });
    });
}
```

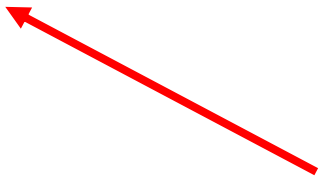
We can now launch lambdas over the items in the group!

ComputeCpp

ComputeCpp™

Hierarchical parallelism:

```
auto command_group = [&](handler& cgh){
    cgh.parallel_for_work_group<class kernel>(range<1>(256),
                                              range<1>(16),
                                              [=](group<1> g)
        {
            private_memory<int> pr(g);
            parallel_for_work_item(g, [=](item<1> i)
            {
                acc[g.get_local_range()*g.get()+i] = pr(i);
            });
        });
}
```



We need to transform the indices back to write to a global buffer

ComputeCpp

Hierarchical parallelism:

Why all this fuss???

ComputeCpp

Hierarchical parallelism - Why all this fuss?

- Yes, you can write the same in plain OpenCL or CUDA
- But these abstractions and helper types were designed to create better tools for expressing operation across ranges and groups
- You'll less likely make mistakes, and spend less time debugging...

ComputeCpp

Hierarchical parallelism - Why all this fuss?

- Yes, you can write the same in plain OpenCL or CUDA
- But these abstractions and helper types were designed to create better tools for expressing operation across ranges and groups
- You'll less likely make mistakes, and spend less time debugging...

And we all know that there is practically no real good way of debugging GPU code☺

Summary

SYCL is a very versatile tool

and a better way to code on GPU

More materials:

- Wigner GPU Day talks from 2015 and 2016 by Codeplay
- The SYCL specification from the Khronos registry
- Projects building on SYCL: sycl.tech

Summary

Projects building on SYCL: sycl.tech

Two important ones to try out:

- Eigen library, SYCL accelerated version
  - This template library is for linear algebra, now on GPUs too!
- SYCL Parallel STL
  - This is a SYCL based implementation of the C++17 parallel algorithms, introducing a sycl execution policy to run algorithms on the GPU.