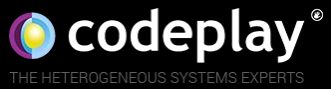


Parallelism in Modern C++; from CPU to GPU

Gordon Brown, Michael Wong
Codeplay C++ Std and SYCL team

CppCon 2018

Exercises

The exercises will take the form of implementing parallel versions of a number of standard algorithms

The exercises can be found here:

https://github.com/AerialMantis/cppcon2018-parallelism-class

The master branch of this repo has the exercises and the solutions branch has the solutions to the exercises

Setting up ComputeCpp SYCL

Some of the exercises will require you to have ComputeCpp SYCL configured

- You can do this by installing ComputeCpp and the OpenCL drivers on your machine
- Or you can do this by using the docker image we are providing

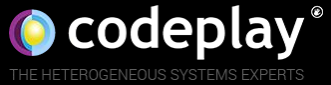
All of the instructions for this are in the Github README

Wifi

CppCon
Password is: stepanov

Schedule

Day 1	Day 2
Welcome	
Chapter 1: Importance of Parallelism Chapter 2: Standard Algorithms	Chapter 8: Data parallelism Chapter 9: CPU & GPU Architecture
Break	
Exercise 1: Sequential Algorithms Chapter 3: Fundamentals of Parallelism	Chapter 10: C++ for GPUs (SYCL) Exercise 3: GPU Algorithms
Lunch	
Chapter 4: Performance Portability Chapter 5: C++ Multi-threaded programming	Chapter 11: C++ for GPUs (SYCL) cont.
Break	
Exercise 2: Thread Parallel Algorithms Chapter 6: Synchronization & Advanced Abstraction Chapter 7: Atomics & Memory Model	Exercise 4: GPU Algorithms cont. Chapter 12: Guidelines for Parallel Computing Question & Answer


codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Chapter 8: Data Parallelism

Prerequisite

1. Previous Chapters
2. Transition from CPU to GPU parallelism
3. Latency vs throughput

You will learn:

1. Data vs task parallelism
2. Flynn’s taxonomy
3. Multicore CPU vs manycore GPU
4. Current state of SIMD computing
5. Auto-vectorization
6. C++ Standard effort for SIMD

Task vs data parallelism

Task parallelism:

- Few large tasks with different operations / control flow
- Optimized for latency

Data parallelism:

- Many small tasks with same operations on multiple data
- Optimized for throughput

Review of Latency, bandwidth, throughput

- **Latency** is the amount of time it takes to travel through the tube.
- **Bandwidth** is how wide the tube is.
- The amount of water flow will be your **throughput**

Definition and examples

Latency is the time required to perform some action or to produce some result. Latency is measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.

Throughput is the number of such actions executed or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. The term "memory bandwidth" is sometimes used to specify the throughput of memory systems.

bandwidth is the maximum rate of data transfer across a given path.

Example

An assembly line is manufacturing cars. It takes eight hours to manufacture a car and that the factory produces one hundred and twenty cars per day.

The latency is: 8 hours.

The throughput is: 120 cars / day or 5 cars / hour.


DMV
LET'S PUT IT THIS WAY. WHEN I GOT IN LINE, I DIDN'T HAVE A BEARD!
LINE 2

Flynn's Taxonomy

- Distinguishes multi-processor computer architectures along the two independent dimensions
  - ***Instruction*** and ***Data***
  - Each dimension can have one state: ***Single*** or ***Multiple***
- SISD: Single Instruction, Single Data
  - Serial (non-parallel) machine
- SIMD: Single Instruction, Multiple Data
  - Processor arrays and vector machines
- MISD: Multiple Instruction, Single Data (weird)
- MIMD: Multiple Instruction, Multiple Data
  - Most common parallel computer systems

Assuming power is the constraint

What kind of processors are we building?

- CPU
  - complex control hardware
  - Increasing flexibility + performance
  - Expensive in power
- GPU
  - Simpler control structure
  - More HW per computation
  - Potentially more efficient in ops/watt
  - More restrictive programming model

Multicore CPU vs Manycore GPU

- Each core optimized for a single thread
- Fast serial processing
- Must be good at everything
- Minimize latency of 1 thread
  - Lots of big on chip caches
  - Sophisticated controls

- Cores optimized for aggregate throughput, deemphasizing individual performance
- Scalable parallel processing
- Assumes workload is highly parallel
- Maximize throughput of all threads
  - Lots of big ALUs
  - Multithreading can hide latency, no big caches
  - Simpler control, cost amortized over ALUs via SIMD

SIMD hard knocks

- SIMD architectures use data parallelism
- Improves tradeoff with area and power
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler
- Hard for a compiler to exploit SIMD
- Hard to deal with sparse data & branches
  - C and C++ Difficult to vectorize, Fortran better
- So
  - Either forget SIMD or hope for the autovectorizer
  - Use compiler intrinsics

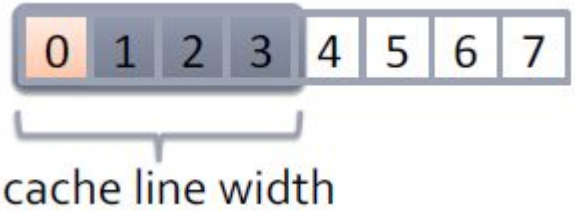
Memory

- Many core gpu is a device for turning a compute bound problem into a memory bound problem

- Lots of processors but only one socket
- Memory concerns dominate performance tuning

Memory is SIMD too

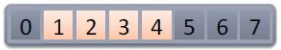
- Virtually all processors have SIMD memory subsystems



- This has 2 effects
  - Sparse access wastes bandwidth
  -

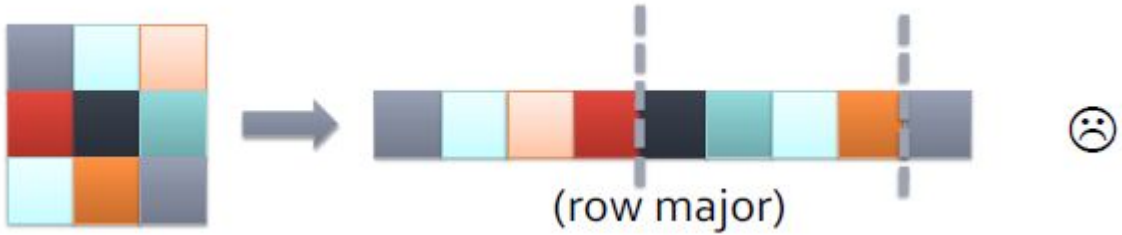
2 words used, 8 words loaded:
¼ effective bandwidth

  - Unaligned access wastes bandwidth

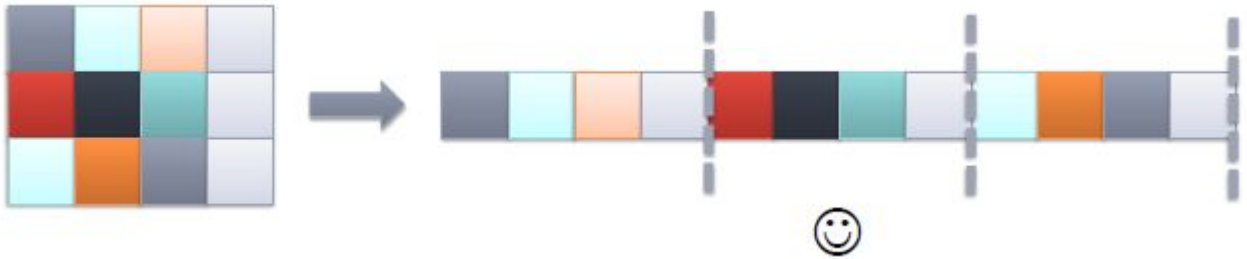


4 words used, 8 words loaded:
½ effective bandwidth

Data Structure Padding



- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern



Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests (cache line)
- GPUs have a coalescer which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

Power of Computing

- 1998, when C++ 98 was released
  - Intel Pentium II: 0.45 GFLOPS
  - No SIMD: SSE came in Pentium III
  - No GPUs: GPU came out a year later
- 2011: when C++11 was released
  - Intel Core-i7: 80 GFLOPS
  - AVX: 8 DP flops/HZ*4 cores *4.4 GHz= 140 GFlops
  - GTX 670: 2500 GFLOPS
- Computers have gotten so much faster, how come software have not?
  - Data structures and algorithms

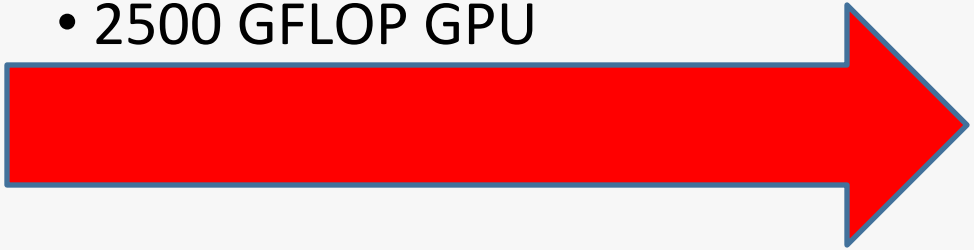
In 1998, a typical machine had the following flops

- .45 GFLOP, 1 core

- Single threaded C++98/C99/Fortran dominated this picture

In 2011, a typical machine had the following flops

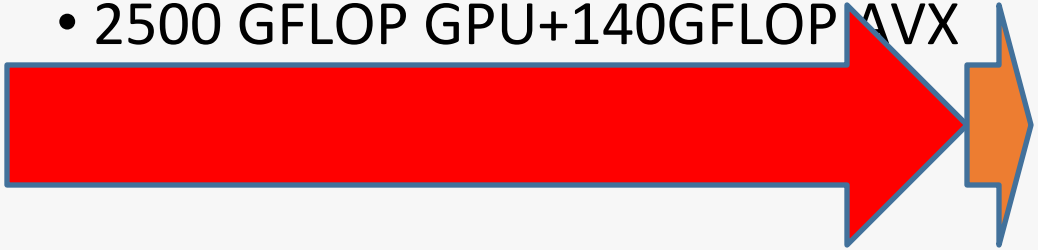
- 2500 GFLOP GPU



- To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP

In 2011, a typical machine had the following flops

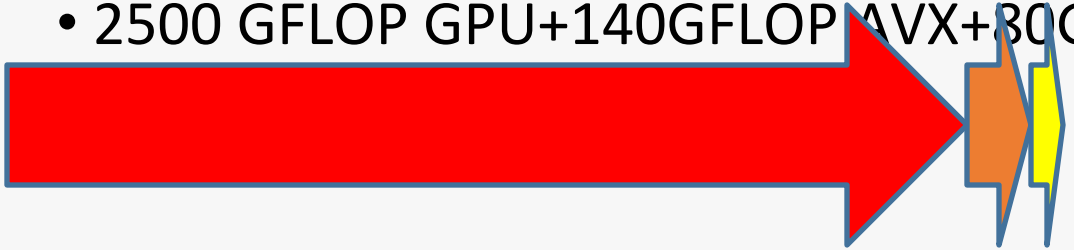
- 2500 GFLOP GPU+140GFLOP AVX



- To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use Intrinsics, OpenCL, or auto-vectorization

In 2011, a typical machine had the following flops

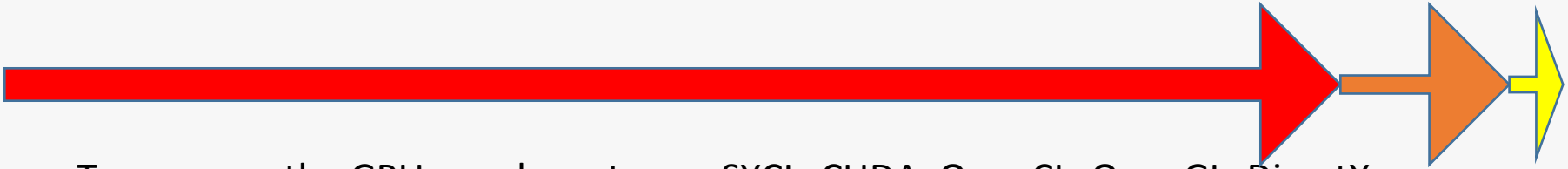
- 2500 GFLOP GPU+140GFLOP AVX+80GFLOP 4 cores



- To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use Intrinsics, OpenCL, or auto-vectorization
- To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, OpenCL

In 2017, a typical machine had the following flops

- 4600 GFLOP GPU+560 GFLOP AVX+140 GFLOP



- To program the GPU, you have to use SYCL, CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization, OpenMP
- To program the CPU, you might use C/C++11/14/17, SYCL, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, OpenMP, parallelism TS, Concurrency TS, OpenCL

SIMD Language Extensions

- IBM currently has 7 SIMD architectures
  - VMX, VMX128, VSX, SPE, BGL, BGQ, QPX
  - Each has its own proprietary language extension
  - Code written for one language extension can't be moved without a rewrite
  - We don't even have compatibility within our own company
- Intel has 7 SIMD architectures
  - MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX-512/MIC
  - MMX, SSEx, AVX each have a different language extension
  - Code written for one language extension can't be moved without a rewrite

“The Great Hope”- Auto-Vectorizing compilers

- SIMD floating point entered the market 15 years ago
  - (Intel Pentium III, Motorola G4, AMD K6-2)
- Software industry held its breath waiting for a “magic” auto-vectorizing compiler (including Microsoft)
- Despite 15 years of research and development the industry still doesn’t have a good auto-vectorizing compiler
- Industry instead ended up with primitive language support
  - Multiple non-compatible language extensions
  - Compiler intrinsics
- Using intrinsics humans still produce superior vector code but at great pain

Why autovectorization is hard?

- SIMD register width has increased from 128-256-512, 1024 soon
- Instructions are more powerful and complex
  - Hard for compiler to select proper instruction
  - Code pattern needs to be recognized by the compiler
  - Precision requirements often inhibit SIMD codegen

What sort of loops can be vectorized?

- Countable
- Single entry, single exit
- Straight-line code
- Innermost loop of a nest
- No function calls
- Certain non-contiguous memory access
- Some Data dependencies
- Efficient Alignment
- Mixed data types
- Non-unit stride between elements
- Loop body too complex (register pressure)

Industry needs better language support for SIMD

- 80% of Cell programmers time spent vectorizing code
- Need to reduce programming effort
  - Fewer code modifications to vectorize
  - Rapid conversion of scalar to vector code
- Code portability
  - Don’t rewrite for every SIMD architecture
- Less code maintenance
  - Intrinsics impossible to maintain
  - Easier to rewrite then figuring out what the code is doing
- Support required vendor-specific extensions

Emerging C++ Parallelism TS2

- No standard!
- Boost.SIMD
- Proposal N3571 by Mathias Gaunard et. al., based on the Boost.SIMD library.
- Proposal N4184 by Matthias Kretz, based on Vc library.
- Unifying efforts and expertise to provide an API to use SIMD portably
- Within C++
- P0193 status report
- P0203 design considerations
- P0214 latest SIMD paper

SIMD from Matthias Kretz

- std::simd<T, N, Abi>
  - simd<T, N> SIMD register holding N elements of type T
  - simd<T> same with optimal N for the currently targeted architecture
  - Abi Defaulted ABI marker to make types with incompatible ABI different
  - Behaves like a value of type T but applying each operation on the N values it contains, possibly in parallel.
- Constraints
  - T must be an integral or floating-point type (tuples/struct of those once we get reflection)
  - N parameter under discussion, probably will need to be power of 2.

Operations on SIMD

- Built-in operators
- All usual binary operators are available, for all:
  - simd<T, N> simd<U, N>
  - simd<T, N> U, U simd<T, N>
- Compound binary operators and unary operators as well
  - simd<T, N> convertible to simd<U, N>
  - simd<T, N>(U) broadcasts the value

- No promotion:
  - simd<uint8_t>(255) + simd<uint8_t>(1) == simd<uint8_t>(0)
- Comparisons and conditionals:
  - ==, !=, <, <=,> and >= perform element-wise comparison return mask<T, N, Abi>
  - if(cond) x = y is written as where(cond, x) = y
  - cond ? x : y is written as if_else(cond, x, y)

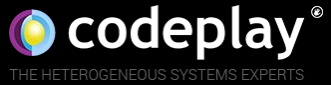
Key takeaways

Task vs Data Parallelism

Multicore CPU vs Manycore GPU

Auto vectorisation (implicit SIMD) is not as reliable as explicit SIMD

We need a standard to converge the many SIMD formats


codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Chapter 9: CPU & GPU Architecture

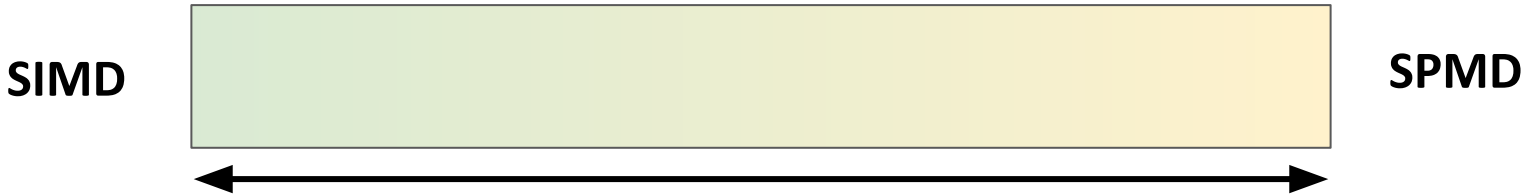
Prerequisite:

1. Previous chapters
2. Latency vs throughput vs bandwidth
3. Flynn's taxonomy
4. SIMD

You will learn:

1. Overview of CPU architecture
2. Overview of GPU architecture
3. The SPMD programming model
4. Writing optimal SPMD code

SIMD vs SPMD

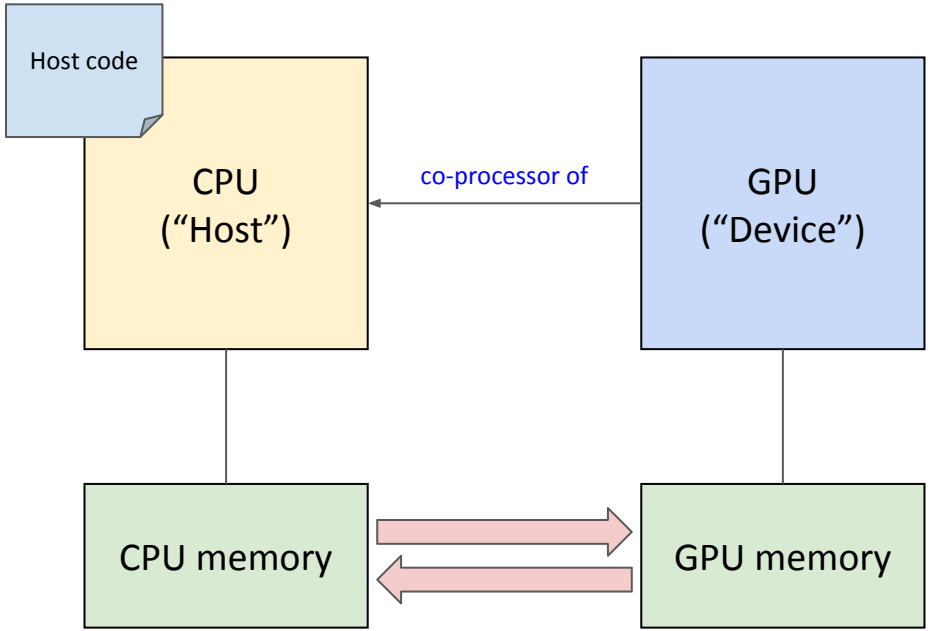


SPMD: Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data.

You can launch multiple threads, each using their respective SIMD lanes

SPMD is a parallel execution model and assumes multiple cooperating processors executing a program.

Programming on heterogeneous systems

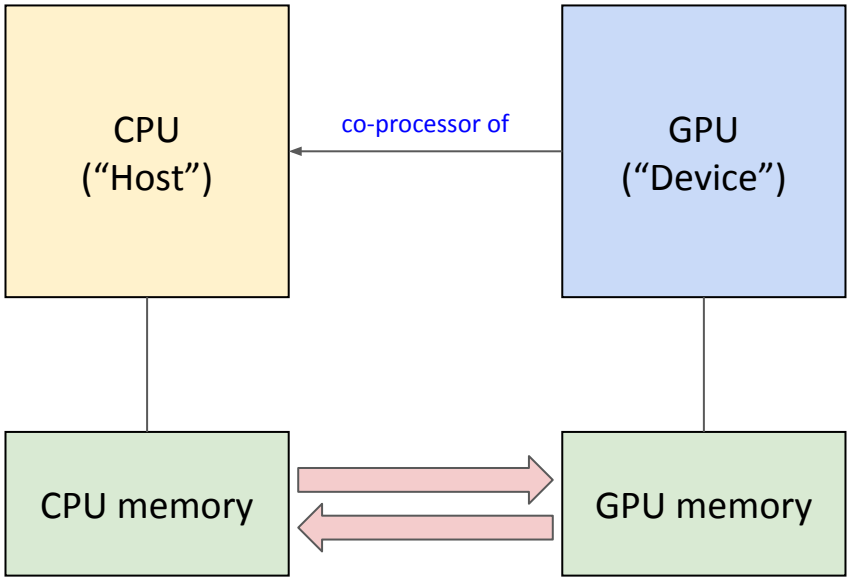


CPU vs GPU

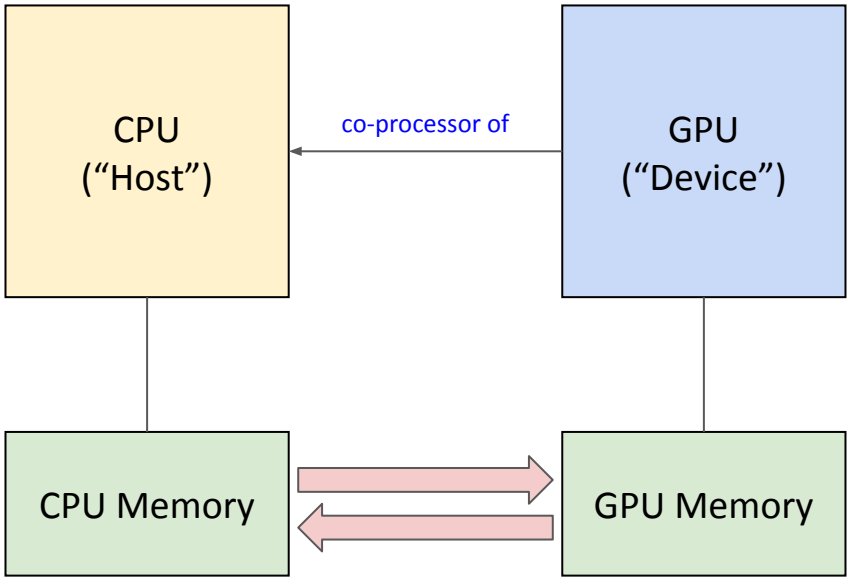
- Small number of large processors
- More control structures and less processing units
  - Can do more complex logic
  - Requires more power
- Optimise for latency
  - Minimising the time taken for one particular task
- Flexible programming model

- Large number of small processors
- Less control structures and more processing units
  - Can do less complex logic
  - Lower power consumption
- Optimised for throughput
  - Maximising the amount of work done per unit of time
- Restricted programming model

Programming on heterogeneous systems

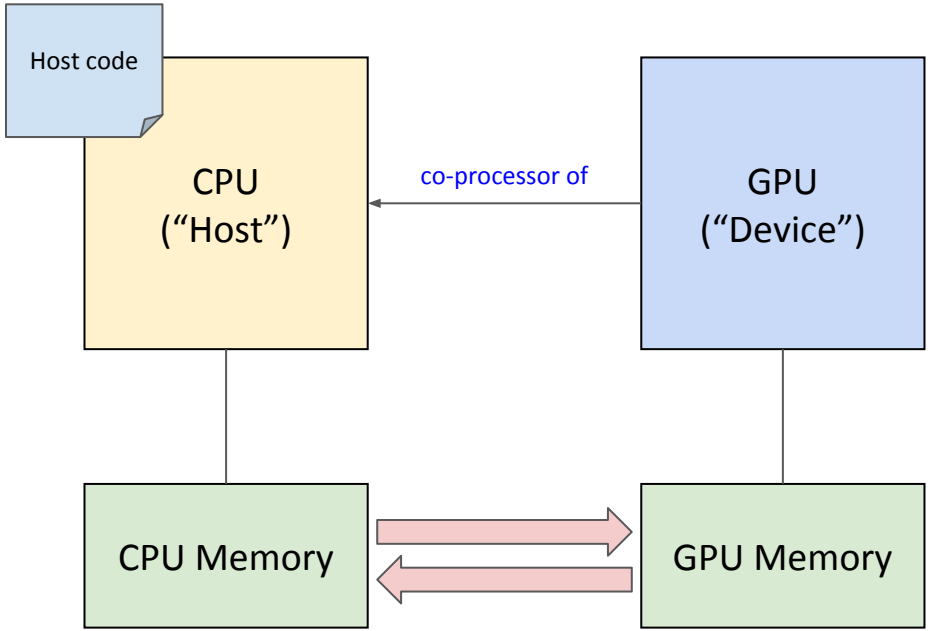


Programming on heterogeneous systems

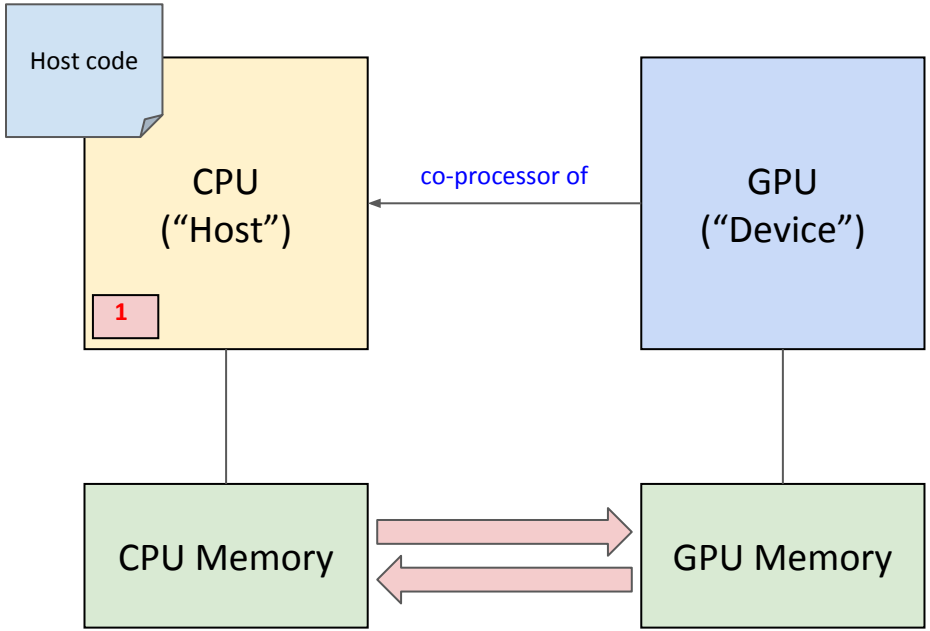


Explicit heterogeneous programming models can allow you to write SPMD code that will run on SIMD CPUs and GPUs

Programming on a SIMD CPU

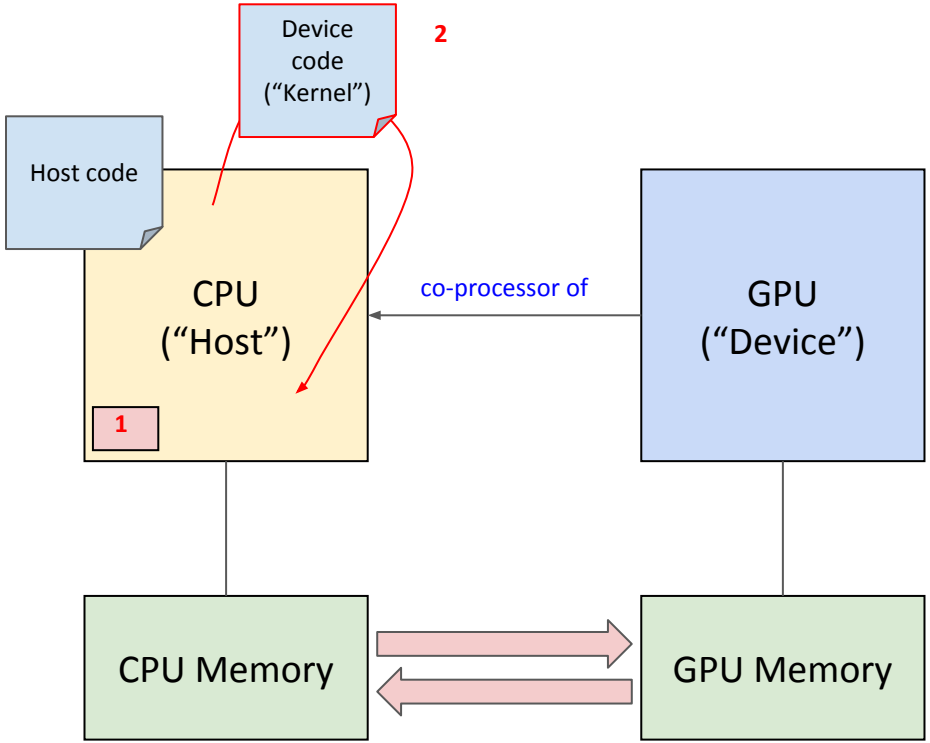


Programming on a SIMD CPU



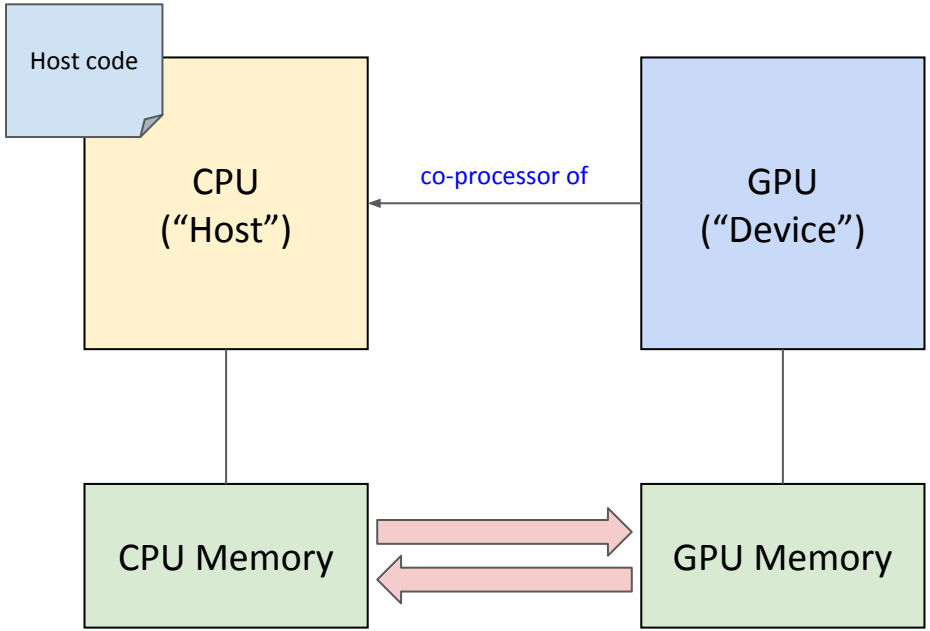
1. The CPU allocates memory on the CPU

Programming on a SIMD CPU

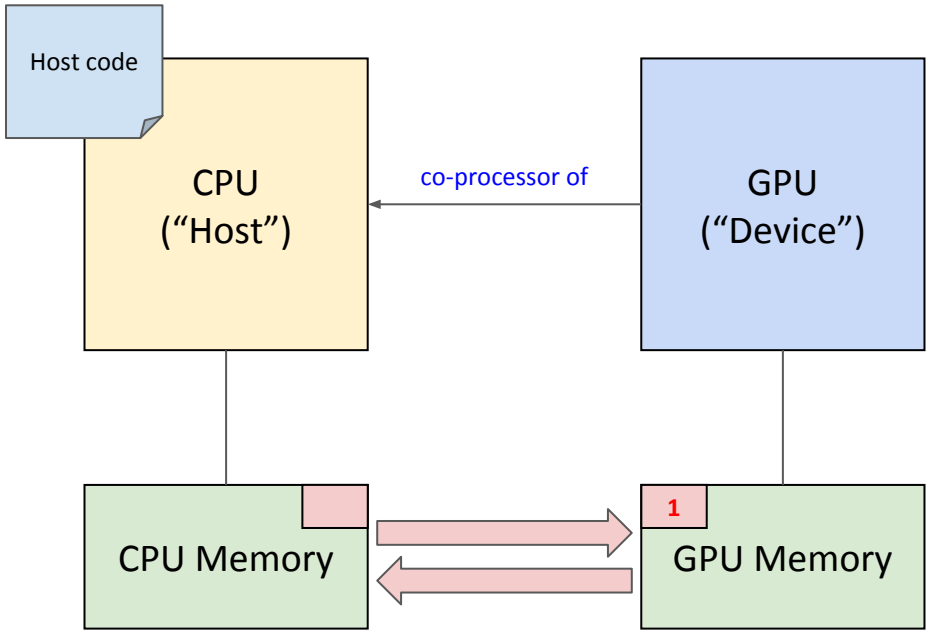


1. The CPU allocates memory on the CPU
2. The CPU executes a kernel using threads and SIMD instructions

Programming on a GPU

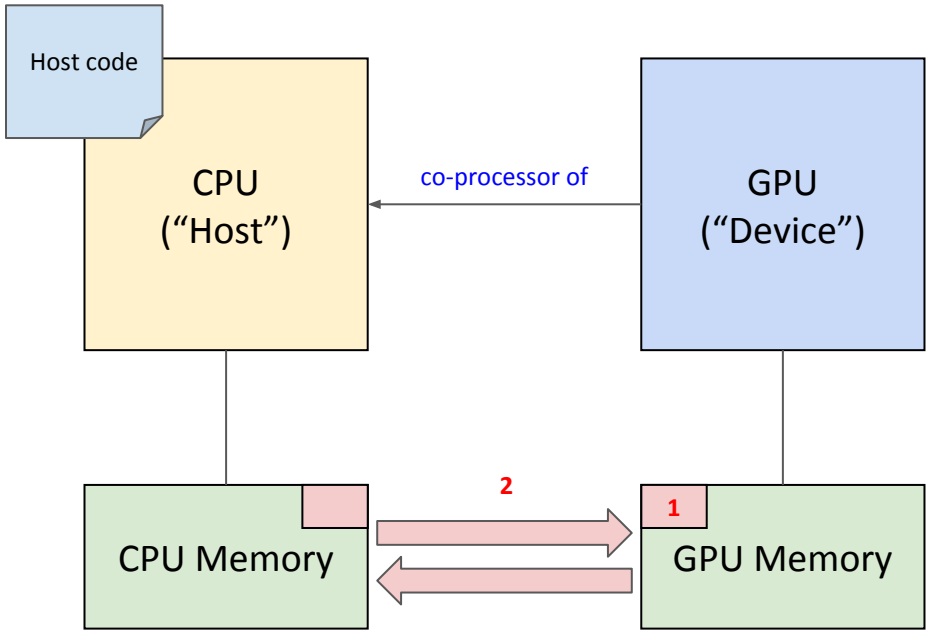


Programming on a GPU



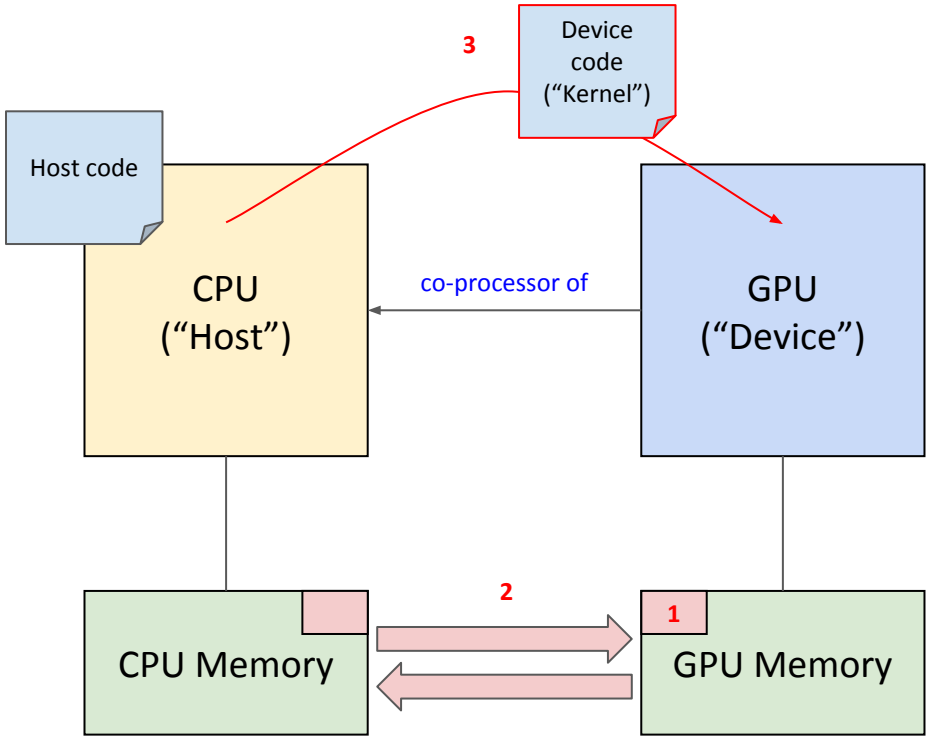
1. The CPU allocates memory on the GPU

Programming on a GPU



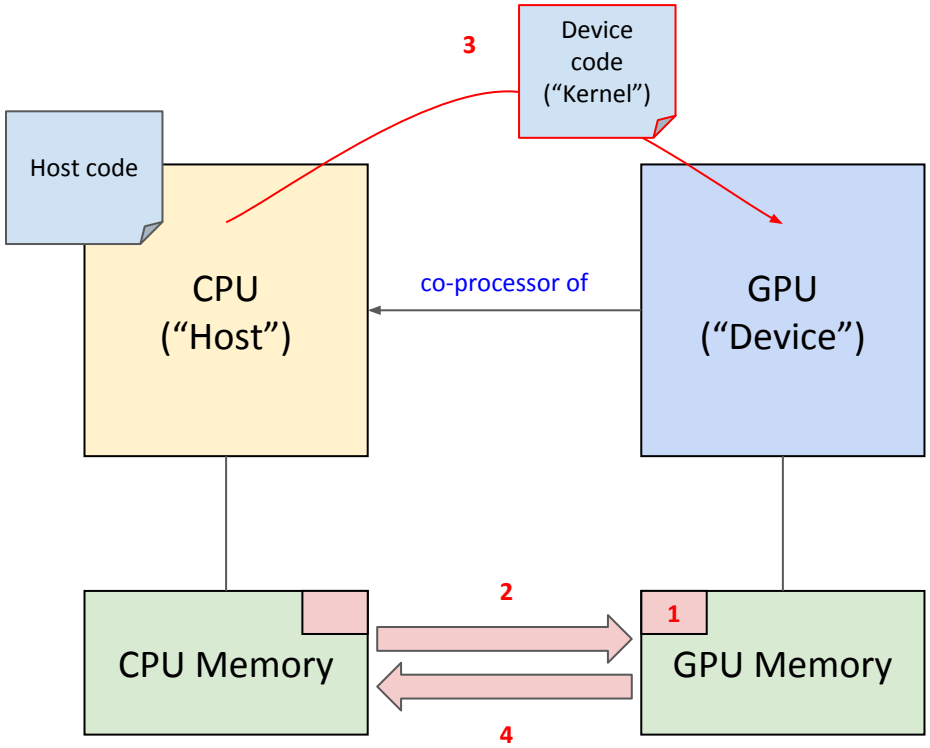
1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU

Programming on a GPU



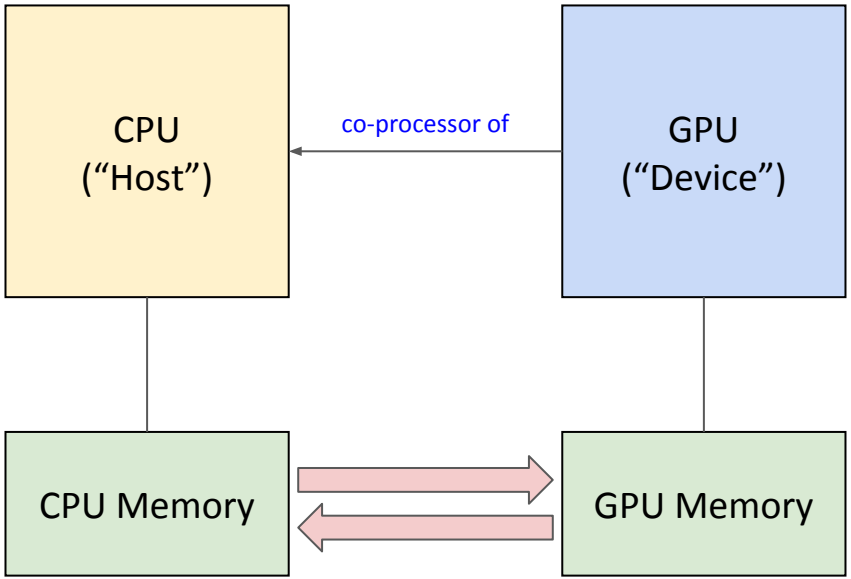
1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU

Programming on a GPU

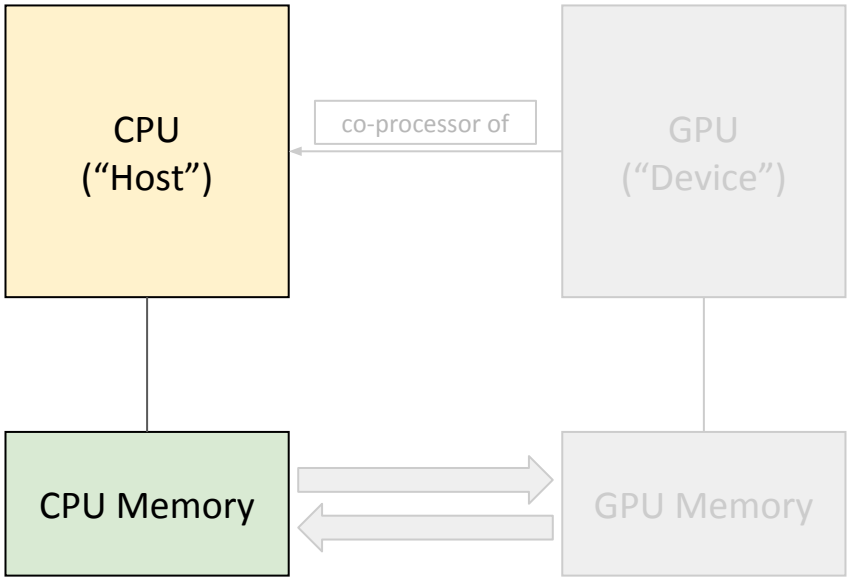


1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU
4. The CPU copies data to CPU from GPU

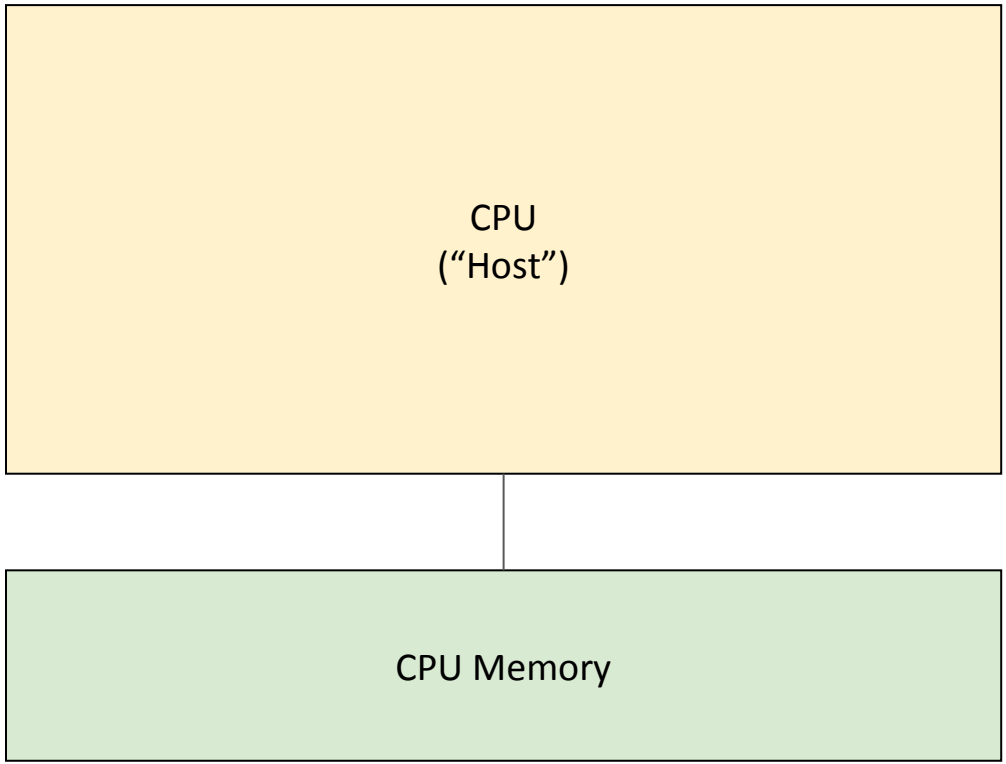
So let’s take a look now at each of these



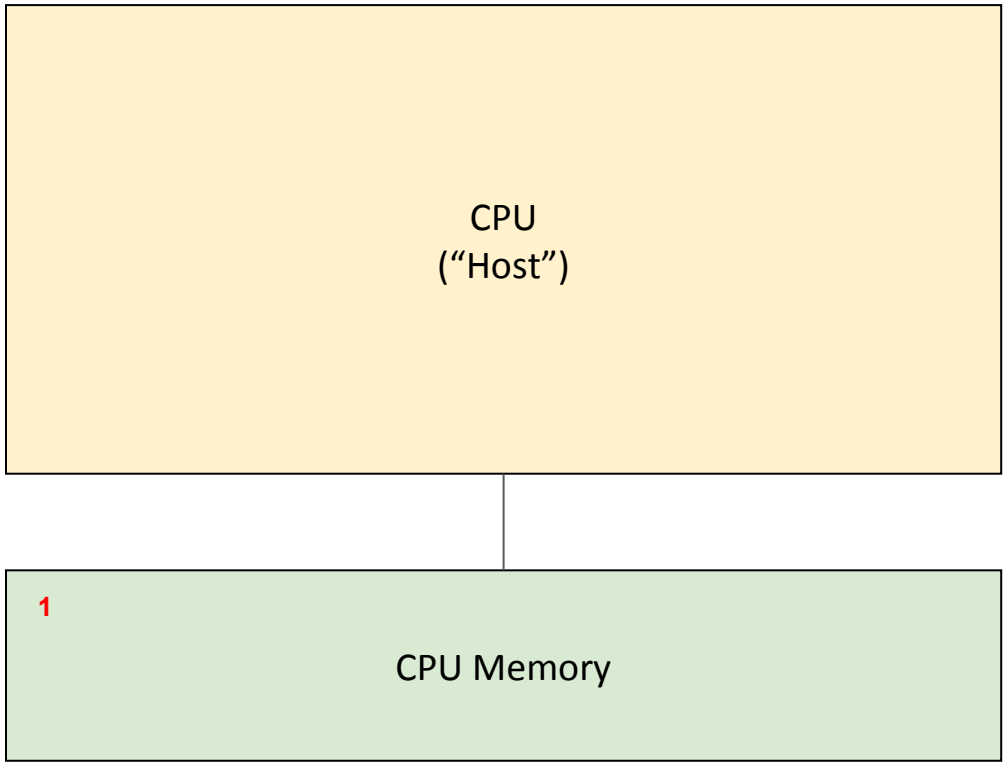
Let’s take a look at the CPU...



Let’s take a look at the CPU…

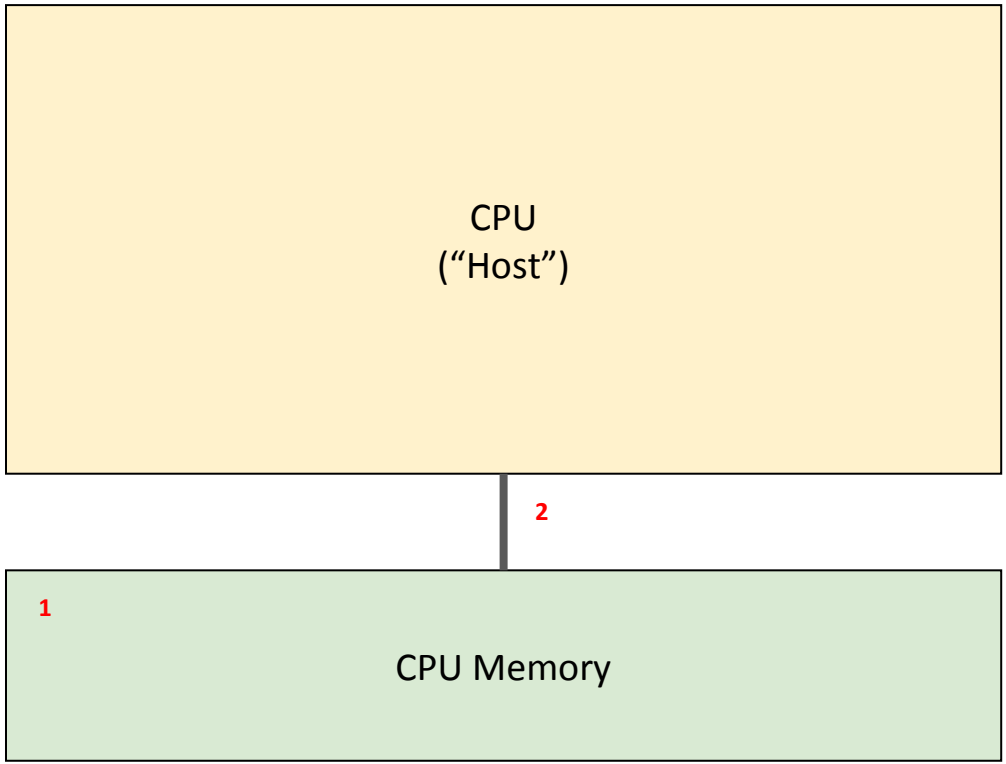


Let’s take a look at the CPU…



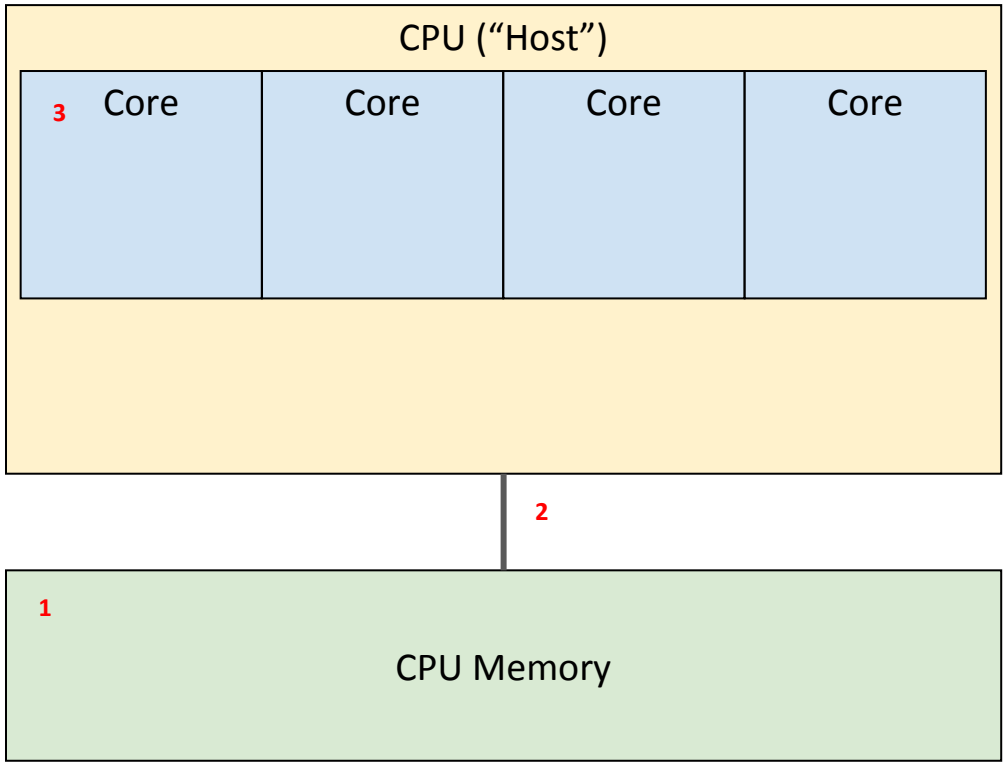
1. A CPU has a region of dedicated memory

Let’s take a look at the CPU…



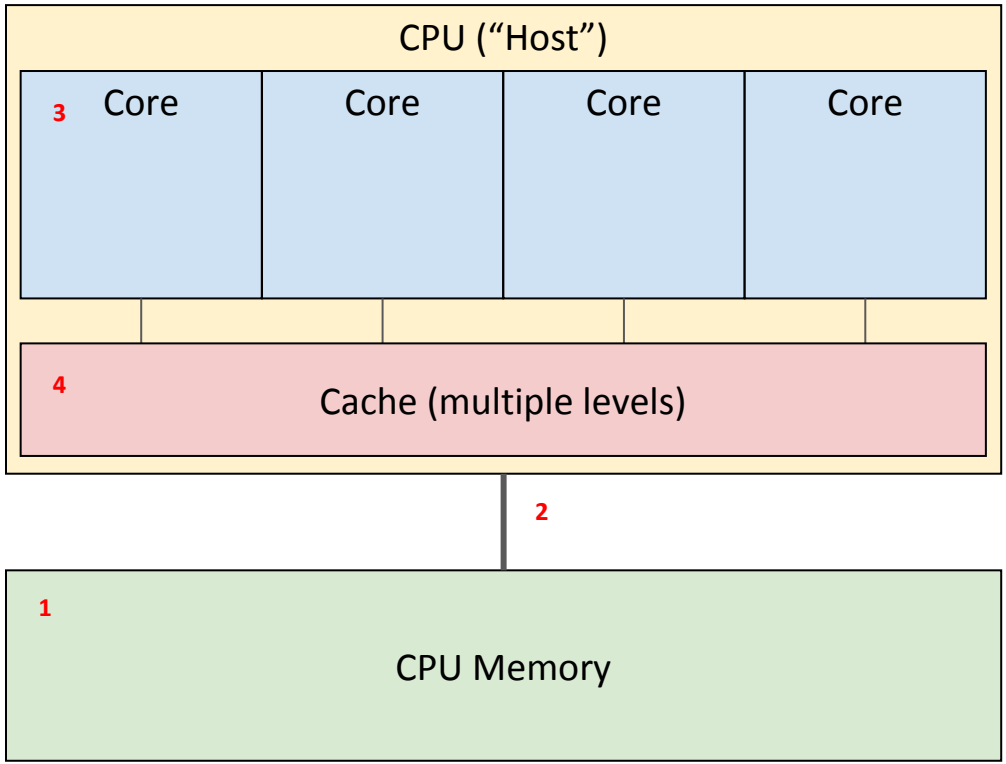
1. A CPU has a region of dedicated memory
2. CPU memory is connected to the CPU via a bus

Let’s take a look at the CPU…



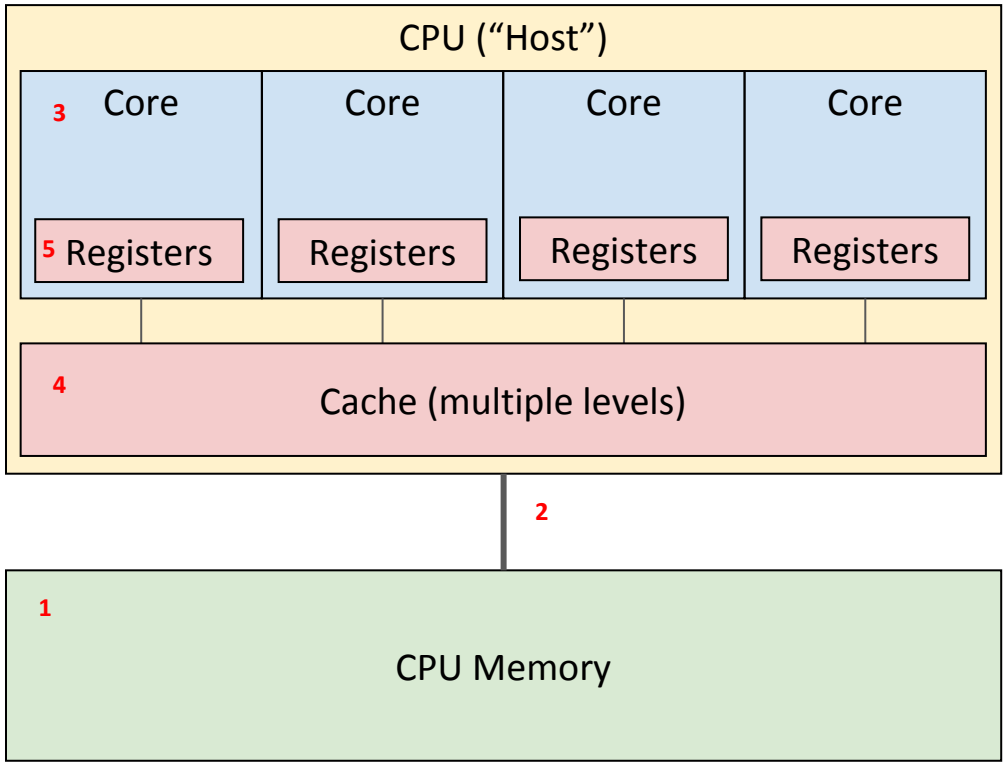
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores

Let’s take a look at the CPU…



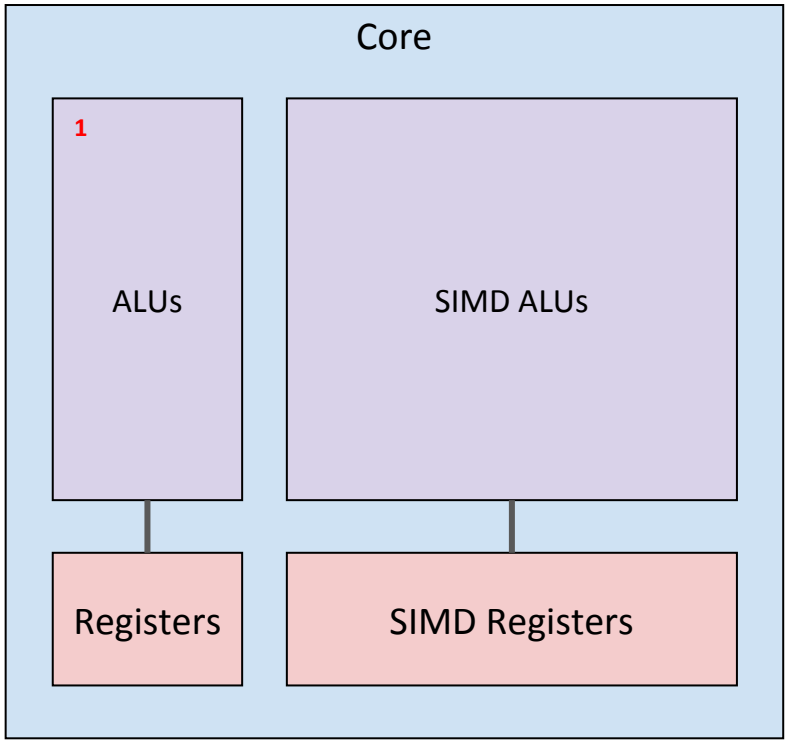
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores
4. A CPU has a number of caches of different levels

Let’s take a look at the CPU…



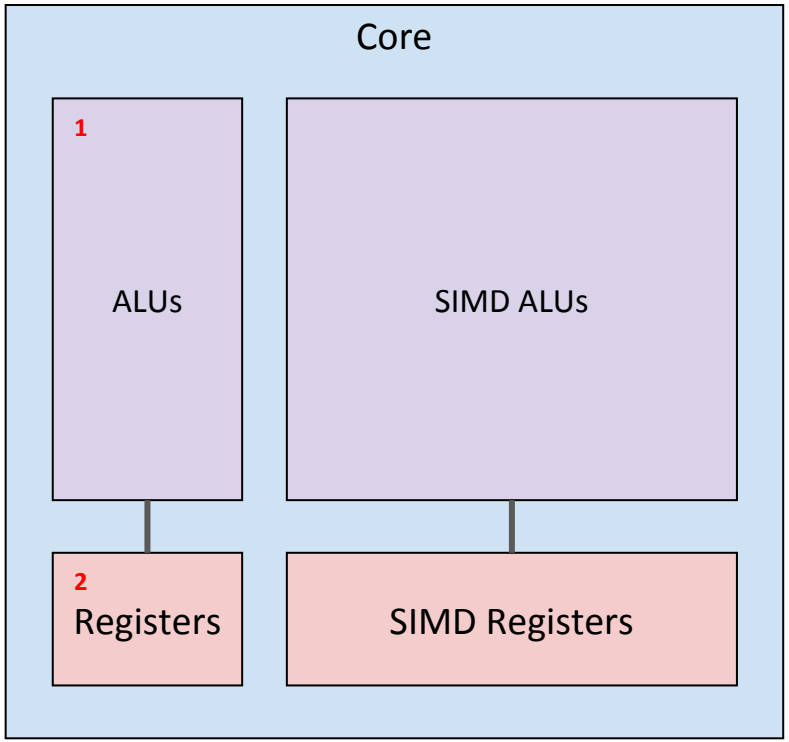
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores
4. A CPU has a number of caches of different levels
5. Each CPU core has dedicated registers

Inside a CPU core



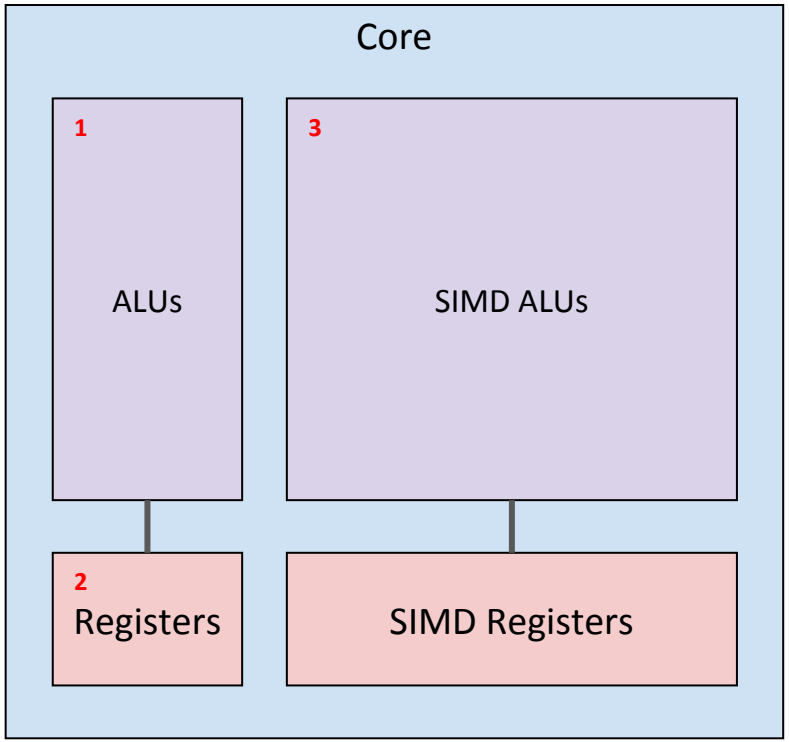
1. A CPU core you have a number of standard ALUs

Inside a CPU core



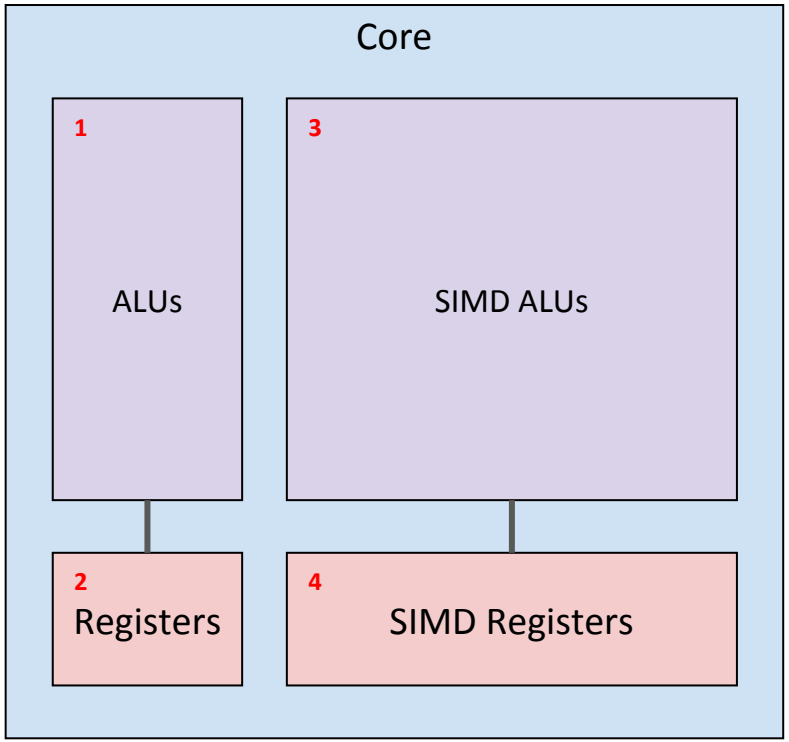
1. A CPU core you have a number of standard ALUs
2. These are connected to standard registers

Inside a CPU core



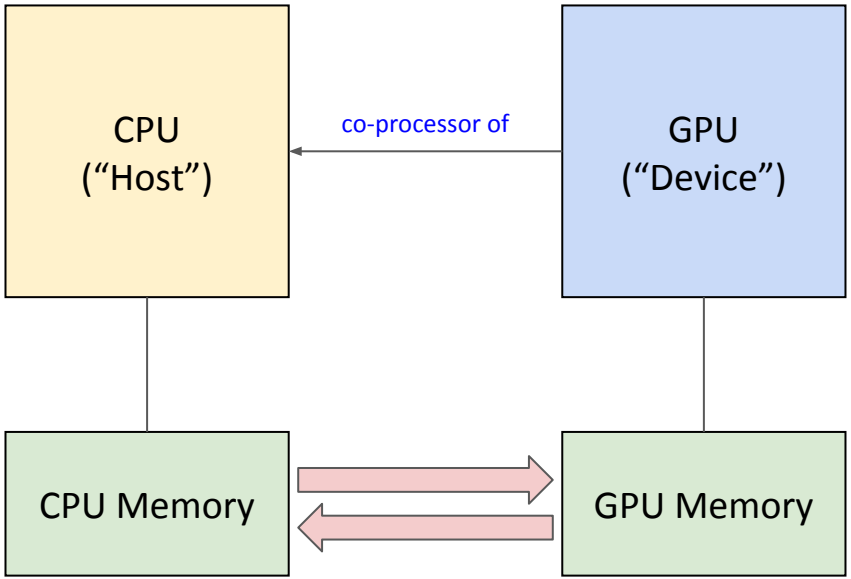
1. A CPU core you have a number of standard ALUs
2. These are connected to standard registers
3. A CPU core also have SIMD ALUs

Inside a CPU core

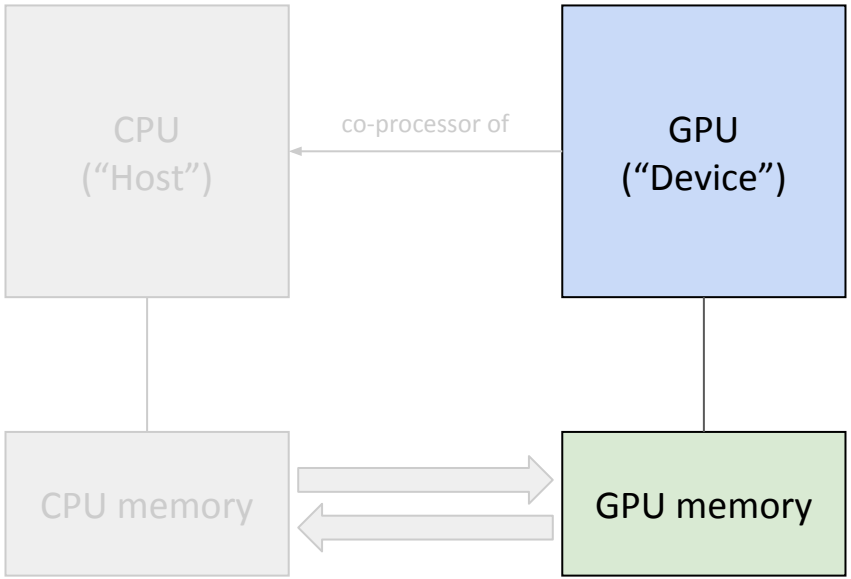


1. A CPU core you have a number of standard ALUs
2. These are connected to standard registers
3. A CPU core also have SIMD ALUs
4. These are connected to SIMD registers

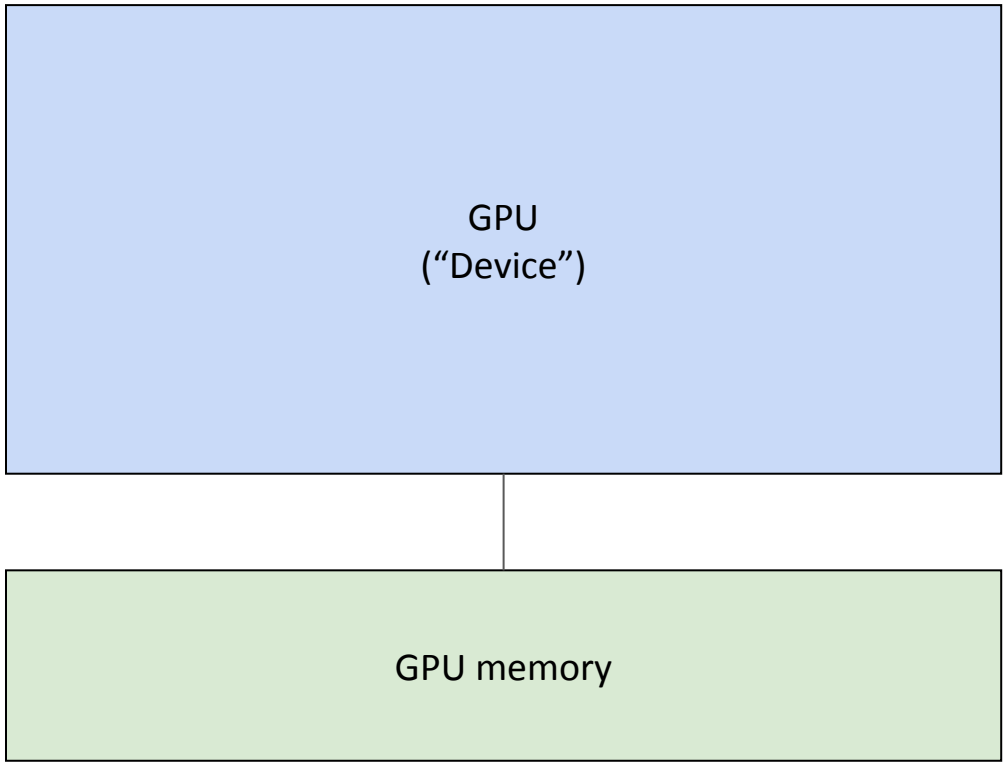
Now let’s take a look at the GPU...



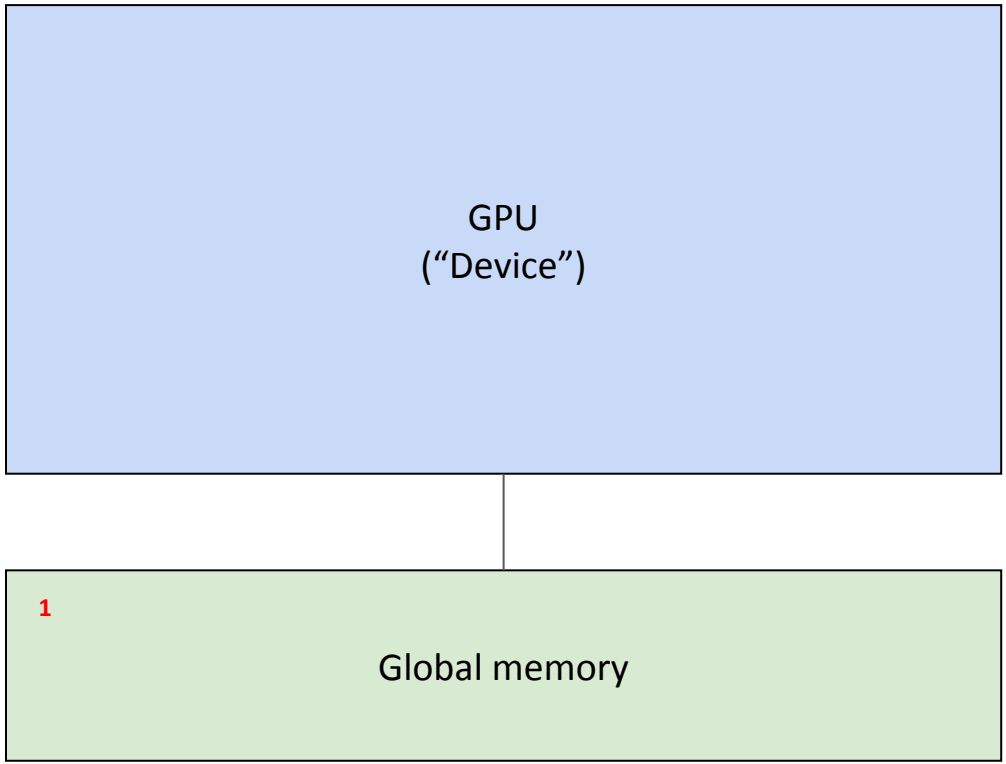
Now let’s take a look at the GPU...



Now let’s take a look at the GPU...

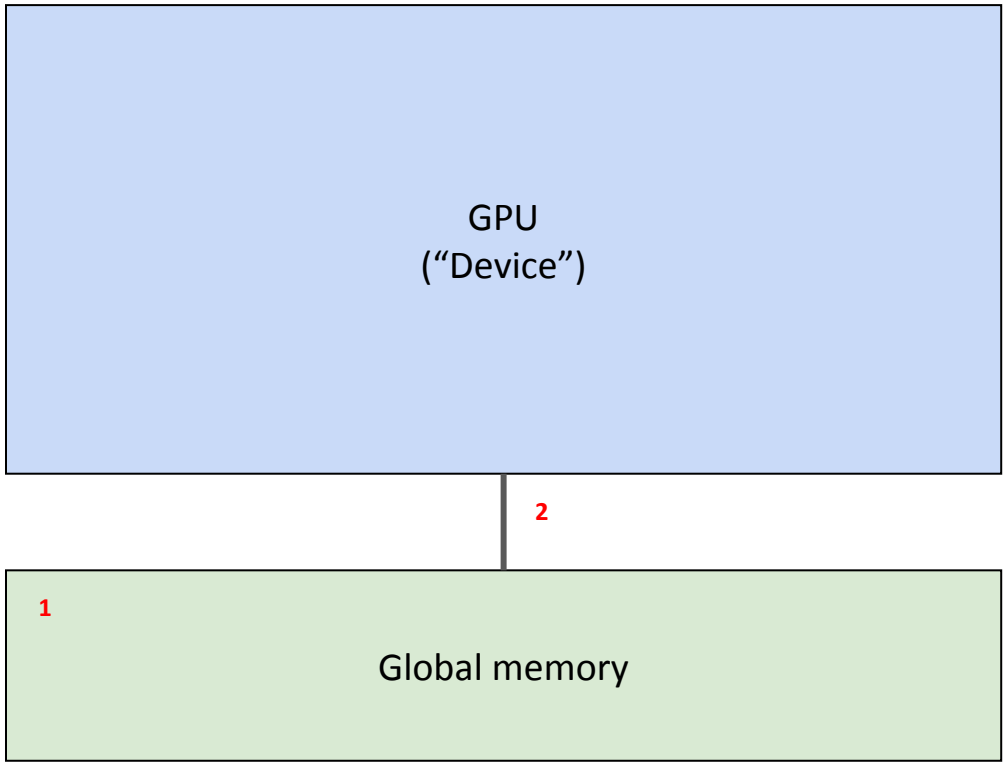


Now let’s take a look at the GPU...



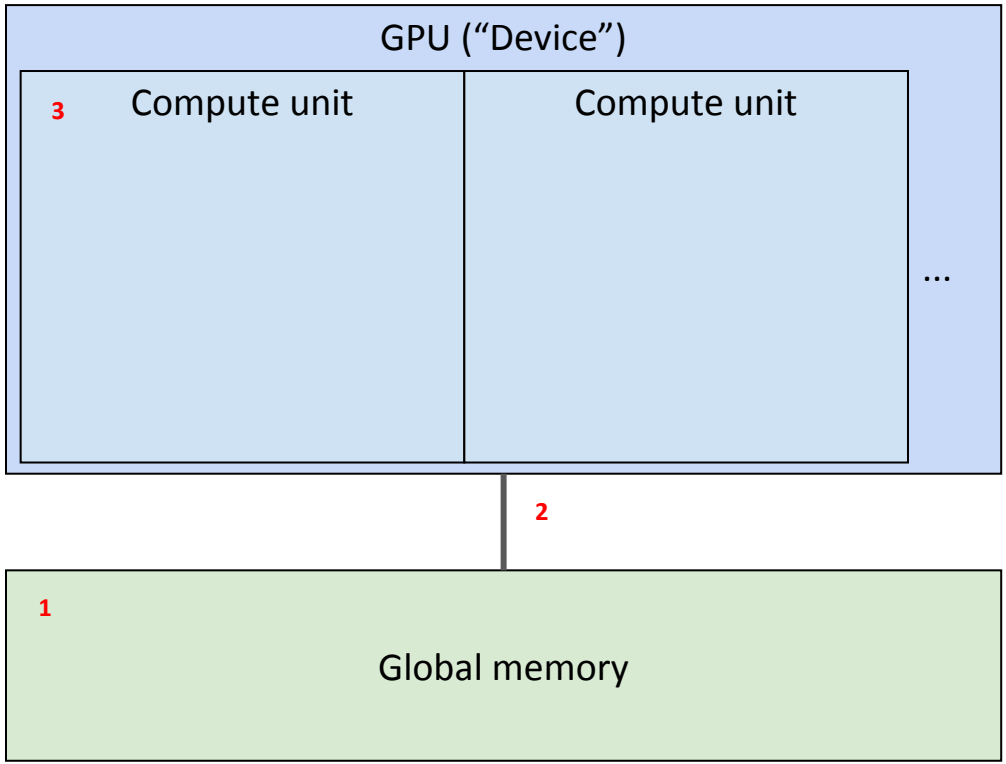
1. A GPU has a region of dedicated global memory

Now let’s take a look at the GPU...



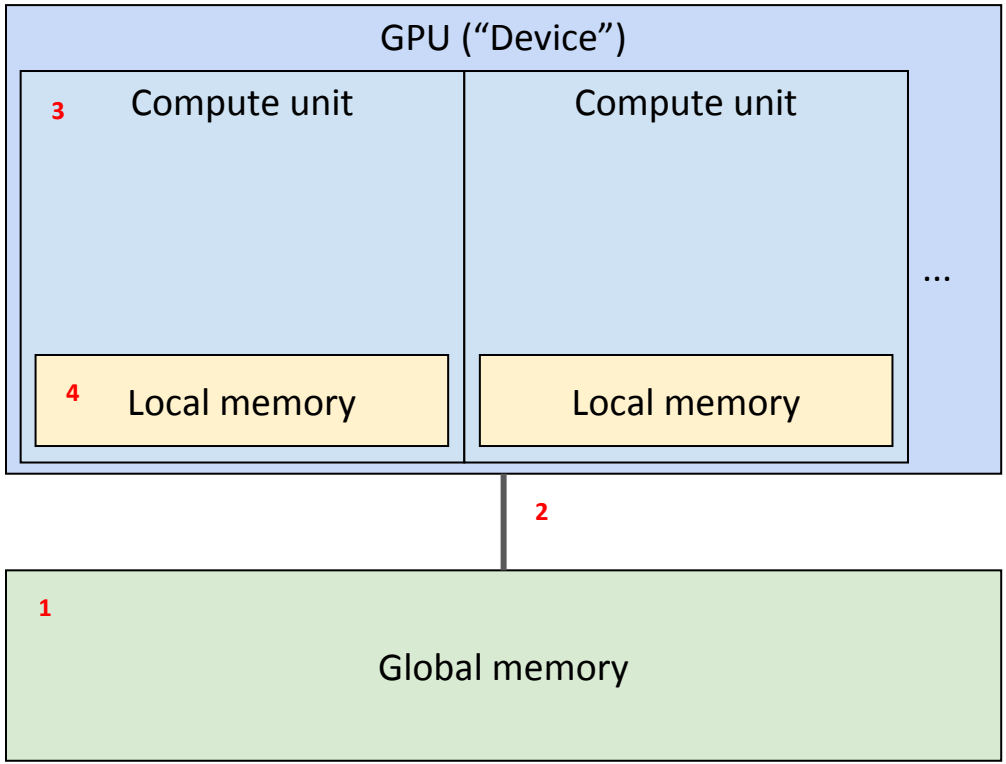
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus

Now let’s take a look at the GPU...



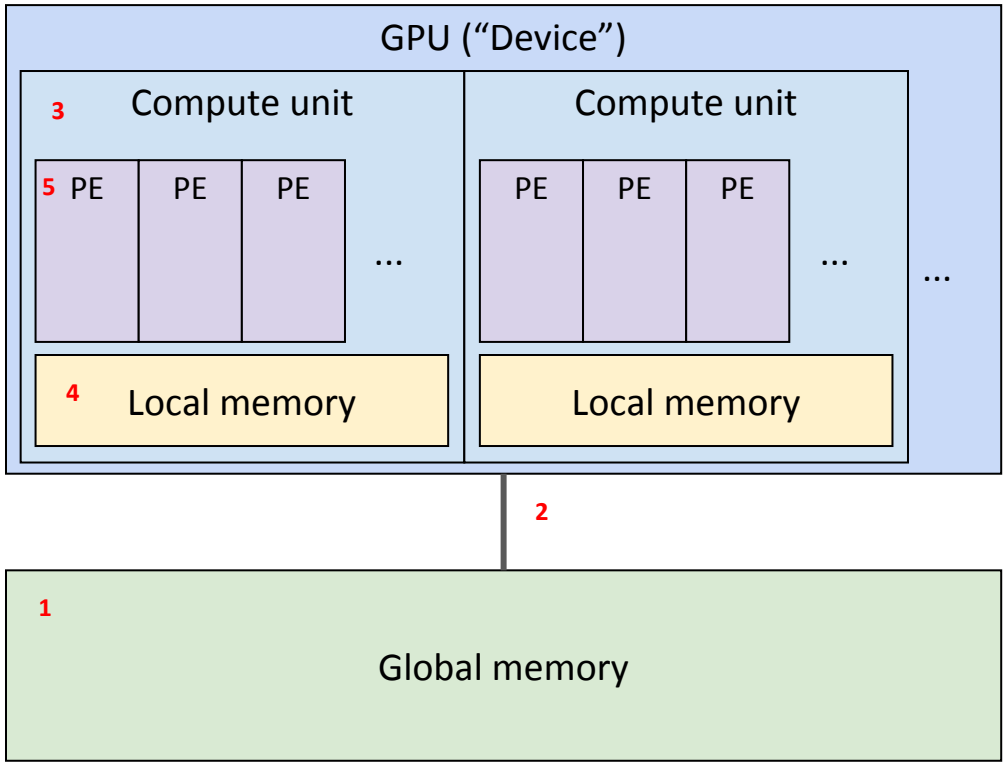
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units

Now let's take a look at the GPU...



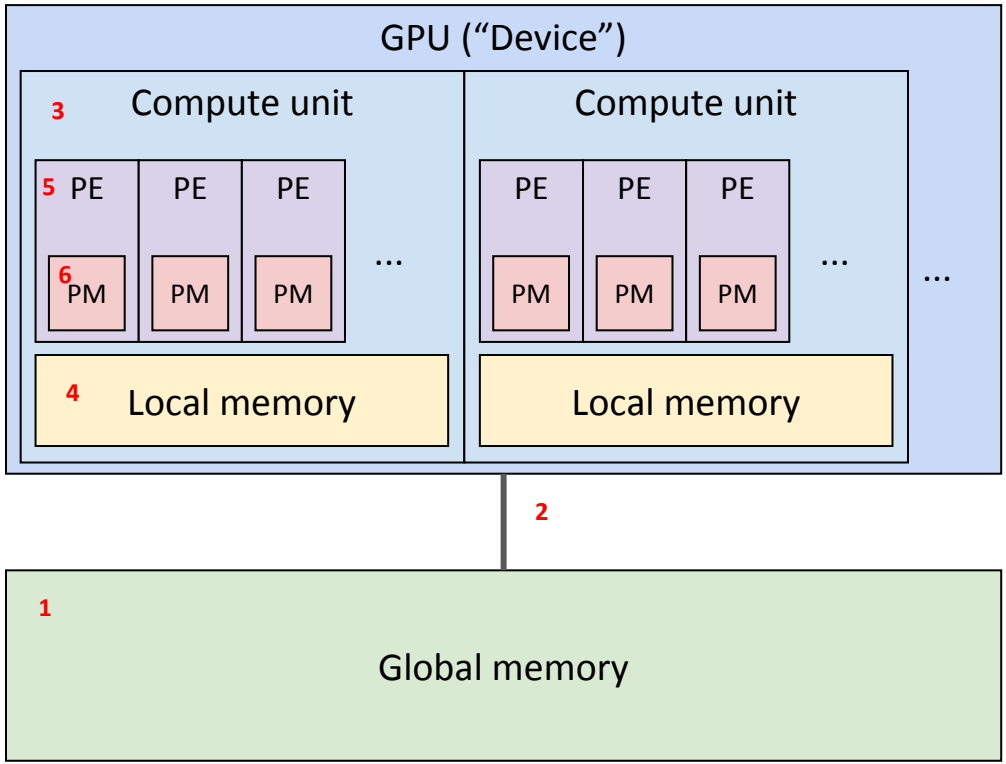
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory

Now let's take a look at the GPU...



1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements

Now let's take a look at the GPU...



1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements
6. Each processing element has dedicated private memory

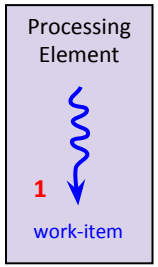
The SPMD programming model

Many heterogeneous languages and models like SYCL use an SPMD programming model

This model can be applied both to:

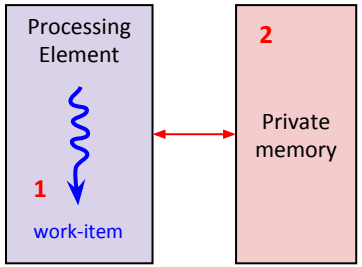
- SIMD CPUs
- GPUs
- Many other heterogeneous devices

The SPMD programming model



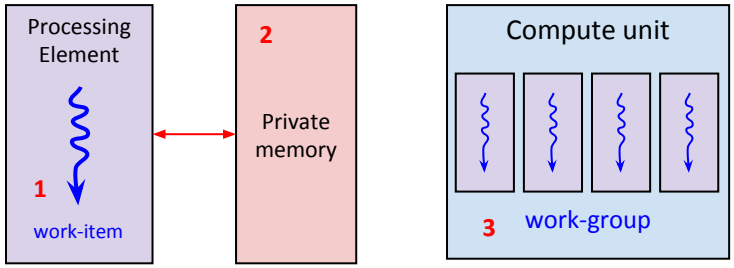
1. A processing element executes a single work-item

The SPMD programming model



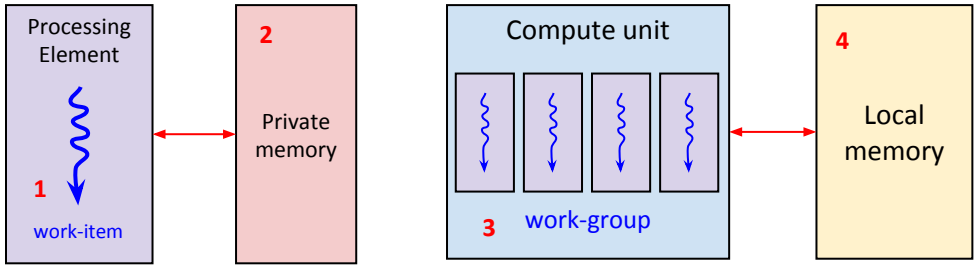
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element

The SPMD programming model



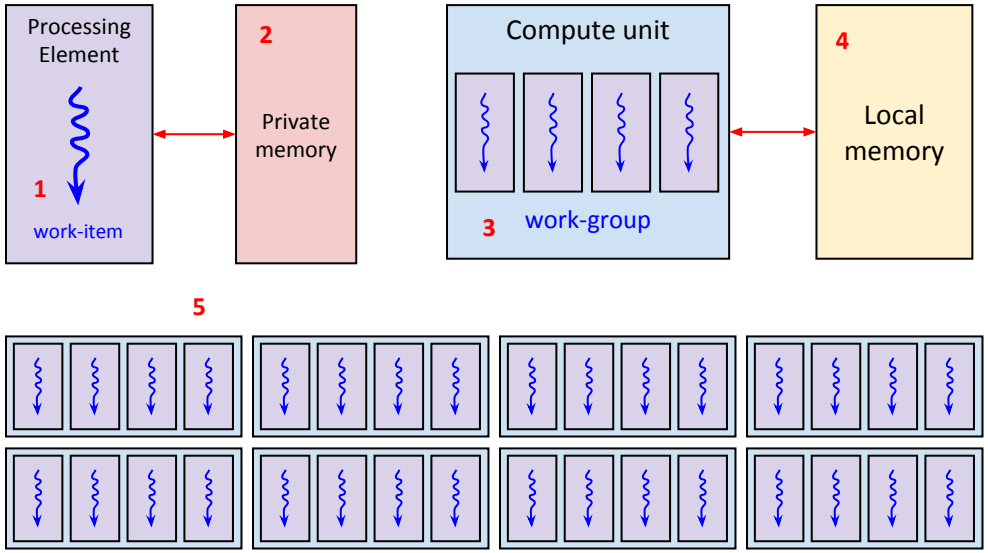
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit

The SPMD programming model



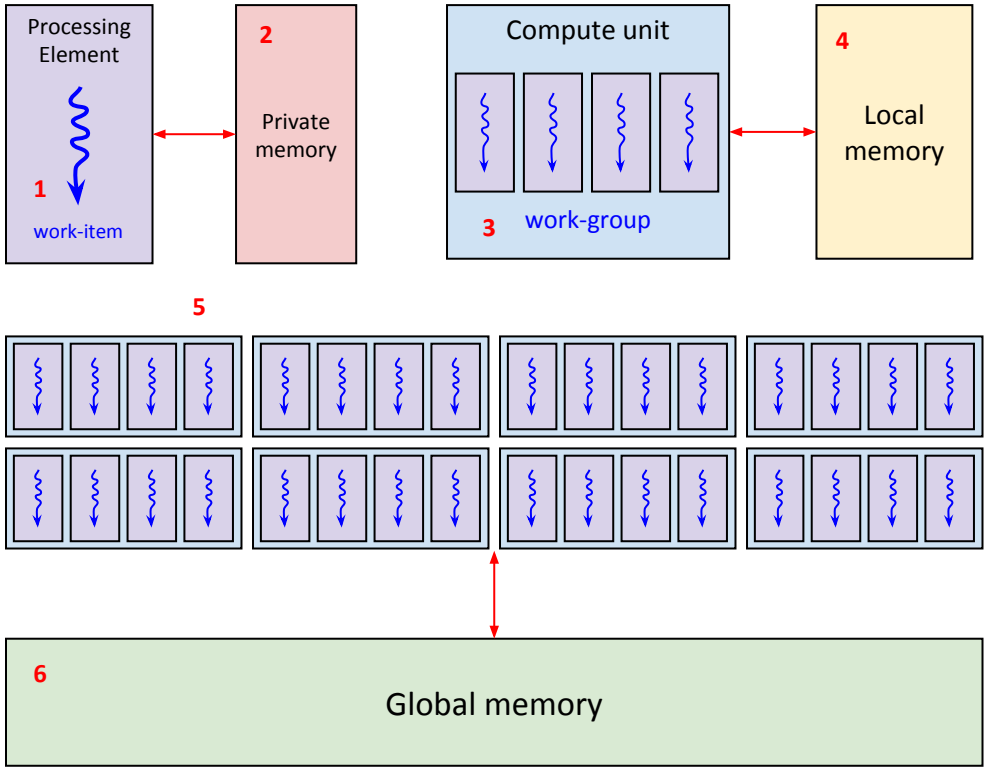
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access local memory, a dedicated memory region for each compute unit

The SPMD programming model



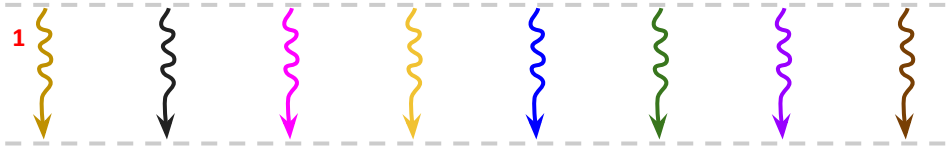
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access local memory, a dedicated memory region for each compute unit
5. A GPU executes multiple work-groups

The SPMD programming model



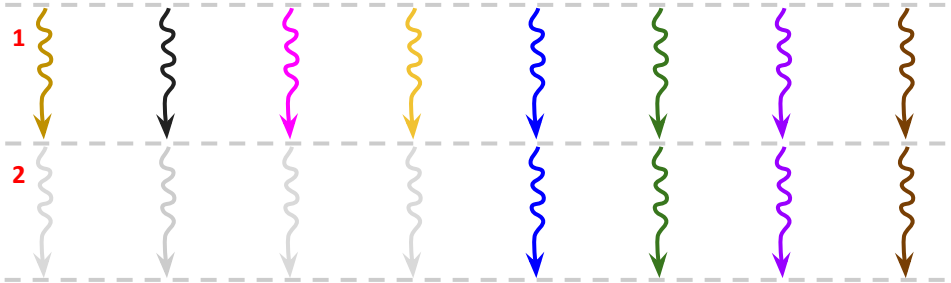
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access local memory, a dedicated memory region for each compute unit
5. A GPU executes multiple work-groups
6. Each work-item can access global memory, a single memory region available to all processing elements on the GPU

Synchronization



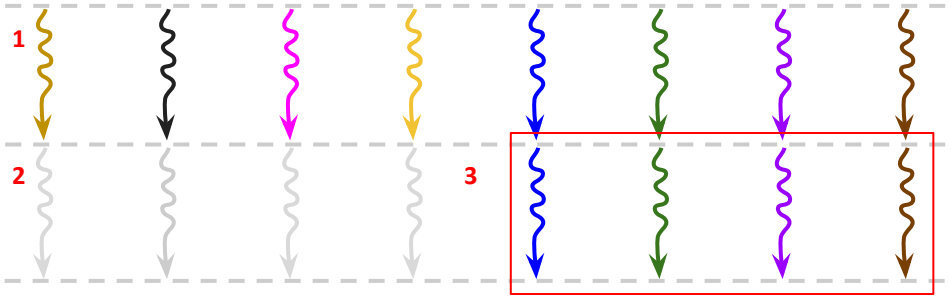
1. Multiple work-items will execute concurrently

Synchronization



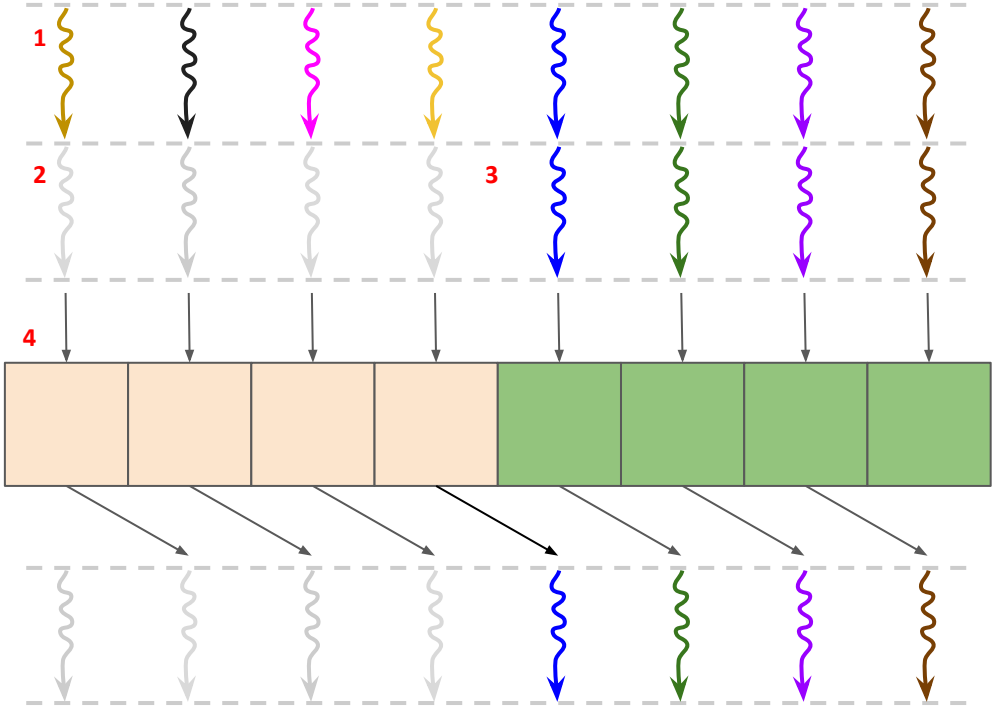
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly

Synchronization



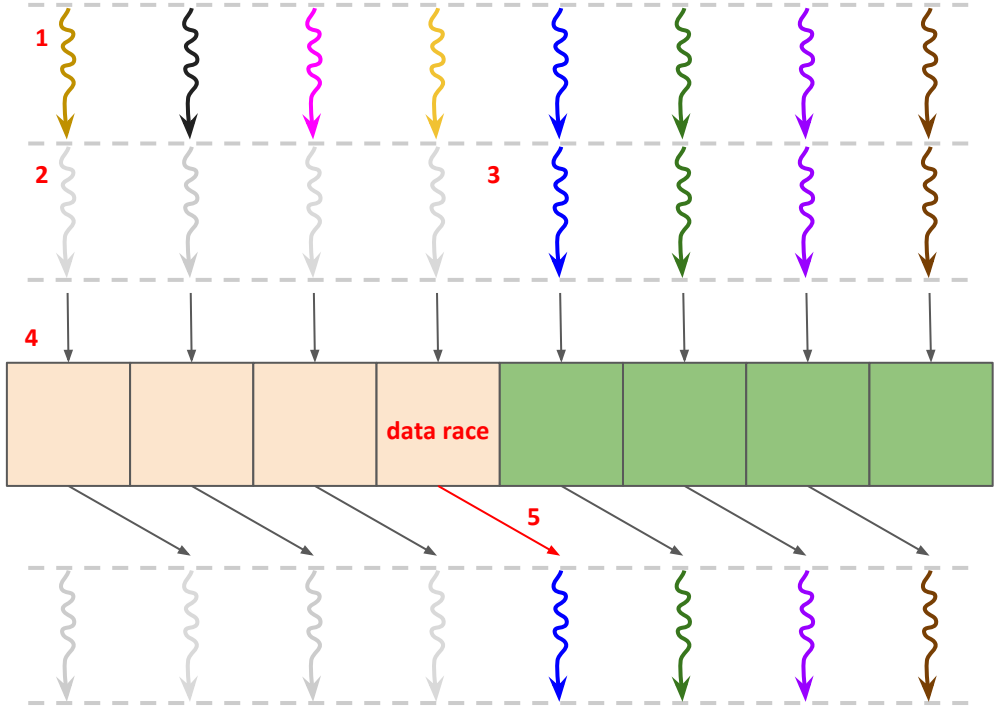
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs and SIMD lanes do execute a number of work-items uniformly (lock-step), but that number is unspecified

Synchronization



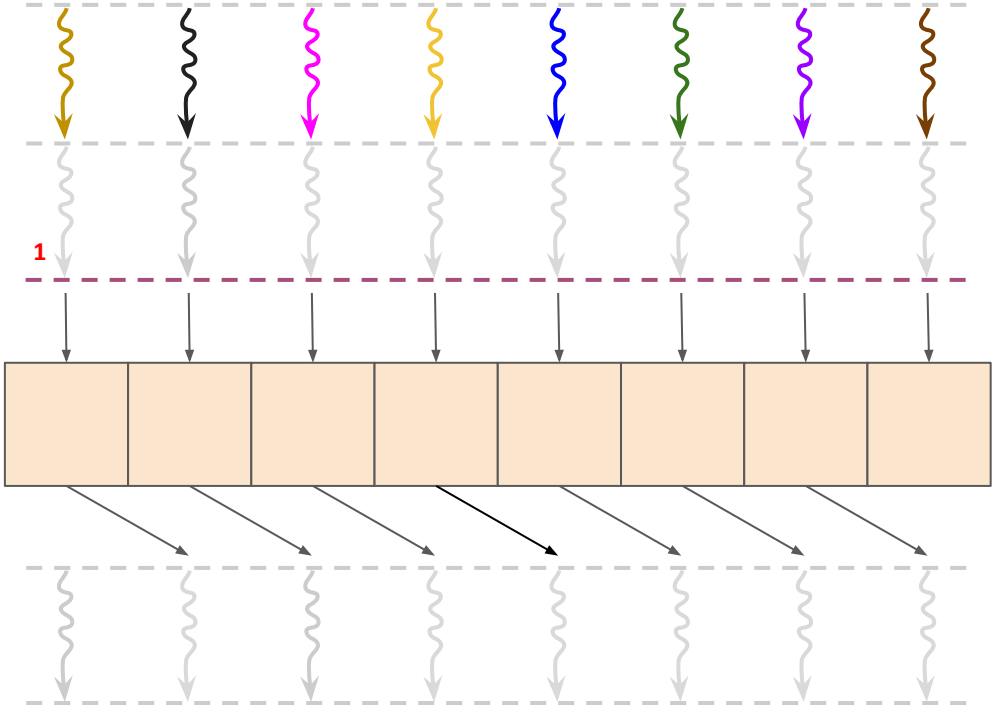
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs and SIMD lanes do execute a number of work-items uniformly (lock-step), but that number is unspecified
4. A work-item can share results with other work-items via local and global memory

Synchronization



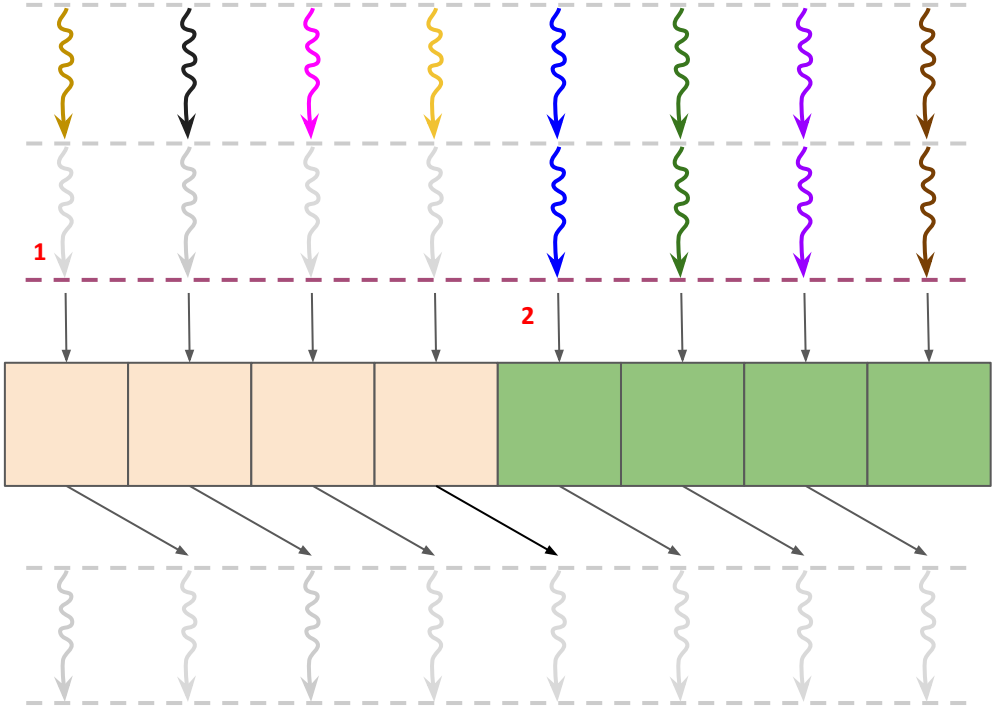
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs and SIMD lanes do execute a number of work-items uniformly (lock-step), but that number is unspecified
4. A work-item can share results with other work-items via local and global memory
5. However this means that it's possible for a work-item to read a result that hasn't yet been written to, creating a data race

Work-group barriers



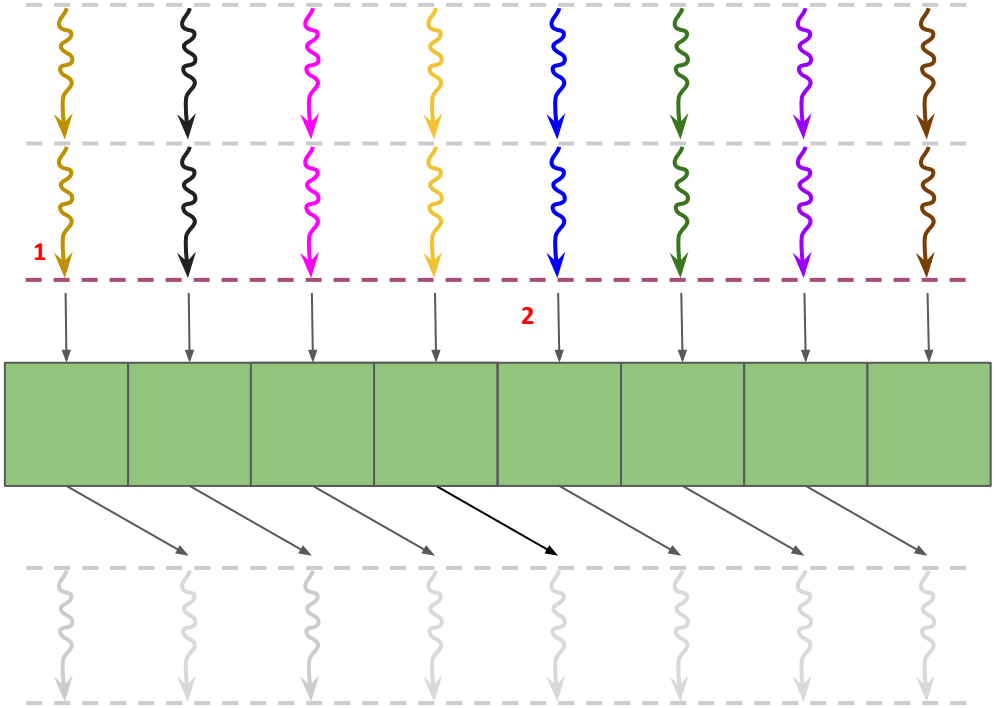
1. This problem can be solved by a synchronisation primitive called a work-group barrier

Work-group barriers



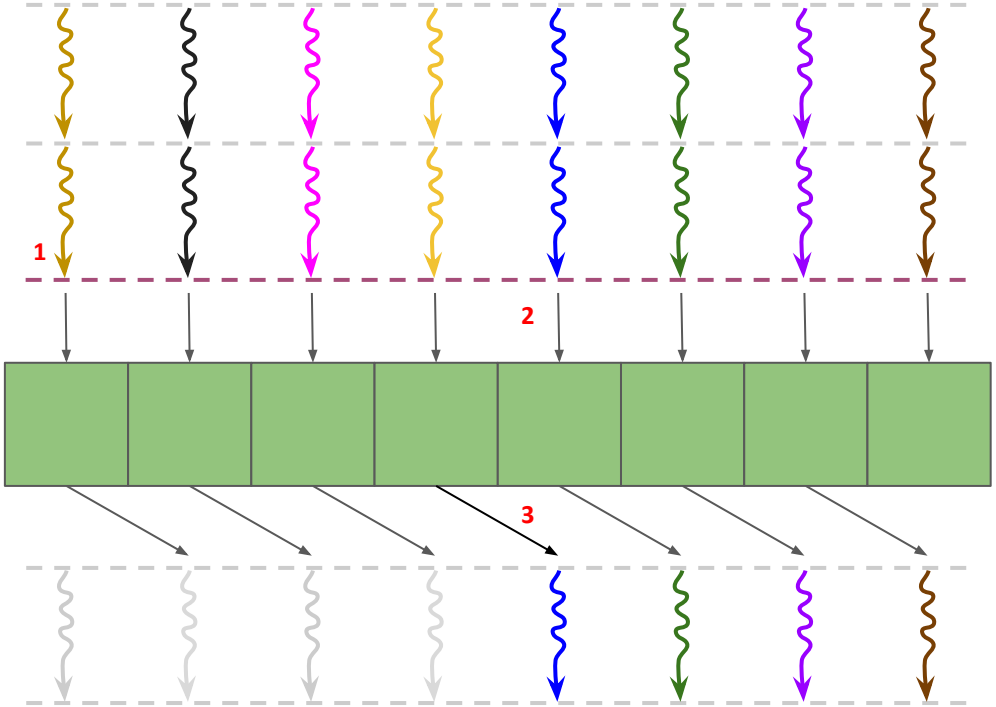
1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point

Work-group barriers



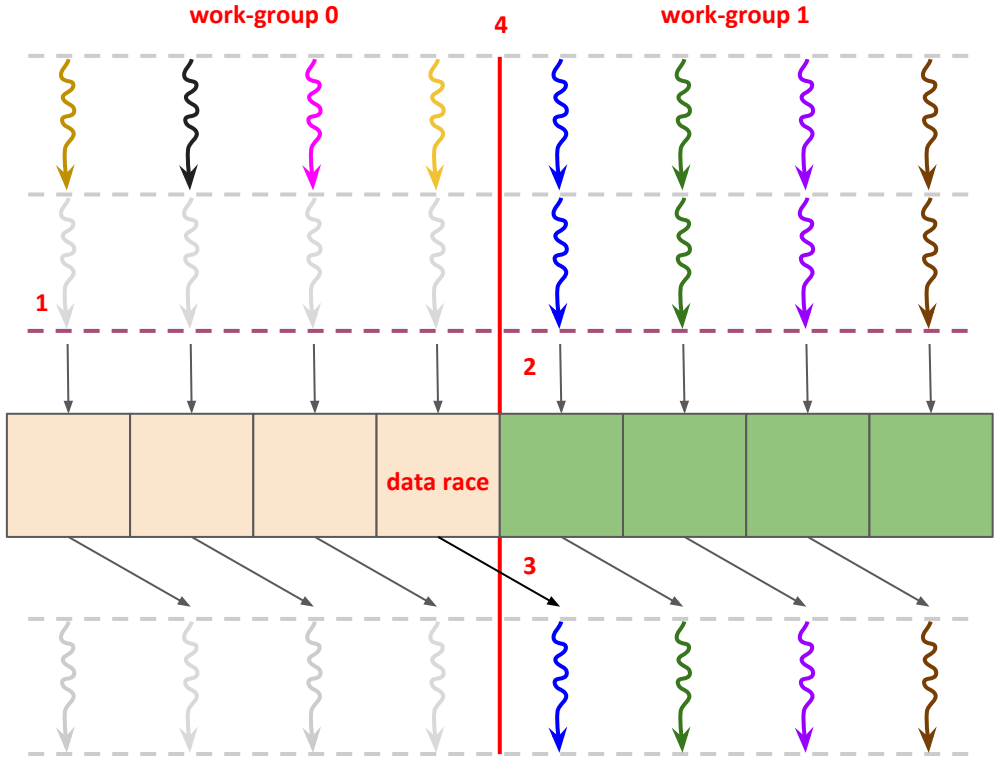
1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point
3.

Work-group barriers



1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point
3. So now you can be sure that all of the results that you want to read from have been written to

Work-group barriers

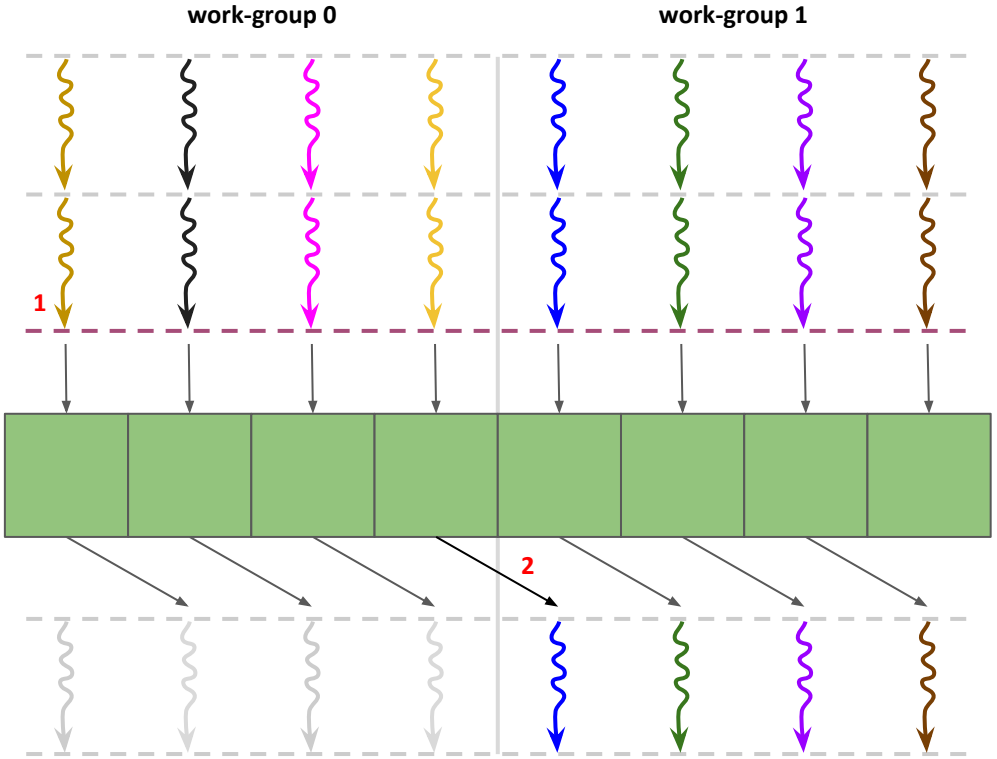


1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point
3. So now you can be sure that all of the results that you want to read from have been written to
4. However this does not apply across work-group boundaries, and you have a data rance again

Kernel barriers

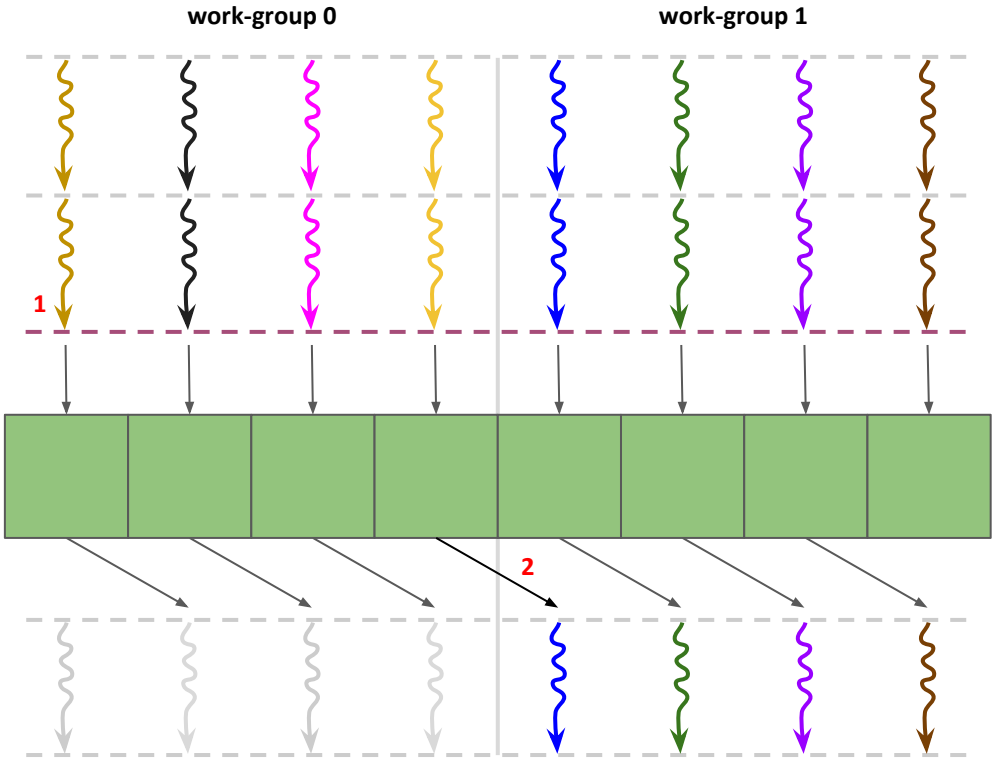
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)

Kernel barriers



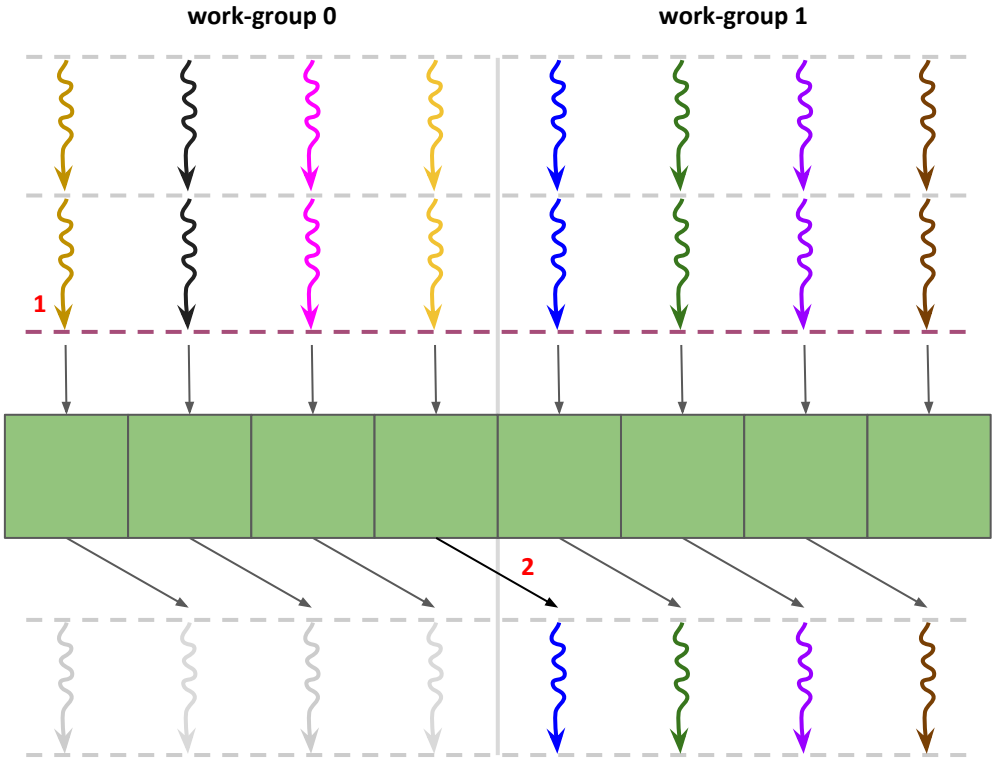
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)
2. Again you can be sure that all of the results that you want to read from have been written to

Kernel barriers



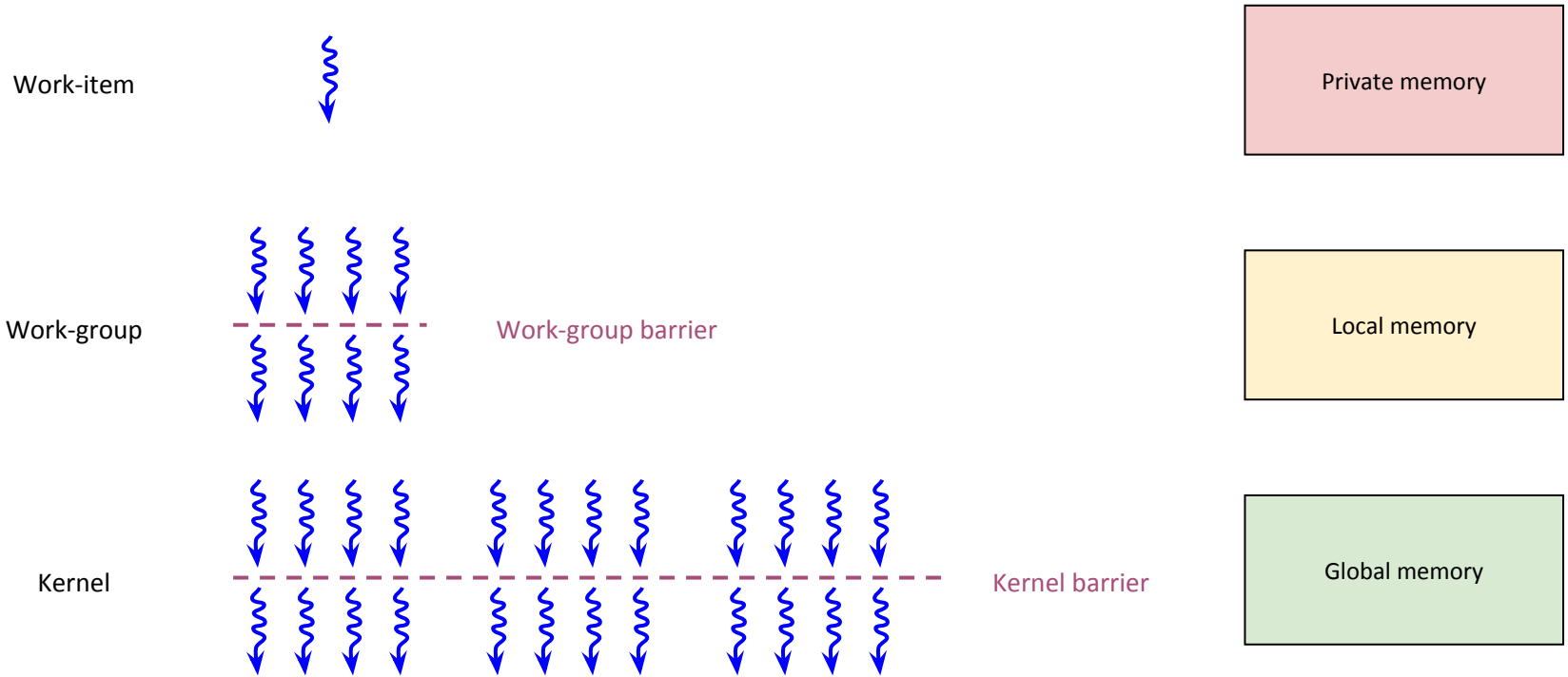
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)
2. Again you can be sure that all of the results that you want to read from have been written to
3. However kernel barriers have a higher overhead as they require you to launch another kernel

Kernel barriers



1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)
2. Again you can be sure that all of the results that you want to read from have been written to
3. However kernel barriers have a higher overhead as they require you to launch another kernel
4. And kernel barriers require results to be stored in global memory, local memory is not persistent across kernels

Summary of SPMD programming model



What is SPMD good at?

SPMD execution is very efficient at launching a large number of work-items

- Unlike a task parallelism where launching threads is expensive, SPMD launches thousands of work-items

SPMD is optimised for throughput

- You’re not getting the full benefit of a GPU or SIMD CPU unless you are using as many work-items as possible

What is SPMD bad at?

SPMD execution is bad at divergent control flow

- Due to lock-step execution, divergent control flow can be very inefficient

GPUs also have some restrictions in what you can do within a kernel

- You cannot use dynamic allocation (i.e. non-placement new)
- You cannot use recursion
- You cannot use function pointers or virtual functions

How do you write an SPMD program?

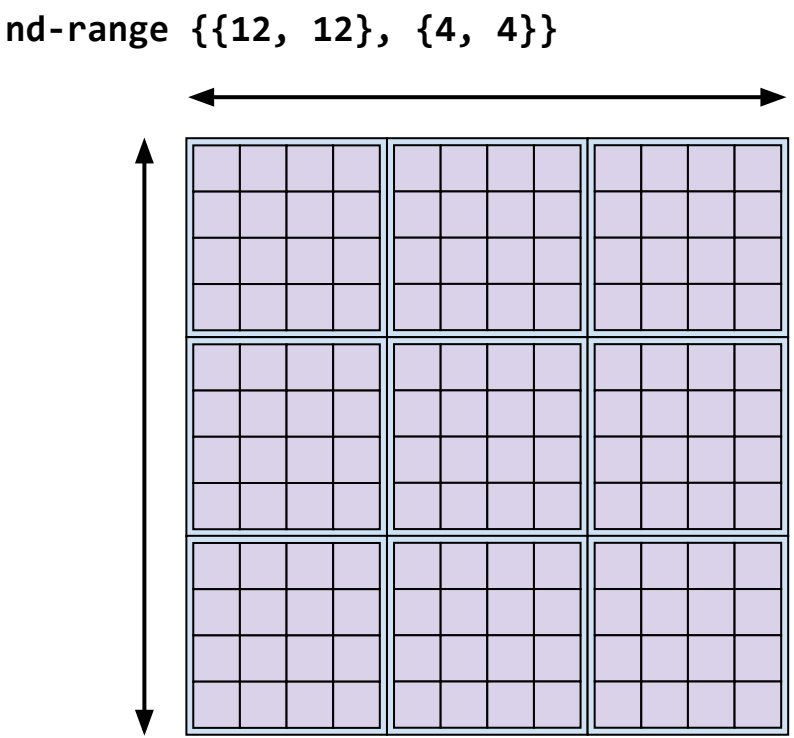
Single instruction single data (SISD)

```
void calc(int *in, int *out) {
  for (int i = 0; i < 1024; i++) {
    out[i] = in[i] * in[i];
  }
}



calc(in, out);
```

Single program multiple data (SPMD)

```
void calc(int *in, int *out, int id) {
  out[id] = in[id] * in[id];


}


/* specify degree of parallelism */
parallel_for(calc, in, out, 1024);
```

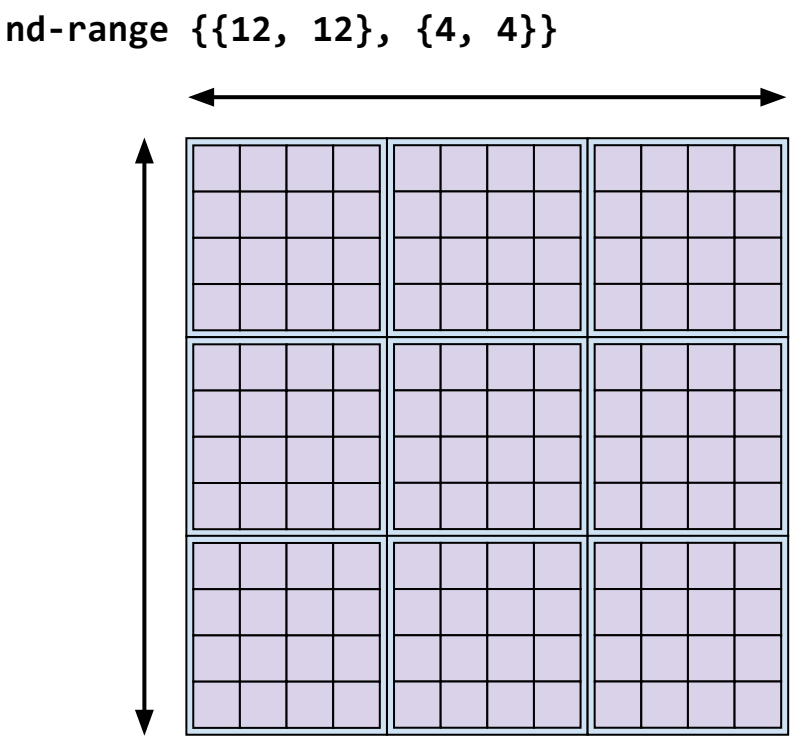
Launching SPMD kernels



Kernels are launched in the form of an nd-range

- An nd-range can be 1, 2 or 3 dimensions
- An nd-range describes a number of work-items divided into equally sized work-groups

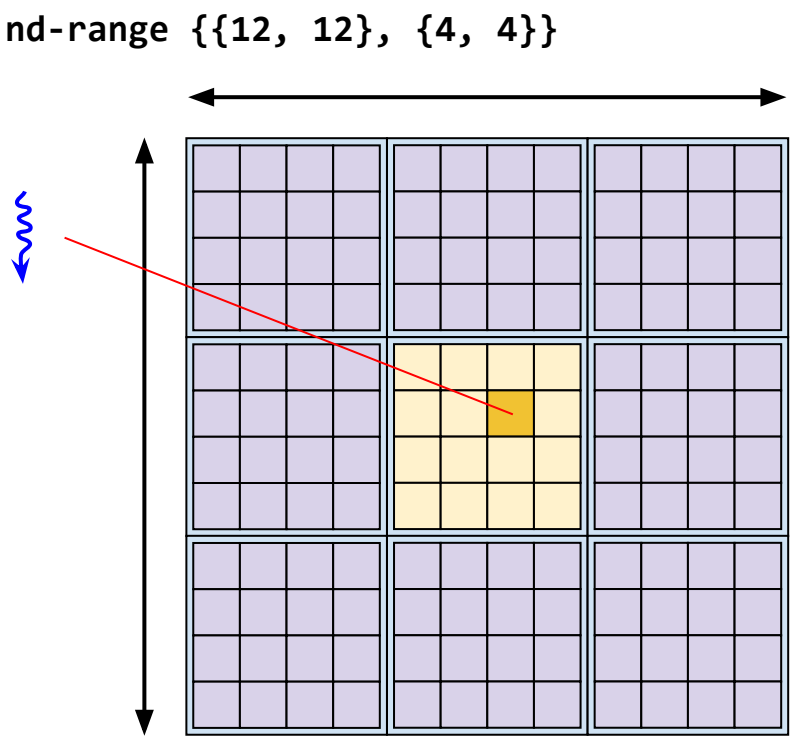
Launching SPMD kernels



An nd-range is mapped to the underlying hardware

- Work-groups are mapped to compute units
- Work-items are mapped to processing units

Launching SPMD kernels



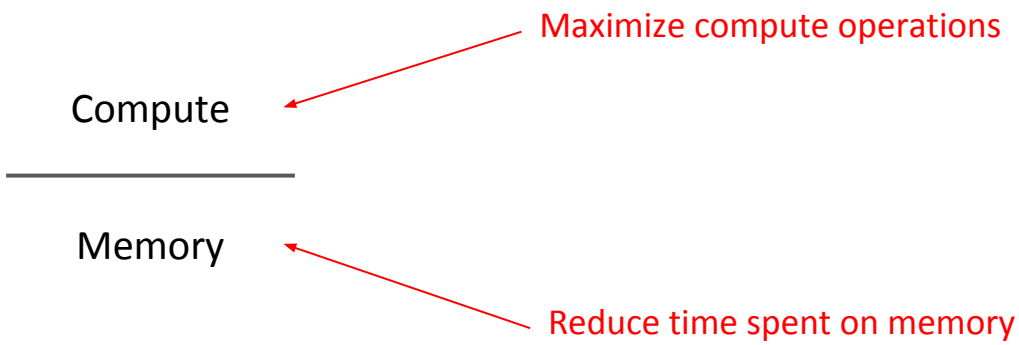
The kernel is executed once per work-item in the nd-range

Each work item knows it's index within the nd-range

- global range {12, 12}
- local range {4, 4}
- group range {3, 3}
- global id {6, 5}
- local id {2, 1}
- group id {1, 1}

Writing efficient SPMD programs

Maximize arithmetic intensity



Reduce time spent on memory

Move frequently accessed data to fast memory

- Private memory > Local memory > Global memory

Reduce time spent on memory

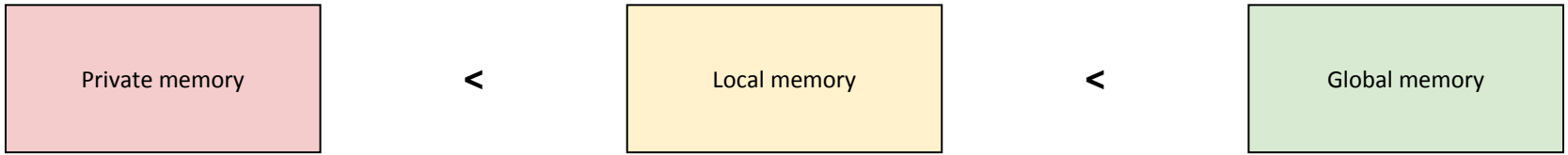
Move frequently accessed data to fast memory

- Private memory > Local memory > Global memory

Coalesce access to global memory

- Access strides of global memory following the execution of work-items

GPU memory access latency



Coalesce global memory access

Accessing global memory is very high latency

Coalesce global memory access

Accessing global memory is very high latency

On GPUs and SIMD CPUs memory is loaded in strides

Coalesce global memory access

Accessing global memory is very high latency

On GPUs and SIMD CPUs memory is loaded in strides

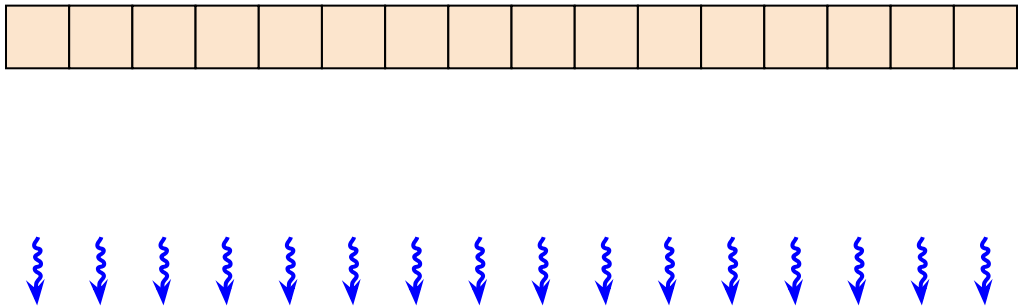
So it is efficient to access global memory in strides, and very inefficient to access global memory randomly

Coalesce global memory access

```
struct str {
  float f;
};
```

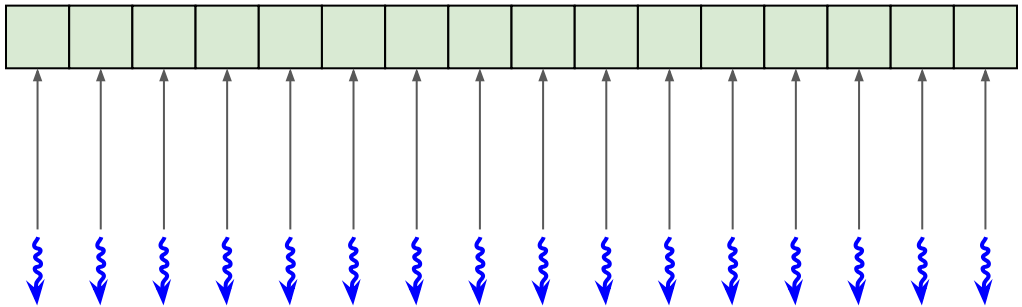
Coalesce global memory access

```
struct str {
  float f;
};

a[globalId] =
  globalId;
```



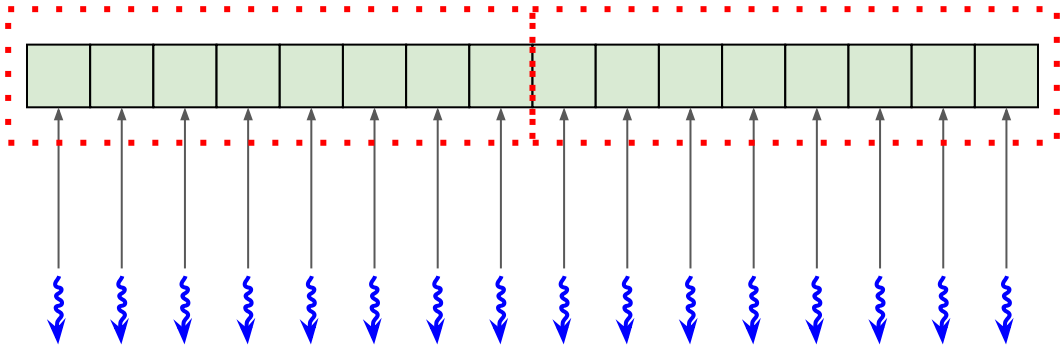
Coalesce global memory access

```
struct str {
  float f;
};

a[globalId] =
  globalId;
```



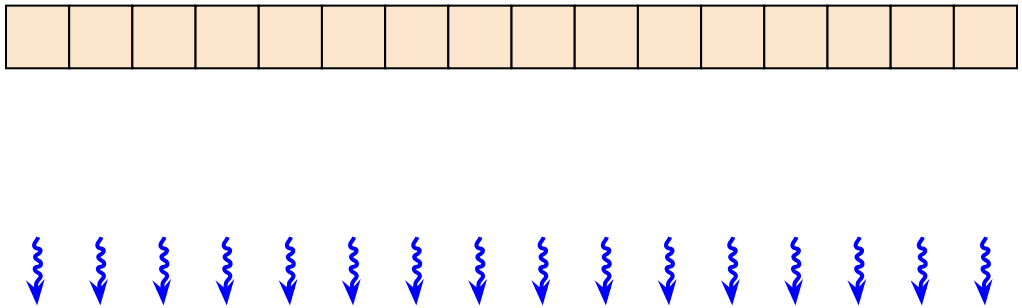
Coalesce global memory access

```
struct str {
  float f;
};

a[globalId] =
  globalId;
```



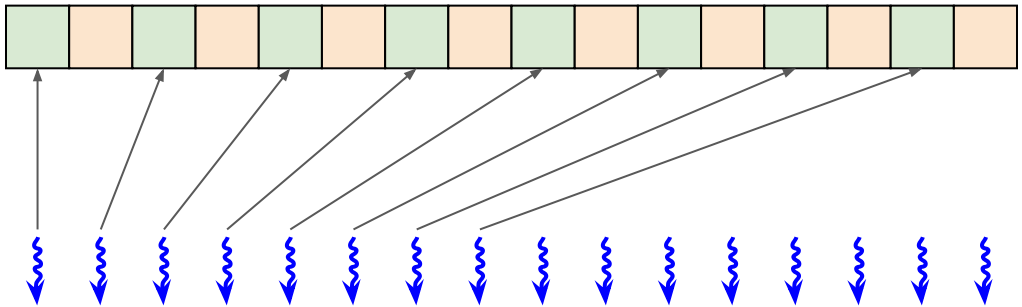
Coalesce global memory access

```
struct str {
  float f;
};

a[globalId] =
  globalId * 2;
```



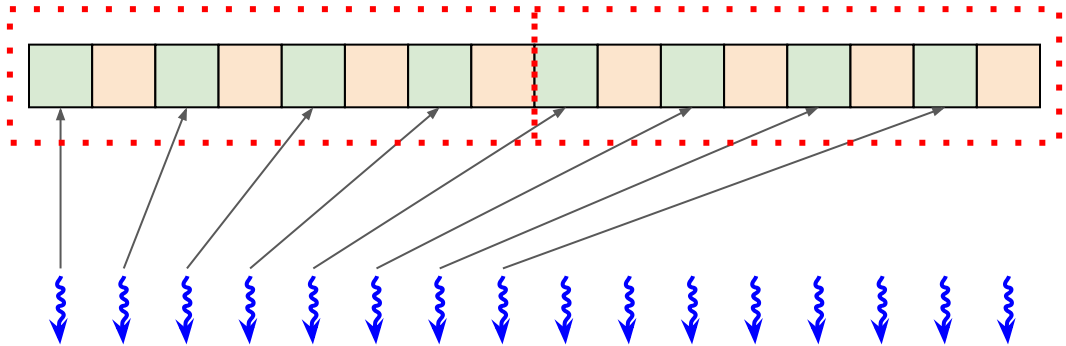
Coalesce global memory access

```
struct str {
  float f;
};

a[globalId] =
  globalId * 2;
```



Coalesce global memory access

```
struct str {
  float f;
};

a[globalId] =
  globalId * 2;
```



AoS vs SoA

AoS and SoA describe different ways to structure your data

AoS vs SoA

AoS and SoA describe different ways to structure your data

- AoS = “array of structures”

AoS vs SoA

AoS and SoA describe different ways to structure your data

- AoS = “array of structures”
- SoA = “structure of arrays”

AoS vs SoA

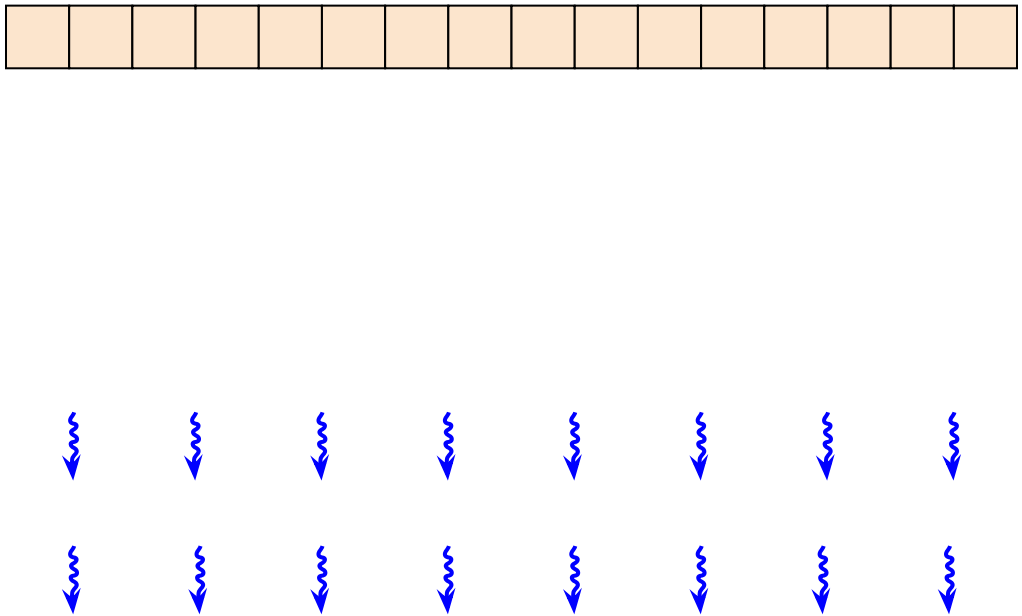
AoS and SoA describe different ways to structure your data

- AoS = “array of structures”
- SoA = “structure of arrays”

When accessing global memory depending on how you access global memory you may want to arrange your data in one layout or the other

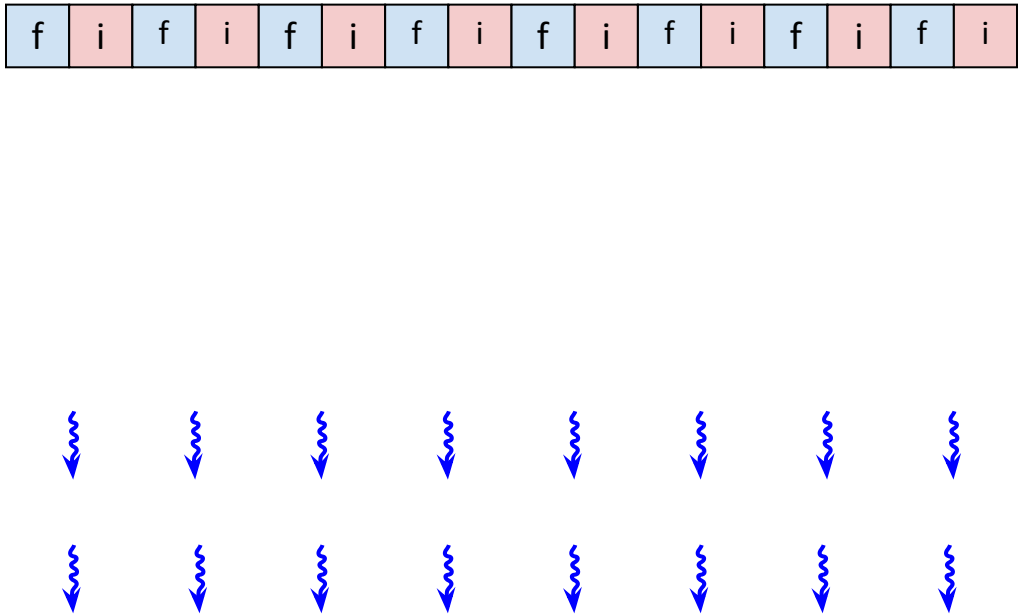
AoS vs SoA

```
struct str {
  float f;
  int i;
};

str s[N];

... = s[globalId].f;

... = s[globalId].i;
```



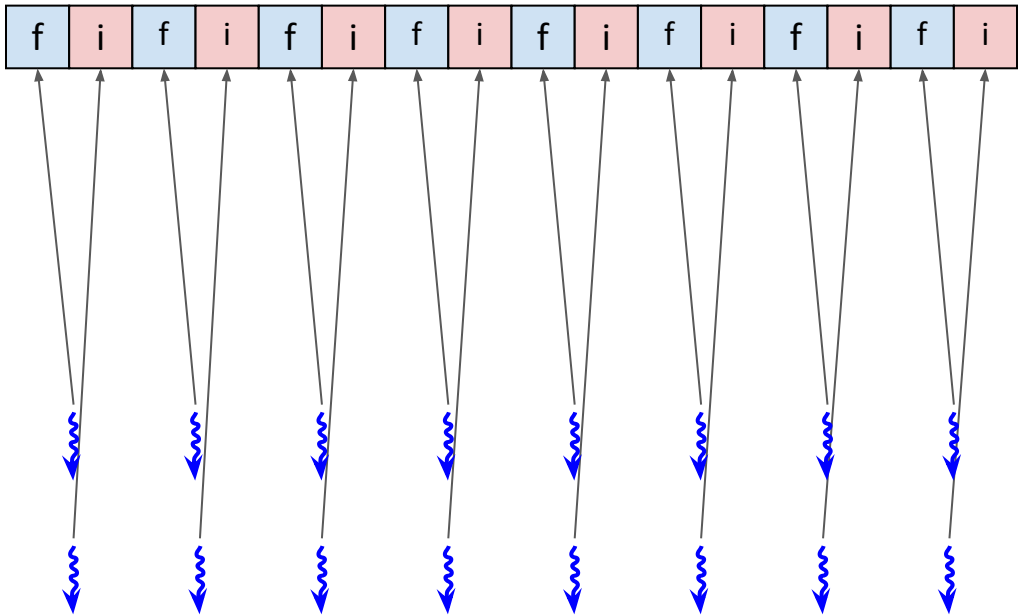
AoS vs SoA

```
struct str {
  float f;
  int i;
};

str s[N];

... = s[globalId].f;

... = s[globalId].i;
```



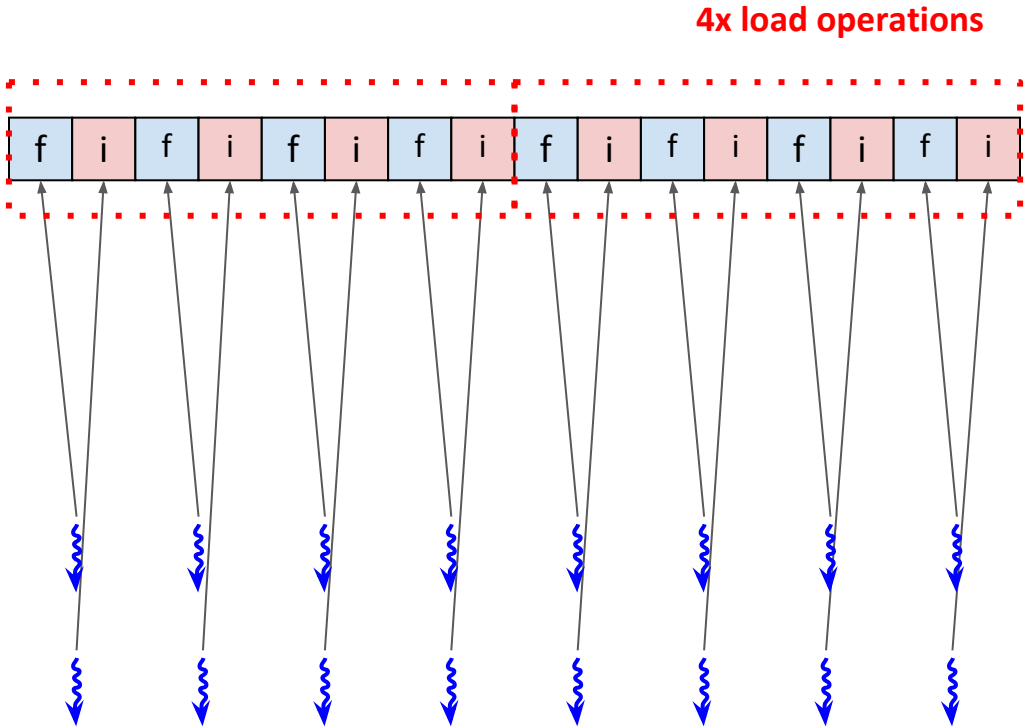
AoS vs SoA

```
struct str {
  float f;
  int i;
};

str s[N];

… = s[globalId].f;

… = s[globalId].i;
```



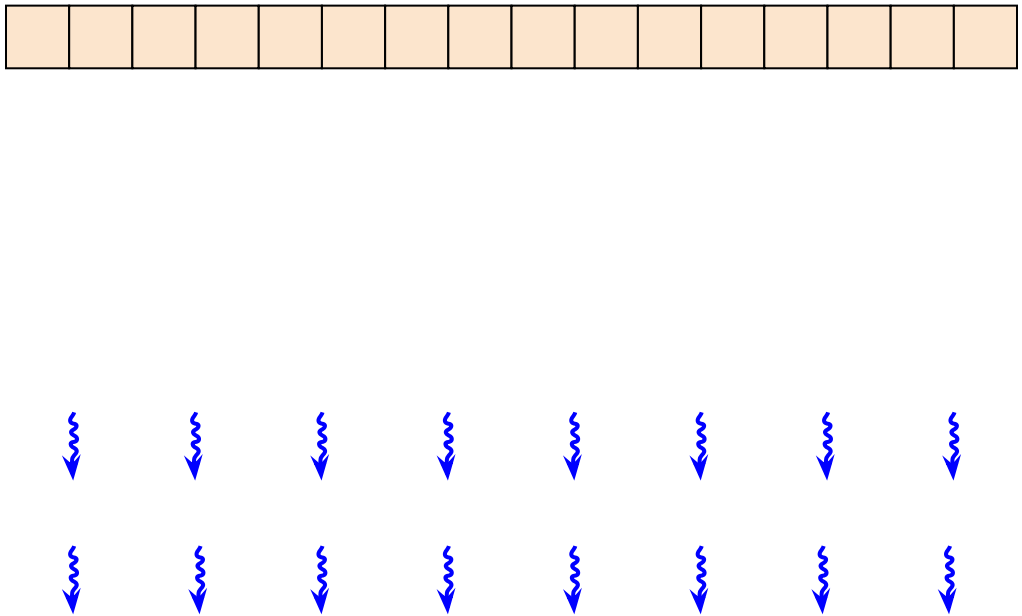
AoS vs SoA

```
struct str {
  float f;
  int i;
};

str s[N];

... = s[globalId].f;

... = s[globalId].i;
```



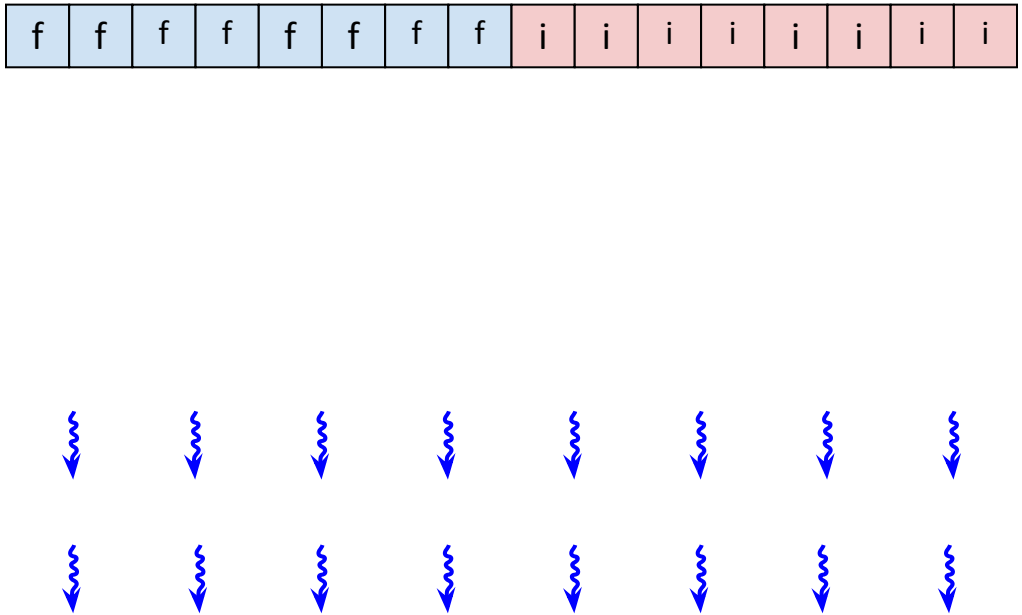
AoS vs SoA

```
struct str {
  float f[N];
  int i[N];
};

str s;

... = s.f[globalId];

... = s.i[globalId];
```



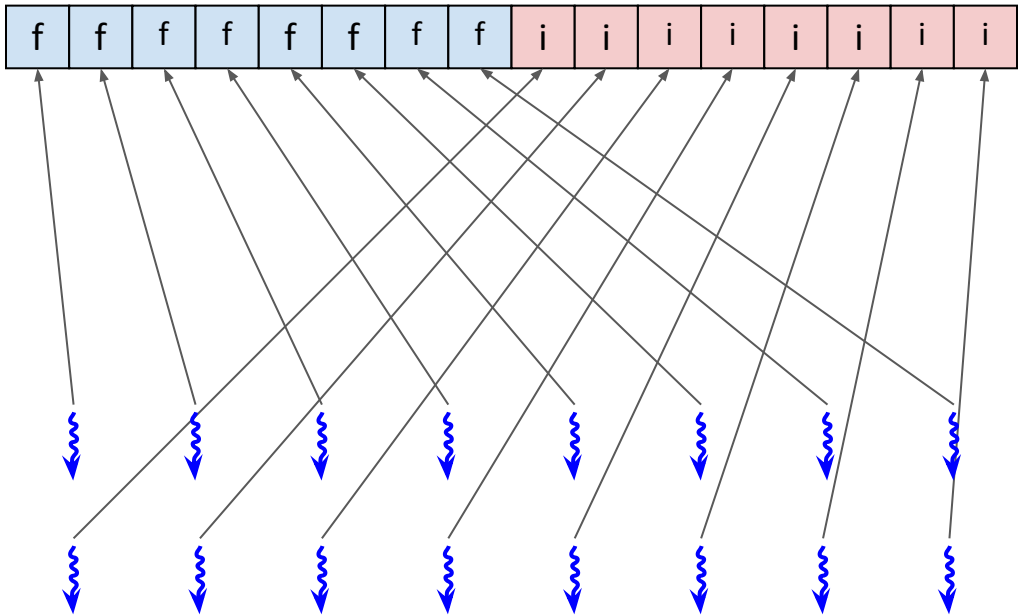
AoS vs SoA

```
struct str {
  float f[N];
  int i[N];
};

str s;

... = s.f[globalId];

... = s.i[globalId];
```



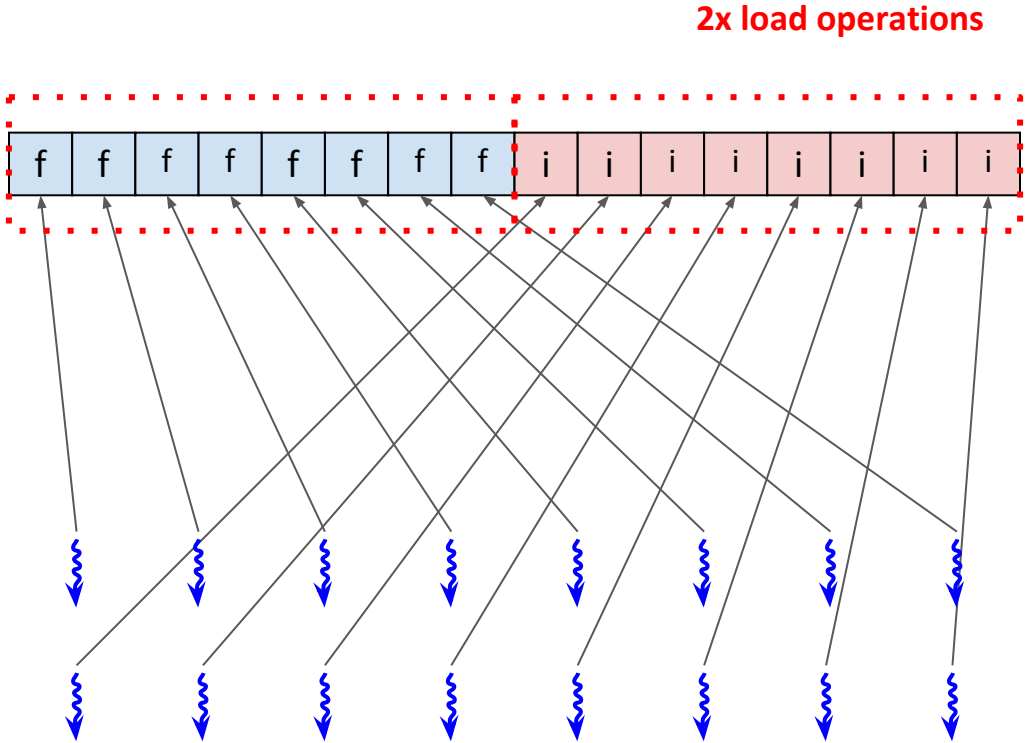
AoS vs SoA

```
struct str {
  float f[N];
  int i[N];
};

str s;

… = s.f[globalId];

… = s.i[globalId];
```



AoS vs SoA

```
struct str {
  float f[N];
  int i[N];
};

str s;

... = s.f[globalId];

... = s.i[globalId];
```



Row-major vs column-major

Another consideration to make when structuring your data comes when you have multi-dimensional representation

- This is common for image processing

Row-major vs column-major

Another consideration to make when structuring your data comes when you have multi-dimensional representation

- This is common for image processing

Row-major or column-major effectively describe which dimension of your data structure is the faster moving

Row-major vs column-major

Another consideration to make when structuring your data comes when you have multi-dimensional representation

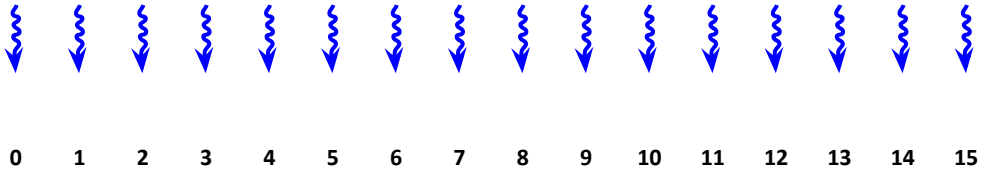
- This is common for image processing

Row-major or column-major effectively describe which dimension of your data structure is the faster moving

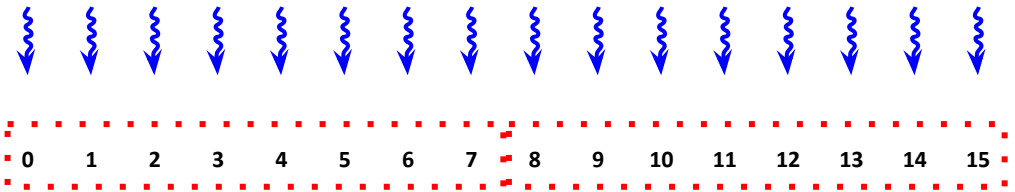
This has interesting implications in SPMD execution

Row-major vs column-major

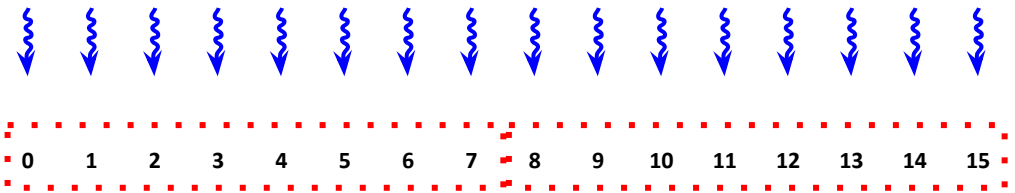
Row-major vs column-major



Row-major vs column-major

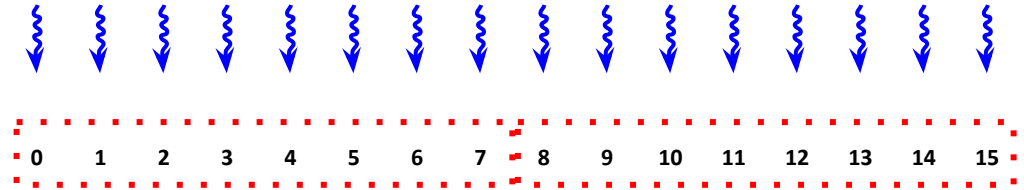


Row-major vs column-major

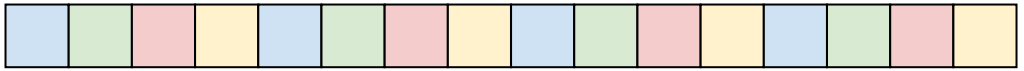


0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

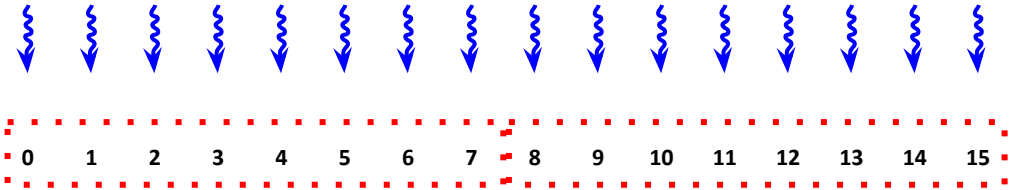
Row-major vs column-major



0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15



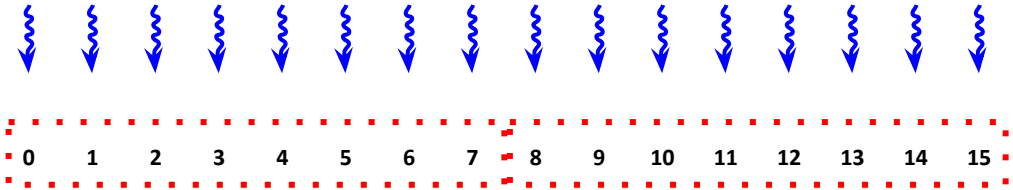
Row-major vs column-major



0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

0	4	8	9	1	5	8	10	2	6	8	11	3	7	8	12

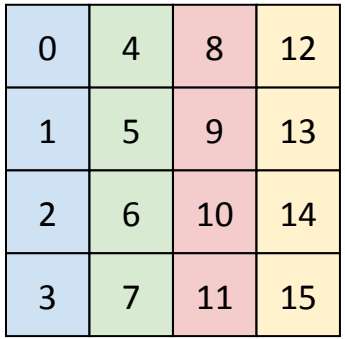
Row-major vs column-major

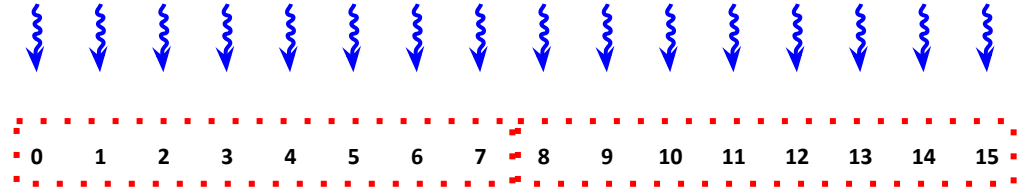


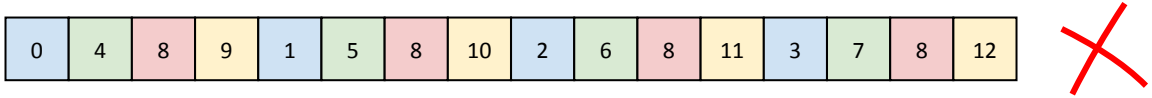
0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

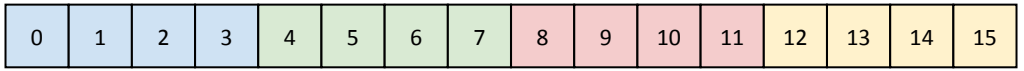
0	4	8	9	1	5	8	10	2	6	8	11	3	7	8	12

Row-major vs column-major



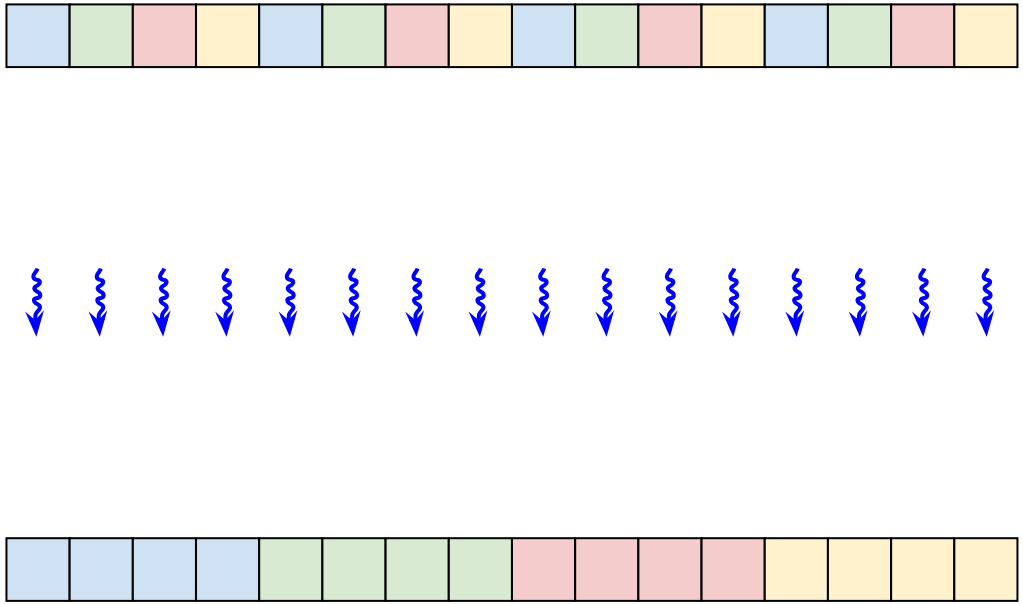






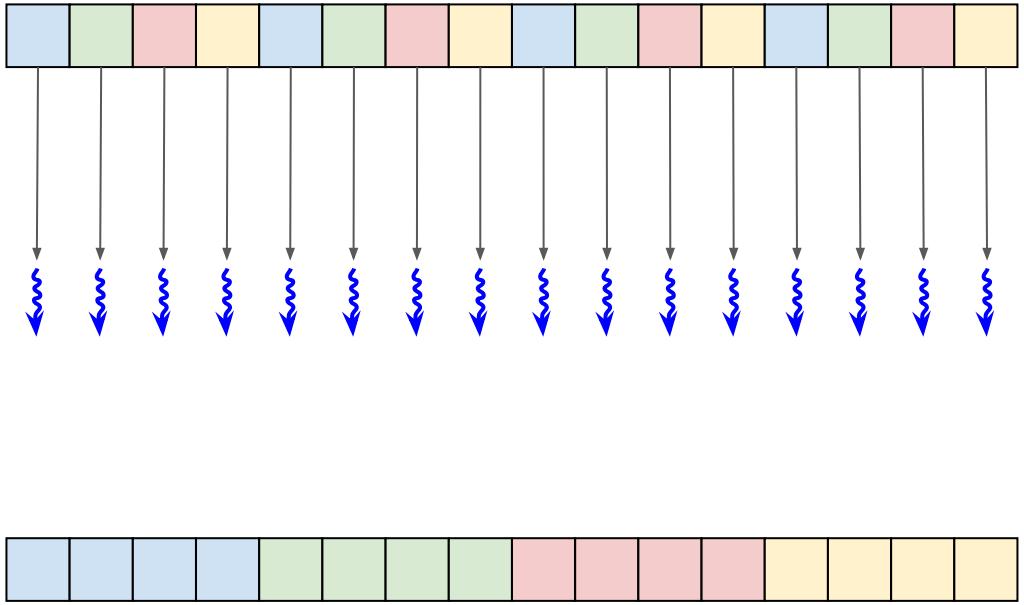
Row-major vs column-major

```
int x = globalId[0];
int y = globalId[1];
int stride = 4;

out[(x * stride) + y] =
  in[(y * stride) + x];
```



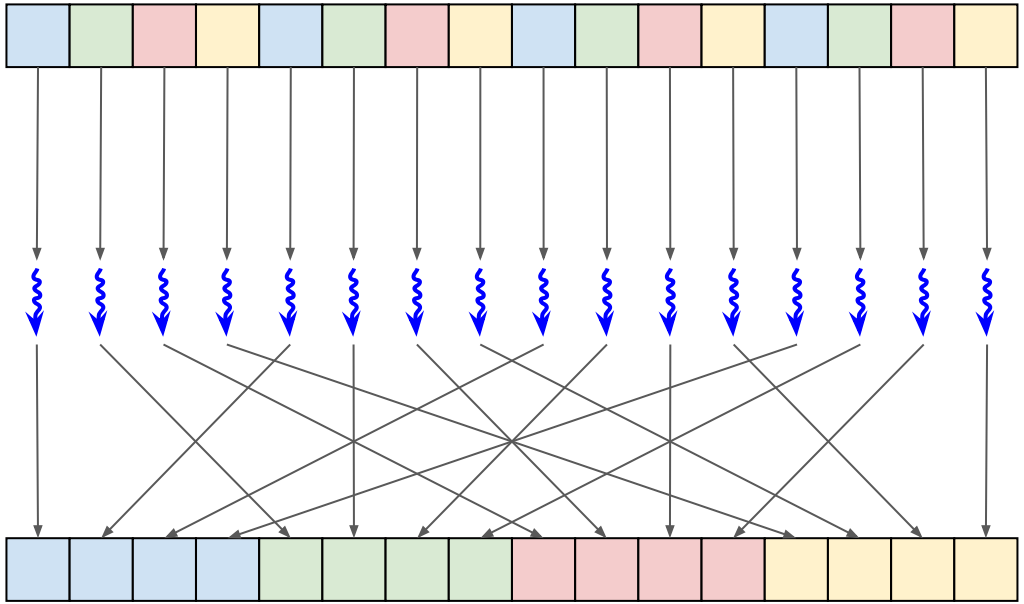
Row-major vs column-major

```
int x = globalId[0];
int y = globalId[1];
int stride = 4;

out[(x * stride) + y] =
  in[(y * stride) + x];
```



Row-major vs column-major

```
int x = globalId[0];
int y = globalId[1];
int stride = 4;

out[(x * stride) + y] =
  in[(y * stride) + x];
```



Lock-step execution

SPMD architectures such as SIMD CPUs and GPUs execute in lock-step

Lock-step execution

SPMD architectures such as SIMD CPUs and GPUs execute in lock-step

This means that an instruction is loaded and then executed on a number of elements of data concurrently

Lock-step execution

SPMD architectures such as SIMD CPUs and GPUs execute in lock-step

This means that an instruction is loaded and then executed on a number of elements of data concurrently

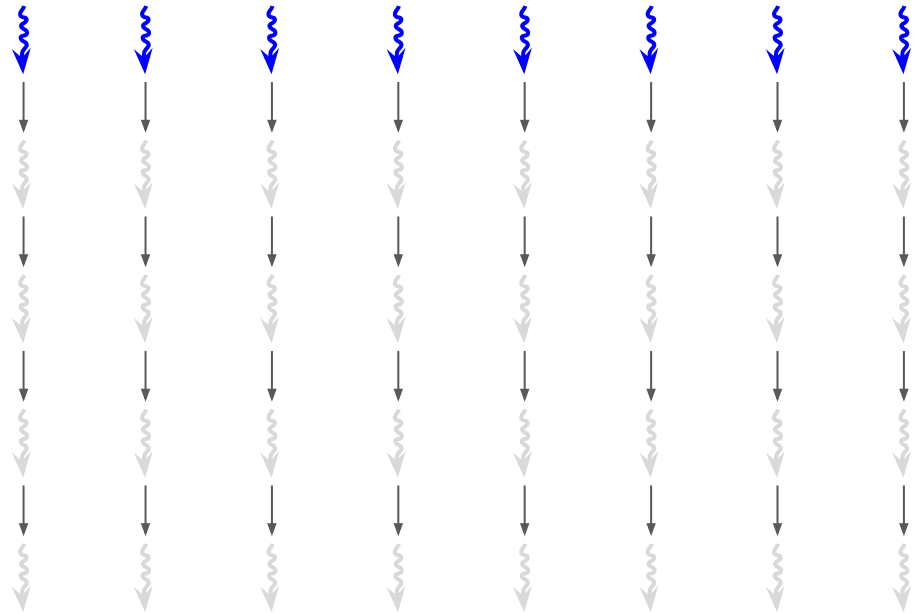
This has interesting implications for program control-flow

Lock-step execution

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```

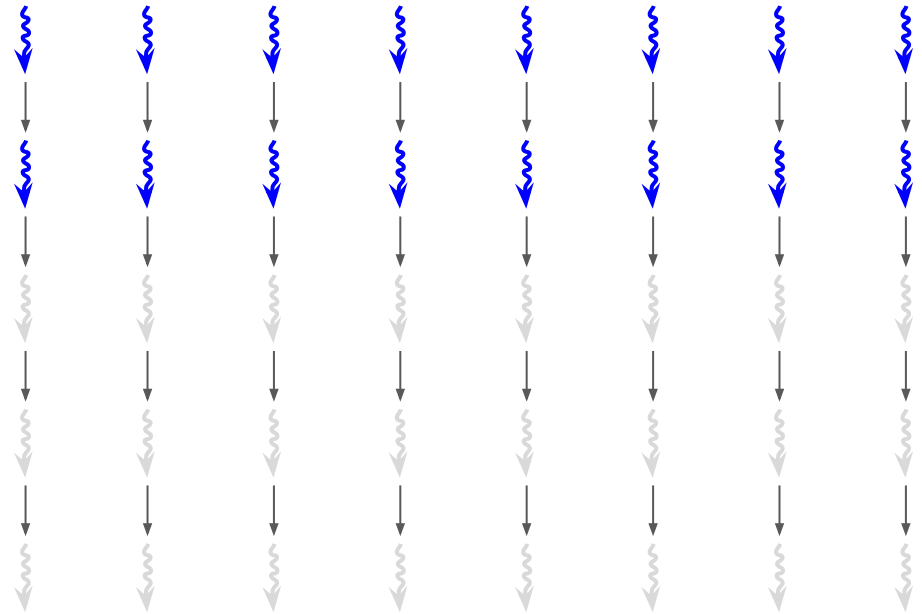
Lock-step execution

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



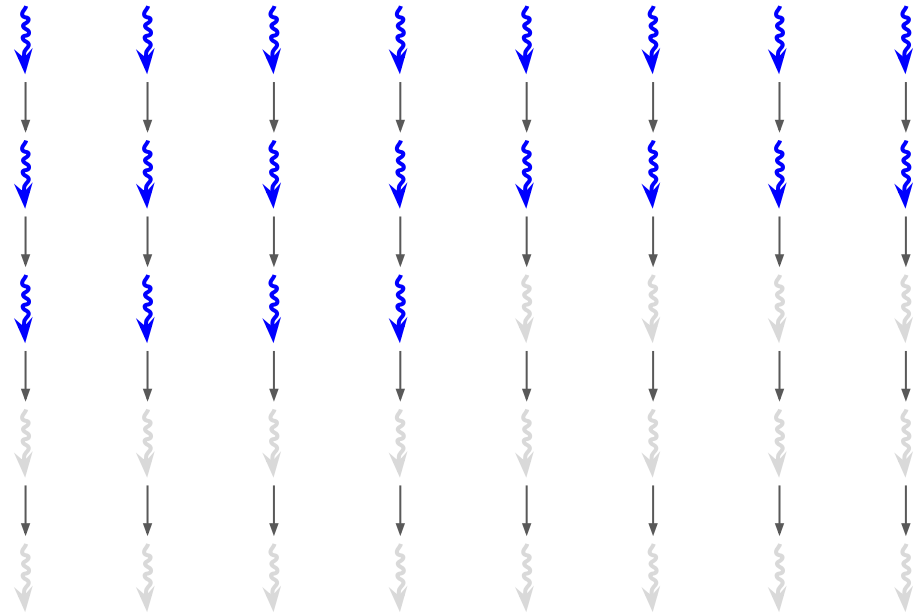
Lock-step execution

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



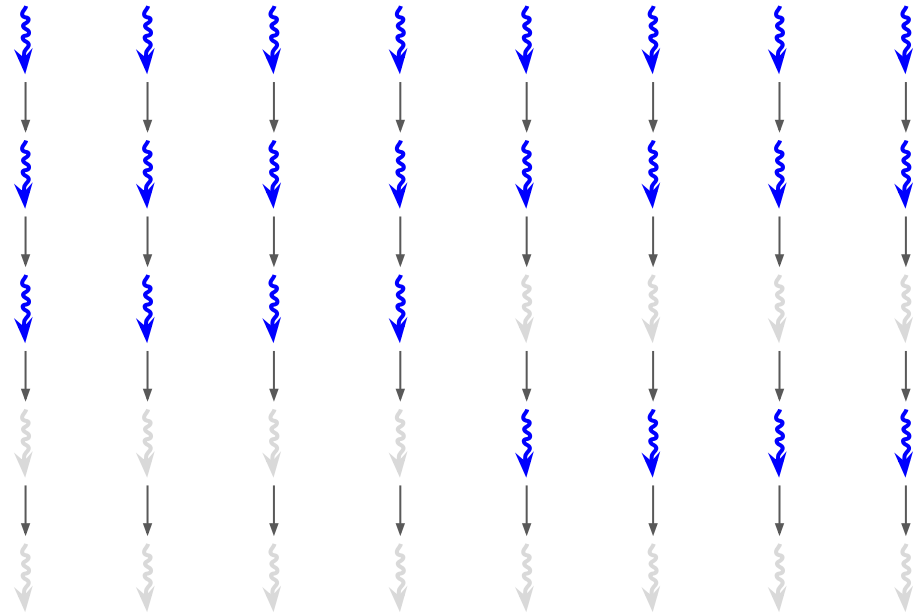
Lock-step execution

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



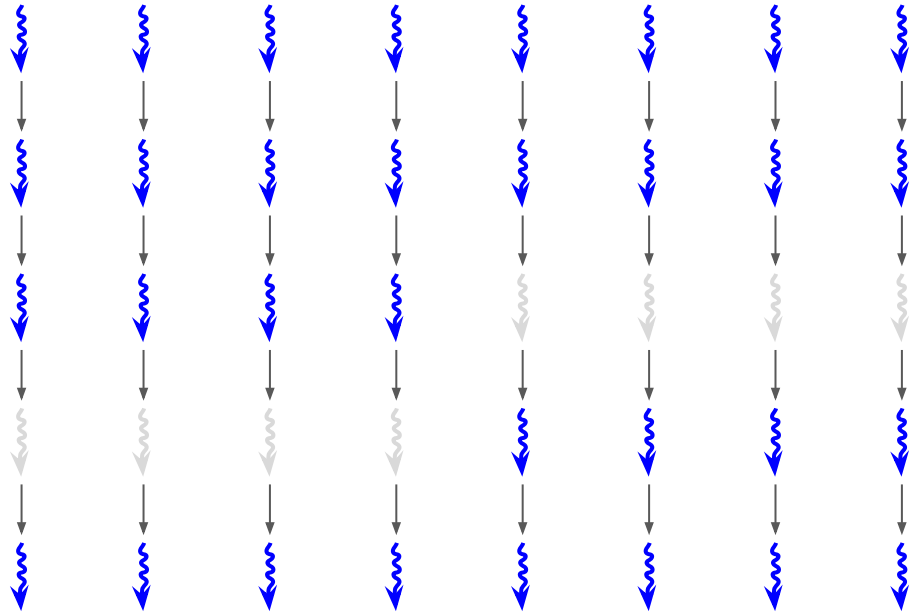
Lock-step execution

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



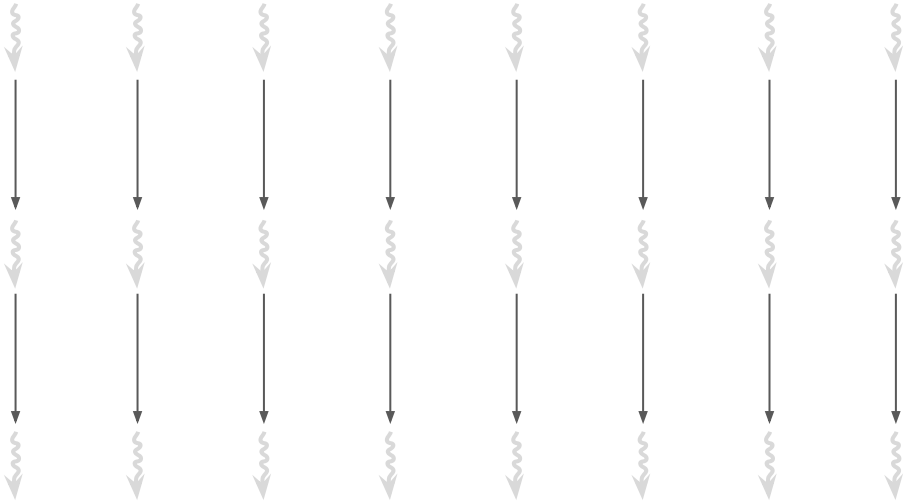
Lock-step execution

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



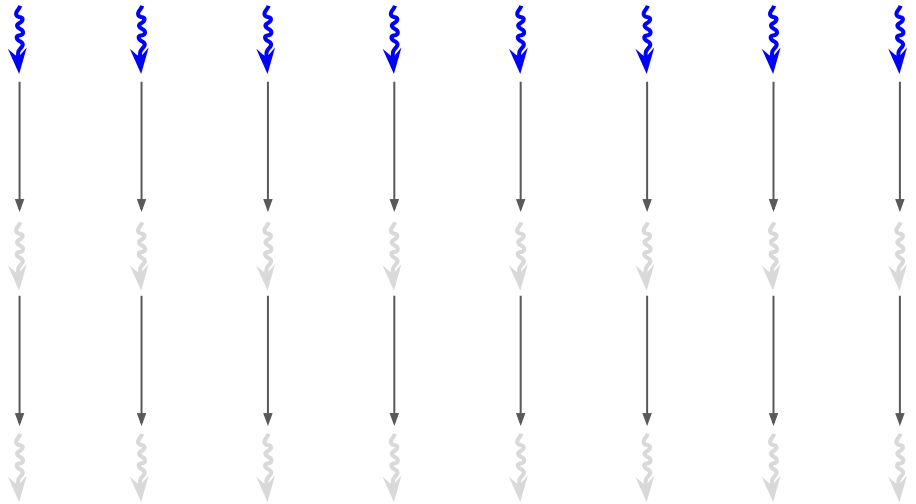
Loop divergence

```
…

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

…
```



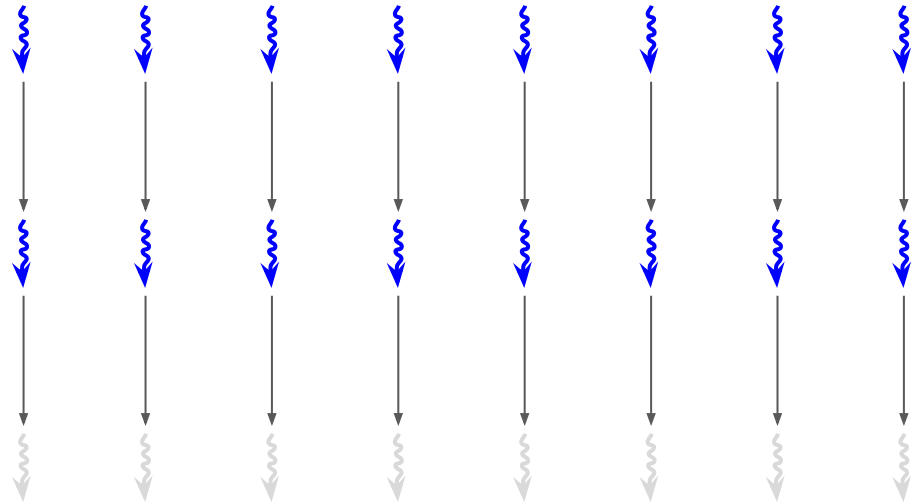
Loop divergence

```
…

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

…
```



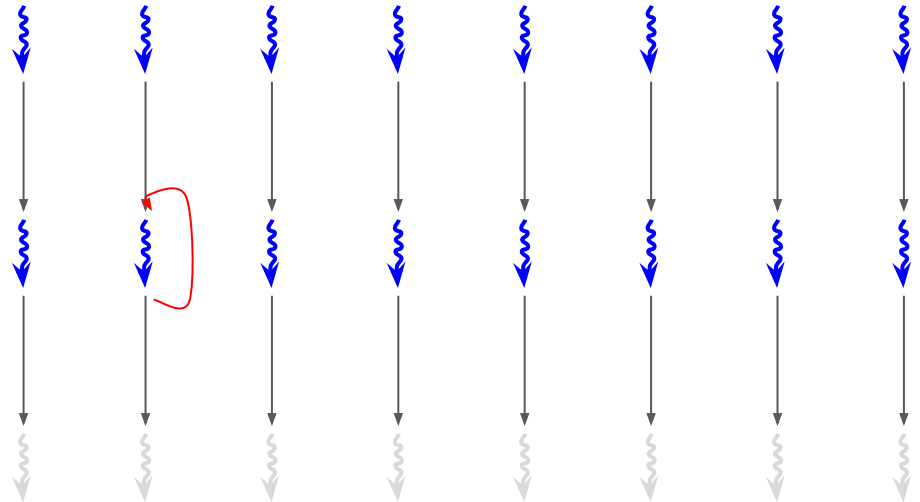
Loop divergence

```
…

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

…
```



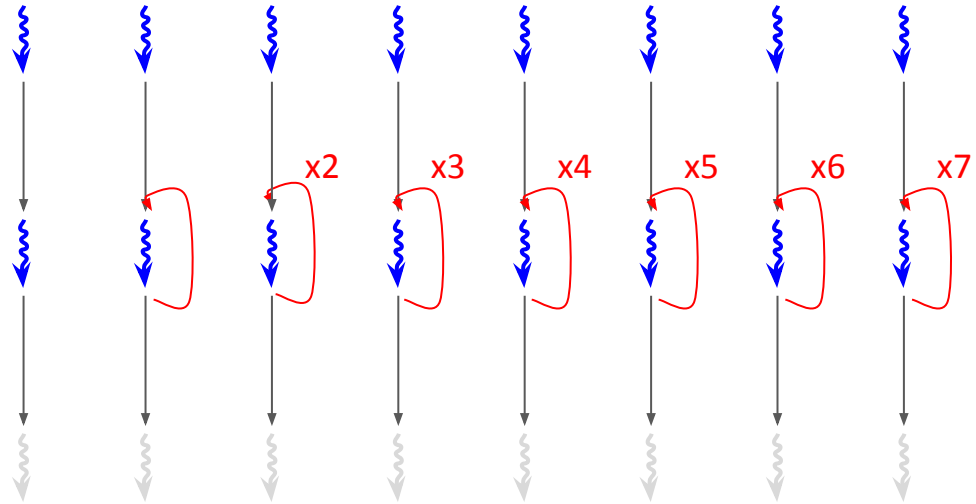
Loop divergence

```
…

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

…
```



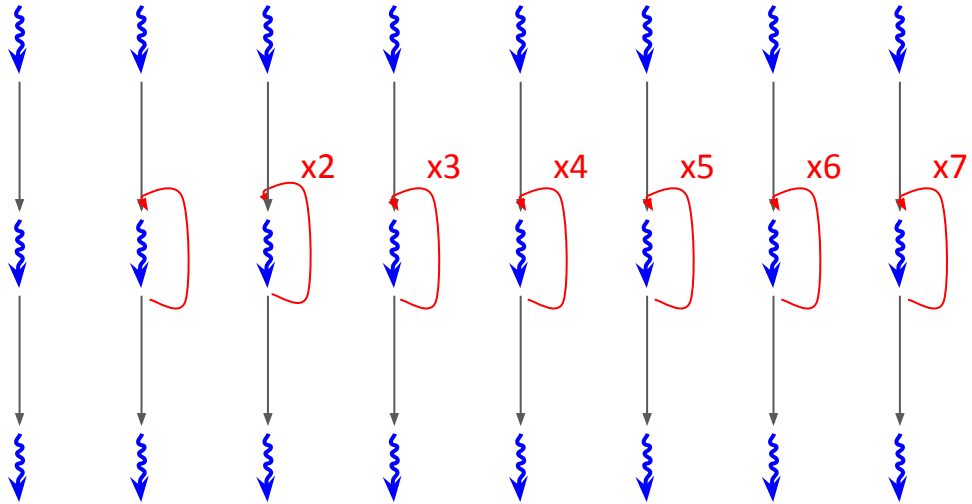
Loop divergence

```
…

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

…
```



Loop divergence

```
…

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

…
```



Maximize compute operations

Copying data from CPU memory to GPU memory is generally very expensive

- This varies depending on the architecture
- This can also not be the case if your system has unified memory or shared virtual memory

Sometimes the cost of copying data to the GPU can outweigh the benefit of performing a computation on the GPU

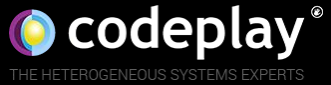
- Maximize the amount of work that can be done on the GPU
- If possible overlap data movement with computation

Key takeaways

CPUs have a large number of small cores and is optimized for latency while GPUs have a small number of large cores and is optimized for throughput

The SPMD programming model consists of a hierarchy of execution groupings, synchronisation primitives and memory regions

Optimizing for a GPU means maximizing computation while minimizing time spent on memory


codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Chapter 10: C++ for GPUs (SYCL)

Prerequisite:

1. Previous chapters
2. Limitations of SPMD programming model

You will learn:

1. Why you may want to use SYCL
2. What a typical SYCL application looks like
3. How SYCL single source compilation works

SYCL for OpenCL

Cross-platform, single-source, high-level, C++ programming layer
Built on top of OpenCL and based on standard C++11
Delivering a heterogeneous programming solution for C++

Why use SYCL for GPU programming?

- Enables programming heterogeneous devices such as GPUs using standard C++
- Provides a high-level abstraction for development of complex parallel software applications
- Provides efficient data dependency analysis and task scheduling and synchronisation

SYCL is entirely standard C++

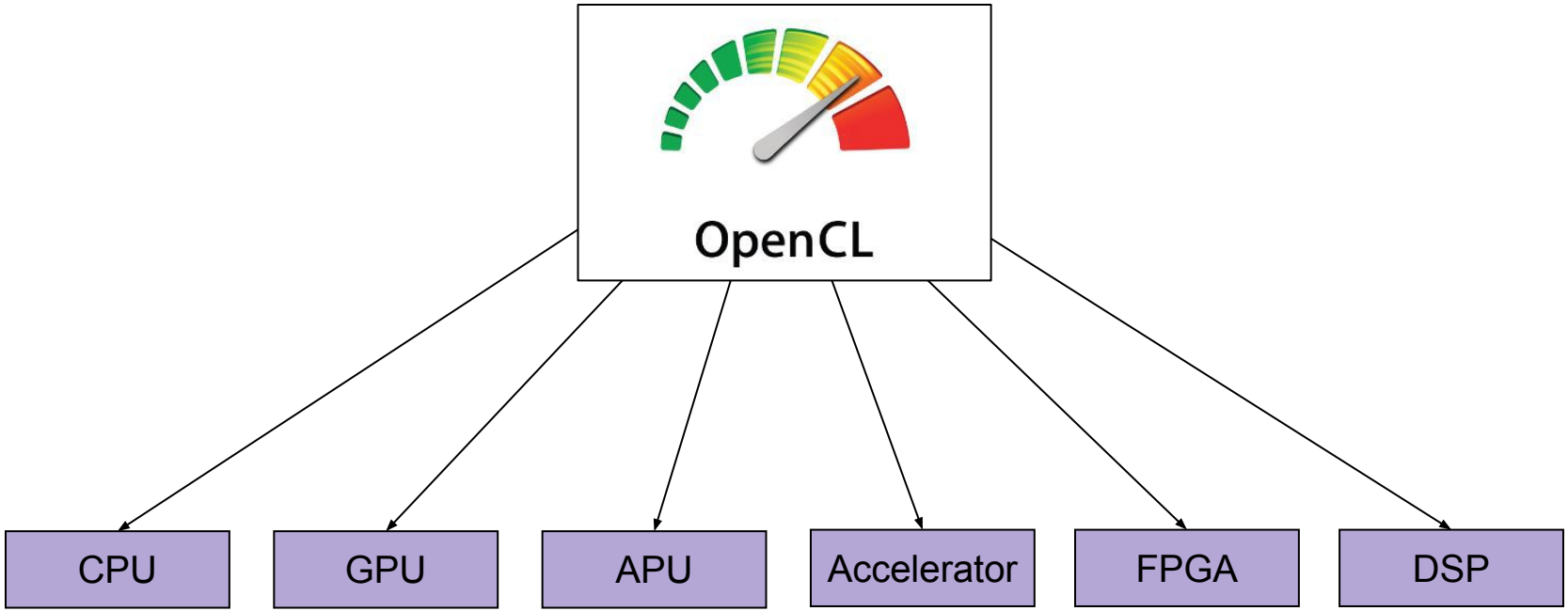
```
__global__ vec_add(float *a, float *b, float *c) {
  return c[i] = a[i] + b[i];
}

float *a, *b, *c;
vec_add<<<range>>>(a
```

```
vector<float> a, b, c;

#pragma parallel_for
for(int i = 0; i < a.size(); i++) {
```

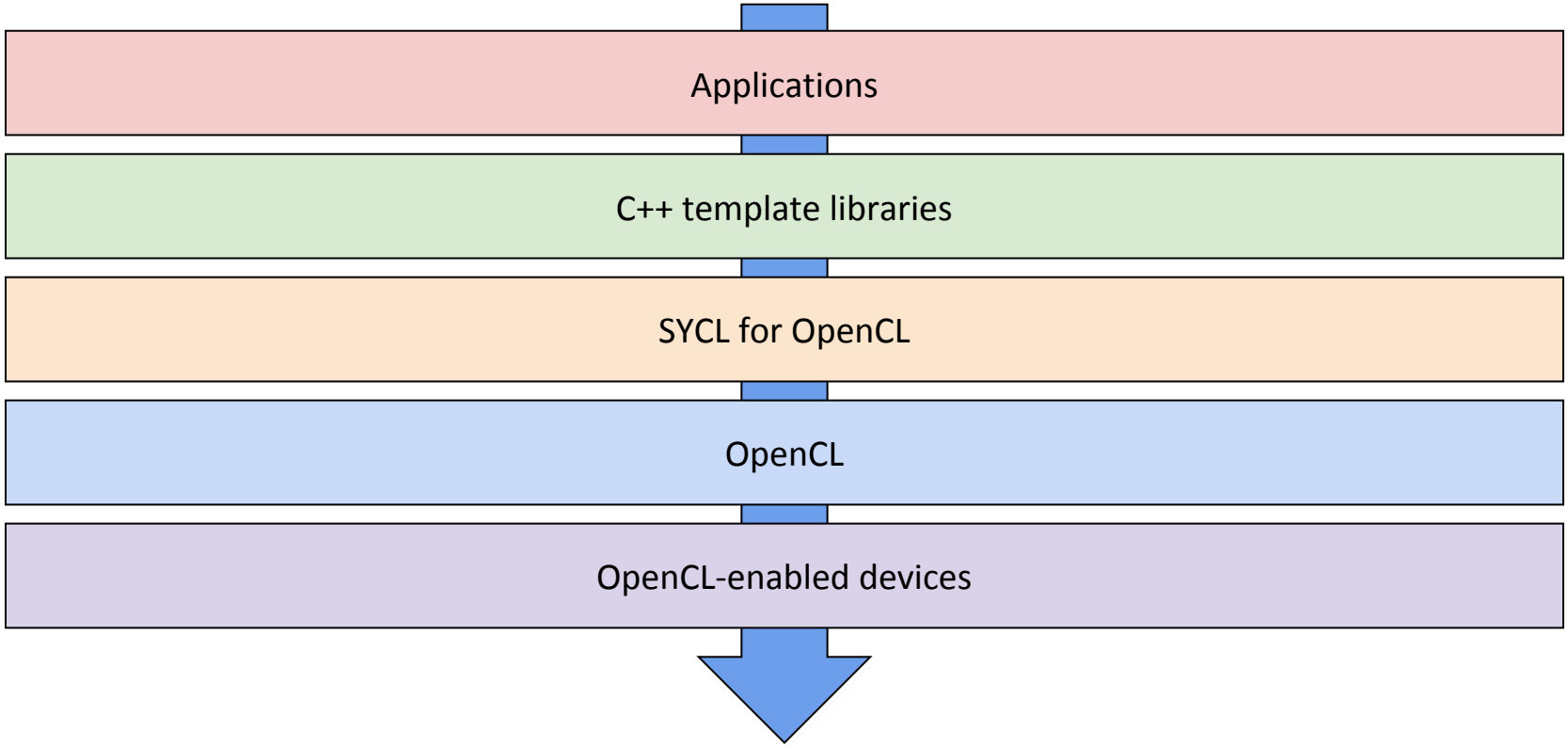
```
array_view<float> a, b, c;
extent<2> e(64, 64);

parallel_for_each(e, [=](index<2> idx) restrict(amp) {
  c[idx] = a[idx] + b[idx];
});
```

```
cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
  c[idx] = a[idx] + c[idx];
}));
```

SYCL can target a wide range of OpenCL devices



The SYCL ecosystem



So how do you write GPU code using SYCL?

This example will cover everything you need to know to write a simple vector add using SYCL

Vector add example

```
#include <vector>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {









}
```

Let’s start with the prototype for a function which takes two input vectors and an output vector

We’re going to implement this function using SYCL

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>


void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {









}
```

First we include the CL/sycl.hpp header

This header contains the entire SYCL API

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;









}
```

Next we import the cl::sycl namespace to reduce the verbosity of the code

The entire SYCL API is defined within the cl::sycl namespace

Selecting a device to execute on

In SYCL devices are selected using what's called a device selector

A device selector picks the best device based on a particular heuristic

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;









}
```

Now we create a queue that we can enqueueing work on

A default constructed queue will have the SYCL runtime choose a device for you using the default selector

Enqueuing work on a queue

In SYCL all work is enqueued to a queue via command groups

A command group represents the kernel function, an nd-range and the data dependencies

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;









}
```

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;




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







  });
}
```

We create a command group to enqueue work to the queue

Managing memory across host and device

In SYCL the storage and access of memory is separated via buffers and accessors

A buffer manages a region of memory across host and one or more devices

An accessor represents an instance of access to a particular buffer

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;




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









  });
}
```

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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









  });
}
```

We create buffer objects for the data of each vector to manage the data across the host and device(s)

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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









  });
}
```

Buffers synchronise and copy their data back to the original pointer when they are destroyed

So in this case this on returning from the parallel_add function

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);



  });
}
```

We create an accessor for each buffer with suitable access modes

The two inputs are read and the output is write

Launching kernel functions

In SYCL there are several ways to launch kernel functions which express different forms of parallelism

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);




  });
}
```

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>



void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    cgh.parallel_for<   >(range<1>(a.size()), [=](id<1> i) {

    });
  });
}
```

We launch a kernel function by calling a kernel invocation API

In this case we are using parallel_for, which takes a range and a function object

Vector add example

```
#include <vector>
#include <CL/sycl.hpp>

class add;

void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> i) {

    });
  });
}
```

We use a template parameter to name the device function

This is required for portability

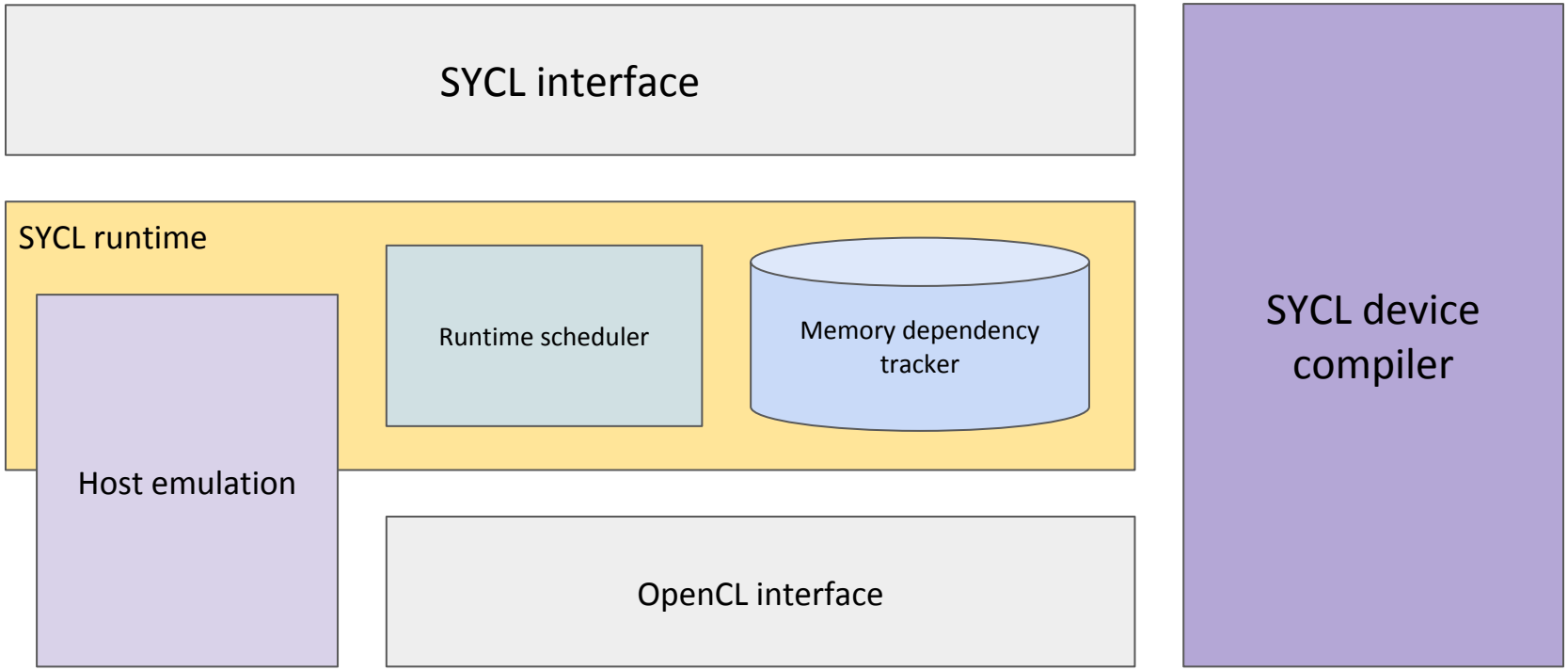
Vector add example

```
#include <vector>
#include <CL/sycl.hpp>

class add;

void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(a.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(b.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> i) {
      oAcc[i] = aAcc[i] + bAcc[i];
    });
  });
}
```

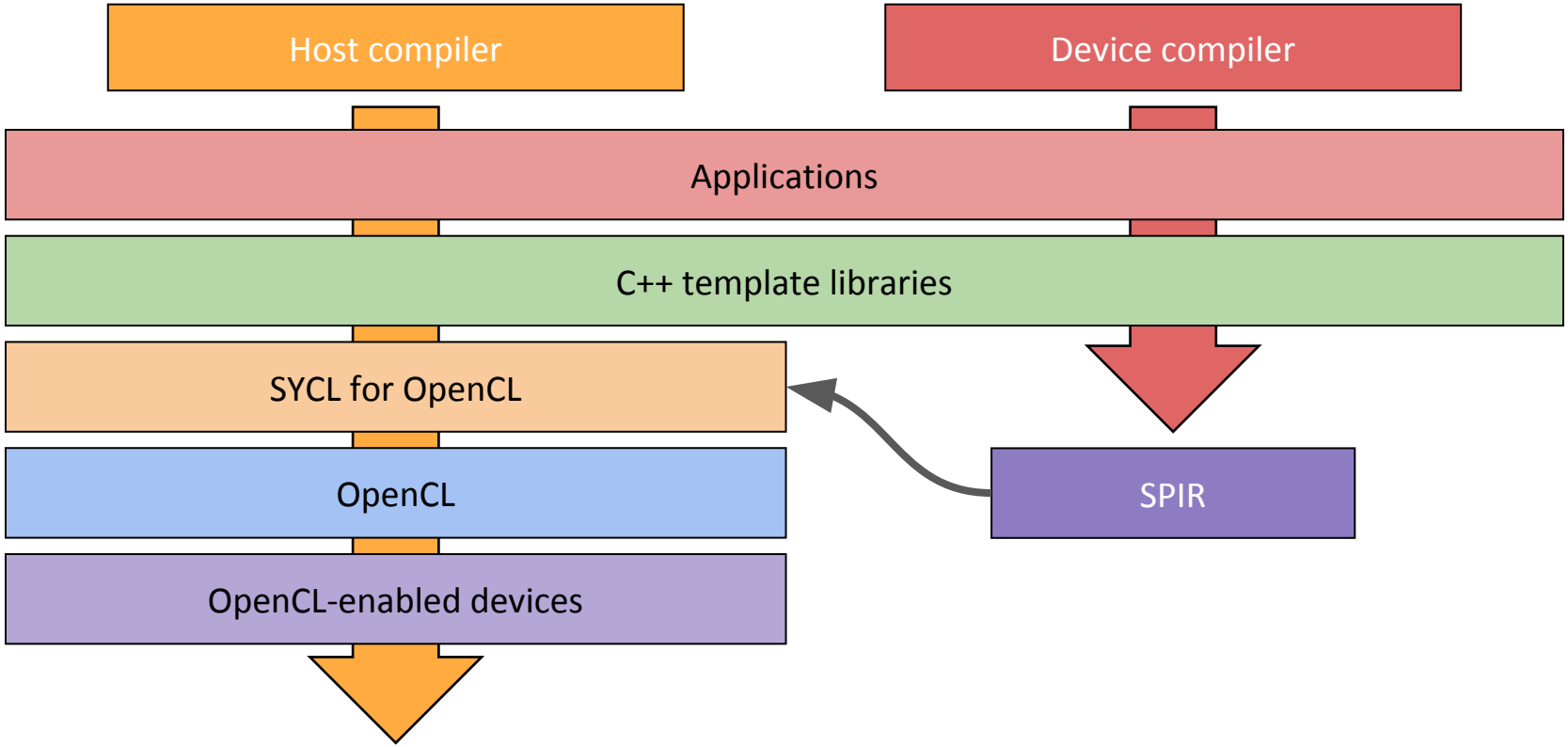
The lambda defines the kernel function

In the body of lambda we add the elements of the two input accessors and assign it to the output accessor

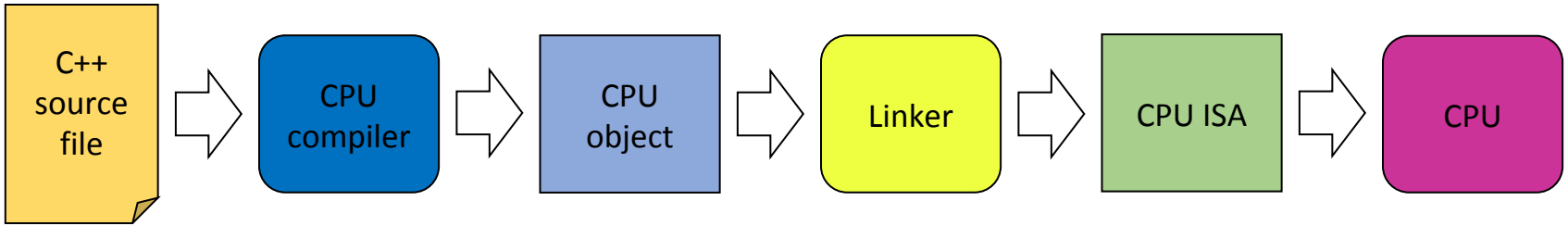
What do you get in a SYCL implementation?



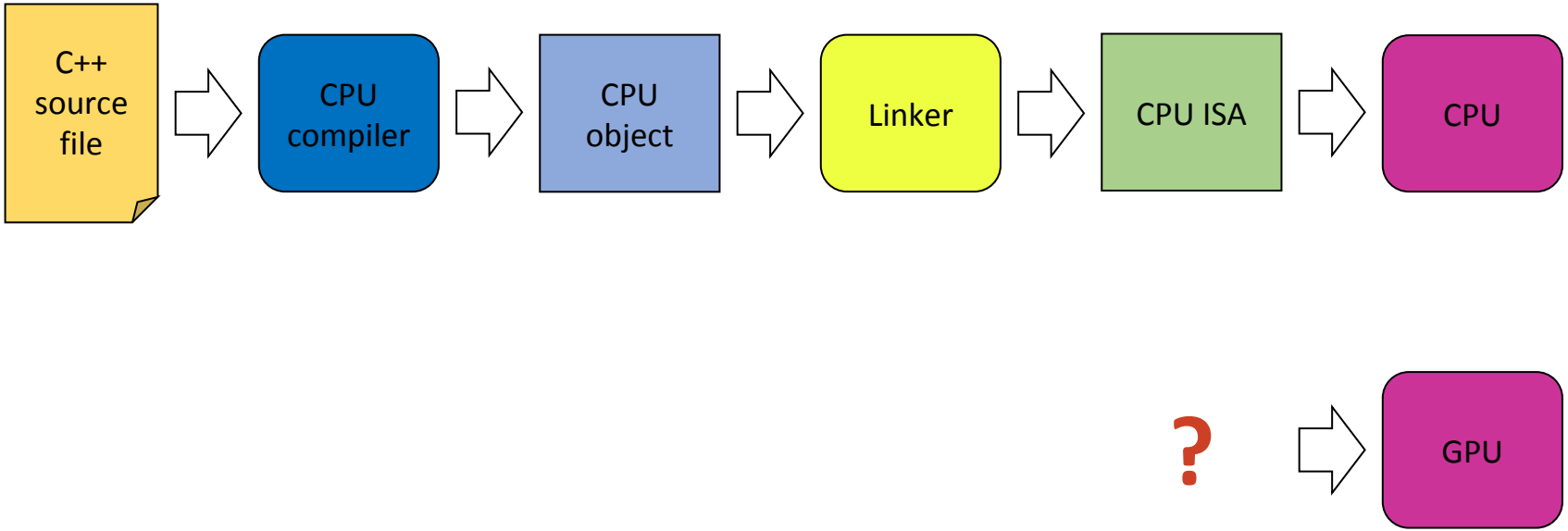
SYCL single source



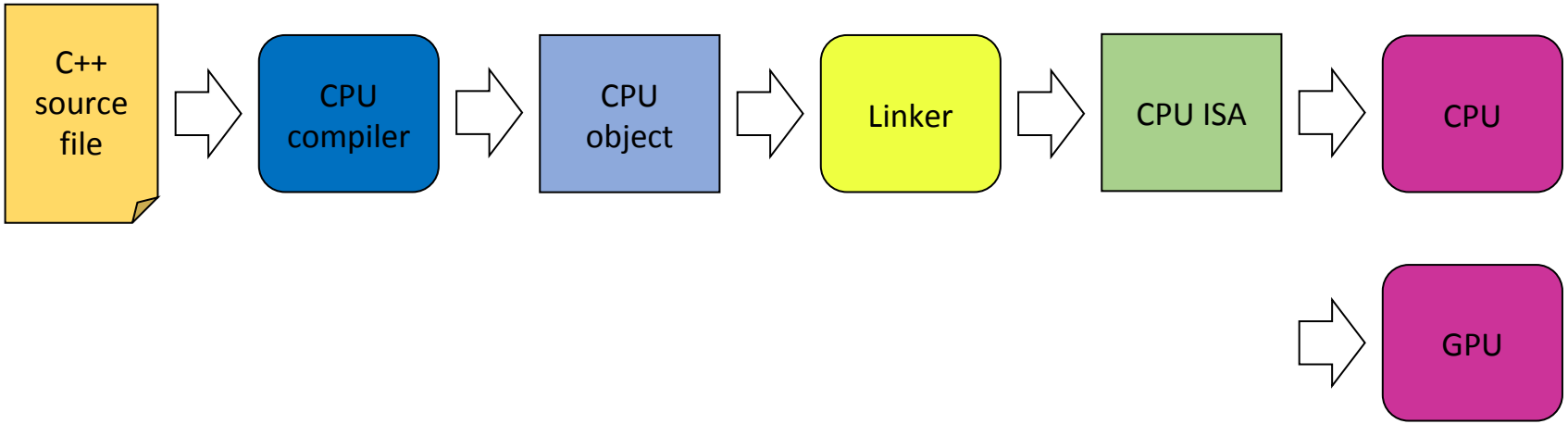
Std C++ compilation model



Std C++ compilation model

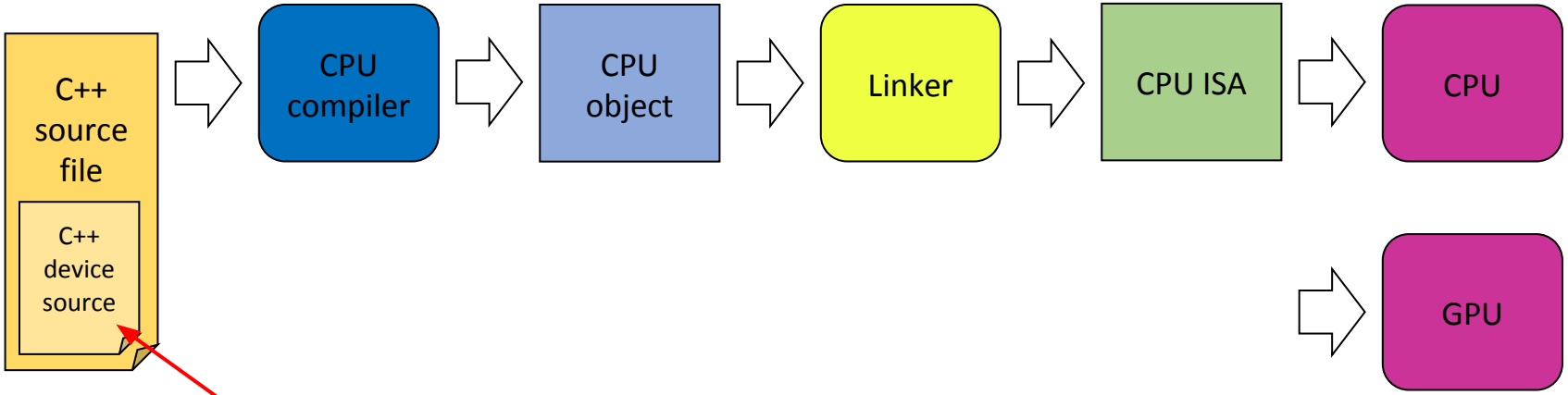


SYCL single source compilation model



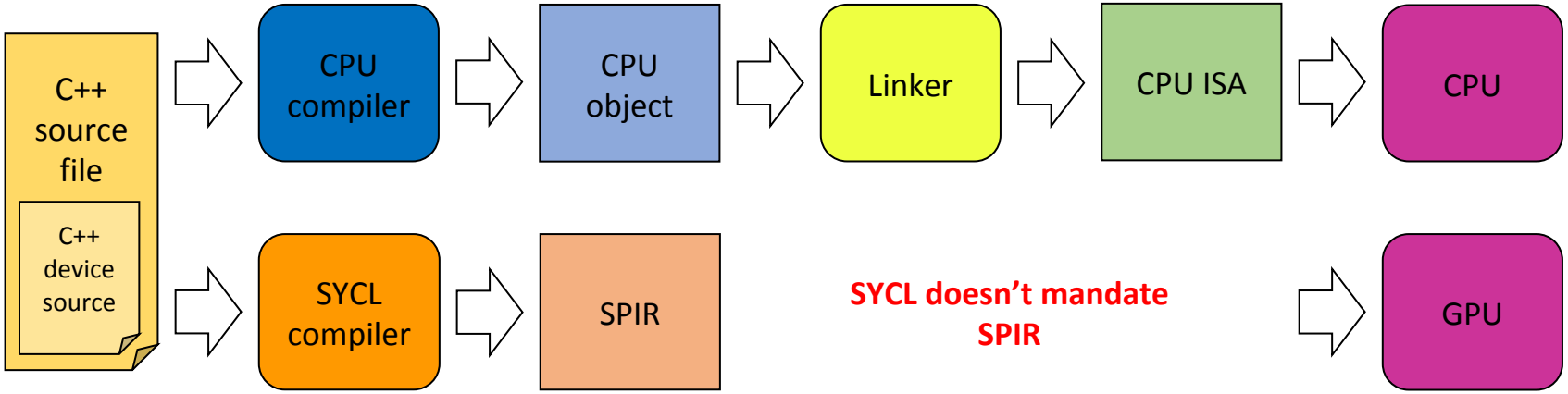

```
auto aAcc = aBuf.get_access<access::read>(cgh);
auto bAcc = bBuf.get_access<access::read>(cgh);
auto oAcc = oBuf.get_access<access::write>(cgh);
cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> idx) {
  oAcc[i] = aAcc[i] + aAcc[i];
}));
```

SYCL single source compilation model



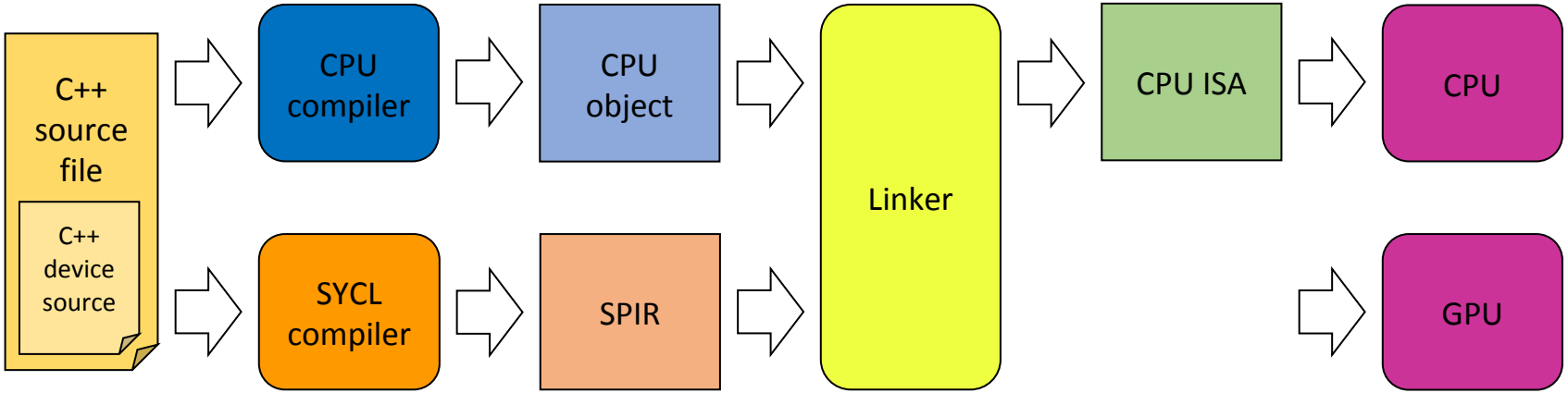

```
auto aAcc = aBuf.get_access<access::read>(cgh);
auto bAcc = bBuf.get_access<access::read>(cgh);
auto oAcc = oBuf.get_access<access::write>(cgh);
cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> idx) {
  oAcc[i] = aAcc[i] + aAcc[i];
}));
```

SYCL single source compilation model



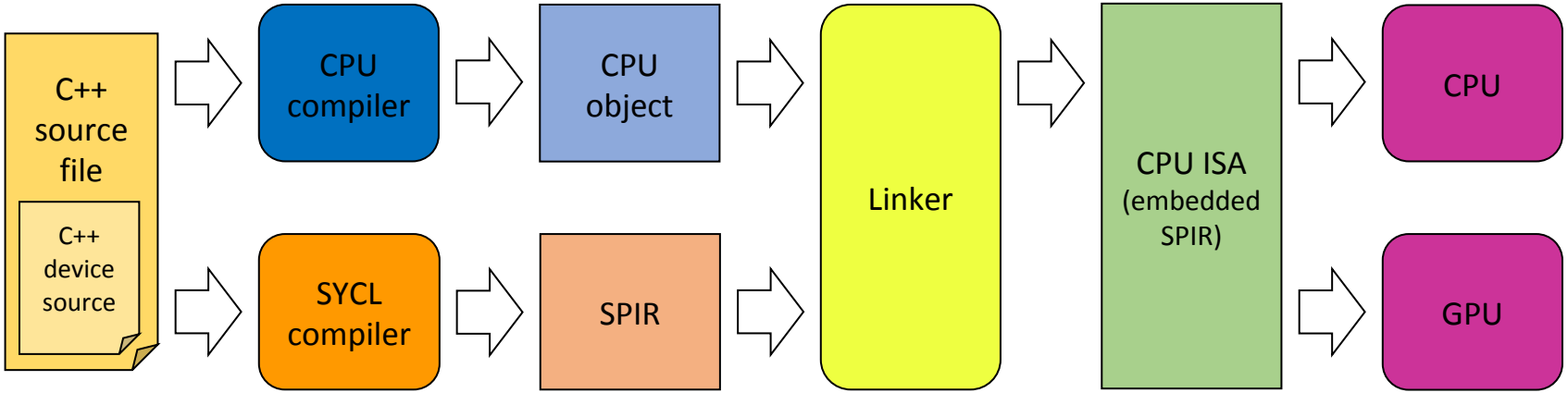

```
auto aAcc = aBuf.get_access<access::read>(cgh);
auto bAcc = bBuf.get_access<access::read>(cgh);
auto oAcc = oBuf.get_access<access::write>(cgh);
cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> idx) {
  oAcc[i] = aAcc[i] + aAcc[i];
}));
```

SYCL single source compilation model




```
auto aAcc = aBuf.get_access<access::read>(cgh);
auto bAcc = bBuf.get_access<access::read>(cgh);
auto oAcc = oBuf.get_access<access::write>(cgh);
cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> idx) {
  oAcc[i] = aAcc[i] + aAcc[i];
}));
```

SYCL single source compilation model



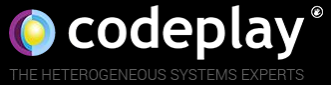

```
auto aAcc = aBuf.get_access<access::read>(cgh);
auto bAcc = bBuf.get_access<access::read>(cgh);
auto oAcc = oBuf.get_access<access::write>(cgh);
cgh.parallel_for<add>(range<1>(a.size()), [=](id<1> idx) {
  oAcc[i] = aAcc[i] + aAcc[i];
}));
```

Key takeaways

SYCL is a single source, cross platform, high-level programming model for heterogeneous systems

SYCL allows you to write programs for CPUs, GPUs and other heterogeneous devices using standard C++

SYCL uses a single source programming model where the code is compiled by both a “Host” compiler and a SYCL “Device” compiler


codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Exercise 3:
Implementing GPU Algorithms

- Implement the GPU variant of `transform`
- Evaluate the performance of the algorithms

Exercise document: https://goo.gl/bGfkZj

Exercise 3:
Implementing GPU Algorithms

```
1.  template <class ContiguousIt1, class ContiguousIt2, class UnaryOperation>
2.  ContiguousIt2 transform(sequential_execution_policy seq, ContiguousIt1 first, ContiguousIt1 last,
3.                          ContiguousIt2 d_first, UnaryOperation unary_op) {
4.
5.    /* implement me */
6.
7.  }
```

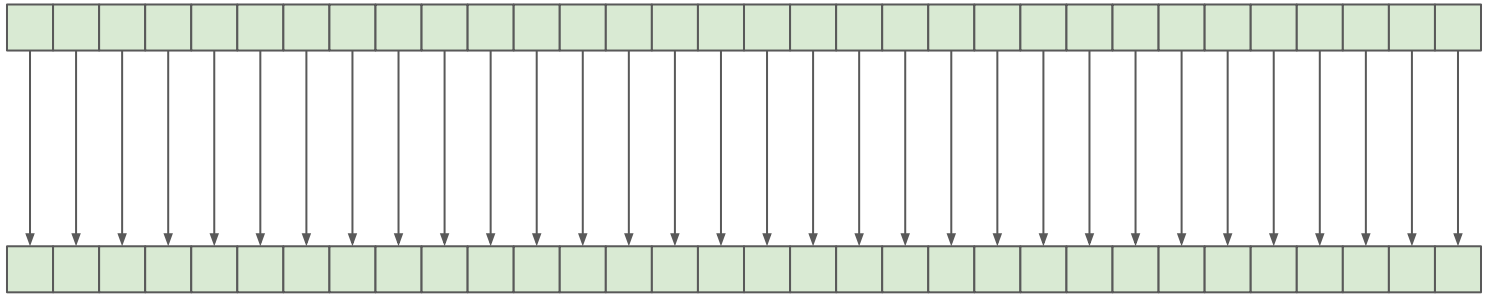
https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/gpu_transform.h

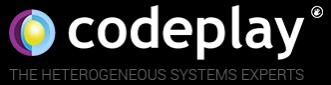
Exercise 3:
Implementing GPU Algorithms

So how do we parallelise transform on a GPU?

- Transform is embarrassingly parallel so there's no communication required

Exercise 3:
Implementing GPU Algorithms




codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Chapter 11: C++ for GPUs (SYCL) cont.

Prerequisite:

1. Previous chapters
2. Introduction to SYCL
3. Buffers and accessors
4. Separate host and device compilers
5. Single source programming model

You will learn:

1. How to select devices
2. How to handle errors and debug
3. How to manage memory
4. How to express parallelism
5. How to construct work-group barriers

Selecting a device to execute work on

In SYCL devices are selected using a device selector

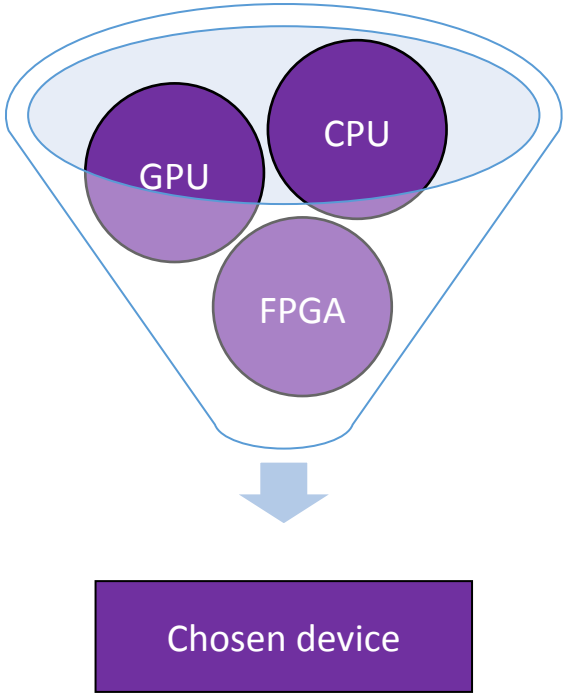
A device selector picks the best device based on a particular heuristic

Device selectors

Device selectors allow you to choose a device based on a custom heuristic

Evaluates all devices within a systems topology and scores them

For example: a GPU device from vendor X that supports double



Creating a device selector

Device selectors are C++ function objects whose function call operator takes a reference to a SYCL device

- The function object is called for each OpenCL device in the system and the host device

The function call operator returns a score for each device

- The device with the highest score is chosen, a device with a negative score is never chosen

Creating a device selector

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









int main(int argc, char *argv[]) {

  queue defaultQueue;

}
```

Creating a device selector

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









int main(int argc, char *argv[]) {

  queue defaultQueue;

}
```

A default constructed queue will have the SYCL runtime choose a device for you using the default selector

Creating a device selector

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

struct gpu_selector : public device_selector {

  int operator()(const device& d) const override {









  }
};

int main(int argc, char *argv[]) {

  queue defaultQueue;

}
```

A device selector must inherit from the default selector

A device selector must hav a function call operator which takes a reference to a device

Creating a device selector

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

struct gpu_selector : public device_selector {

  int operator()(const device& d) const override {

    if (d.is_gpu()){
      return 1;
    }
    else {
      return -1;
    }
  }
};

int main(int argc, char *argv[]) {

  queue defaultQueue;

}
```

The body of the function call operator defines the heuristic for selecting devices

Here we specify we want a GPU device

Creating a device selector

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

struct gpu_selector : public device_selector {

  int operator()(const device& d) const override {

    if (d.is_gpu()){
      return 1;
    }
    else {
      return -1;
    }
  }
};

int main(int argc, char *argv[]) {

  queue gpuQueue(gpu_selector{});

}
```

If you construct a queue from the device selector type then it will use that to select it's device

Here we construct a queue from the GPU selector

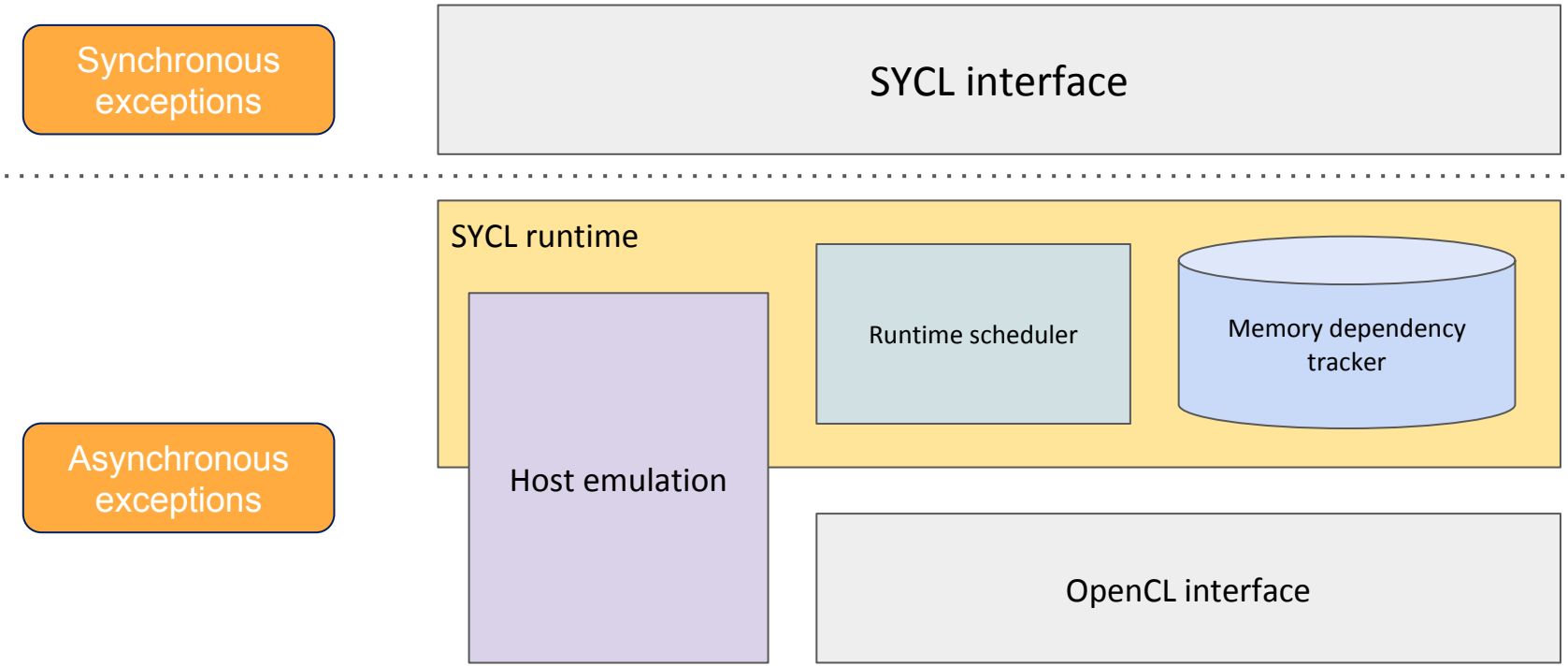
How do you handle errors in SYCL?

In SYCL it’s crucial that you handle errors, otherwise you won’t know if something has gone wrong in your application

SYCL errors are handled by throwing exceptions

There are two kinds of SYCL errors; synchronous exceptions and asynchronous exceptions

Handling errors



Handling errors

```
void parallel_add(std::vector<float> &a, std::vector<float> &b,
                  std::vector<float> &o) {
  using namespace cl::sycl;


    queue defaultQueue;



    /* create buffers */

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

      /* create accessors */

      /* launch kernel */
    });




}
```

Handling errors

```
void parallel_add(std::vector<float> &a, std::vector<float> &b,
                  std::vector<float> &o) {
  using namespace cl::sycl;

  try {
    queue defaultQueue;



    /* create buffers */

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

      /* create accessors */

      /* launch kernel */
    });


  } catch (exception e) {
    std::cout << “exception caught: ” + e.what() << “\n”;
  }
}
```

To handle synchronous exceptions you must everything in a try-catch block

Handling errors

```
void parallel_add(std::vector<float> &a, std::vector<float> &b,
                  std::vector<float> &o) {
  using namespace cl::sycl;

  try {
    queue defaultQueue([=](exception_list eL) {
      for (auto e : eL) { std::rethrow_exception(e); }
    });

    /* create buffers */

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

      /* create accessors */

      /* launch kernel */
    });

    defaultQueue.wait_and_throw();
  } catch (exception e) {
    std::cout << “exception caught: ” + e.what() << “\n”;
  }
}
```

To handle asynchronous exceptions you must also provide an async handler and instruct the queue to throw

How do you debug a SYCL kernel?

SYCL kernels can be debugged by executing them on the host device

The host device emulates the SYCL programming model meaning you can debug kernel code using a standard C++ debugger

Debugging a SYCL kernel

```
#include <vector>
#include <CL/sycl.hpp>

class add;

void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> aBuf(a.data(), range<1>(dA.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(dB.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(dO.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(range<1>(dA.size()), [=](id<1> i) {
      oAcc[i] = aAcc[i] + bAcc[i];
    });
  });
}
```

Debugging a SYCL kernel

```
#include <vector>
#include <CL/sycl.hpp>

class add;

void parallel_add(std::vector<float> &a, std::vector<float> &b, std::vector<float> &o) {
  using namespace cl::sycl;

  queue hostQueue(host_selector{});

  buffer<float, 1> aBuf(a.data(), range<1>(dA.size()));
  buffer<float, 1> bBuf(b.data(), range<1>(dB.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(dO.size()));

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

    auto aAcc= aBuf.get_access<access::mode::read>(cgh);
    auto bAcc = bBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(range<1>(dA.size()), [=](id<1> i) {
      oAcc[i] = aAcc[i] + bAcc[i];
    });
  });
}
```

Creating a queue with the host selector will result in the queue only executing on the host device

How do you log text from a kernel?

For logging from a kernel function SYCL provides a buffered ostream

Logging from a SYCL kernel

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

int main(int argc, char *argv[]) {

  queue defaultQueue;
  {
    defaultQueue.submit([&](handler &cgh){



      cgh.parallel_for<class add>(range<1>(dA.size()), [=](id<1> idx){



      });
    });
  }
}
```

Logging from a SYCL kernel

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

int main(int argc, char *argv[]) {

  queue defaultQueue;
  {
    defaultQueue.submit([&](handler &cgh){

      stream os(1024, 128, cgh);

      cgh.parallel_for<class add>(range<1>(dA.size()), [=](id<1> idx){

        os << idx << ": hello world!\n";

      });
    });
  }
}
```

Stream object can be constructed similarly to an accessor

Stream objects can be used to log out using stream operators

Logging from a SYCL kernel

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

int main(int argc, char *argv[]) {

  queue defaultQueue;
  {
    defaultQueue.submit([&](handler &cgh){

      stream os(1024, 128, cgh);

      cgh.parallel_for<class add>(range<1>(dA.size()), [=](id<1> idx){

        os << idx << ": hello world!\n";

      });
    });
  }
}
```

Streamed text is output to the console when the kernel completes

```
{0}: hello world!
{1}: hello world!
{2}: hello world!
{3}: hello world!
{4}: hello world!
{5}: hello world!
{6}: hello world!
{7}: hello world!
{8}: hello world!
{9}: hello world!
…
```

Types supported by streams

Supported types

- Fundamental C++ types
- Pointer types
- Vector types
- Range and id types

Supported modifiers

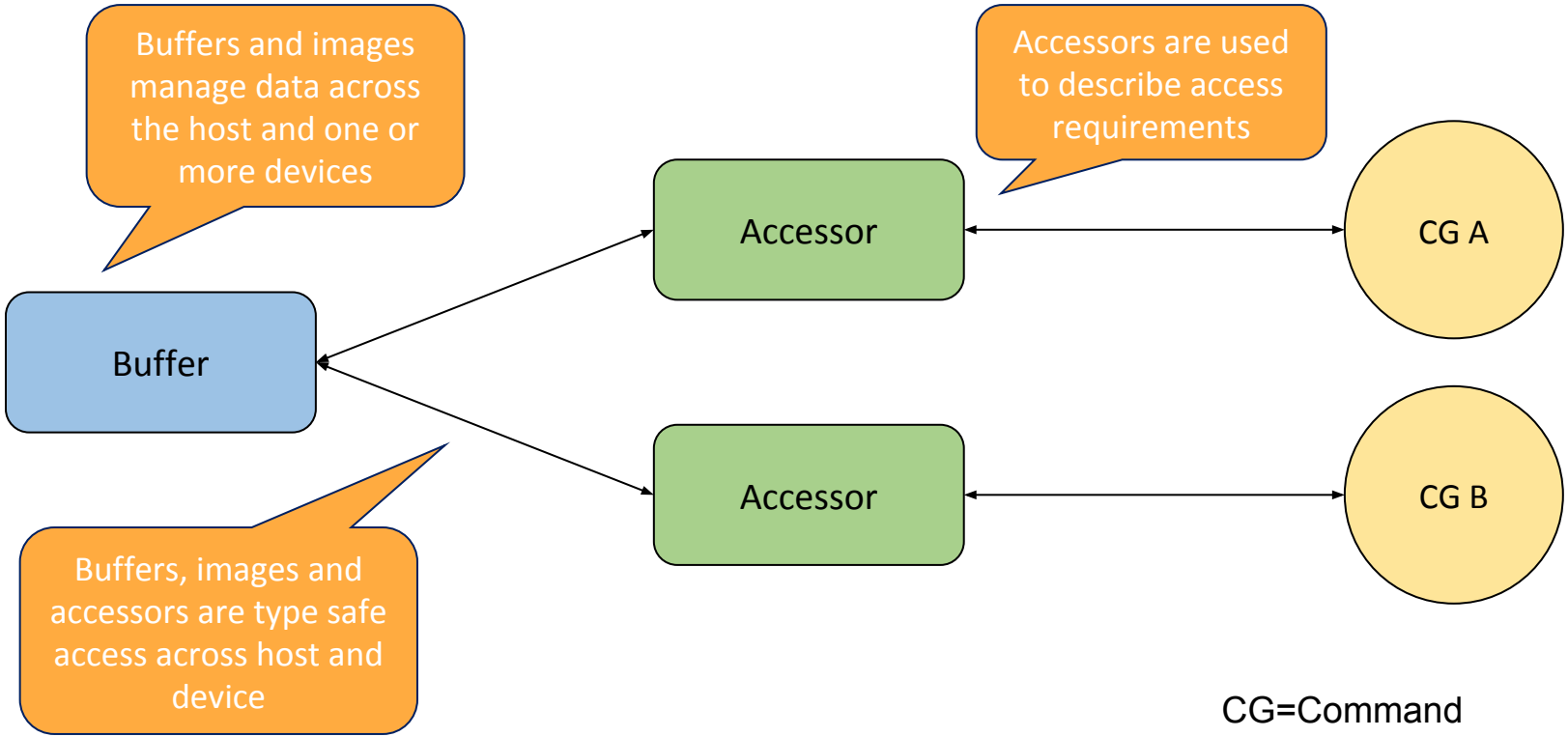
- setprecision(int), setw(int)
- showbase, showpos, hex, oct, scientific, fixed, hexfloat, defaultfloat, endl

How you manage memory in SYCL?

SYCL separates the storage and access of data through the use of buffers and accessors

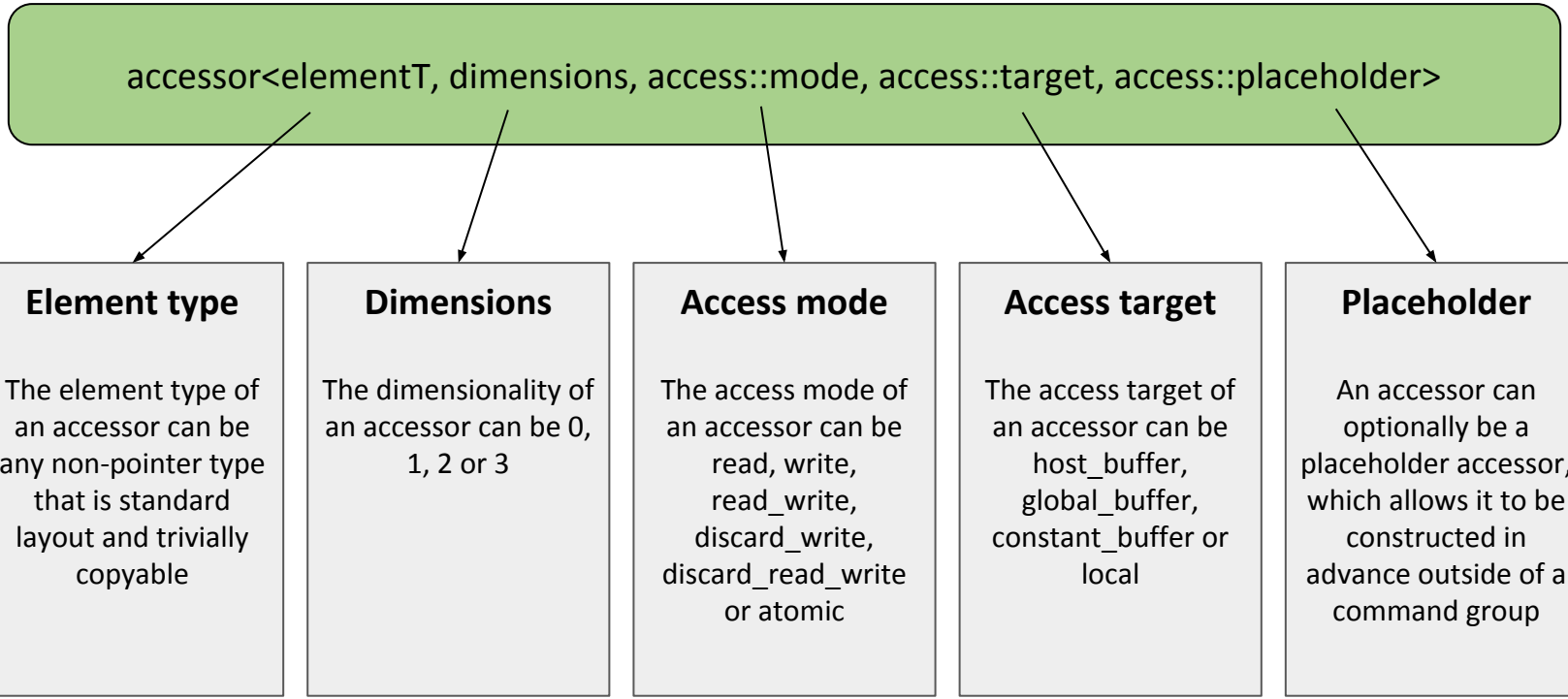
SYCL provides data dependency tracking based on accessors to optimise the scheduling of tasks

Separating storage and access

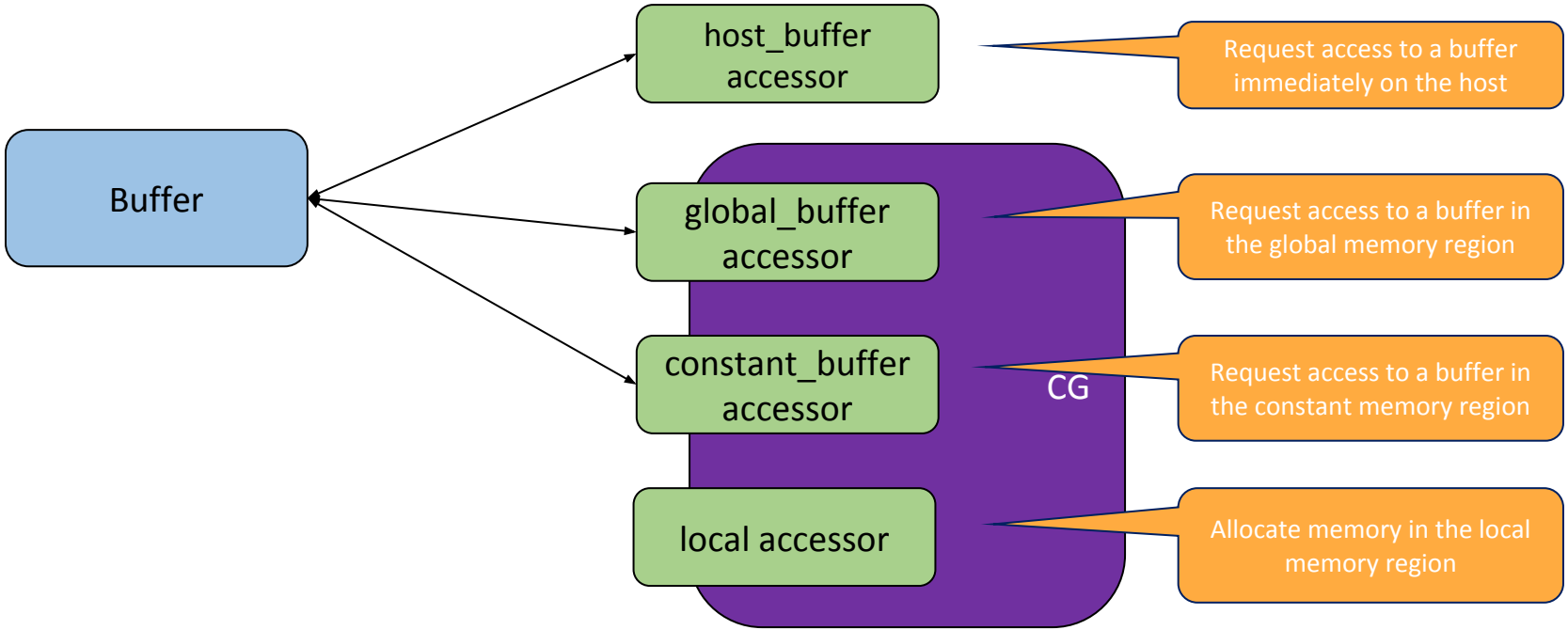


CG=Command Group/A kernel

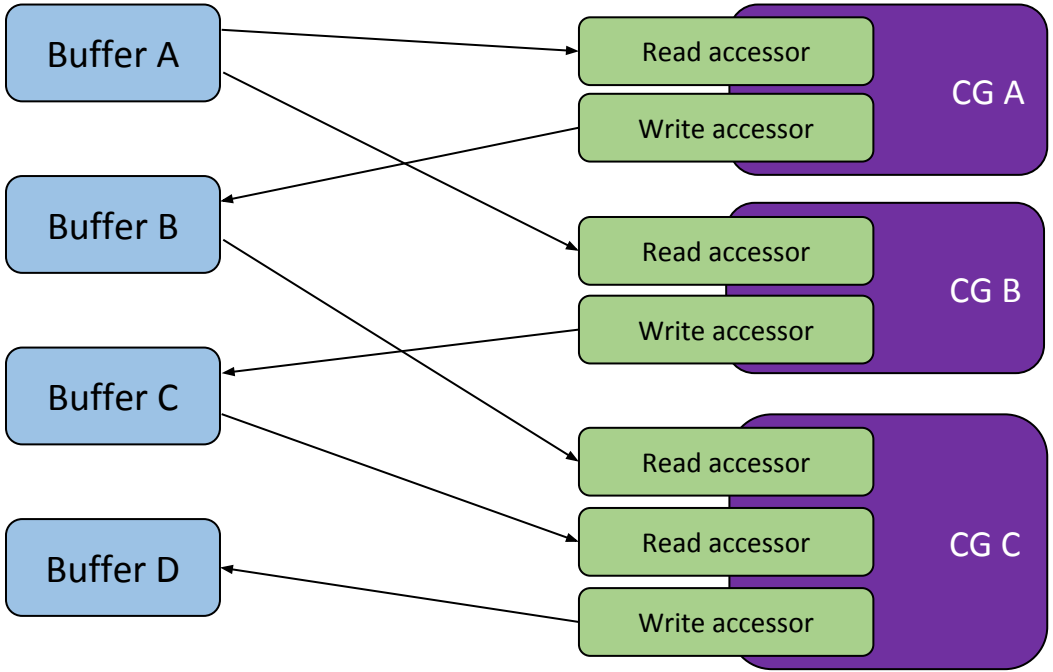
Anatomy of an accessor

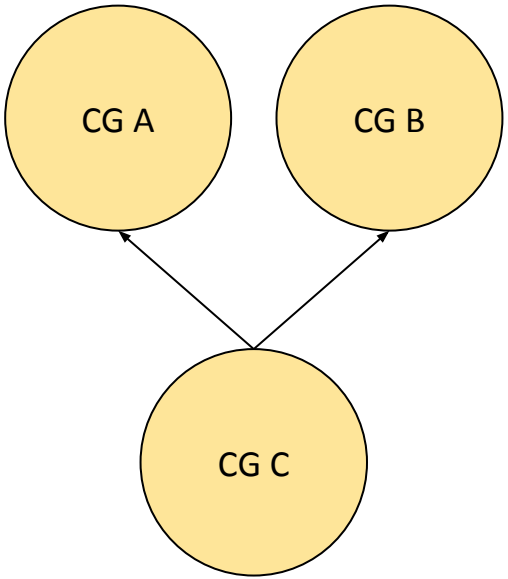


Accessing memory in different regions



Data dependency task graphs





Benefits of data dependency graphs

Allows you to describe your problems in terms of relationships

- Removes the need to en-queue explicit copies
- Removes the need for complex event handling

Allows the runtime to make data movement optimizations

- Preemptively copy data to a device before kernels
- Avoid unnecessarily copying data back to the host after execution on a device
- Avoid copies of data that you don’t need

Expressing parallelism in SYCL

In SYCL there are several ways to launch kernels that express parallelism in different ways

Different forms of parallelism

```
cgh.single_task<T>([=](){

  /* task executed by a single
     work item */

});
```

```
cgh.parallel_for<T>(range<2>(64, 64),
                    [=](id<2> idx){

  /* data parallel work executed
     across a range */

});
```

```
cgh.parallel_for_work_group(range<2>(64, 64),
                            [=](group<2> gp){

  /* data parallel work executed once
     per work group */

  parallel_for_work_item(gp, [=](h_item<2> it){

    /* data parallel work executed once
       per work item */

  });

});
```

Single task

```
cgh.single_task<add>([=](){
  for (int idx = 0; idx < 1024; idx++) {
    oAcc[idx] = aAcc[idx] + bAcc[idx];
  }
});
```

The single_task function launches a single work-item

Parallel for

```
cgh.parallel_for<add>(range<2>(32, 32), [=](id<2> idx) {
  oAcc[idx] = aAcc[idx] + bAcc[idx];
});
```

The parallel_for functions launch a range of work-items in parallel

The first variant takes a range and expects a kernel function which receives an id

Parallel for

```
cgh.parallel_for<add>(range<2>(32, 32), [=](id<2> idx) {
  oAcc[idx] = aAcc[idx] + bAcc[idx];
});
```

The range describes the total number of work-items to be launched

The id specifies the current work-item being executed

Parallel for

```
cgh.parallel_for<add>(range<2>(32, 32),
                      [=](item<2> item) {
  size_t rng = item.get_range();
  size_t idx = item.get_id();
  oAcc[idx] = aAcc[idx] + bAcc[idx];
});
```

The second variant of the parallel_for function takes a range and expects a kernel function which receives an item

Parallel for

```
cgh.parallel_for<add>(range<2>(32, 32),
                      [=](item<2> item) {
  size_t rng = item.get_range();
  size_t idx = item.get_id();
  oAcc[idx] = aAcc[idx] + bAcc[idx];
});
```

The range describes the total number of work-items to be launched

The item specifies the range of work-items and the current work-item being executed

Parallel for

```
auto ndRange = nd_range<2>(range<2>(32, 32), range<2>(8, 4));

cgh.parallel_for<class add>(ndRange, [=](nd_item<2> ndItem){
  size_t globalRange = ndItem.get_global_range();
  size_t localRange = ndItem.get_local_range();
  size_t globalId = ndItem.get_global_id();
  size_t localId = ndItem.get_local_id();
  size_t groupId = ndItem.get_group();
  oAcc[globalId] = aAcc[globalId] + bAcc[globalId];
});
```

The third variant of the parallel_for function takes an nd_range and expects a kernel function which receives an nd_item

Parallel for

```
auto ndRange = nd_range<2>(range<2>(32, 32), range<2>(8, 4));

cgh.parallel_for<add>(ndRange, [=](nd_item<2> ndItem){
  size_t globalRange = ndItem.get_global_range();
  size_t localRange = ndItem.get_local_range();
  size_t globalId = ndItem.get_global_id();
  size_t localId = ndItem.get_local_id();
  size_t groupId = ndItem.get_group();
  oAcc[globalId] = aAcc[globalId] + bAcc[globalId];
});
```

The range describes the total number of work-items to be launched and the number of work-items in each work-group

Parallel for

```
auto ndRange = nd_range<2>(range<2>(32, 32), range<2>(8, 4));

cgh.parallel_for<add>(ndRange, [=](nd_item<2> ndItem){
  size_t globalRange = ndItem.get_global_range();
  size_t localRange = ndItem.get_local_range();
  size_t globalId = ndItem.get_global_id();
  size_t localId = ndItem.get_local_id();
  size_t groupId = ndItem.get_group();
  oAcc[globalId] = aAcc[globalId] + bAcc[globalId];
});
```

The nd_item specifies the range of work-items and work-groups and the current work-item being executed (relative to the full range of work items and to the current work-group)

Parallel for

```
auto ndRange = nd_range<2>(range<2>(32, 32), range<2>(8, 4));

cgh.parallel_for<add>(ndRange, [=](nd_item<2> ndItem){

  auto temp = oAcc[globalId];

  ndItem.barrier(access::memory_fence::global_and_local);

  oAcc[globalId + 1] = temp;
});
```

The nd_item also provides synchronisation primitives such as barriers

Work-group barriers

In SYCL you can insert work-group barriers using the `barrier` member function of the `nd_item` class

Using local memory and barriers

```
#include <vector>
#include <CL/sycl.hpp>

class reverse;

void parallel_reverse(std::vector<float> &i, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> iBuf(i.data(), range<1>(i.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto iAcc= iBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);



    cgh.parallel_for<reverse>(nd_range<1>(range<1>(i.size()), rane<1>(128)), [=](nd_item<1> item) {



    });
  });
}
```

Here we have a SYCL kernel which takes input and output global accessors

We specify an explicit work group size of 128

Using local memory and barriers

```
#include <vector>
#include <CL/sycl.hpp>

class reverse;

void parallel_reverse(std::vector<float> &i, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> iBuf(i.data(), range<1>(i.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto iAcc= iBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    accessor<float, 1, access::mode::read_write, access::target::local> scratch(128, cgh);

    cgh.parallel_for<reverse>(nd_range<1>(range<1>(i.size()), rane<1>(128)), [=](nd_item<1> item) {



    });
  });
}
```

A local accessor allocates an amount of local memory per work-group

We create a local accessor of 128 elements, one for each work-item in the work-group

Using local memory and barriers

```
#include <vector>
#include <CL/sycl.hpp>

class reverse;

void parallel_reverse(std::vector<float> &i, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> iBuf(i.data(), range<1>(i.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto iAcc= iBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    accessor<float, 1, access::mode::read_write, access::target::local> scratch(128, cgh);

    cgh.parallel_for<reverse>(nd_range<1>(range<1>(i.size()), rane<1>(128)), [=](nd_item<1> item) {
      scratch[item.get_local_id(0)] = iAcc[item.get_global_id(0)];


    });
  });
}
```

In the SYCL kernel we read from the element of global memory for the current work-item and write it to the element of local memory for the curren work-item

Using local memory and barriers

```
#include <vector>
#include <CL/sycl.hpp>

class reverse;

void parallel_reverse(std::vector<float> &i, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> iBuf(i.data(), range<1>(i.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto iAcc= iBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    accessor<float, 1, access::mode::read_write, access::target::local> scratch(128, cgh);

    cgh.parallel_for<reverse>(nd_range<1>(range<1>(i.size()), rane<1>(128)), [=](nd_item<1> item) {
      scratch[item.get_local_id(0)] = iAcc[item.get_global_id(0)];
      item.barrier(access::fence_space::local_space);

    });
  });
}
```

We insert a work-group barrier to ensure all work-items in each work-group have copied before moving on

Using local memory and barriers

```
#include <vector>
#include <CL/sycl.hpp>

class reverse;

void parallel_reverse(std::vector<float> &i, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> iBuf(i.data(), range<1>(i.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto iAcc= iBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    accessor<float, 1, access::mode::read_write, access::target::local> scratch(128, cgh);

    cgh.parallel_for<reverse>(nd_range<1>(range<1>(i.size()), rane<1>(128)), [=](nd_item<1> item) {
      scratch[item.get_local_id(0)] = iAcc[item.get_global_id(0)];
      item.barrier(access::fence_space::local_space);
      oAcc[item.get_global_id(0)] = 128 - scratch[item.get_local_id(0)];
    });
  });
}
```

We read from the element of local memory for the current work-item, reverse it by subtracting it from 128, and write it to the element of global memory for the curren work-item

Using local memory and barriers

```
#include <vector>
#include <CL/sycl.hpp>

class reverse;

void parallel_reverse(std::vector<float> &i, std::vector<float> &o) {
  using namespace cl::sycl;

  queue defaultQueue;

  buffer<float, 1> iBuf(i.data(), range<1>(i.size()));
  buffer<float, 1> oBuf(o.data(), range<1>(o.size()));

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

    auto iAcc= iBuf.get_access<access::mode::read>(cgh);
    auto oAcc = oBuf.get_access<access::mode::write>(cgh);

    accessor<float, 1, access::mode::read_write, access::target::local> scratch(128, cgh);

    cgh.parallel_for<reverse>(nd_range<1>(range<1>(i.size()), rane<1>(128)), [=](nd_item<1> item) {
      scratch[item.get_local_id(0)] = iAcc[item.get_global_id(0)];
      item.barrier(access::fence_space::local_space);
      oAcc[item.get_global_id(0)] = 128 - scratch[item.get_local_id(0)];
    });
  });
}
```

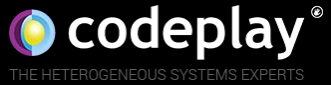
This SYCL kernel reverses all elements within each work-group

Key takeaways

The SYCL API allows you to discover devices, manage memory and express parallelism

SYCL allows you to describe dependencies to your kernel functions and automatically optimises kernel scheduling and data movement

SYCL allows you to express parallelism in many different ways


codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

Exercise 4:
Implementing GPU Algorithms cont.

- Implement the GPU variant of `reduce`
- Implement the GPU variant of `transform_reduce`
- Evaluate the performance of the algorithms

Exercise document: https://goo.gl/5j5DyN

Exercise 4:
Implementing GPU Algorithms cont.

```
1.  template <class ContiguousIt, class T, class BinaryOperation, class UnaryOperation>
2.  T reduce(sequential_execution_policy seq, ContiguousIt first, ContiguousIt last,
3.           T init, BinaryOperation binary_op) {
4.
5.    /* implement me */
6.
7.  }
```

https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/gpu_reduce.h

Exercise 4:
Implementing GPU Algorithms cont.

```
1.  template <class ContiguousIt, class T, class BinaryOperation>
2.  T transform_reduce(sequential_execution_policy seq, ContiguousIt first, ContiguousIt last,
3.                     T init, BinaryOperation binary_op, UnaryOperation unary_op) {
4.
5.    /* implement me */
6.
7.  }
```

https://github.com/AerialMantis/cppcon2018-parallelism-class/blob/master/include/bits/gpu_transform_reduce.h

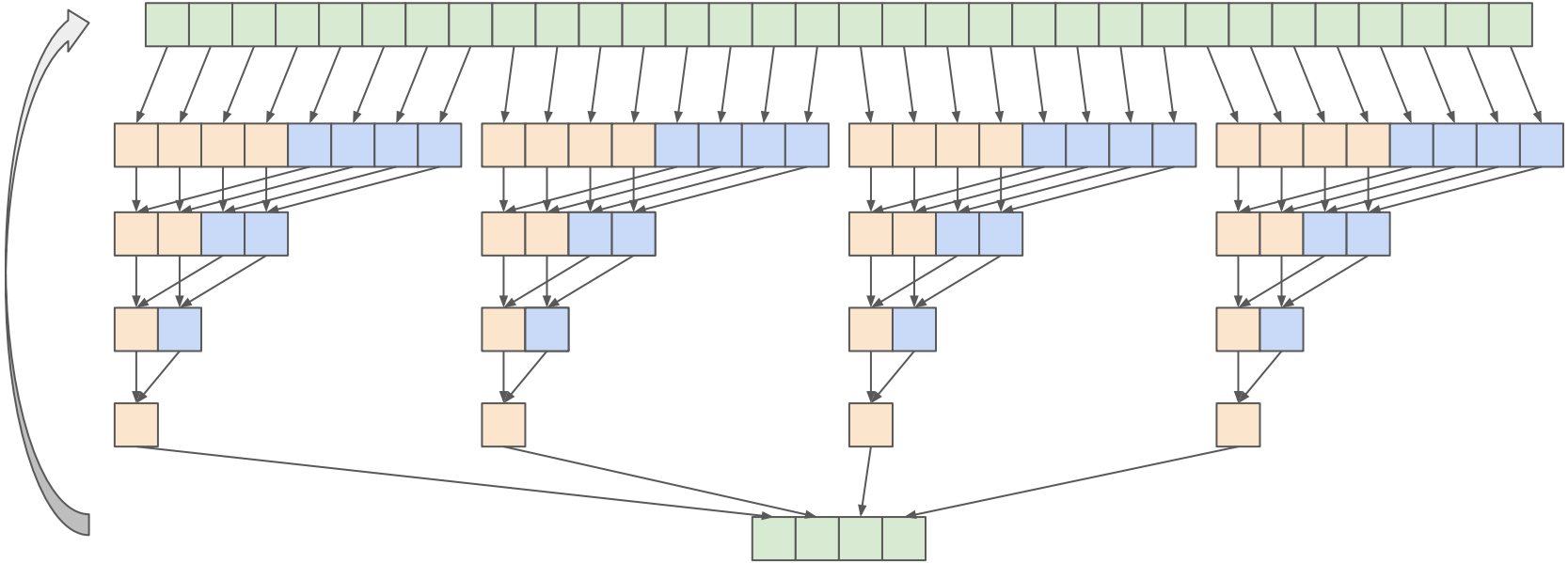
Exercise 4:
Implementing GPU Algorithms cont.

So how do we parallelise reduce on a GPU?

- We want to utilize the available hardware
- We want to keep dependencies to a minimum
- We want to make efficient use of local memory and work-group synchronization

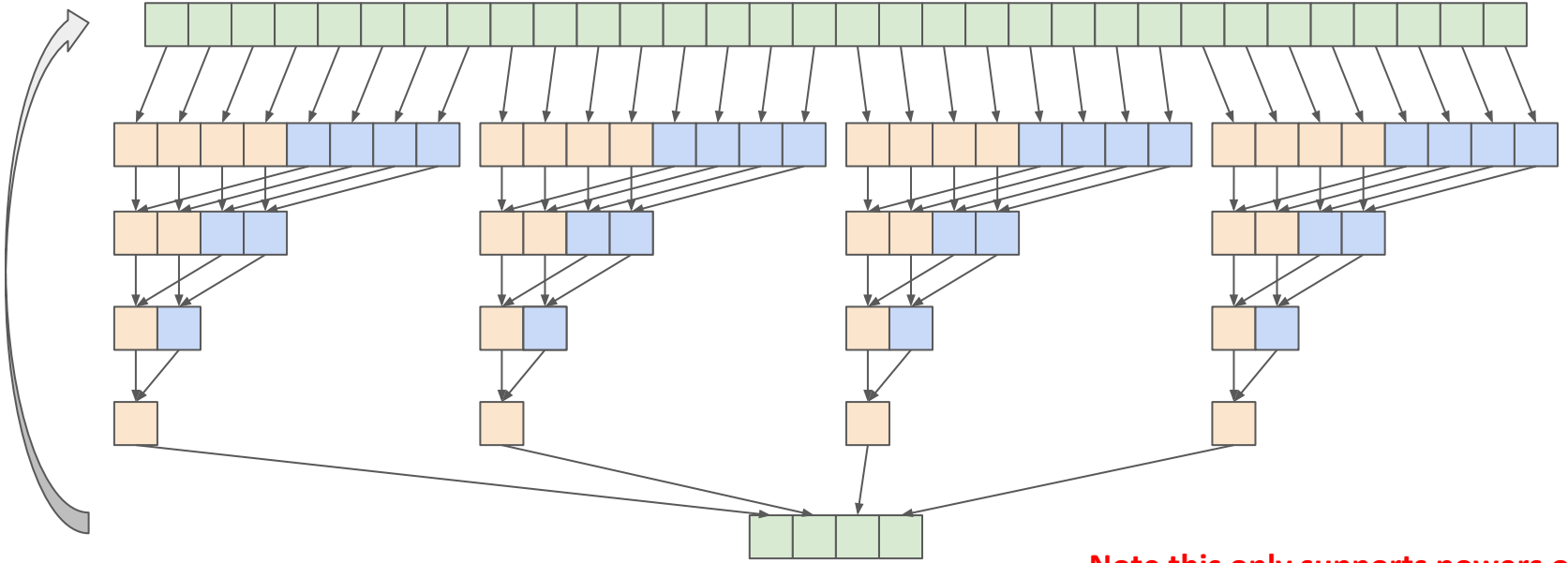
Exercise 4:

Implementing GPU Algorithms cont.

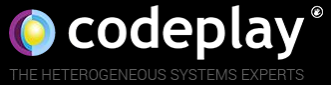


Exercise 4:

Implementing GPU Algorithms cont.



Note this only supports powers of 2


codeplay
THE HETEROGENEOUS SYSTEMS EXPERTS

SYCL Ecosystem

- ComputeCpp - https://codeplay.com/products/computesuite/computecpp
- triSYCL - https://github.com/triSYCL/triSYCL
- SYCL - http://sycl.tech
- SYCL ParallelSTL - https://github.com/KhronosGroup/SyclParallelSTL
- VisionCpp - https://github.com/codeplaysoftware/visioncpp
- SYCL-BLAS - https://github.com/codeplaysoftware/sycl-blas
- TensorFlow-SYCL - https://github.com/codeplaysoftware/tensorflow
- Eigen http://eigen.tuxfamily.org

References

In-depth multithreading and lock-free programming from my mentor: a free book too: Is Parallel Programming Hard, And If So, What Can You Do About It?

http://www.rdrop.com/~paulmck/

C++ concurrency in action: C++ specific
https://www.amazon.com/dp/1933988770/?tag=googhydr-20&hvadid=177208251545&hvpos=1t1&hvnetw=g&hvrand=15339107147013507384&hvpone=&hvptwo=&hvqmt=b&hvdev=c&hvdvcmdl=&hvlocint=&hvlocphy=9060230&hvtargid=aud-509611686427:kwd-13530201978&ref=pd_sl_2223bpmrge_b

Art of Multiprocessor Programming: classic general text now by the father of multiprocessor computing

https://www.amazon.com/Art-Multiprocessor-Programming-Revised-Reprint/dp/0123973376/ref=sr_1_1?s=books&ie=UTF8&qid=1538328077&sr=1-1&keywords=art+of+multiprocessor+programming

SYCL reference card

https://www.khronos.org/files/sycl/sycl-12-reference-card.pdf

Heterogeneous Computing with OpenCL

https://www.amazon.com/Heterogeneous-Computing-OpenCL-David-Kaeli/dp/0128014148

More References

Lots of books on CUDA

Heterogeneous System Architecture: a good book on a low latency AMD-led specification

https://www.amazon.com/Heterogeneous-System-Architecture-Platform-Infrastructure-ebook/dp/B019K7CMOK

One of the first common scientific programs to run faster on GPUs than CPUs was an implementation of LU factorization

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.193.7712&rep=rep1&type=pdf

Three 2016 simd talks:

https://www.youtube.com/watch?v=2HsLsTRxfbA

https://www.youtube.com/watch?v=h6Q-5Q2N5ck

https://www.youtube.com/watch?v=UgaQCg-0ZoU

And one in 2018 that we saw and liked:

https://cppcon2018.sched.com/event/FnKL/compute-more-in-less-time-using-c-simd-wrapper-libraries

Still want more?

Join the Khronos SYCL Standardization effort (by your company, or as an advisor to us, or adopter to build a product)

https://www.khronos.org/

Join ISO C++ SG14 Low Latency/Games/Financial/Embeded

https://groups.google.com/a/isocpp.org/forum/#!forum/sg14

Join ISO C++ SG14/SG1 Heterogeneous C++ call

https://groups.google.com/forum/#!forum/hetero-c

Contact us

michael@codeplay.com

gordon@codeplay.com