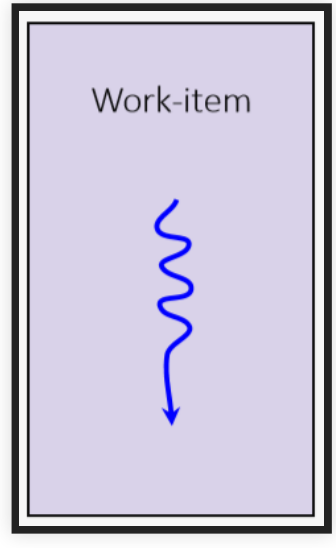
SYCL KERNEL FUNCTIONS

LEARNING OBJECTIVES

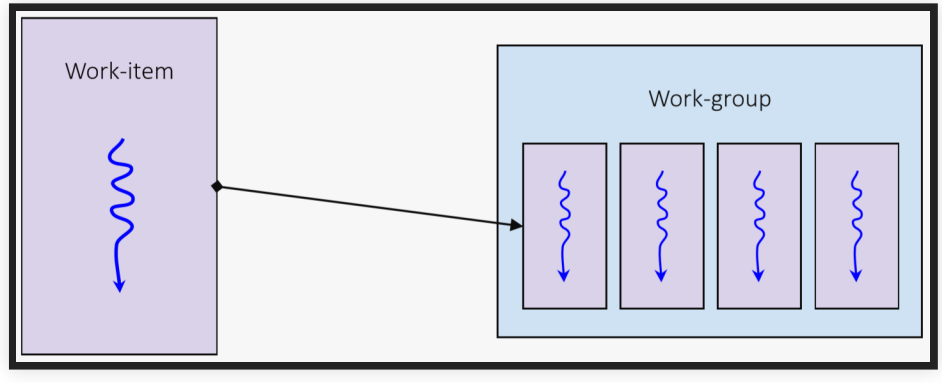
- Learn about the SYCL execution model
- Learn about the SIMT model for describing parallelism
- Learn how to define and invoke SYCL kernel functions
- Learn about the rules and restrictions on kernel functions
- Learn how to use both lambdas and function objects
- Learn how to manually compile a kernel function

THE SYCL EXECUTION MODEL

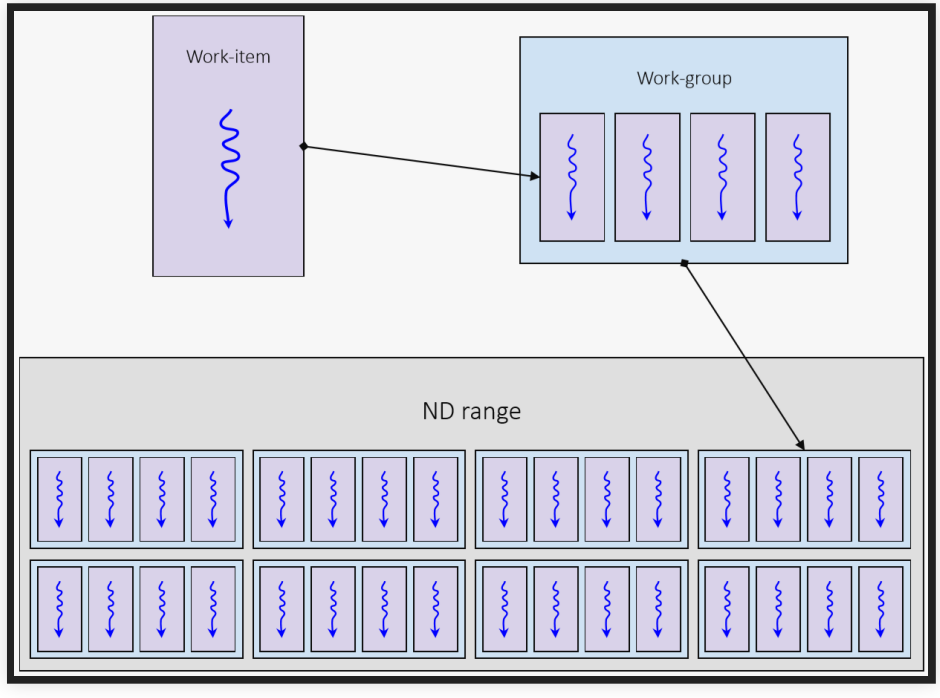
- SYCL kernel functions are executed by **work-items**
- You can think of a work-item as a thread of execution
- Each work-item will execute a SYCL kernel function from start to end
- A work-item can run on CPU threads, SIMD lanes, GPU threads, or any other kind of processing element



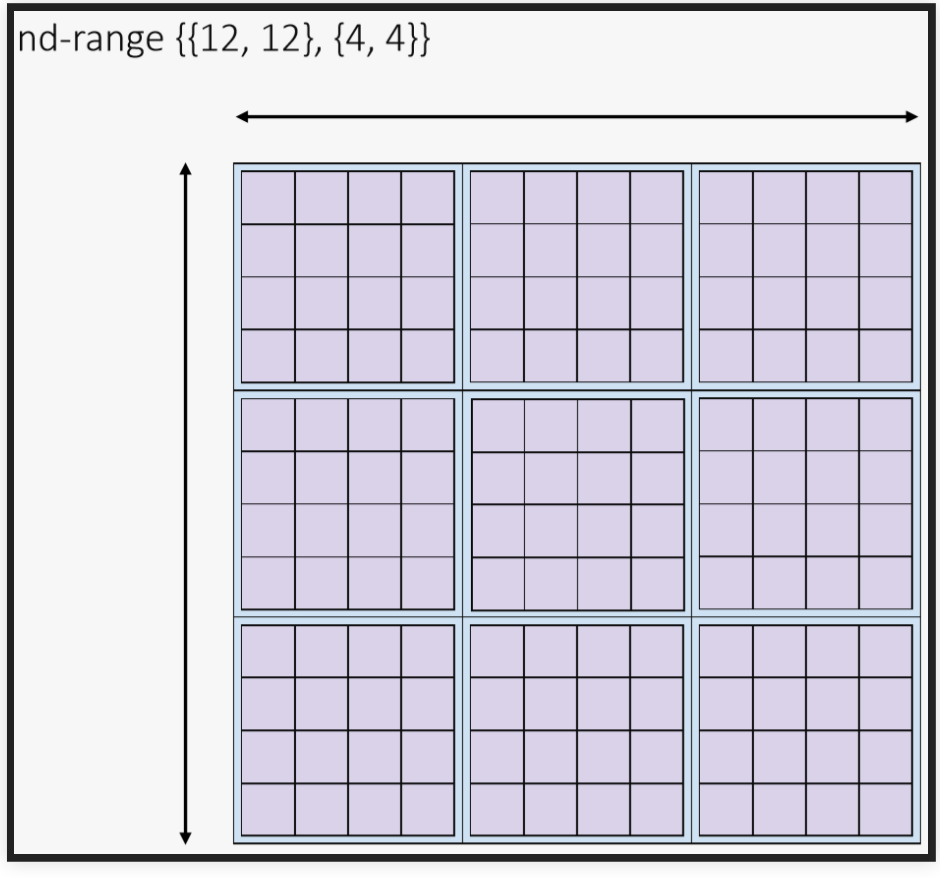
- Work-items are collected together into **work-groups**
- The size of work-groups is generally relative to what is optimal on the device being targeted
- It can also be affected by the resources used by each work-item



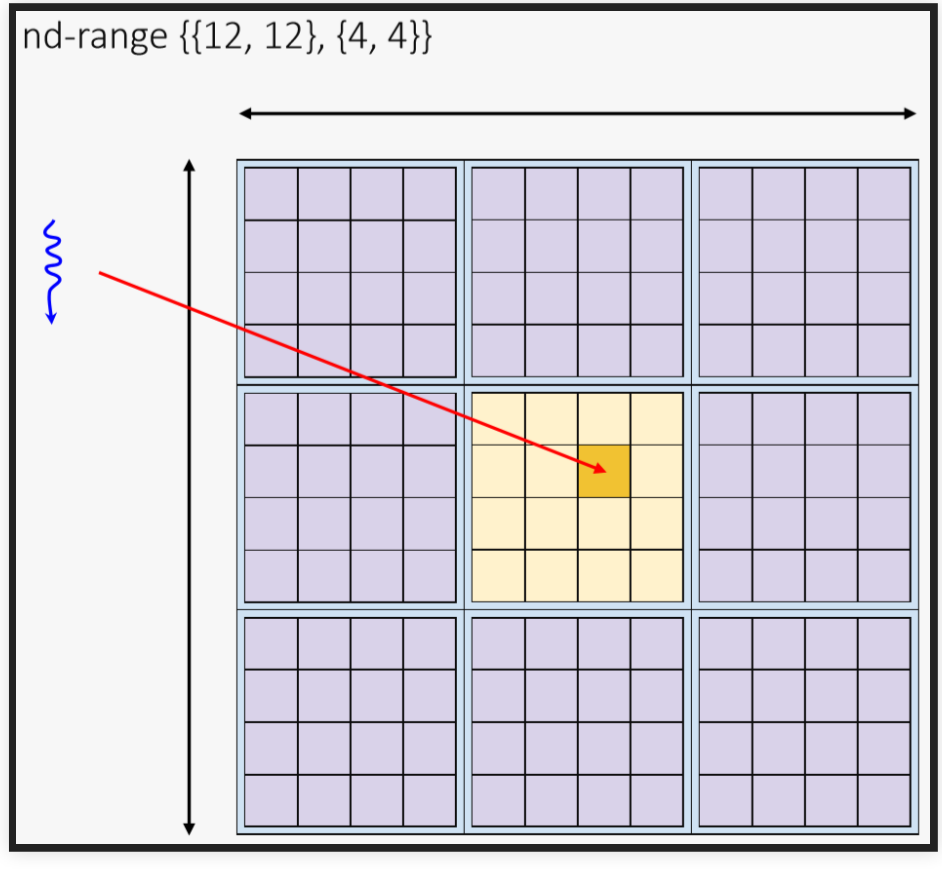
- SYCL kernel functions are invoked within an **nd-range**
- An nd-range has a number of work-groups and subsequently a number of work-items
- Work-groups always have the same number of work-items



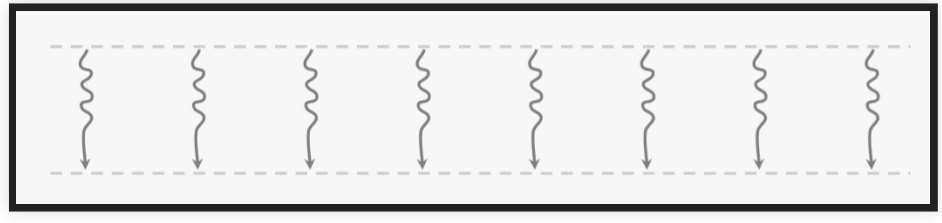
- The nd-range describes an **iteration space**; how the work-items and work-groups are composed
- An nd-range can be 1, 2 or 3 dimensions
- An nd-range has two components
  - The **global-range** describes the total number of workitems in each dimension
  - The **local-range** describes the number of work-items in a work-group in each dimension



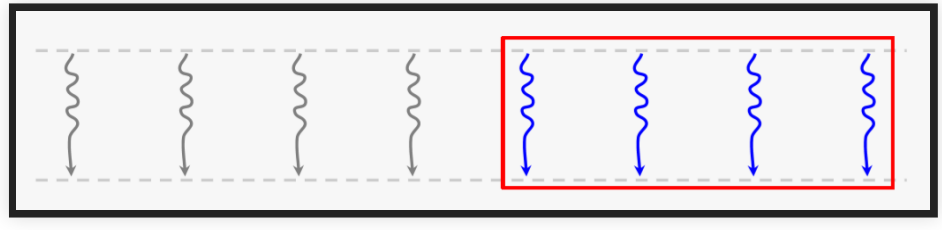
- Each invocation in the iteration space of an nd-range is a work-item
- Each invocation knows which work-item it is on and can query certain information about its position in the nd-range
- Each work-item has the following:
  - **Global range**: {12, 12}
  - **Global id**: {6, 5}
  - **Group range**: {3, 3}
  - **Group id**: {1, 1}
  - **Local range**: {4, 4}
  - **Local id**: {2, 1}



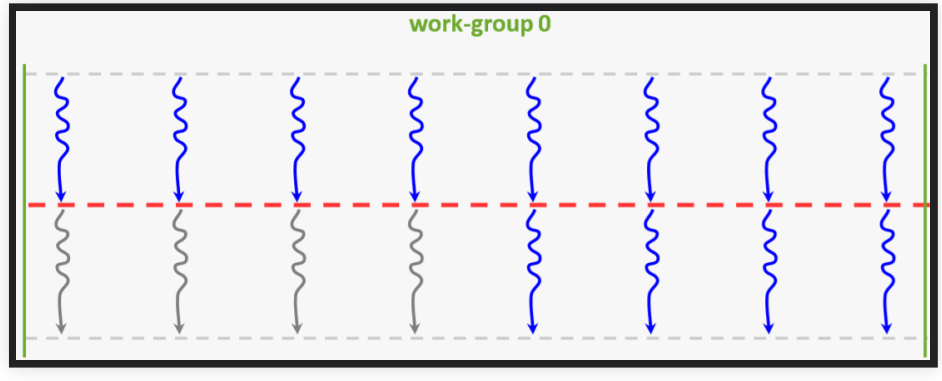
Typically an nd-range invocation SYCL will execute the SYCL kernel function on a very large number of work-items, often in the thousands



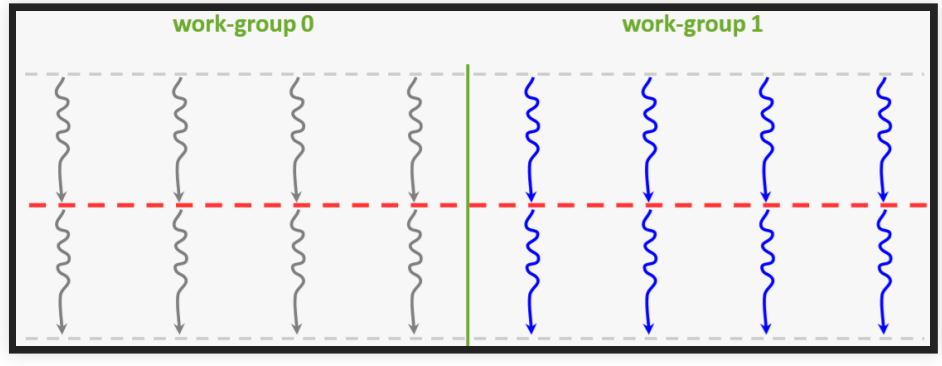
- Multiple work-items will generally execute concurrently
- On vector hardware this is often done in lock-step, which means the same hardware instructions
- The number of work-items that will execute concurrently can vary from one device to another
- Work-items will be batched along with other work-items in the same work-group
- The order work-items and workgroups are executed in is implementation defined



- Work-items in a work-group can be synchronized using a work-group barrier
  - All work-items within a work-group must reach the barrier before any can continue on



- SYCL does not support synchronizing across all work-items in the nd-range
- The only way to do this is to split the computation into separate SYCL kernel functions



SYCL IS AN SIMT PROGAMMING MODEL

Sequential CPU code

```
void calc(int *in, int *out) {
  // all iterations are run in the same
  // thread in a loop
  for (int i = 0; i < 1024; i++){
    out[i] = in[i] * in[i];
  }
}

// calc is invoked just once and all
// iterations are performed inline
calc(in, out);
```

Parallel SIMT code

```
void calc(int *in, int *out, int id) {
  // function is described in terms of
  // a single iteration
  out[id] = in[id] * in[id];
}

// parallel_for invokes calc multiple
// times in parallel
parallel_for(calc, in, out, 1024);
```

ENQUEUEING SYCL KERNEL FUNCTIONS

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

class add;

int main(int argc, char *argv[]) {
  queue gpuQueue(gpu_selector{});

  gpuQueue.submit([&](handler &cgh){

    cgh.parallel_for<add>(range<1>(1024),
      [=](id<1> i) {
      // kernel code });
  });
  gpuQueue.wait();
}
```

- SYCL kernel functions are defined and invoked using one of the kernel function invoke APIs provided by the **handler** class
- These add a SYCL kernel function command to the command group
- There can only be one SYCL kernel function command in a command group

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

class add;

int main(int argc, char *argv[]) {
  queue gpuQueue(gpu_selector{});

  gpuQueue.submit([&](handler &cgh){

    cgh.parallel_for<add>(range<1>(1024),
      [=](id<1> i) {
      // kernel code });
  });
  gpuQueue.wait();
}
```

- The lambda or function object represents the SYCL device function
- This is the part that is compiled for the device

```
#include <CL/sycl.hpp>
using namespace cl::sycl;

class add;

int main(int argc, char *argv[]) {
  queue gpuQueue(gpu_selector{});

  gpuQueue.submit([&](handler &cgh){

    cgh.parallel_for<add>(range<1>(1024),
      [=](id<1> i) {
      // kernel code });
  });
  gpuQueue.wait();
}
```

- There are a number of different APIs for expressing different forms of parallelism, complexity and functionality
- Each takes some representation of the nd-range and expects a certain parameter type that describes the current index into the iteration space
- These types have a number of member functions for retrieving different index and range information about the current interation

Expressing Parallelism

```
cgh.single_task([=](){
  // SYCL kernel function is executed
  // once on a single work-item
});
```

```
cgh.parallel_for_work_group(range<2>(64, 64),
                            [=](group<2> gp){
  // SYCL kernel function is executed
  // once per work-group
```

```
cgh.parallel_for(range<2>(64, 64),
                           [=](id<2> idx){
  // SYCL kernel function is executed
  // on an nd-range of work-items
});
```

```
  parallel_for_work_item(gp, [=](h_item<2> it){
    // SYCL kernel function is executed
    // once per work-item
  });
});
```

```
cgh.parallel_for<kernel>(range<1>(1024),
    [=](id<1> idx){

  // kernel code

});
```

- Overload taking a **range** object specifies the global range, runtime decides local range
- An **id** parameter represents the index within the global range

---

```
cgh.parallel_for<kernel>(range<1>(1024),
    [=](item<1> item){

  // kernel code

});
```

- Overload taking a **range** object specifies the global range, runtime decides local range
- An **item** parameter represents the global range and the index within the global range

---

```
cgh.parallel_for<kernel>(nd_range<1>(range<1>(1024),
    range<1>(32)),[=](nd_item<1> ndItem){

  // kernel code

});
```

- Overload taking an **nd_range** object specifies the global and local range
- An **nd_item** parameter represents the global and local range and index

```
cgh.parallel_for<kernel>(range<1>(1024),
    [=](id<1>(512), id<1>idx){

  // kernel code

});
```

```
cgh.parallel_for<kernel>(range<1>(1024),
     id<1>(512), [=](item<1> item){

  // kernel code

});
```

```
cgh.parallel_for<kernel>(nd_range<1>(range<1>(1024),
    range<1>(32), id<1>(512)),
    [=](nd_item<1> ndItem){

  // kernel code

});
```

- All overloads of **parallel_for** also allow you to optionally specify an offset
- The offset, if used, will increment each index into the global index by the specified value
    - E.g. with a range of 1024 and an offset of 512 the indexes would become (512, 1536)

SYCL KERNEL FUNCTION RULES AND RESTRICTIONS

SYCL Kernel Function Rules

- Must be defined using a C++ lambda or function object, they cannot be a function pointer or std::function
- Must always capture or store members by-value
- SYCL kernel functions declared with a lambda must be named using a forward declarable C++ type, declared in global scope
- SYCL kernel function names follow C++ ODR rules, which means you cannot have two kernels with the same name

SYCL Kernel Function Restrictions

- No dynamic allocation
- No dynamic polymorphism
- No function pointers
- No recursion

SYCL KERNELS AS FUNCTION OBJECTS

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try {
    queue gpuQueue(gpu_selector{},
      async_handler{});
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ out[i] = inA[i] + inB[i]; });
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- All the examples of SYCL kernel functions up until now have been defined using lambdas

```
struct add {
  using read_accessor_t =
    accessor<float, 1,
    access::mode::read,
    access::target::global_buffer>
  using write_accessor_t =
    accessor<float, 1
    access::mode::write,
    access::target::global_buffer>

  read_accessor_t inA_, inB_;
  write_accessor_t out_;

void operator()(id<1> i){
  out_[i] = inA_[i] + inB_[i];
  }
};
```

- As well as defining SYCL kernels using lambdas you can also define a SYCL kernel using a regular C++ function object
- Where the accessors are stored as members and the function call operator takes the appropriate parameter

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try {
    queue gpuQueue(gpu_selector{},
      async_handler{});
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for(range<1>(dA.size()),
        add{inA, inB, out});
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

```
struct add {
  using read_accessor_t =
    accessor<float, 1,
    access::mode::read,
    access::target::global_buffer>
  using write_accessor_t =
    accessor<float, 1
    access::mode::write,
    access::target::global_buffer<

  read_accessor_t inA_, inB_;
  write_accessor_t out_;

  void operator()(id<1> i) ){
  out[i] = inA[i] + inB[i];
  }
};
```

To use a C++ function object you simply construct an instance of the type initialising the accessors and pass it to parallel_for

Notice you no longer need to name the SYCL kernel

PRE-COMPILING SYCL KERNEL FUNCTIONS

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try {
    queue gpuQueue(gpu_selector{},
      async_handler{});
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for(range<1>(dA.size()),
        add{inA, inB, out});
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- When you emqueue a SYCL kernel function the runtime has to just-in-time compile the kernel for the device that the queue is targeting
- This means the first time a kernel function is enqueued will take longer
- However you can avoid this by pre-compiling the kernel
- Pre-compiling kernels also gives you more control over how the kernel is compiled

```
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try {
    queue gpuQueue(gpu_selector{},
      async_handler{});

    program addProgram(gpuQueue.get_context());

    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for(range<1>(dA.size()),
        add{inA, inB, out});
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- To pre-compile a kernel you need to create a program that is associated with the context you are executing the kernel on
- A program is represented by the **program** class

```
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try {
    queue gpuQueue(gpu_selector{},
      async_handler{});

    program addProgram(gpuQueue.get_context());
    addProgram.build_with_kernel_type<add>();

    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for(range<1>(dA.size()),
        add{inA, inB, out});
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- The program can be then be compiled from a SYCL kernel function name by calling **compile_with_kernel_type** and specifying the kernel name as a template parameter
- If the kernel function was defined as a function object then the name would be the function object type

```
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try {
    queue gpuQueue(gpu_selector{}, async_handler{});

    program addProgram(gpuQueue.get_context());
    addProgram.build_with_kernel_type<add>();
    auto addKernel = addProgram.get_kernel<add>();

    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer>float, 1< bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for(range<1>(dA.size()),
            add{inA, inB, out});
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- Once the program is compiled you can retrieve the kernel
- A kernel is represented by the **kernel** class
- This is retrieved from a program object by calling the **get_kernel** member function and specifying the kernel name as a template parameter

```
int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };
  try{
    queue gpuQueue(gpu_selector{}, async_handler{});

    program addProgram(gpuQueue.get_context());
    addProgram.build_with_kernel_type<add>();
    auto addKernel = addProgram.get_kernel<add>();

    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));
    gpuQueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for(addKernel,
       range<1>(dA.size()), [=](id<1> i)
       { out[i] = inA[i] + inB[i]; });
    });
    gpuQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- Finally the kernel object can then be passed to parallel_for in order to specify that the invocation should use the precompiled kernel

QUESTIONS