GPU PROGRAMMING PRINCIPLES

LEARNING OBJECTIVES

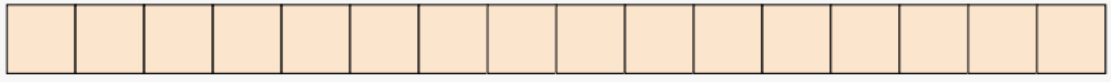
- Learn about coalesced global memory access
- Learn about the performance impact
- Learn about local memory and how to use it.

COALESCED GLOBAL MEMORY

- Reading from and writing to global memory is generally very expensive.
- It often involves copying data across an off-chip bus.
    - This means you generally want to avoid unnecessary accesses.
- Memory access operations is done in chunks.
    - This means accessing data this is physically close together in memory is more efficient.

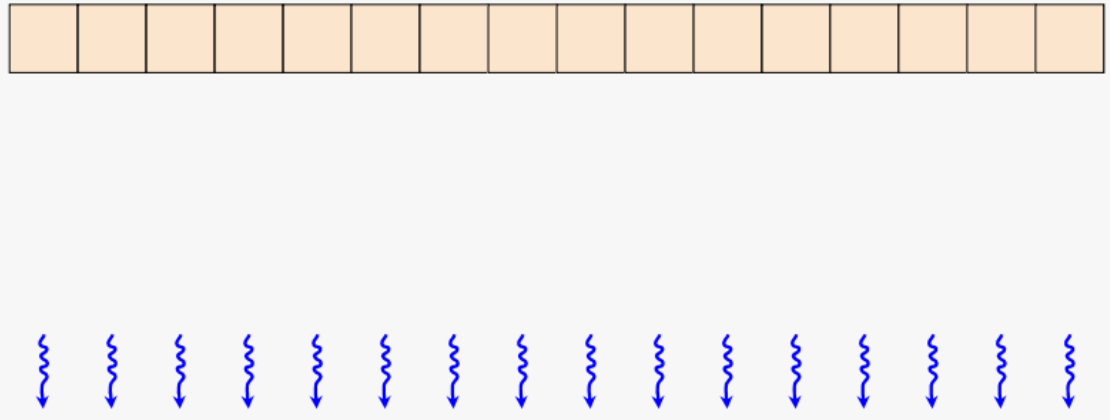
COALESCED GLOBAL MEMORY

```
float data[size];
```



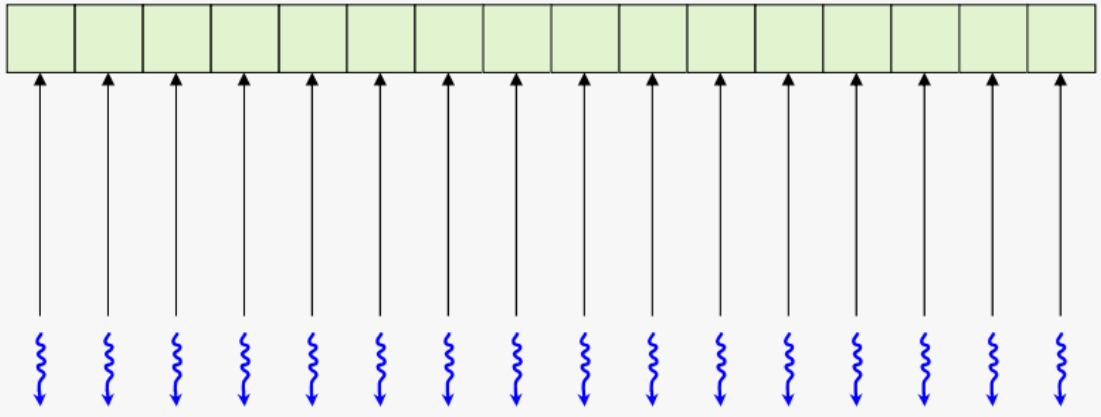
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId]);
```



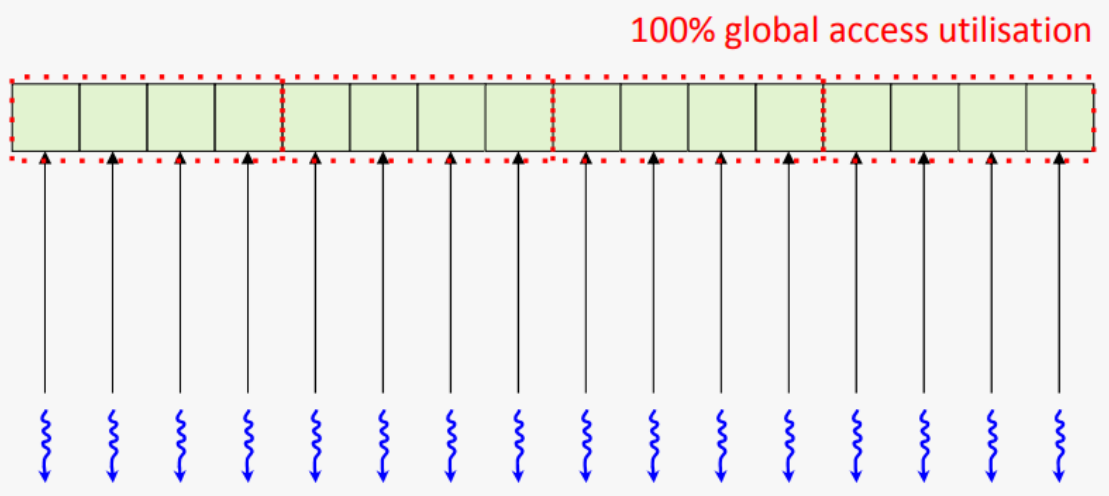
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId]);
```



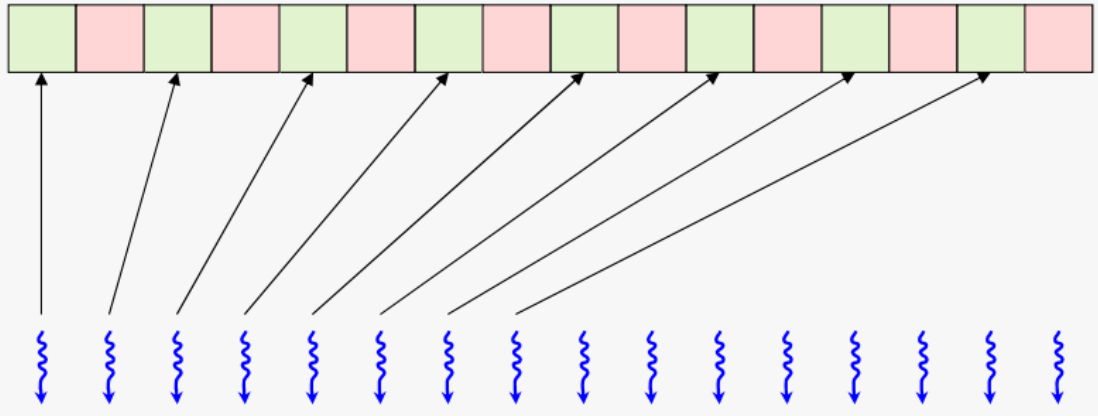
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId]);
```



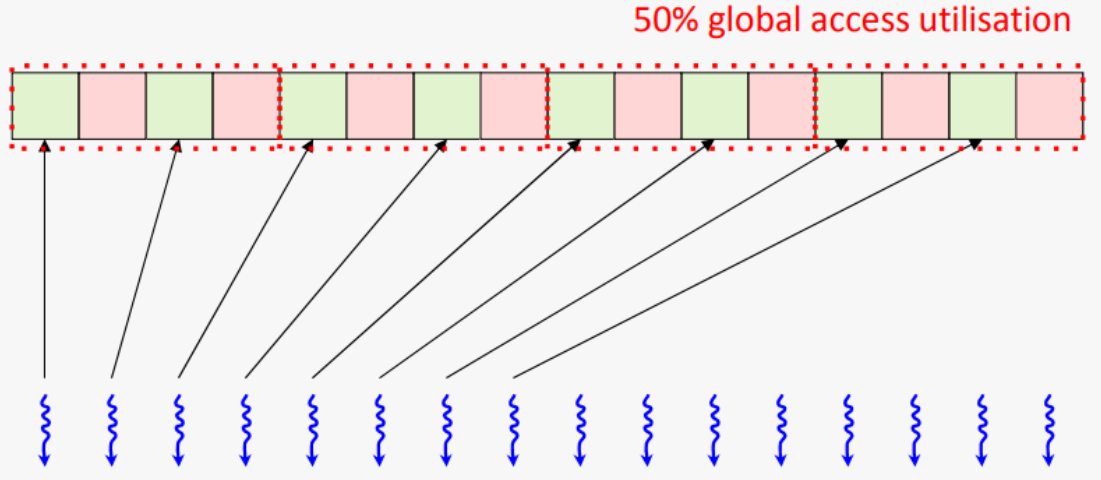
COALESCED GLOBAL MEMORY

```
float data[size];

...

f(a[globalId * 2]);
```



COALESCED GLOBAL MEMORY

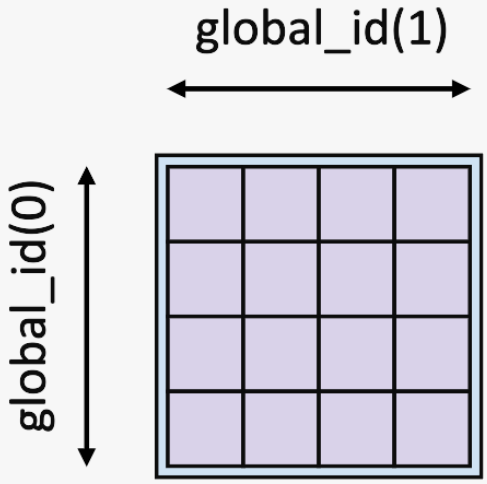
```
float data[size];

...

f(a[globalId * 2]);
```



ROW-MAJOR VS COLUMN-MAJOR

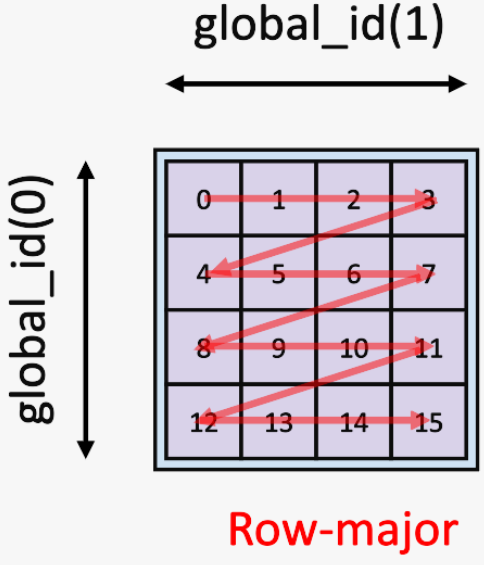
- Coalescing global memory access is particularly important when working in multiple dimensions.
- This is because when doing so you have to convert from a position in 2d space to a linear memory space.
- There are two ways to do this; generally referred to as row-major and column-major.

ROW-MAJOR VS COLUMN-MAJOR



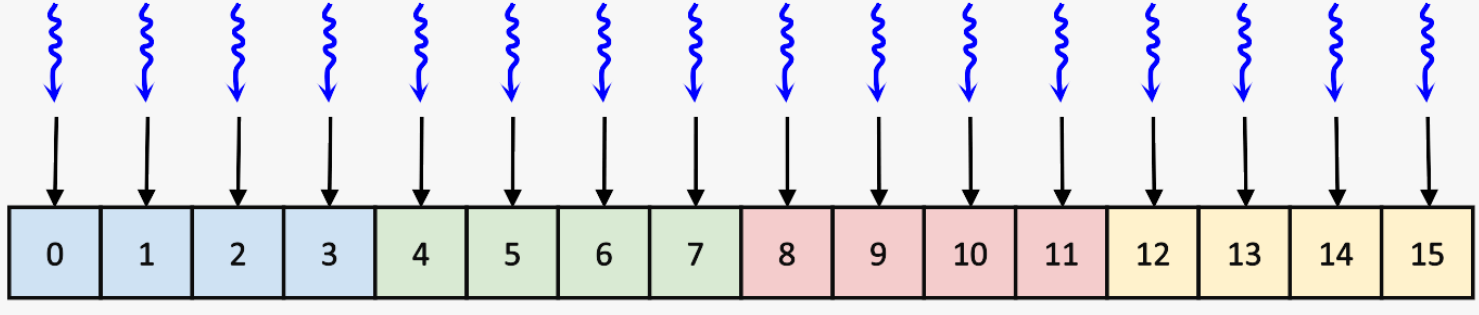
Row-major

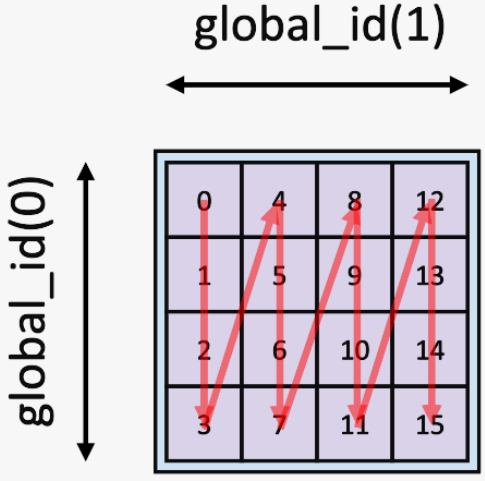
```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id0 * 4) + id1;

a[linearId] = f();
```



Row-major

```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id0 * 4) + id1;
a[linearId] = f();
```

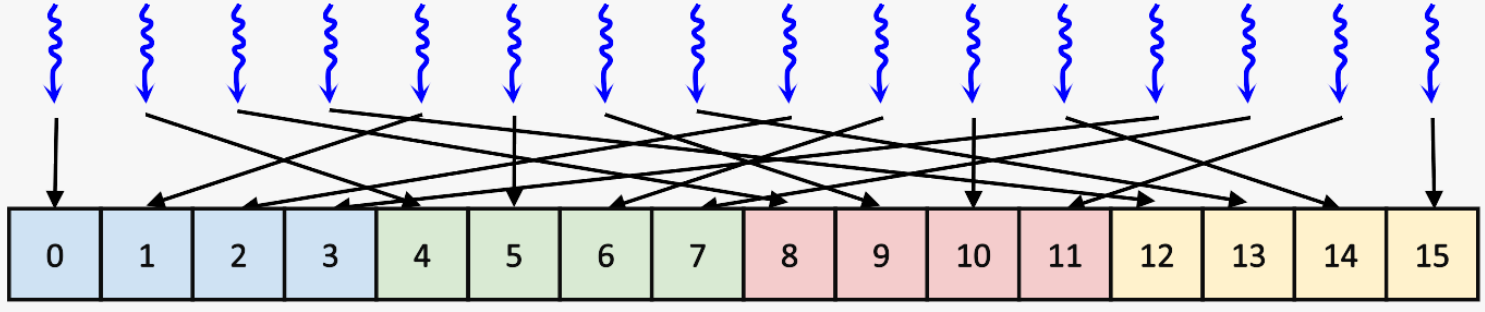




Column-major

Column-major

```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id1 * 4) + id0;

a[linearId] = f();
```



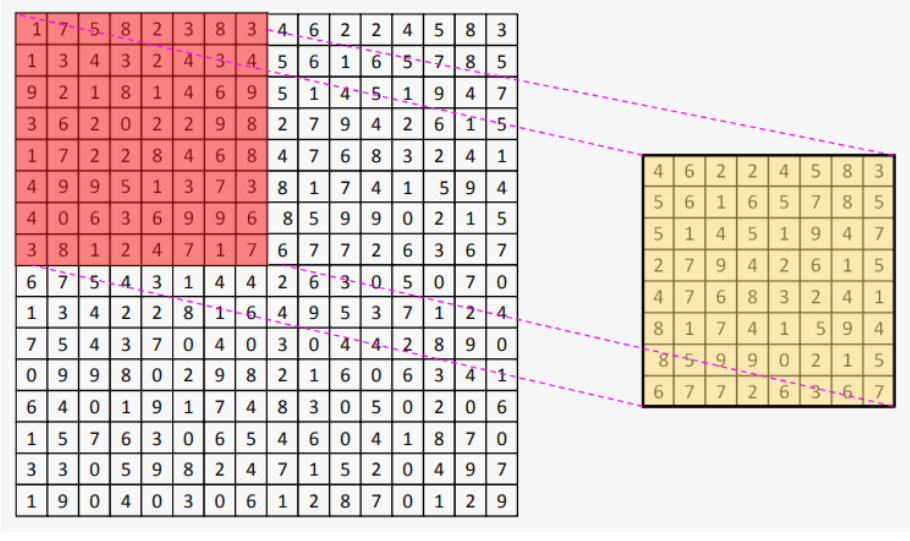
COST OF ACCESSING GLOBAL MEMORY

- Global memory is very expensive to access.
- Even with coalesced global memory access if you are accessing the same elements multiple times that can be expensive.
- Instead you want to cache those values in a lower latency memory.

USING LOCAL MEMORY

- The solution is local memory.
- Local memory is generally on-chip and doesn't have a cache as it's managed manually so is much lower latency.
- Local memory is a smaller dedicated region of memory per work-group.
- Local memory can be used to cache, allowing us to read from global memory just once and then read from local memory instead, often referred to as a scratchpad.
- Local memory can be accessed in an uncoalesced fashion without much performance degradation.

TILING



- The iteration space of the kernel function is mapped across multiple work-groups.
- Each work-group has its own region of local memory.
- You want to split the input image data into tiles, one for each work-group.

LOCAL ACCESSORS

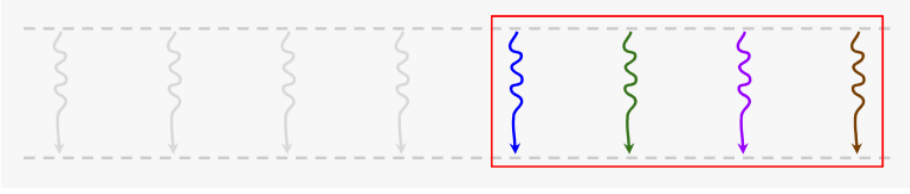
```
auto scratchpad = sycl::local_accessor<int, dims>(sycl::range{workGroupSize}, cgh);
```

- Local memory is allocated via an `accessor` with the `access::target::local` access target.
- Unlike regular `accessors` they are not created with a `buffer`, they allocate memory per work-group for the duration of the kernel function.
- The `range` provided is the number of elements of the specified type to allocate per work-group.

SYNCHRONIZATION

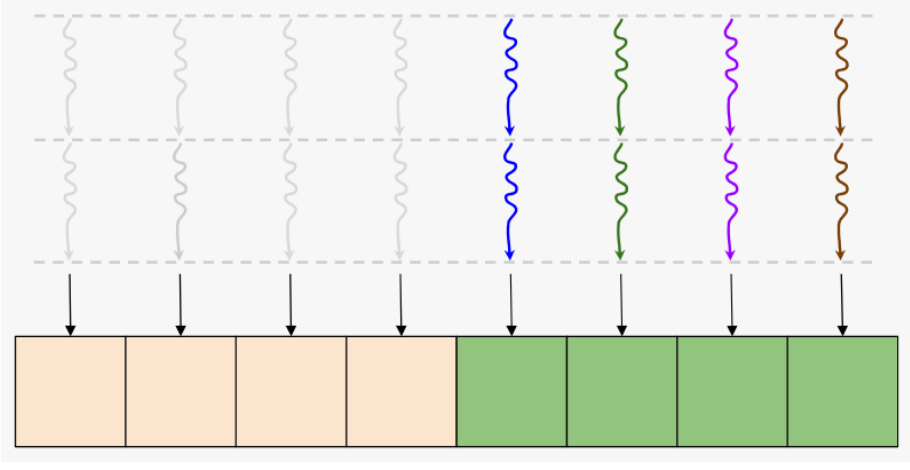
- Local memory can be used to share partial results between work-items.
- When doing so it's important to synchronize between writes and read to memory to ensure all work-items have reached the same point in the program.

SYNCHRONIZATION



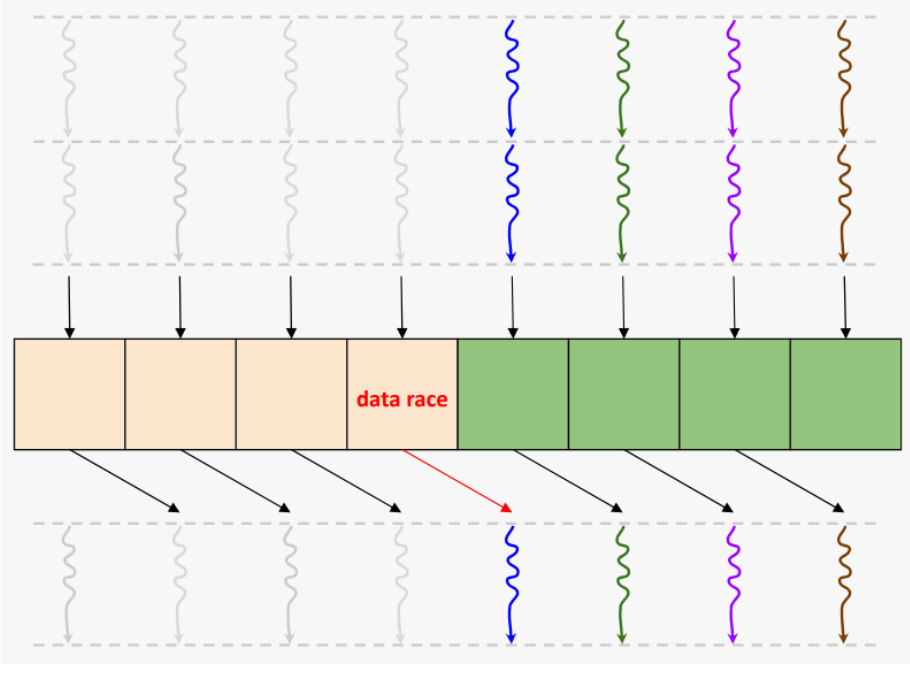
- Remember that work-items are not guaranteed to all execute at the same time (in parallel).

SYNCHRONIZATION



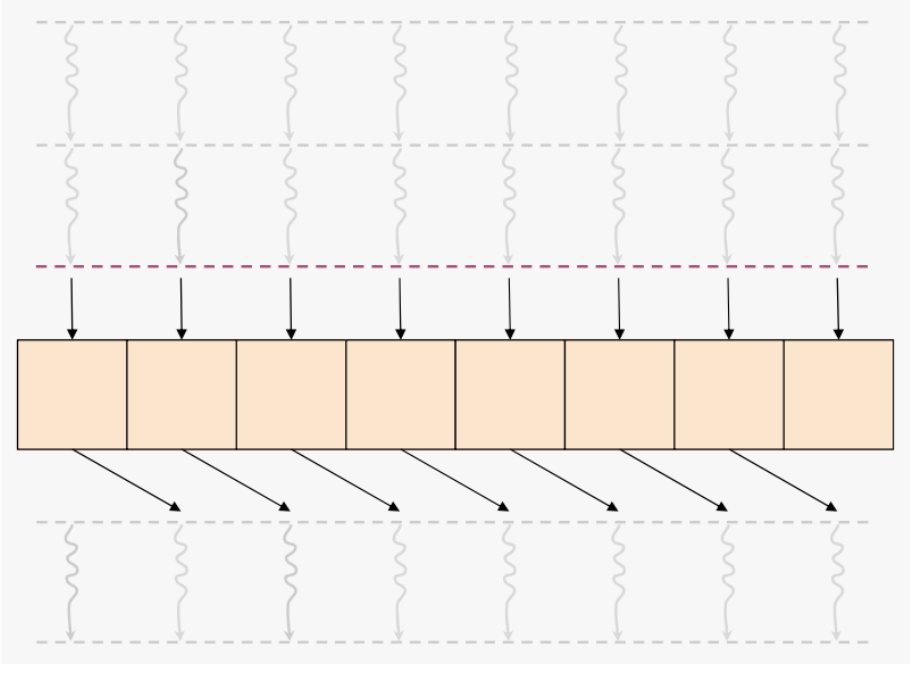
- A work-item can share results with other work-items via local (or global) memory.

SYNCHRONIZATION



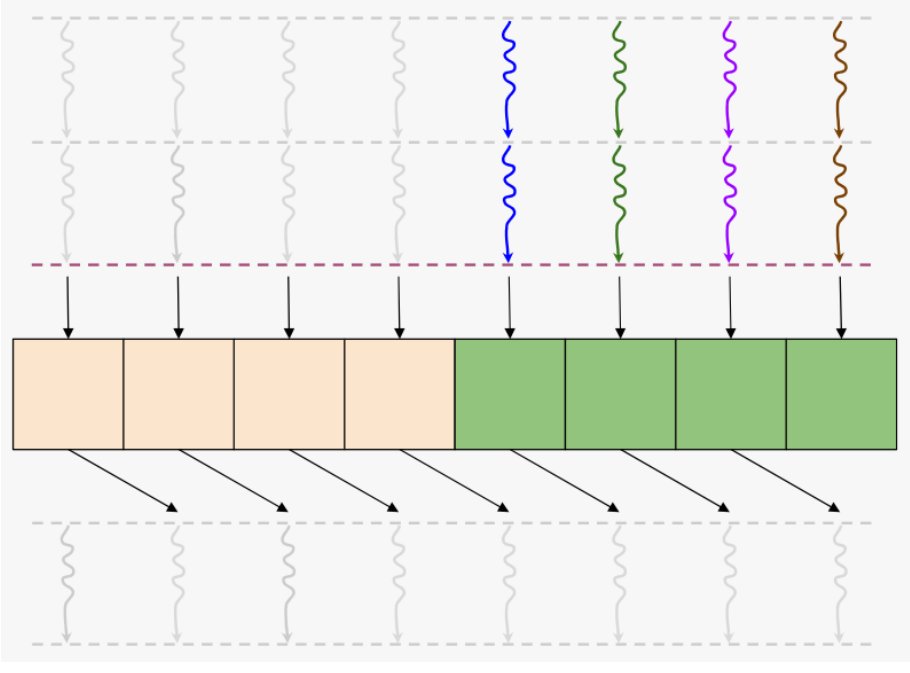
- This means it's possible for a work-item to read a result that hasn't been written to yet.
- This creates a data race.

SYNCHRONIZATION



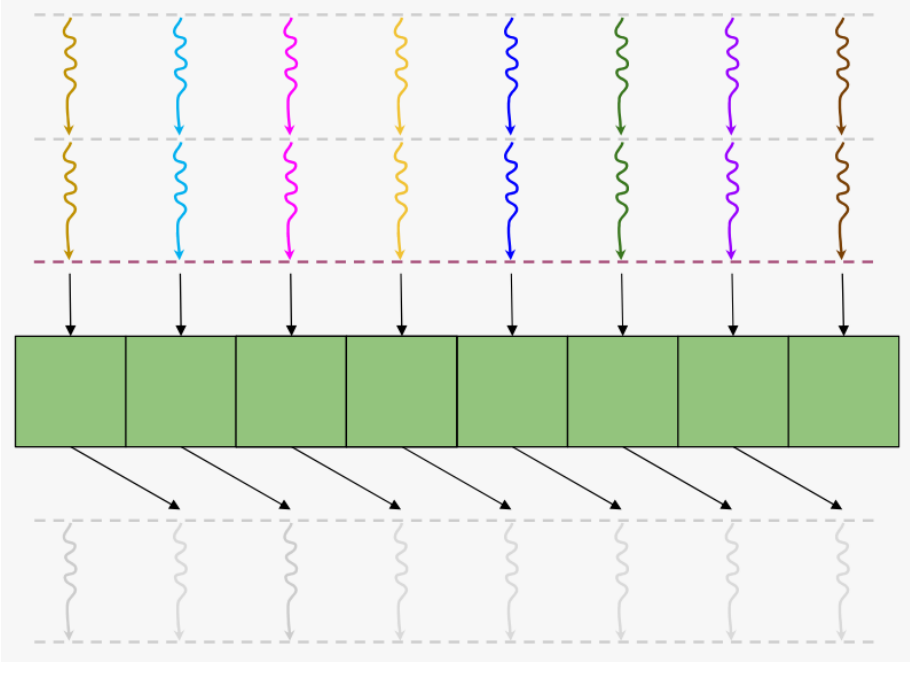
- This problem can be solved with a synchronization primitive called a work-group barrier.

SYNCHRONIZATION



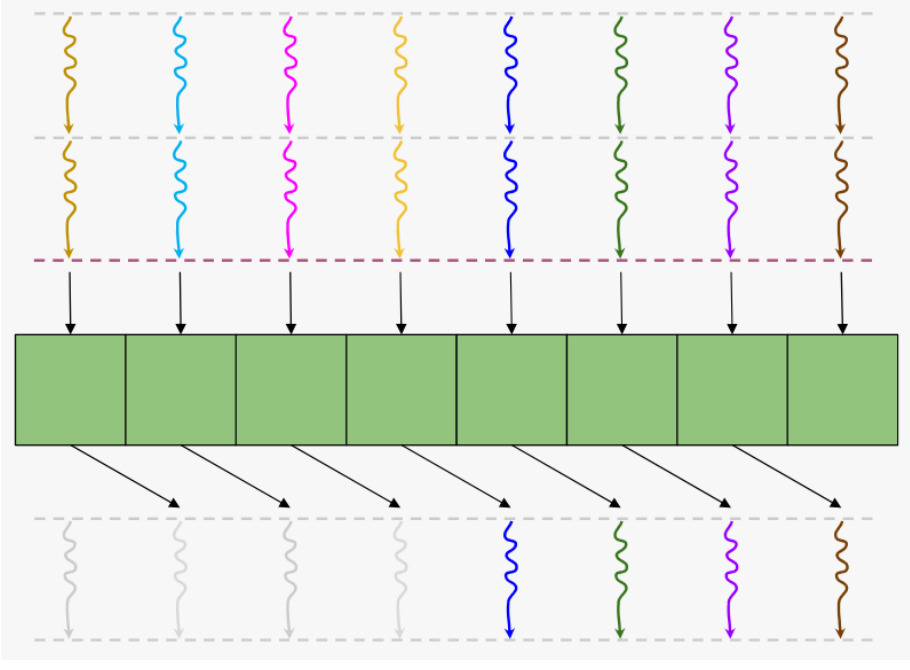
- When a work-group barrier is inserted work-items will wait until all work-items in the work-group have reached that point.

SYNCHRONIZATION



- Only then can any work-items in the work-group continue execution.

SYNCHRONIZATION



- So now you can be sure that all of the results that you want to read have been written to.

SYNCHRONIZATION

- However note that this does not apply across work-group boundaries.
- So if you write in a work-item of one work-group and then read it in a work-item of another work-group you again have a data race.
- Furthermore, remember that work-items can only access their own local memory and not that of any other work-groups.

GROUP_BARRIER

```
sycl::group_barrier(item.get_group());
```

- Work-group barriers can be invoked by calling `group_barrier` and passing a `group` object.
- You can retrieve a `group` object representing the current work-group by calling `get_group` on an `nd_item`.
- Note this requires the `nd_range` variant of `parallel_for`.

QUESTIONS

Code_Exercises/Exercise_08_Matrix_Transpose

Use good memory access patterns to transpose a matrix.