MORE SYCL FEATURES

LEARNING OBJECTIVES

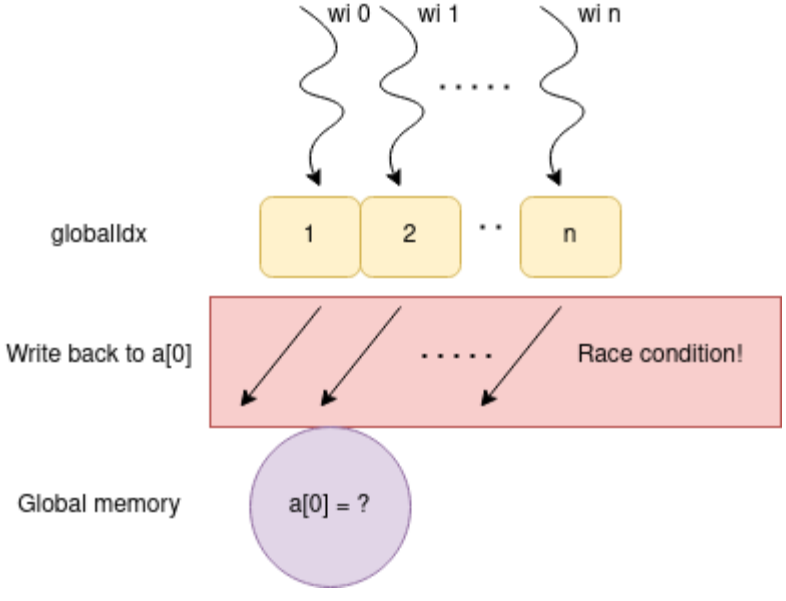
- Learn about atomic operations and how to use them in SYCL kernels
- Learn about SYCL group algorithms
- Learn about SYCL reductions

RACE CONDITION

- In a multithreaded environment, multiple work items writing indiscriminately to the same area of memory causes a race condition.

```
q.parallel_for([=](sycl::item<1> it) {
  // Race condition! Multiple threads
  // writing to same area of memory
  a[0] = it.get_global_linear_id();
});
```

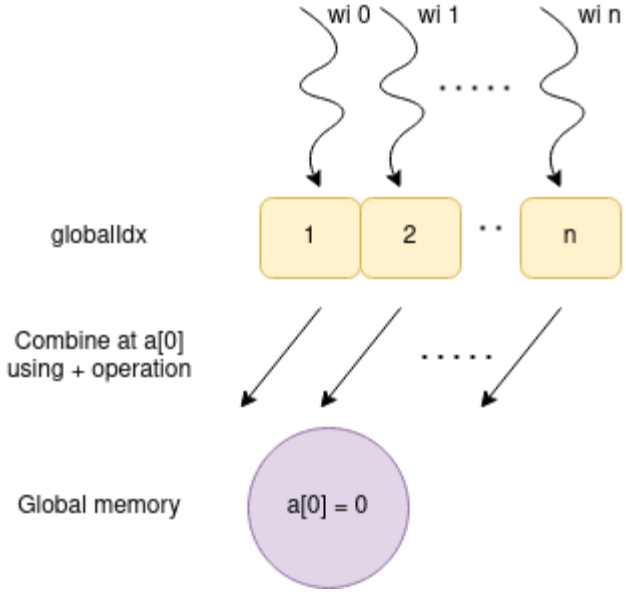
RACE CONDITION




```
q.parallel_for([=](sycl::item<1> it) {
  // Race condition! Multiple threads
  // writing to same area of memory
  a[0] = it.get_global_linear_id();
});
```

- SYCL does not guarantee any particular ordering for the execution of work items.
- When multiple work items concurrently write different values to the same area of memory, there is no way of knowing which value will be held in memory once all work items have finished writing.
- This is called a race condition and can be a source of non determinism in code execution.

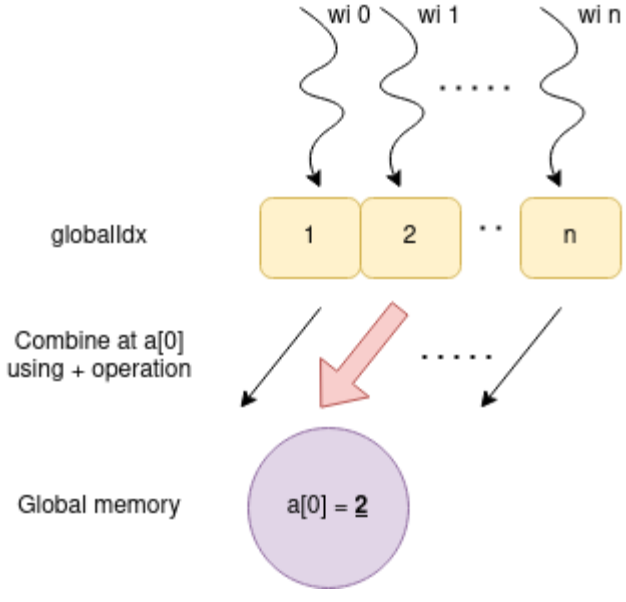
ATOMIC OPERATIONS




```
q.parallel_for([=](sycl::item<1> it) {
  sycl::atomic_ref<T,
          sycl::memory_order_relaxed,
          sycl::memory_scope_device>
            (a[0])
            .fetch_add(it.get_global_linear_id());
});
```

- Atomic operations are needed in order to deterministically combine values from different work items.
- Atomic operations enforce a particular ordering of instructions across work items. Some orderings include `memory_order_relaxed`, `memory_order_acq_rel`, `memory_order_seq_cst`.

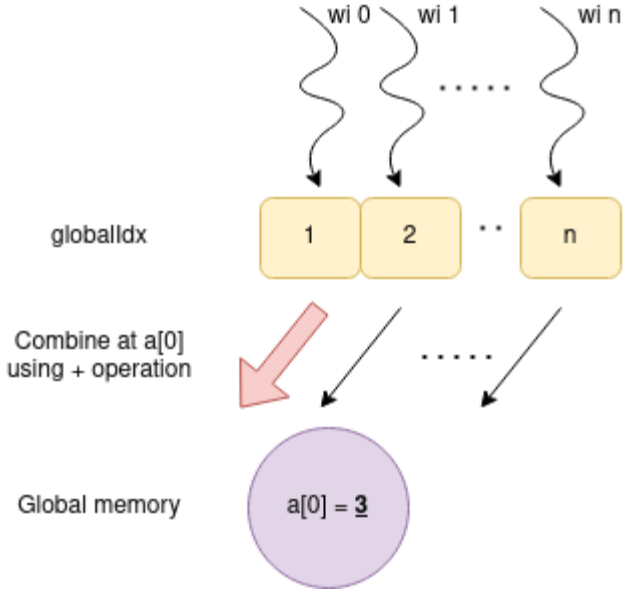
ATOMIC OPERATIONS




```
q.parallel_for([=](sycl::item<1> it) {
  sycl::atomic_ref<T,
          sycl::memory_order_relaxed,
          sycl::memory_scope_device>
            (a[0])
            .fetch_add(it.get_global_linear_id());
});
```

- Atomic operations are needed in order to deterministically combine values from different work items.
- Atomic operations enforce a particular ordering of instructions across work items. Some orderings include `memory_order_relaxed`, `memory_order_acq_rel`, `memory_order_seq_cst`.

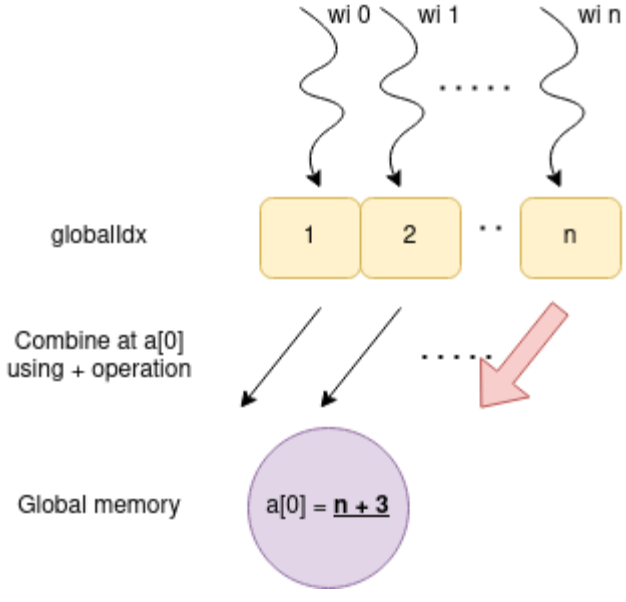
ATOMIC OPERATIONS




```
q.parallel_for([=](sycl::item<1> it) {
  sycl::atomic_ref<T,
          sycl::memory_order_relaxed,
          sycl::memory_scope_device>
            (a[0])
            .fetch_add(it.get_global_linear_id());
});
```

- Atomic operations are needed in order to deterministically combine values from different work items.
- Atomic operations enforce a particular ordering of instructions across work items. Some orderings include `memory_order_relaxed`, `memory_order_acq_rel`, `memory_order_seq_cst`.

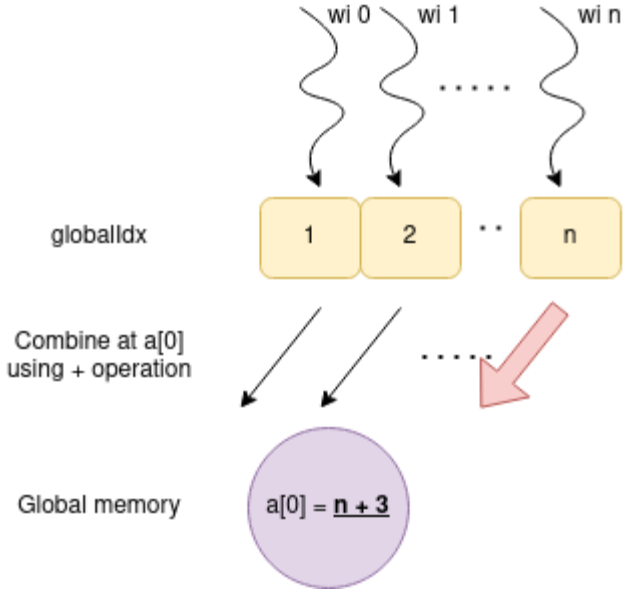
ATOMIC OPERATIONS




```
q.parallel_for([=](sycl::item<1> it) {
  sycl::atomic_ref<T,
          sycl::memory_order_relaxed,
          sycl::memory_scope_device>
            (a[0])
            .fetch_add(it.get_global_linear_id());
});
```

- Atomic operations are needed in order to deterministically combine values from different work items.
- Atomic operations enforce a particular ordering of instructions across work items. Some orderings include `memory_order_relaxed`, `memory_order_acq_rel`, `memory_order_seq_cst`.

ATOMIC OPERATIONS




```
q.parallel_for([=](sycl::item<1> it) {
  sycl::atomic_ref<T,
          sycl::memory_order_relaxed,
          sycl::memory_scope_device>
            (a[0])
            .fetch_add(it.get_global_linear_id());
});
```

- Using atomics, values can be combined across work items without data races.
- `fetch_add`, `fetch_sub`, `fetch_max`, `fetch_min` are some of the ways we can combine values atomically.
- `fetch_and` and `fetch_or` can also be used for integral types
- Please see the SYCL Specification for more details.

ATOMIC OPERATIONS

```
q.parallel_for([=](sycl::item<1> it) {
  sycl::atomic_ref<T,
          sycl::memory_order_relaxed,
          sycl::memory_scope_work_group,
          sycl::access::address_space::local_space>
            (a[0])
            .fetch_add(it.get_global_linear_id());
});
```

- We can also specify the memory space of `a[0]`.
- If `a[0]` is in local memory, we should expect a speedup for using the local memory atomic over the default atomic (which uses the generic address space).

GROUP ALGORITHMS

- SYCL provides group algorithms which perform common operations over a single workgroup.
- `reduce` algorithms perform fast work group reduction operation for some op such as `plus`, `max`, etc.
- `any_of`, `all_of`, `none_of` as well as the `joint_*` counterparts perform some predicate checking and return the same value to all items in a work group.
- `permute_group_by_xor` permutes values among work items in a work group, according to a provided mask.
- `inclusive_scan` and `exclusive_scan`. For a scan of elements: `[x0,...,xn]` the ith result of an exclusive scan is the combination of the elements [x0,...,x{i-1}] and the ith result of an inclusive scan is the combination of elements [x0,...,xi] using some binary op.
- Please see the SYCL specification for more details.

GROUP ALGORITHMS

- Group algorithms can operate on different group scopes, such as `work_group`, `sub_group`.
- All work items in a given group scope must call the function in convergent control flow.
- Please see the SYCL specification for more details.

SYCL REDUCTIONS

```
q.submit([&](sycl::handler &cgh) {
  // Output of reduction will be in ptr
  auto sumReduction = sycl::reduction(ptr,
                        sycl::plus<T>());
  cgh.parallel_for(myNd, sumReduction,
      [=](sycl::nd_item<1> item, auto &sum) {
         sum += devA[item.get_global_linear_id()];
  });
});
```

- SYCL provides reduction operators to perform fast reductions using some binary operation.
- Reductions can be performed with `sycl::plus`, `sycl::maximum`, `sycl::multiplies` etc.
- Please see the SYCL specification for more details.

SYCL REDUCTIONS

```
q.submit([&](sycl::handler &cgh) {
  // Output of reduction will be in ptr
  auto maxReduction = sycl::reduction(ptr,
                      sycl::maximum<T>());
  cgh.parallel_for(myNd, maxReduction,
      [=](sycl::nd_item<1> item, auto &myMax) {
         myMax.combine(devA[item.get_global_linear_id(
  });
});
```

- SYCL provides reduction operators to perform fast reductions using some binary operation.
- Reductions can be performed with `sycl::plus`, `sycl::maximum`, `sycl::multiplies` etc.
- Please see the SYCL specification for more details.

QUESTIONS

EXERCISE

- See how atomics, group algorithms and `sycl::reductions` are used in implementing a simple reduction operation.
- Which code runs fastest? Which code is simplest?