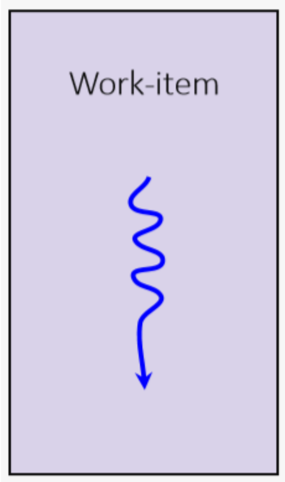
ND RANGE KERNELS

LEARNING OBJECTIVES

- Learn about the SYCL execution and memory model
- Learn how to enqueue an nd-range kernel functions

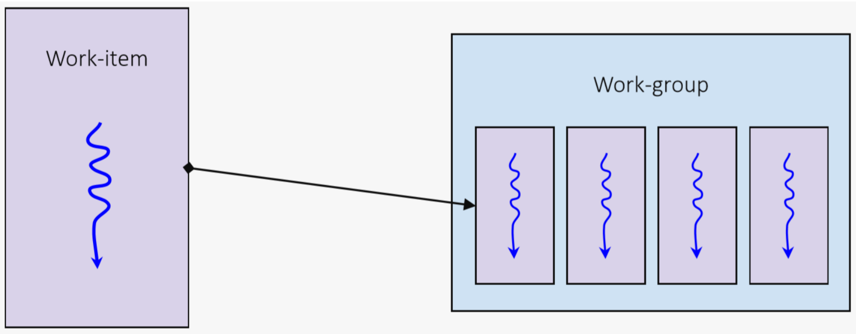
SYCL EXECUTION MODEL

- 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



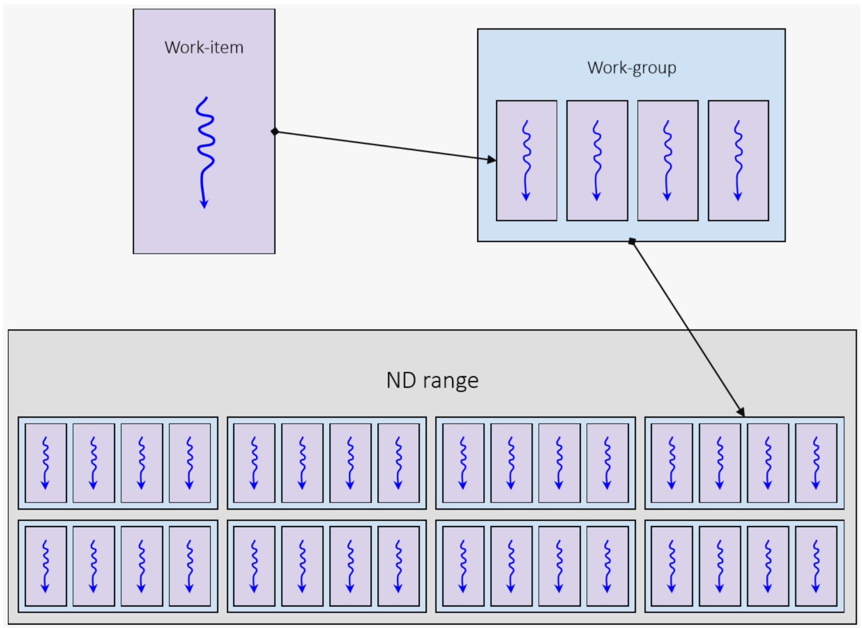
SYCL EXECUTION MODEL

- 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



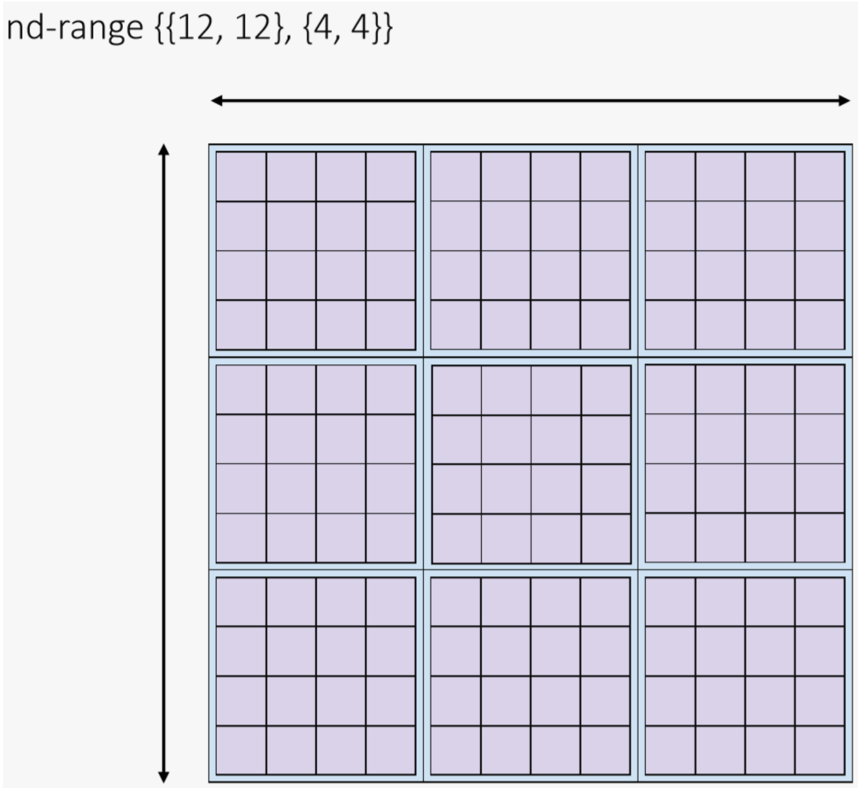
SYCL EXECUTION MODEL

- 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



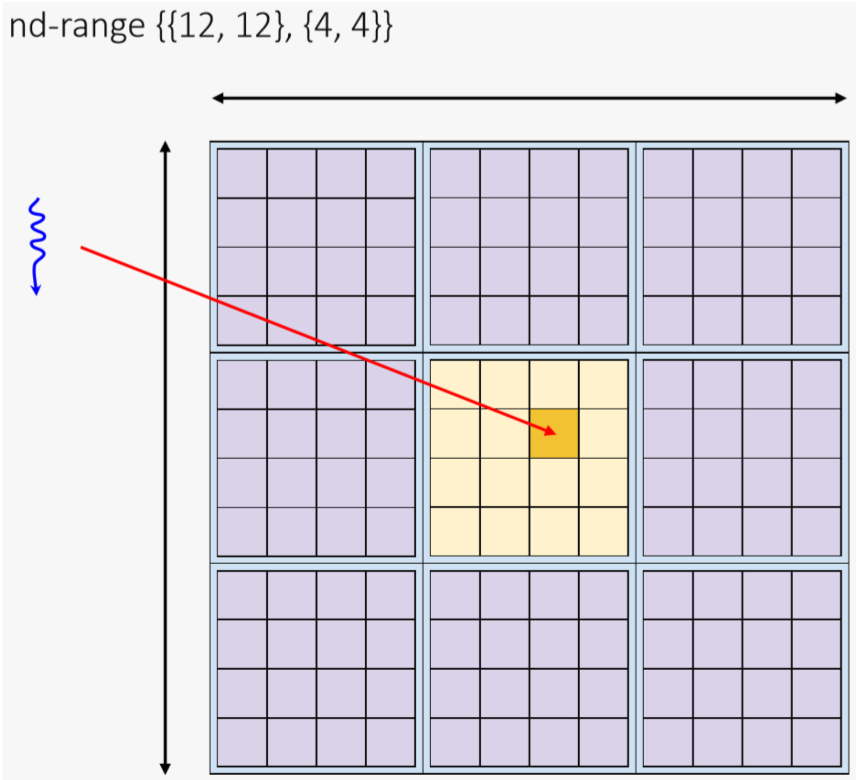
SYCL EXECUTION MODEL

- 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



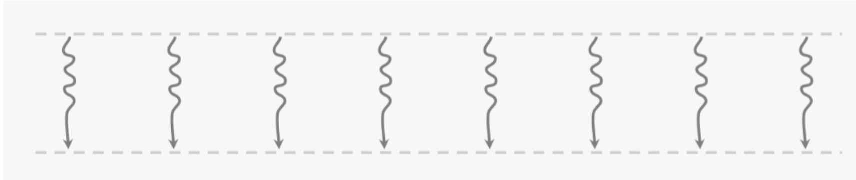
SYCL EXECUTION MODEL

- 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}



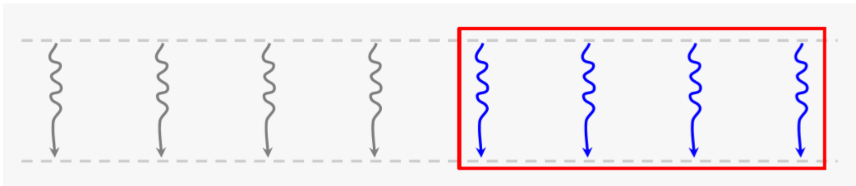
SYCL EXECUTION MODEL

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



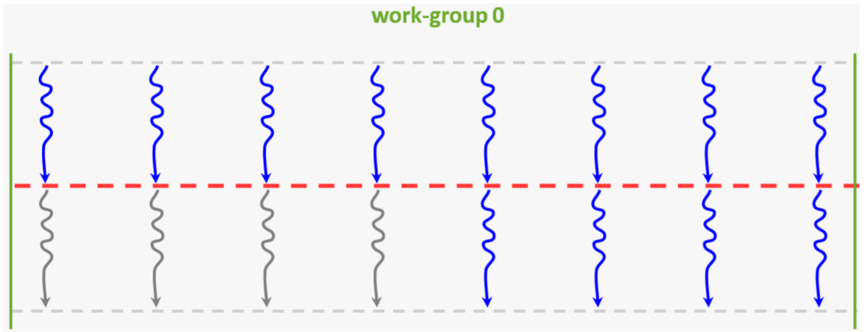
SYCL EXECUTION MODEL

- 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



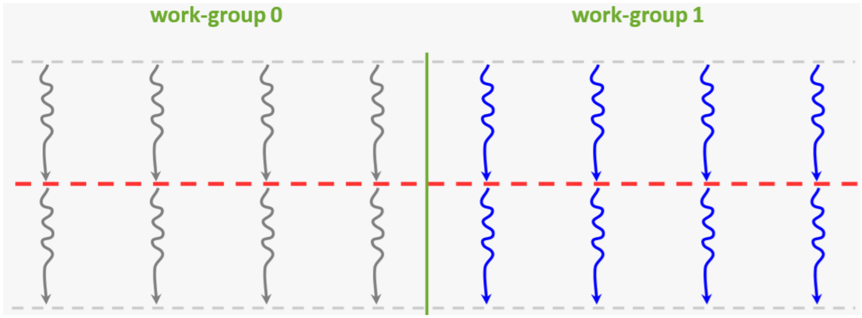
SYCL EXECUTION MODEL

- 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



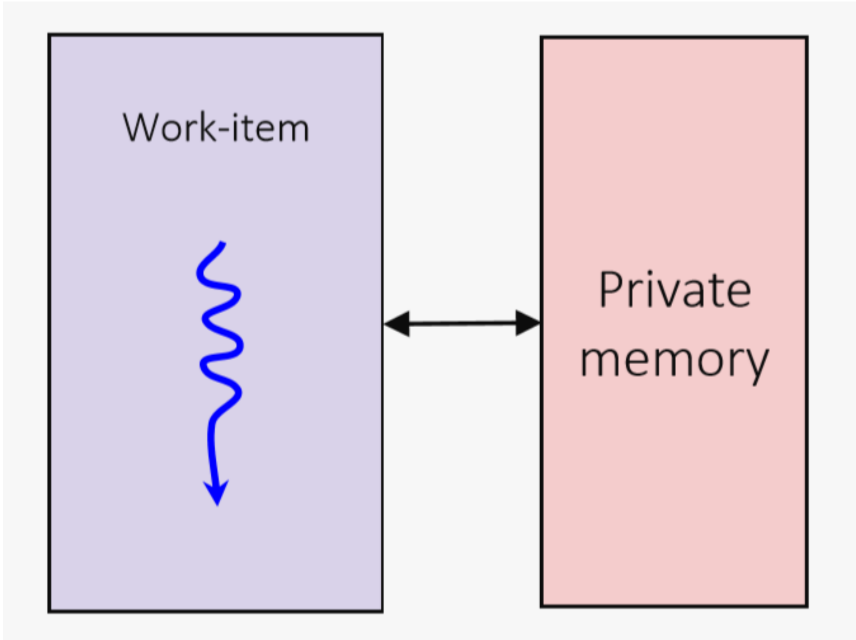
SYCL EXECUTION MODEL

- 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

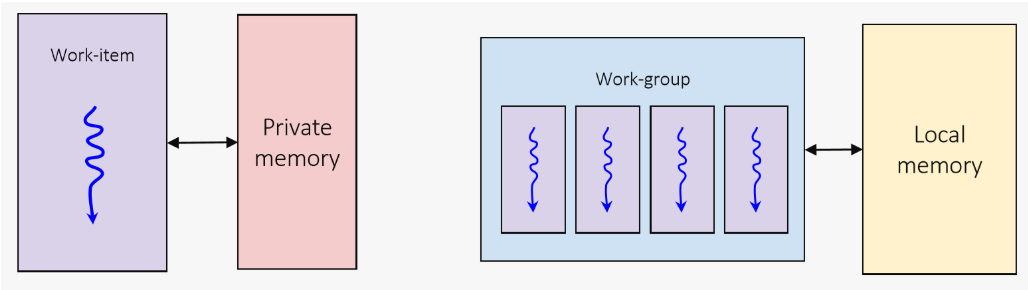


SYCL MEMORY MODEL

- Each work-item can access a dedicated region of **private memory**
- A work-item cannot access the private memory of another work-item

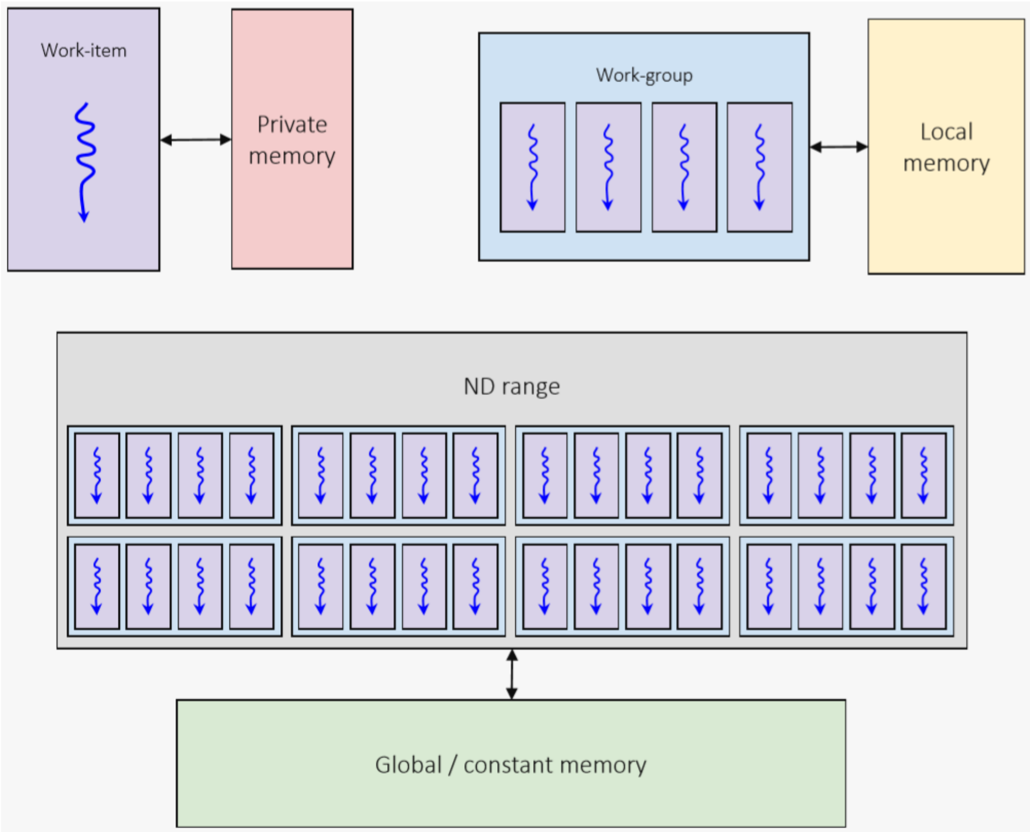


SYCL MEMORY MODEL



- Each work-item can access a dedicated region of **local memory** accessible to all work-items in a work-group
- A work-item cannot access the local memory of another workgroup

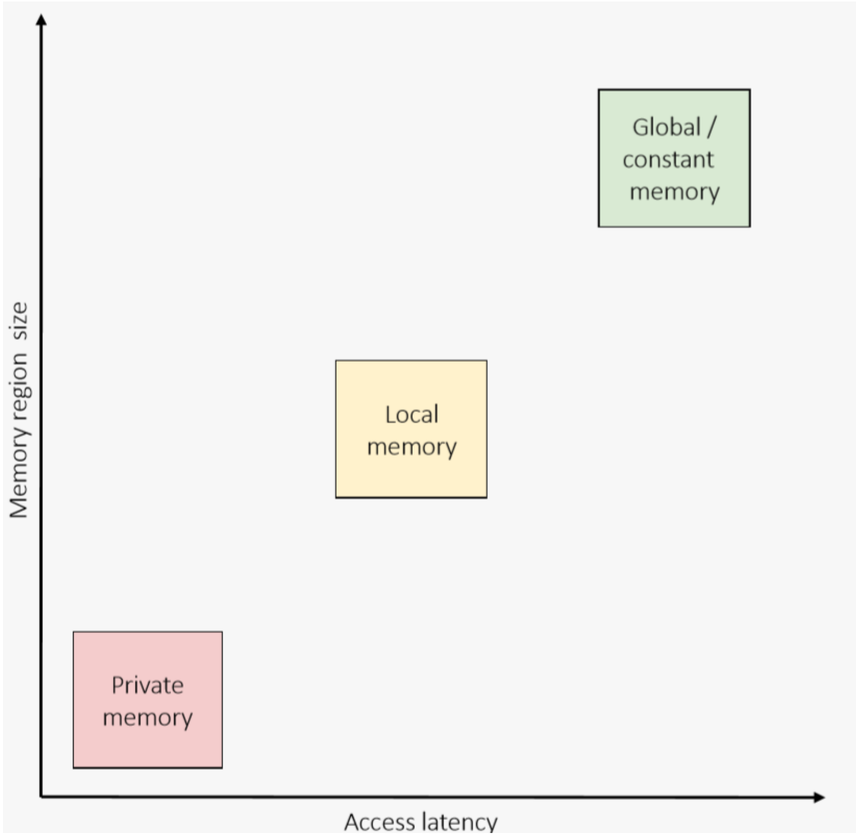
SYCL MEMORY MODEL



- Each work-item can access a single region of **global memory** that's accessible to all work-items in a ND-range
- Each work-item can also access a region of global memory reserved as **constant memory**, which is read-only

SYCL MEMORY MODEL

- Each memory region has a different size and access latency
- Global / constant memory is larger than local memory and local memory is larger than private memory
- Private memory is faster than local memory and local memory is faster than global / constant memory



EXPRESSING PARALLELISM

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

ACCESSING DATA WITH ACCESSORS

- There are a few different ways to access the data represented by an accessor
    - The subscript operator can take an **id**
        - Must be the same dimensionality of the accessor
        - For dimensions > 1, linear address is calculated in row major
- Nested subscript operators can be called for each dimension taking a **size_t**
    - E.g. a 3-dimensional accessor: acc[x][y][z] = …
- A pointer to memory can be retrieved by calling **get_pointer**
    - This returns a **multi_ptr**, which is a wrapper class for pointers to the memory in the relevant memory space

ACCESSING DATA WITH ACCESSORS

```
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];
  });
});
```

- Here we access the data of the `accessor` by passing in the `id` passed to the SYCL kernel function.

ACCESSING DATA WITH ACCESSORS

```
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>(rng, [=](id<3> i){
    auto ptrA = inA.get_pointer();
    auto ptrB = inB.get_pointer();
    auto ptrO = out.get_pointer();
    auto linearId = i.get_linear_id();

    ptrA[linearId] = ptrB[linearId] + ptrO[linearId];
  });
});
```

- Here we retrieve the underlying pointer for each of the `accessors`.
- We then access the pointer using the linearized `id` by calling the `get_linear_id` member function on the `item`.
- Again this linearization is calculated in row-major order.

QUESTIONS

EXERCISE

Code_Exercises/Exercise_14_ND_Range_Kernel/source

Implement a SYCL application that will perform a vector add using `parallel_for`, adding multiple elements in parallel.