GPU Optimization Principals

Gordon Brown & Michael Wong

CppCon 2020 – Sep 2020

- Learning objectives:
  - Learn about the different levels of GPU optimization
  - Learn about good optimization practice
  - Learn about how to pick the right algorithm
  - Learn about avoiding divergent control flow
  - Learn about ensuring coalesced global memory access
  - Learn about vectorization

https://app.slack.com/client/T09P7Q5GB/GKFUQDU1G/thread/GKFUQDU1G-1600797618.098100

Search CppCon

CppCon

- Threads
- All DMs
- Mentions & reactions
- Saved items
- More

Channels

- class_brown_wong
- # general
- instructors 1
- program_committee
- # random
- safety_critical
- # announcements
- # bulletin_board
- # help_desk
- # publicity
- # ride_share
- # sig_academics
- # sig_board_games
- # sig_locals
- # sig_speakers_corner
- # sig_students
- # surveys
- # swag
- + Add a channel

Direct messages

- Slackbot
- Michael Wong (you)
- Aaron R Robinson, Ben Sa...

class_brown_wong

Add a topic

36

Break until 12:15 MDT.

Yesterday

Kirthi Banothu 2:00 PM

The class is doing a great job at introducing the what kinds of things we can do using SYCL and how they can be done. I'm having a hard time putting things into perspective on when they should be used relative to just doing it all in a CPU sequentially. Are there any resources I can look at that might help answer questions such as (listed in the thread):

Is part of this something that might be covered in the later presentations? (edited)

16 replies Last reply 15 hours ago

Sebastian Baginski 2:14 PM

Out of curiosity, is it possible to query for kind of "current timestamp" in the sycl kernel and pass it to the host (kind of std::chrono::**clock_type**::now() but on the device) ?
I mean to use this "device timestamp" to check which of the kernels B / C executed first.

8 replies Last reply 20 hours ago

Gordon Brown 2:24 PM

so to call back to the earlier question about polling for the status of an `event` you can do this by calling `e.get_info<info::event::command_execution_status>()` and this will return `event_command_status` which can be either `submitted`, `running` or `complete`

Steffen Larsen 2:33 PM

Next lecture is 12:50 MDT.

Gordon Brown 2:37 PM

quick note there have been some minor moficaitions to the materials in the repository and I have pushed the presentations for this morning so you should pull to g et these changes

Mark Lorenc 3:02 PM

are the slides for the first presentation on monday available somewhere?

Mark Lorenc 3:02 PM

im guessing if its anywhere its in the repo

2 replies Last reply 19 hours ago

Steffen Larsen 3:15 PM

Message class_brown_wong

Thread

class_brown_wong

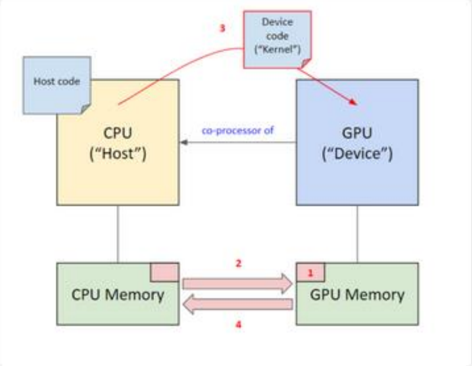
- Host allocates memory on device
- Host copies data from the app in host mem to the allocated device mem
- Host copies over and invokes a kernel on the device
- Host copies data from the allocated device memory back to the host memory to be read by the host app

Are there any referential data points on what the potential range for the overhead of each of these steps can be? If it's something trivial like an single integer and the calc, how long does each step take? Which of the steps take more of the time?
In the code examples we've been doing (like scalar add) - how big does the data have to be before you start seeing performance gains? (including the time for the data transfer + blocking waiting)

Similarly with USM - it seems pretty neat. When would you want to use it over the accessor model when it comes to performance? Are there bounds at which one is better than the other?

I don't really have a grasp on the order of magnitude that these steps take (nanos, mics, millis, etc) and I'm not really sure how to think about the trade offs here.

CPU - GPU steps.png



Steffen Larsen 20 hours ago

Those are some very good questions, but for the most part it comes down to the hardware and the problem. Since the GPU acts as a co-processor to the CPU you are very likely to see overhead from communication latency between the CPU and the GPU as well as a transfer overhead in many cases. Pin-pointing numbers for that is generally difficult, but a good rule of thumb is that if you can think of a way to formulate your problem using some of the ideas Michael talked about earlier, then chances are that, with enough data, you can benefit from using a GPU. Sorry for being a bit hand-wavey.

Steffen Larsen 20 hours ago

Notice also that GPUs, due to their various uses and general structure, have some interesting optimization you can do. I believe @Gordon Brown will be talking about some of these tomorrow.

Kirthi Banothu 20 hours ago

Hmm, that makes sense. Is there any data for any cpu / gpu combination? Would be nice to see relative values or getting a feel for the

ENG 10:02 AM

Answer

- There is no silver bullet, but there are fundamentals that experienced practitoners reach for …

There are different levels of optimizations you can apply

- Choosing the right algorithm
  - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
  - *Such as coalescing global memory access or using local memory*
- Architecture specific optimisations
  - *Optimising for register usage or avoiding bank conflicts*
- Micro-optimisations
  - *Such as floating point dnorm hacks*

There are different levels of optimizations you can apply

- Choosing the right algorithm
  - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
  - *Such as coalescing global memory access or using local memory*
- Architecture specific optimisations
  - *Optimising for register usage or avoiding bank conflicts*
- Micro-optimisations
  - *Such as floating point dnorm hacks*

This class will focus on these two

Choosing the right algorithm

What to parallelise on a GPU

- Find hotspots in your code base
  - *Looks for areas of your codebase that are bottlenecks that are taking up a lot of execution time*

- Look for areas of your codebase that are a good fit for GPU parallelism
  - *Or areas of you codebase that could be adapted for parallelism*

Follow good optimisation practice

- Follow an adaptive optimisation approach such as APOD
  - ***A**nalyse your code base to find opportunities for parallelism*
  - ***P**arallelise a piece of code on the GPU*
  - ***O**ptimise the algorithm to be more efficient on the GPU*
  - ***D**eploy the codebase to evaluate the performance with different inputs*

- Avoid over-optimisation
  - *You may reach a point where optimisations provide diminishing returns*
  - *It may be more efficient to look for another bottleneck in your codebase*

What to look for in an algorithm

- Naturally data parallel
    - *Performing the same operation on multiple items in the computation*
- Large problem
    - *Enough work to utilise the GPU's processing elements*
- Independent progress
    - *Little or no dependencies between items in the computation*
- Non-divergent control flow
    - *Little or no branch or loop divergence*

Embarrassingly parallel algorithms

- Some problems are considered “embarrassingly parallel”
  - *The problem is naturally parallel*
  - *The problem has no communication between items in the computation*

- These kind of problems are perfect for the GPU

Basic GPU programming principles

Optimizing GPU programs means maximizing occupancy and throughput

Maximize compute operations

$$\frac{\text{Compute}}{\text{Memory}}$$

Reduce time spent on memory

Optimizing GPU programs means maximizing throughput

$$\frac{\text{Compute}}{\text{Memory}}$$

- Compute bound
  - *Problems that are bound by the computation time*
- Memory bound
  - *Problems that are bound by the memory operations time*

Optimizing GPU programs means maximizing throughput

$$\frac{\text{Compute}}{\text{Memory}}$$

- Compute bound
  - *Problems that are bound by the computation time*
- Memory bound
  - *Problems that are bound by the memory operations time*

Most GPU problems are memory bound

Optimizing GPU programs means maximizing occupancy and throughput

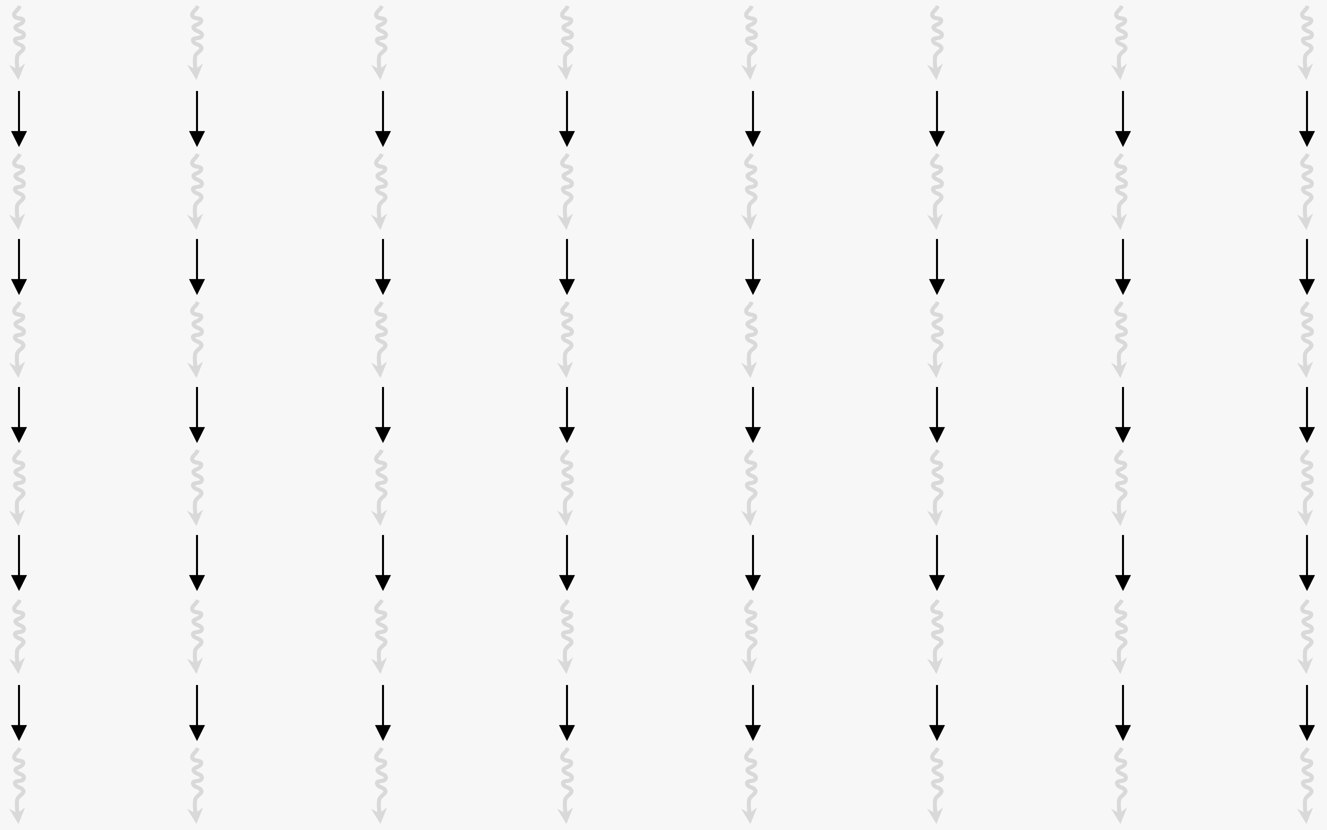
- Maximise compute operations per cycle
  - *Make effective utilisation of the GPU's hardware*

- Reduce time spent on memory operations
  - *Reduce latency of memory access*

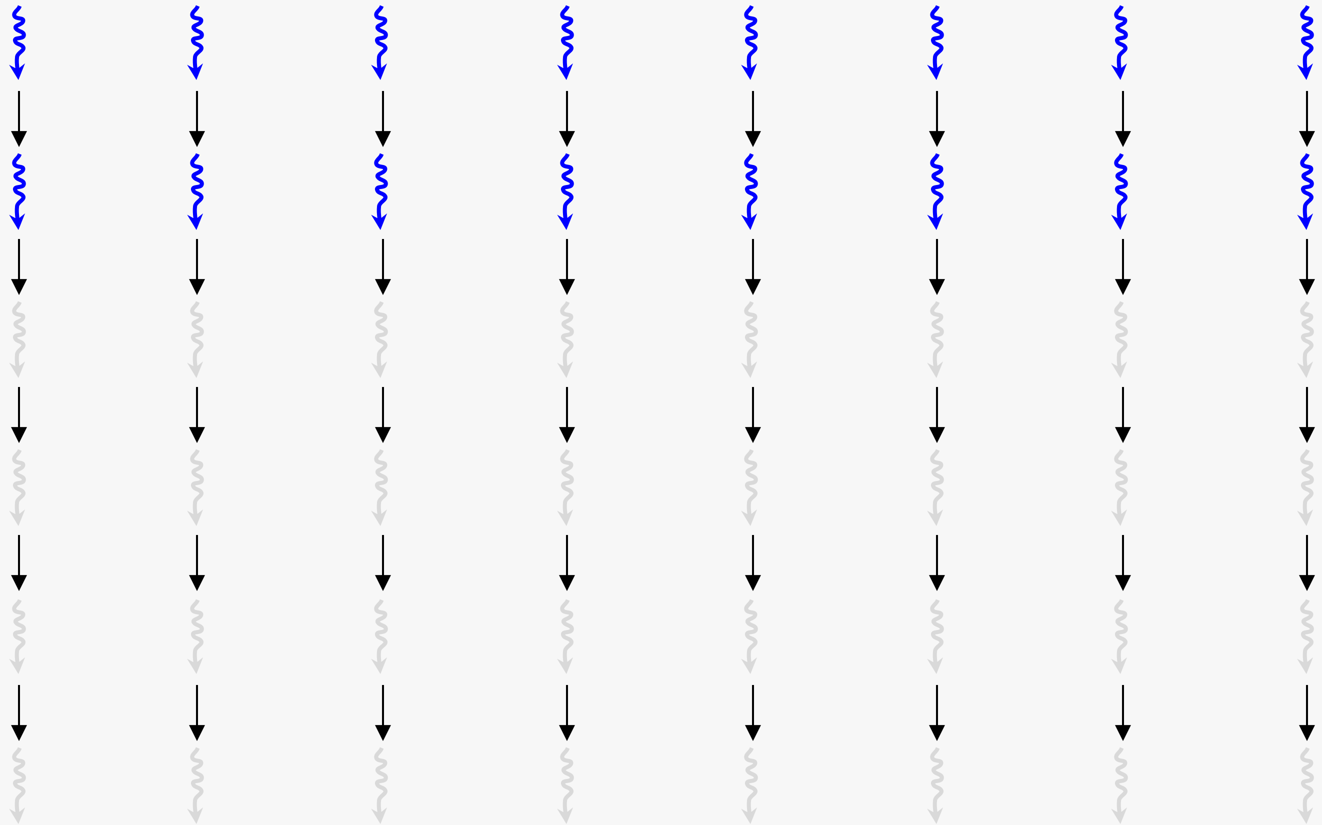
Avoid divergent control flow

- Divergent branches and loops can cause inefficient utilisation
  - *If consecutive work-items execute different branches they must execute separate instructions*
  - *If some work-items execute more iterations of a loop than neighbouring work-items this leaves them doing nothing*

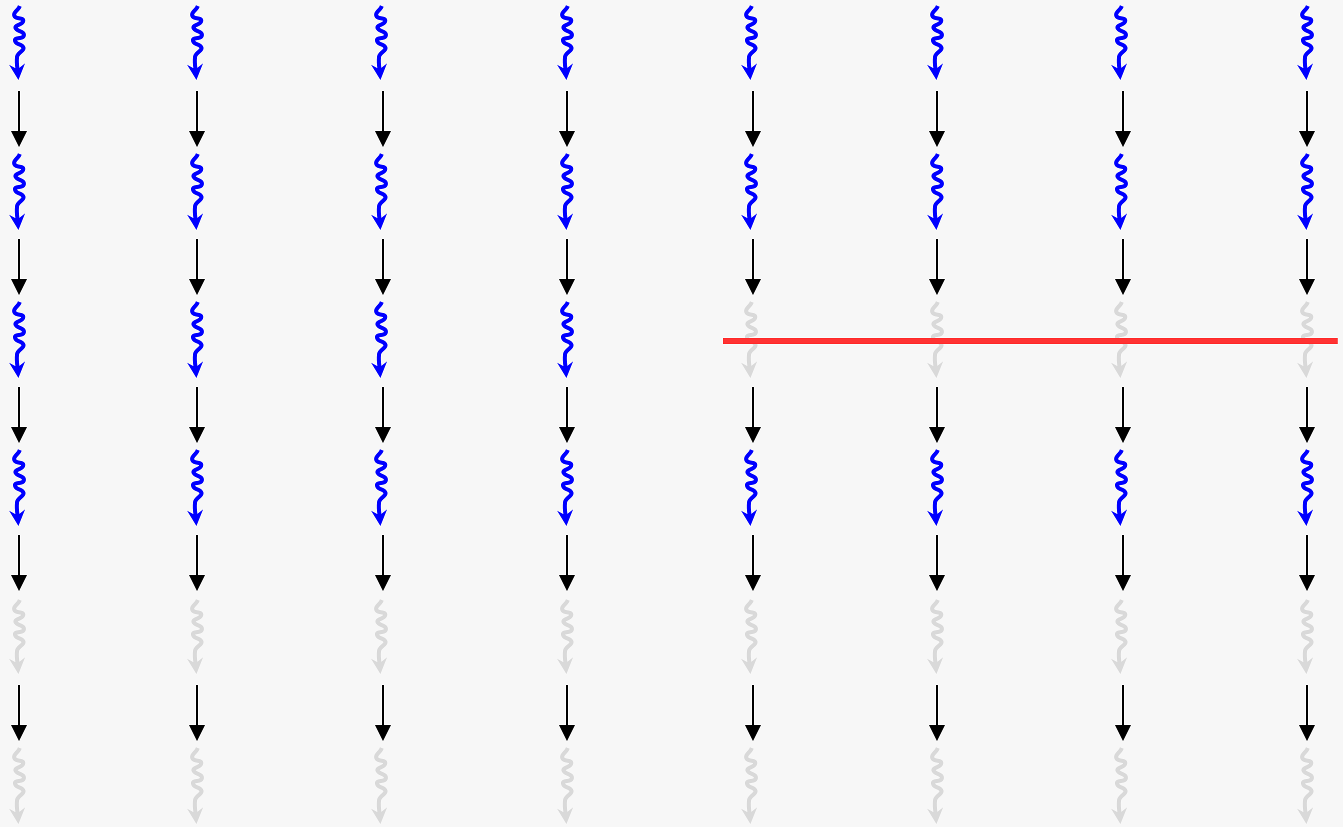
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



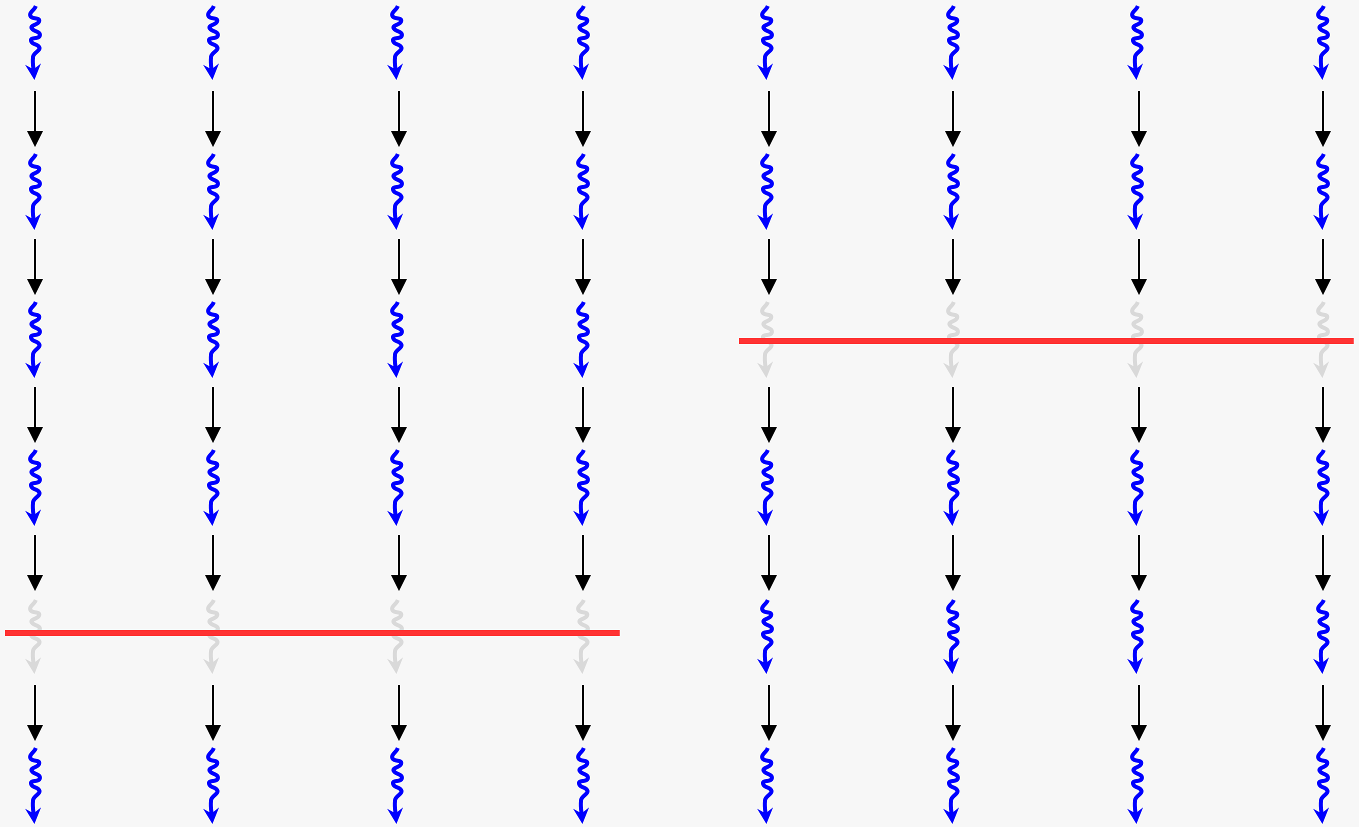
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



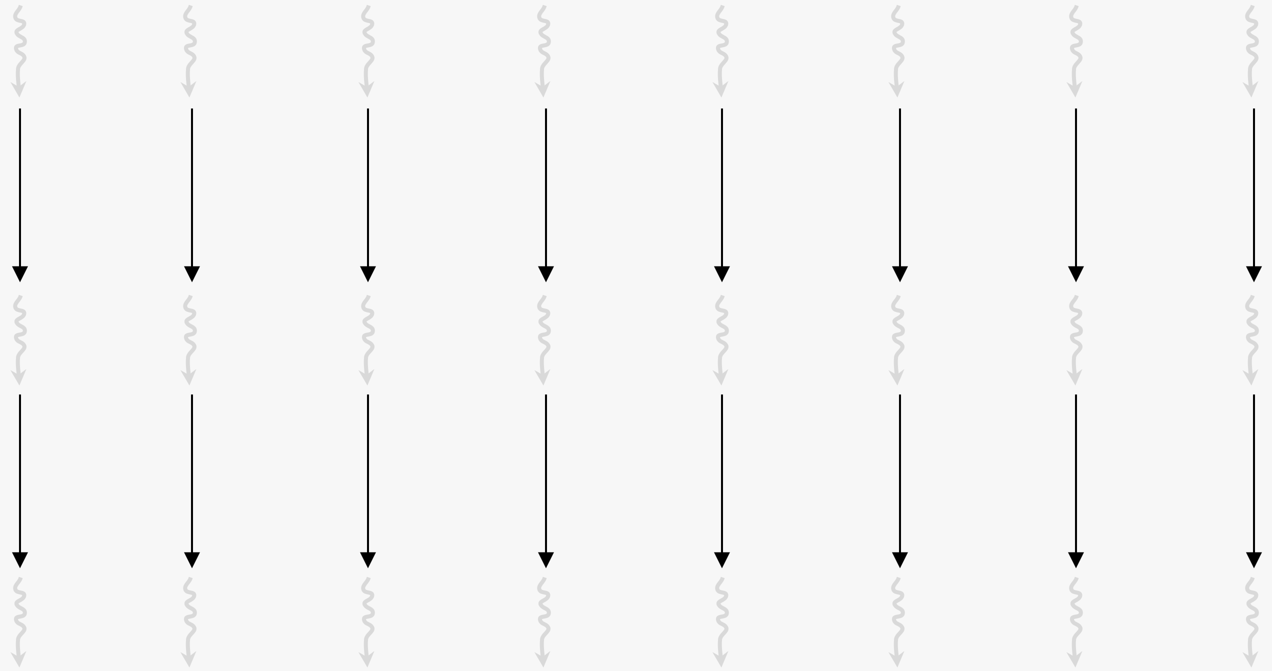
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



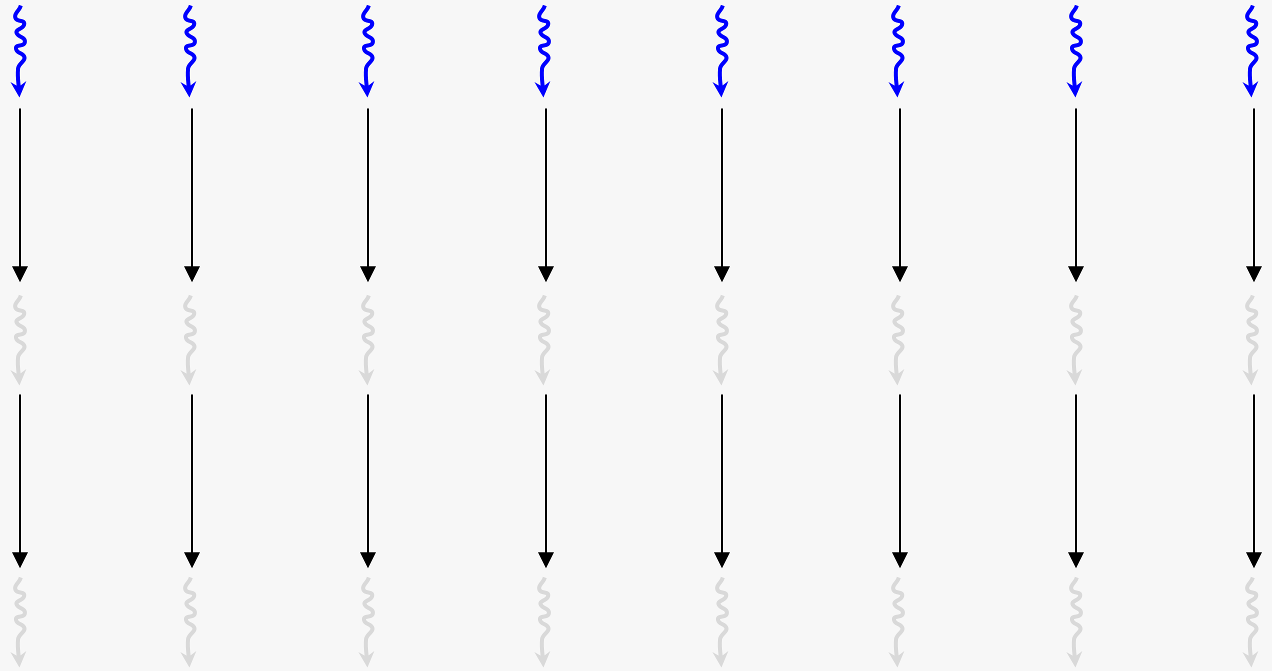
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



```
…

for (int i = 0; i < 
  globalId; i++) {
  do_something();
}

…
```



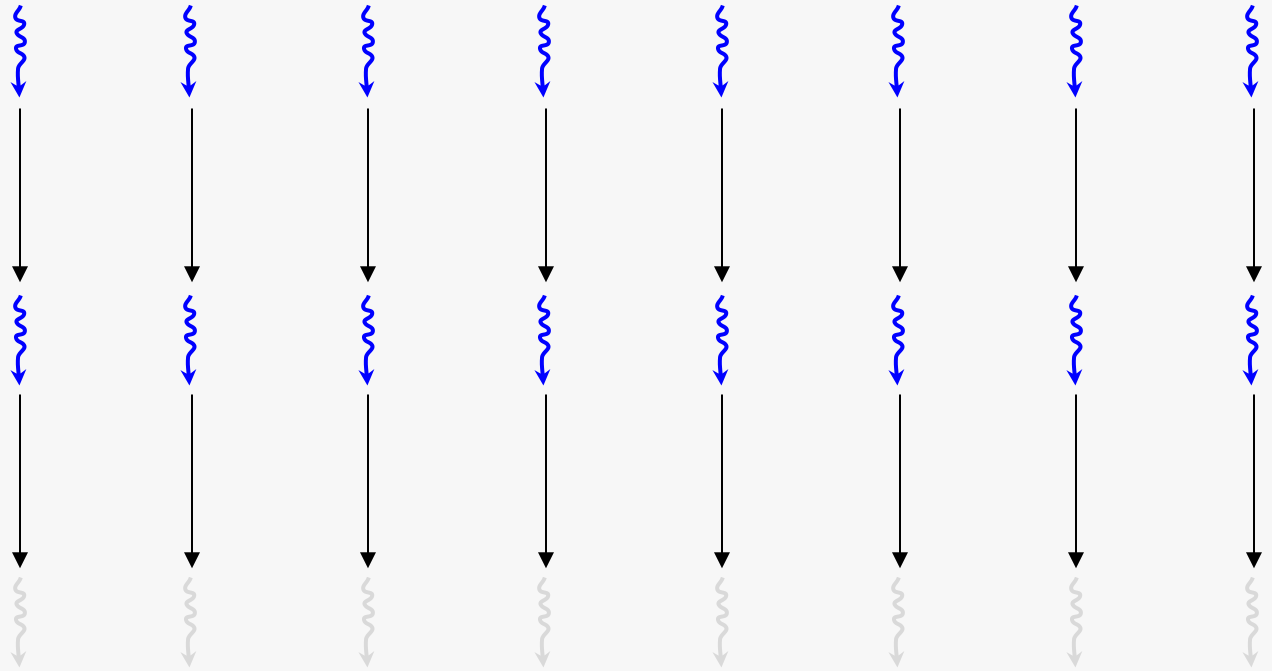
```
...

for (int i = 0; i <
  globalId; i++) {
  do_something();
}

...
```



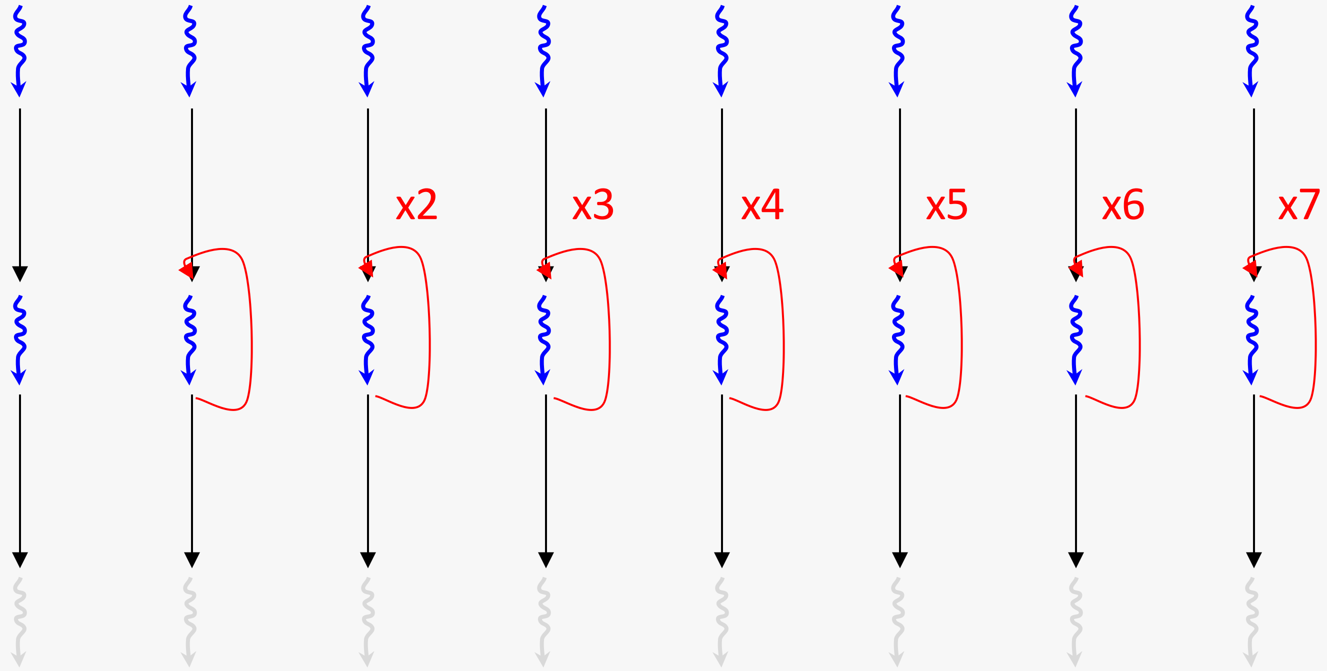
…

```
for (int i = 0; i <
  globalId; i++) {
  do_something();
}
```

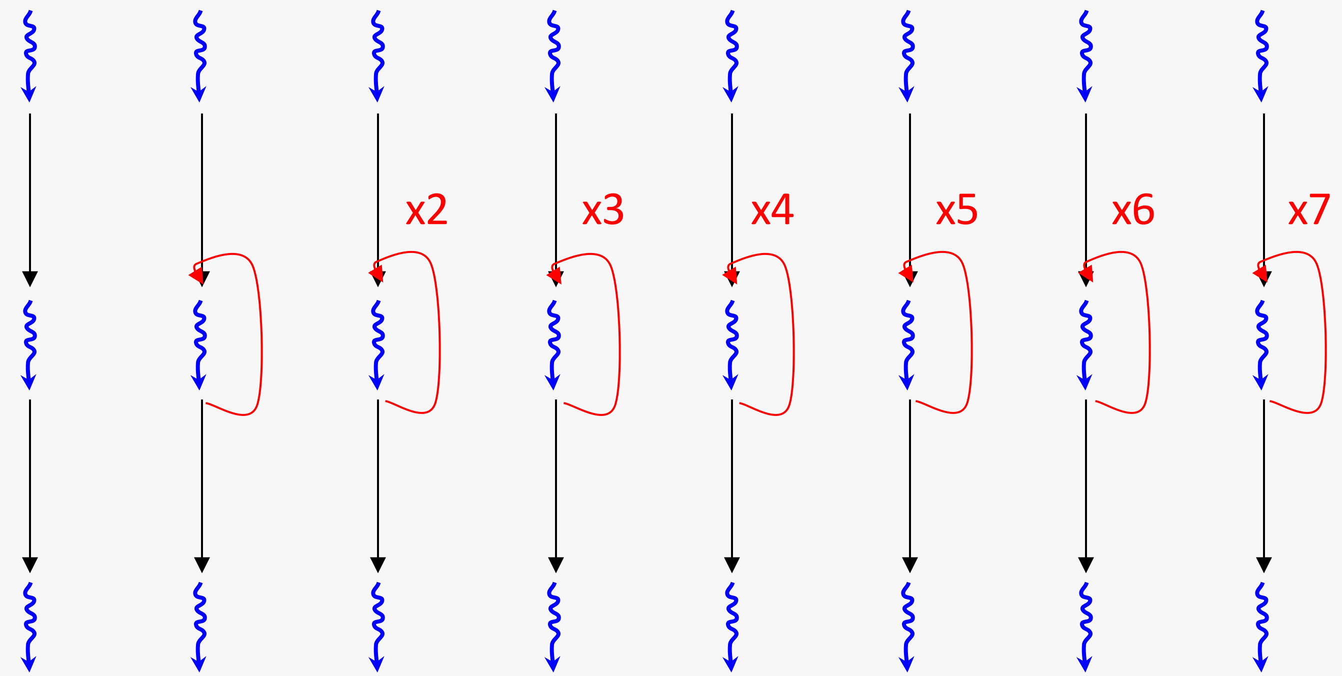
…



```
...

for (int i = 0; i <
  globalId; i++) {
  do_something();
}

...
```



```
...

for (int i = 0; i <
   globalId; i++) {
   do_something();
}

...
```



Coalesced global memory access

- Reading and writing from global memory is very expensive
  - *It often means copying across an off-chip bus*

- Reading and writing from global memory is done in strides
  - *This means accessing data that is physically close together in memory is more efficient*

Make use of vector operations

- GPUs are vector processors
  - *Each processing element is capable of wide instructions which can operate on multiple elements of data at once*

- Many compilers can auto-vectorise
  - *This can affect the amount of performance gain you may see in vectorising your kernels*

Make use of local memory

- Local memory is much lower latency to access than global memory
  - *Cache commonly accessed data and temporary results in local memory rather than reading and writing to global memory*

- Using local memory is not necessarily always more efficient
  - *If data is not accessed frequently enough to warrant the copy to local memory you may not see a performance gain*

Synchronise work-groups when necessary

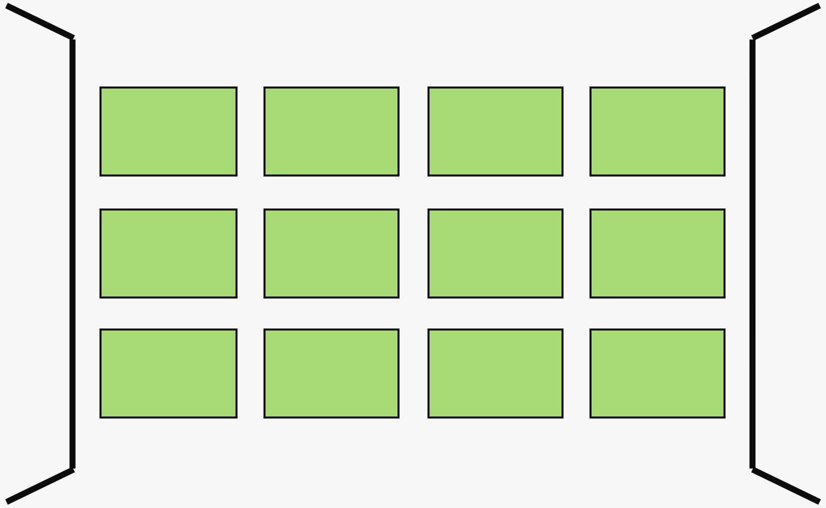
- Synchronising with a work-group barrier waits for all work-items to reach the same point
  - *Use a work-group barrier if you are copying data to local memory that neighbouring work-items will need to access*
  - *Use a work-group barrier if you have temporary results that will be shared with other work-items*

Choosing a good work-group size

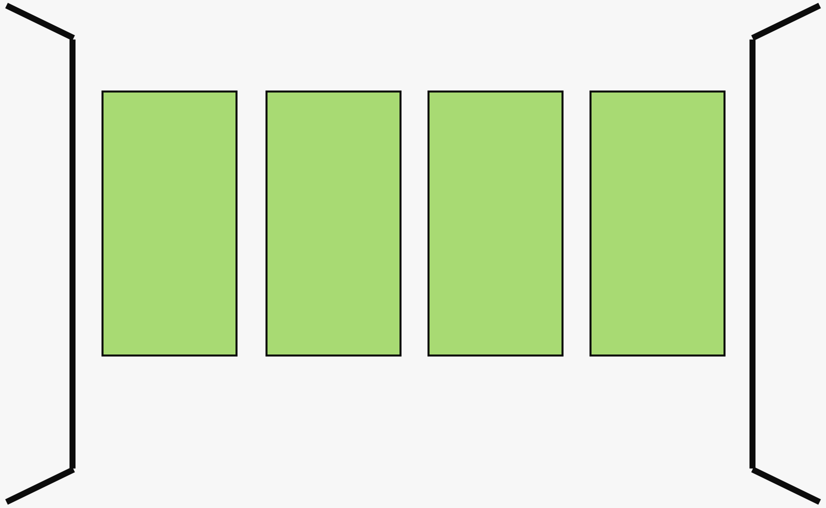
- The occupancy of a kernel can be limited by a number of factors of the GPU
  - *Processing elements, compute units, local memory, registers*
- You can query the preferred work-group size once the kernel is compiled
  - *However this is not guaranteed to give you the best performance*
- It's good practice to benchmark various work-group sizes and choose the best
  - *Different algorithms will work better with different work-group sizes depending on their need for local memory and registers*

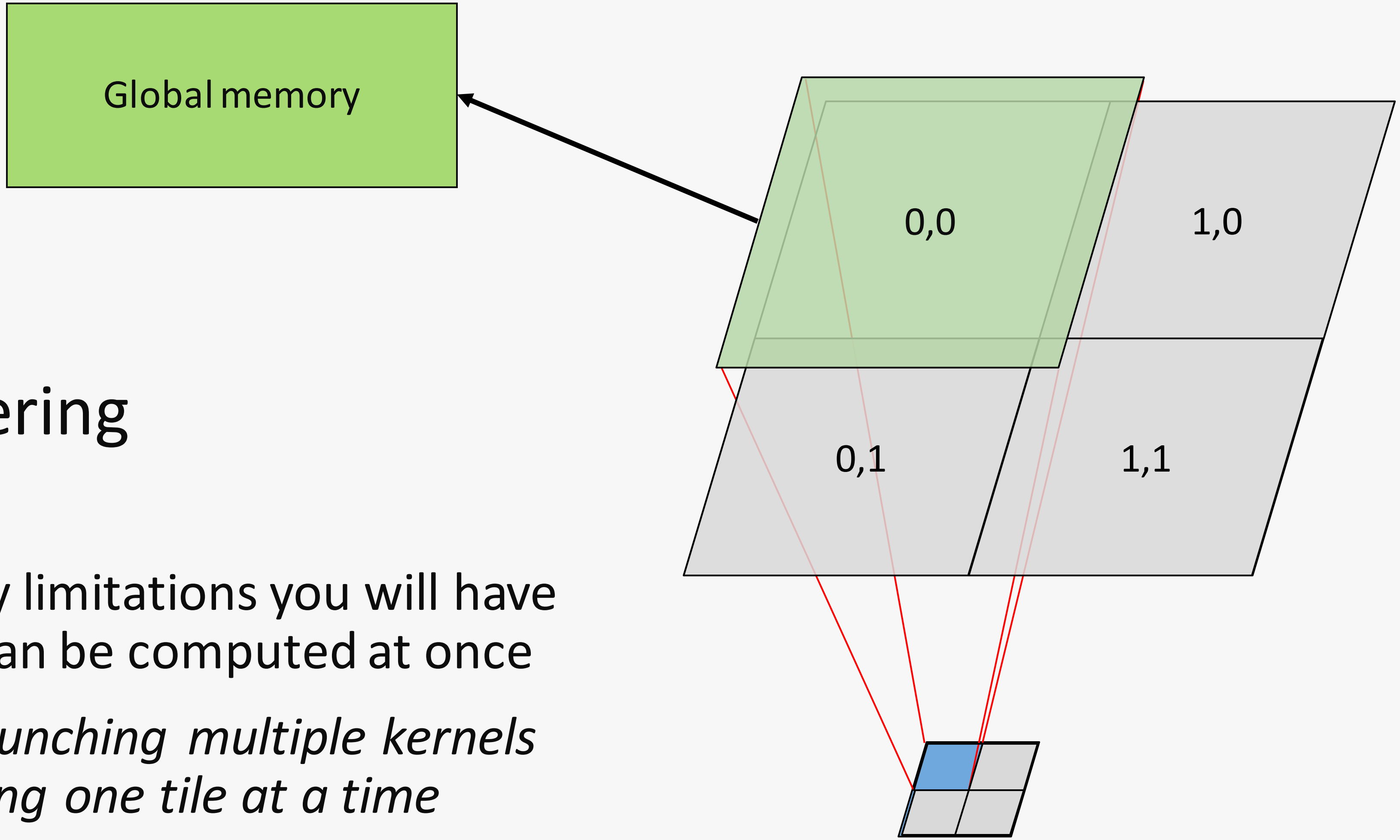
Batch work together

- Hitting occupancy limitations of a GPU can *lead to drops in performance gain*
  - *This is because single work-items are having to do more chunks of work*



- *Batching work for each work-item allows reusing cached data*
  - *Batching work that share neighbouring data allows you to further share local memory and registers*





Use double buffering

- If you hit memory limitations you will have more data than can be computed at once
  - *This means launching multiple kernels each computing one tile at a time*
- If you double buffer then you can hide the latency of data movement

Further tips

- Use profiling tools to gather more accurate information about your programs
  - *SYCL provides kernel profiling*
  - *Most OpenCL implementations provide proprietary profiler tools*

- Follow vendor optimisation guides
  - *Most OpenCL vendors provide optimisation guides that detail recommendations on how to optimise programs for their respective GPU*

Key takeaways

Porting to the GPU can give significant performance gain providing the problem is well suited to GPU parallelism

Reduce time spent on memory operations by using lower latency memory and optimizing the access patterns

Be aware of occupancy limitations and apply optimization to increase occupancy and hide the latency of data movement codeplay®

THE HETEROGENEOUS SYSTEMS EXPERTS

Questions?