codeplay®

THE HETEROGENEOUS SYSTEMS EXPERTS

A Modern C++ Programming Model for GPUs using Khronos SYCL

Michael Wong, Gordon Brown

ACCU 2018

VP of R&D of Codeplay

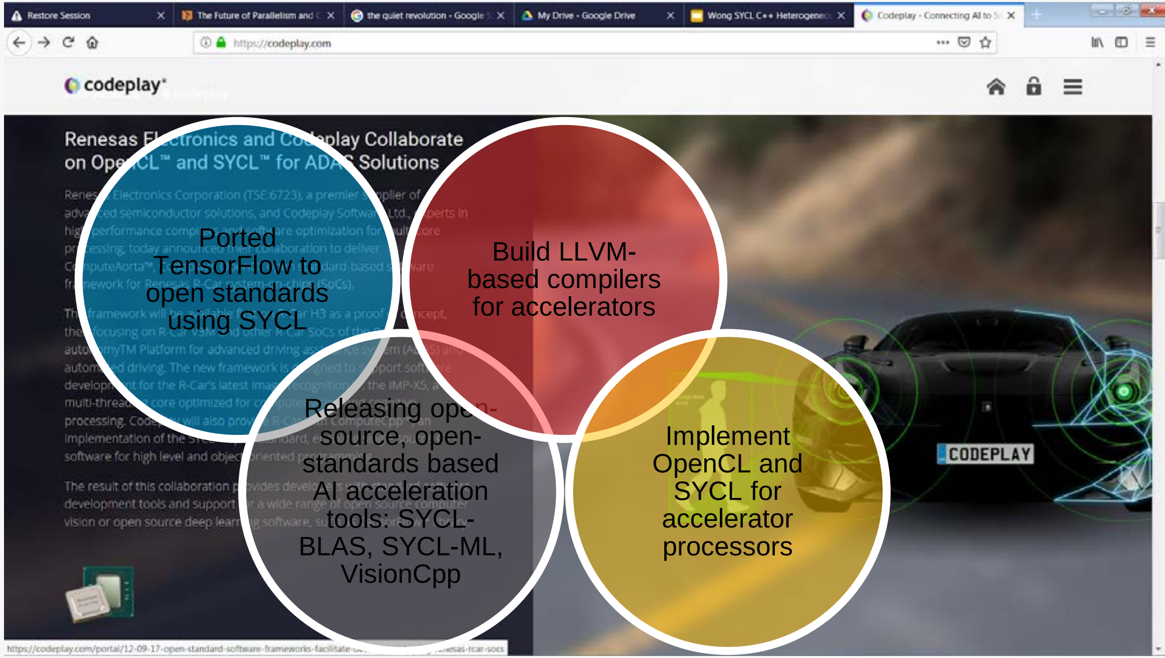
Chair of SYCL Heterogeneous Programming Language
C++ Directions Group
ISOCPP.org Director, VP
http://isocpp.org/wiki/faq/wg21#michael-wong

Head of Delegation for C++ Standard for Canada
Chair of Programming Languages for Standards Council of Canada
Chair of WG21 SG19 Machine Learning
Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
Editor: C++ SG5 Transactional Memory Technical Specification
Editor: C++ SG1 Concurrency Technical Specification
MISRA C++ and AUTOSAR
wongmichael.com/about

We build GPU compilers for semiconductor companies

- **Now working to make AI/MI heteroegneous acceleration safe for autonomous vehicle**

Who am I? Who are we?


Restore Session
The Future of Parallelism and
the quiet revolution - Google
My Drive - Google Drive
Wong SYCL C++ Heterogeneo
Codeplay - Connecting AI to S
https://codeplay.com
codeplay
CODEPLAY

Renesas Electronics and Codeplay Collaborate on OpenCL™ and SYCL™ for ADAS Solutions

Ported TensorFlow to open standards using SYCL

Build LLVM-based compilers for accelerators

Releasing open-source, open-standards based AI acceleration tools: SYCL-BLAS, SYCL-ML, VisionCpp

Implement OpenCL and SYCL for accelerator processors

https://codeplay.com/portal/12-09-17-open-standard-software-frameworks-facilitate-...renesas-rcar-socs

Gordon Brown

- Background in C++ programming models for heterogeneous systems
- Developer with Codeplay Software for 6 years
- Worked on ComputeCpp (SYCL) since it’s inception
- Contributor to the Khronos SYCL standard for 6 years
- Contributor to C++ executors and heterogeneity or 2 years

Acknowledgement Disclaimer

Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.

Specifically, Paul Mckenney, Joe Hummel, Bjarne Stroustru, Botond Ballo for some of the slides.

I even lifted this acknowledgement and disclaimer from some of them.

But I claim all credit for errors, and stupid mistakes. **These are mine, all mine!**

Legal Disclaimer

THIS WORK REPRESENTS THE VIEW OF THE AUTHOR AND DOES NOT NECESSARILY REPRESENT THE VIEW OF CODEPLAY.

OTHER COMPANY, PRODUCT, AND SERVICE NAMES MAY BE TRADEMARKS OR SERVICE MARKS OF OTHERS.

Codeplay - Connecting AI to Silicon

Products

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR™, HSA™ and Vulkan™

Company

High-performance software solutions for custom heterogeneous systems

Enabling the toughest processor systems with tools and middleware based on open standards

Established 2002 in Scotland

~70 employees

Addressable Markets

Automotive (ISO 26262)
IoT, Smartphones & Tablets
High Performance Compute (HPC)
Medical & Industrial

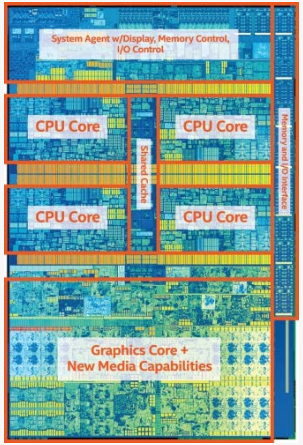
Technologies: Vision Processing
Machine Learning
Artificial Intelligence
Big Data Compute

Customers

BROADCOM. RENESAS
imagination
QUALCOMM Movidius

Partners
intel AMD

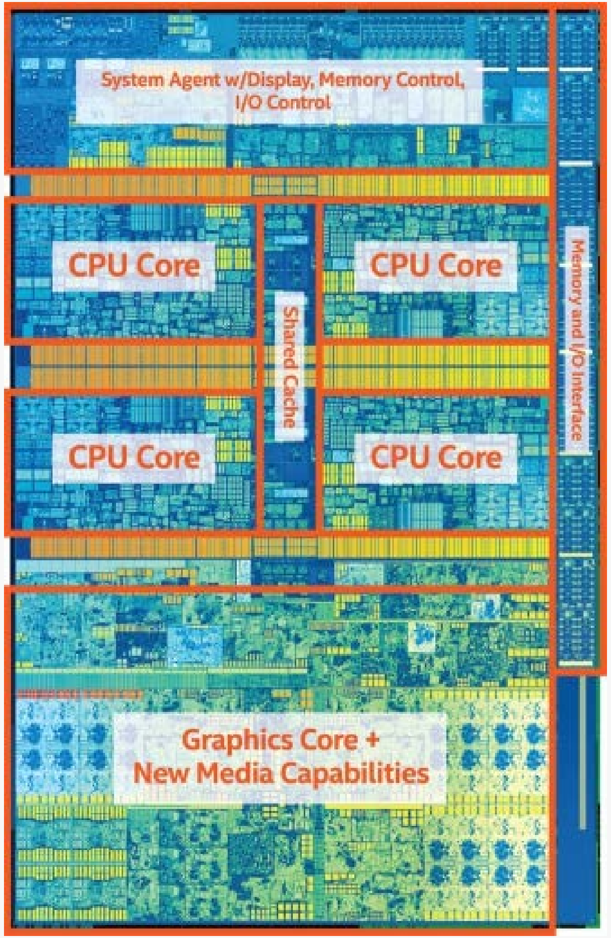
3 Act Play


System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
Memory and I/O Interface
CPU Core
CPU Core
Graphics Core + New Media Capabilities

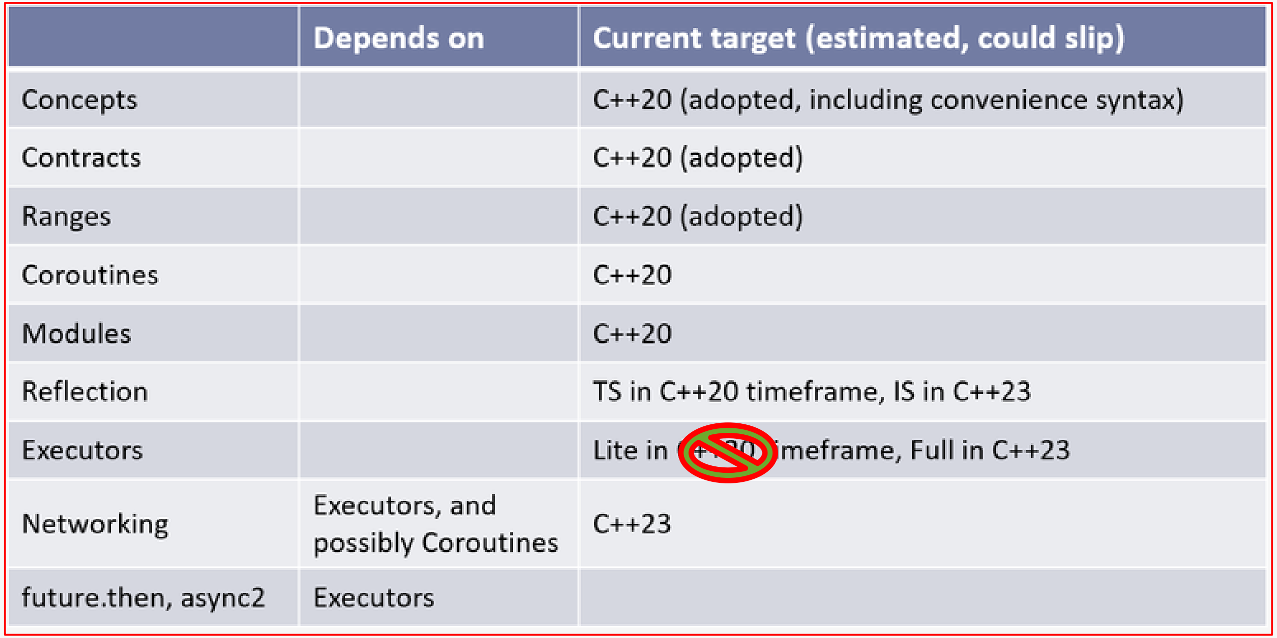
1. What’s still missing from C++?
2. What makes GPU work so fast?
3. What is Modern C++ that works on GPUs, CPUs, everything?

Act 1

1. What's still missing from C++?


System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
Memory and I/O Interface
CPU Core
CPU Core
Graphics Core + New Media Capabilities

What have we achieved so far for C++20?



	Depends on	Current target (estimated, could slip)
Concepts		C++20 (adopted, including convenience syntax)
Contracts		C++20 (adopted)
Ranges		C++20 (adopted)
Coroutines		C++20
Modules		C++20
Reflection		TS in C++20 timeframe, IS in C++23
Executors		Lite in 🚫 C++20 timeframe, Full in C++23
Networking	Executors, and possibly Coroutines	C++23
future.then, async2	Executors	

Use the Proper Abstraction with C++

Abstraction	How is it supported
Cores	C++11/14/17 threads, async
HW threads	C++11/14/17 threads, async, hw_concurrency
Vectors	Parallelism TS2->
Atomic, Fences, lockfree, futures, counters, transactions	C++11/14/17 atomics, Concurrency TS1-> Transactional Memory TS1
Parallel Loops	Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each
Heterogeneous offload, fpga	OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja
Distributed	HPX, MPI, UPC++
Caches	C++17 false sharing support
Numa	
TLS	
Exception handling in concurrent environment	

Task vs data parallelism

Task parallelism

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

What kind of processors should we build

CPU

- 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

GPU

- 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

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

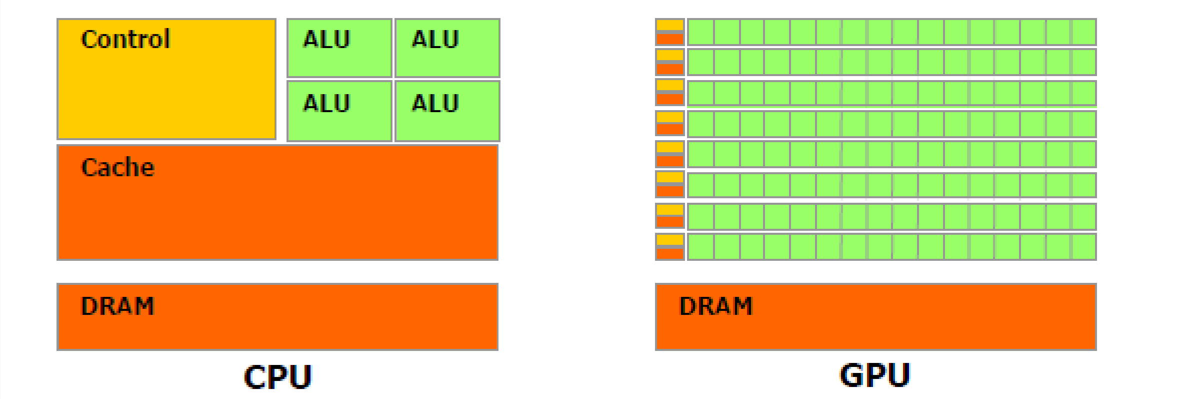
- Cores optimized for aggregate throughput, deemphasizing individual performance
- Scalable parallel processing
- Assumes workload is highly parallel
- Maximize throughput of all threads
  - Lots of big ALUs
  - Multithreading can hide latency, no big caches
  - Simpler control, cost amortized over ALUs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

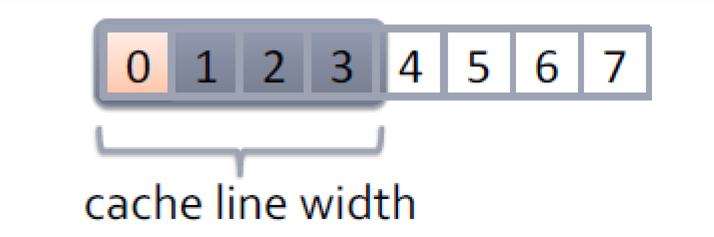
- Many core gpu is a device for turning a compute bound problem into a memory bound problem



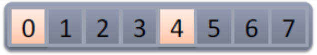
- Lots of processors but only one socket
- Memory concerns dominate performance tuning

Memory is SIMD too

- Virtually all processors have SIMD memory subsystems

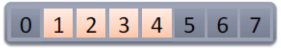


- This has 2 effects
  - Sparse access wastes bandwidth



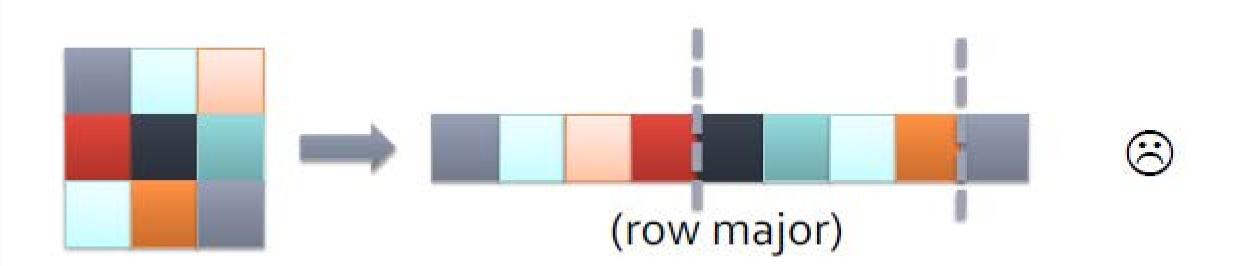
2 words used, 8 words loaded:
¼ effective bandwidth

  - Unaligned access wastes bandwidth

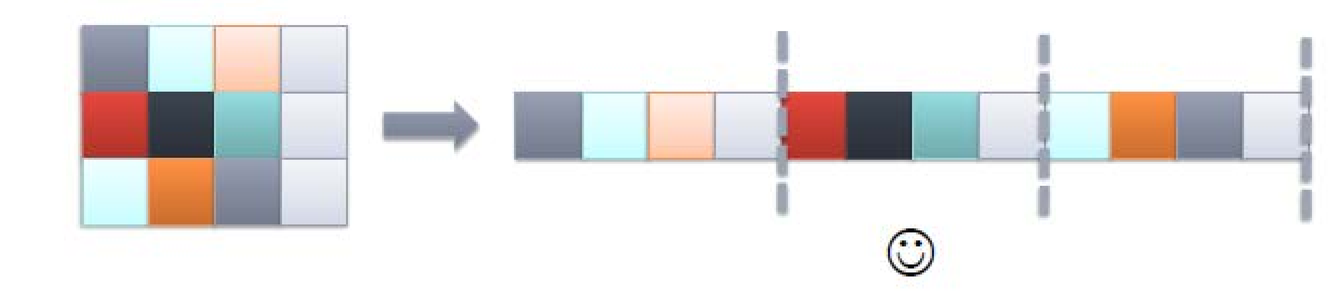


4 words used, 8 words loaded:
½ effective bandwidth

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
  - latency

In 1998, a typical machine had the following flops

.45 GFLOPS, 1 core

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

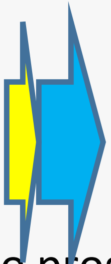
In 2011, a typical machine had the following flops

80 GFLOPS 4 cores

To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, OpenCL

In 2011, a typical machine had the following flops

80 GFLOPS 4 cores+140 GFLOPS AVX

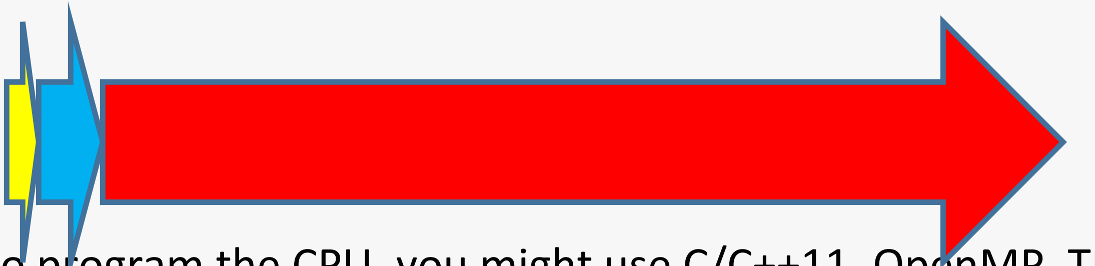


To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, CUDA, OpenCL

To program the vector unit, you have to use Intrinsics, OpenCL, CUDA, or auto-vectorization

In 2011, a typical machine had the following flops

80 GFLOPS 4 cores+140 GFLOPS AVX+2500 GFLOPS GPU



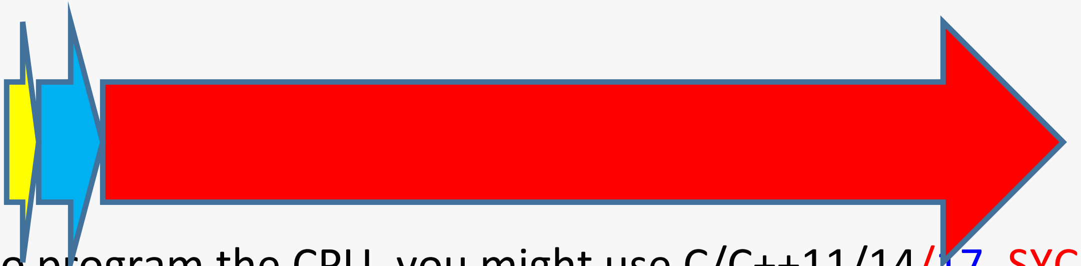
To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, CUDA, OpenCL

To program the vector unit, you have to use Intrinsics, OpenCL, CUDA or auto-vectorization

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

In 2017, a typical machine had the following flops

140 GFLOPS + 560 GFLOPS AVX + 4600 GFLOPS GPU



To program the CPU, you might use C/C++11/14/17, SYCL, OpenMP, TBB, Cilk, CUDA, OpenCL

To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, CUDA or auto-vectorization, OpenMP

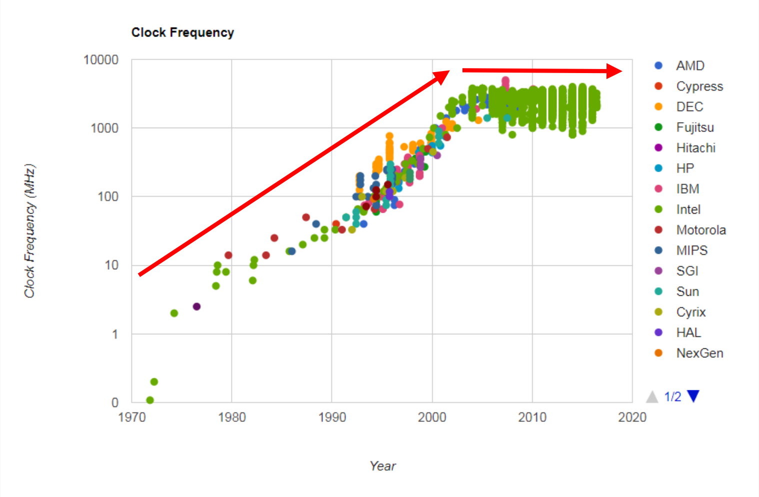
To program the GPU, you have to use SYCL, CUDA, OpenCL, OpenGL, DirectX, Intrinsics, OpenMP

"The end of Moore's Law"

"The free lunch is over"

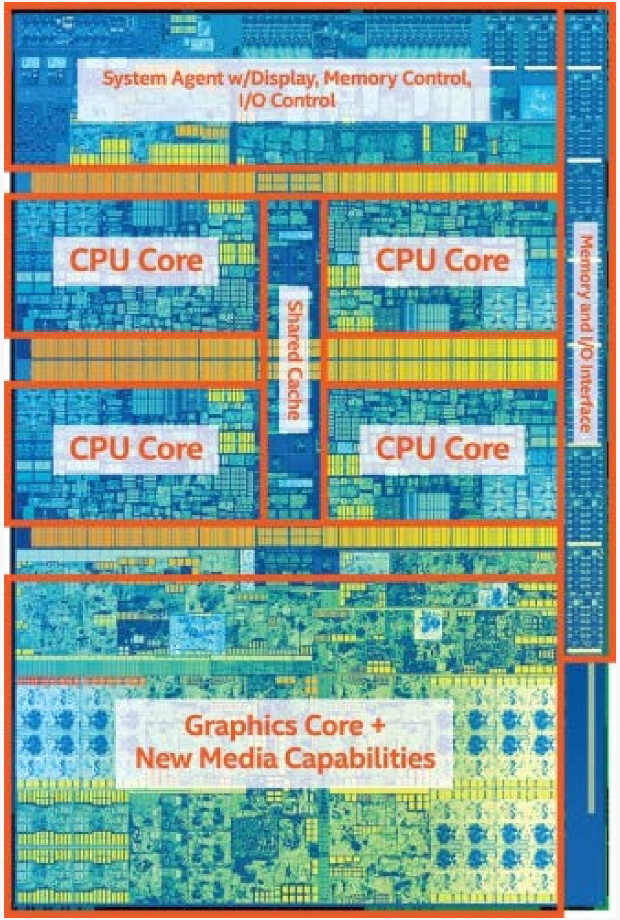
"The future is parallel and heterogeneous"

"GPUs are everywhere"

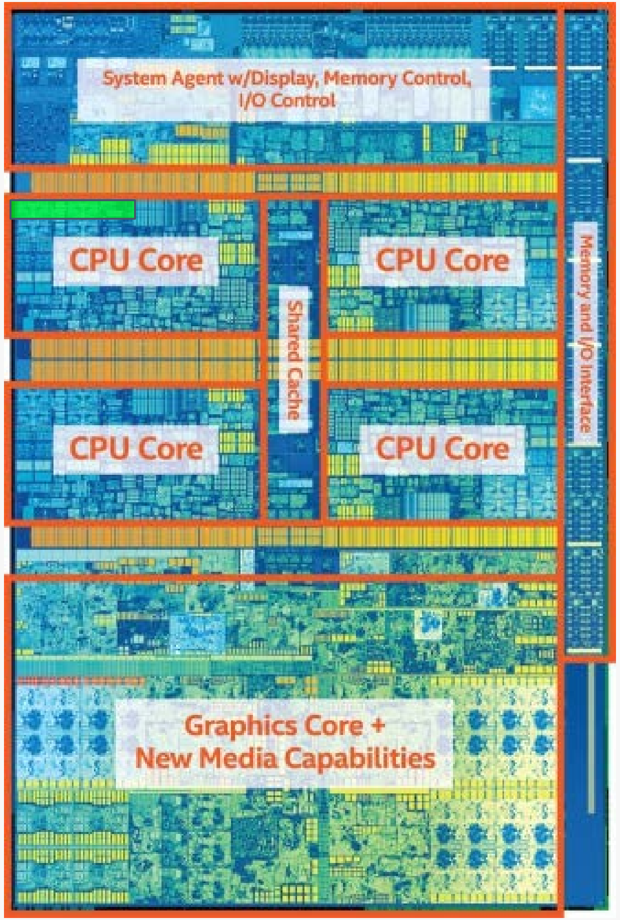


Take a typical Intel chip

- Intel Core i7 7th Gen
  - 4x CPU cores
    - Each with hyperthreading
    - Each with 8-wide AVX instructions
  - GPU
    - With 1280 processing elements

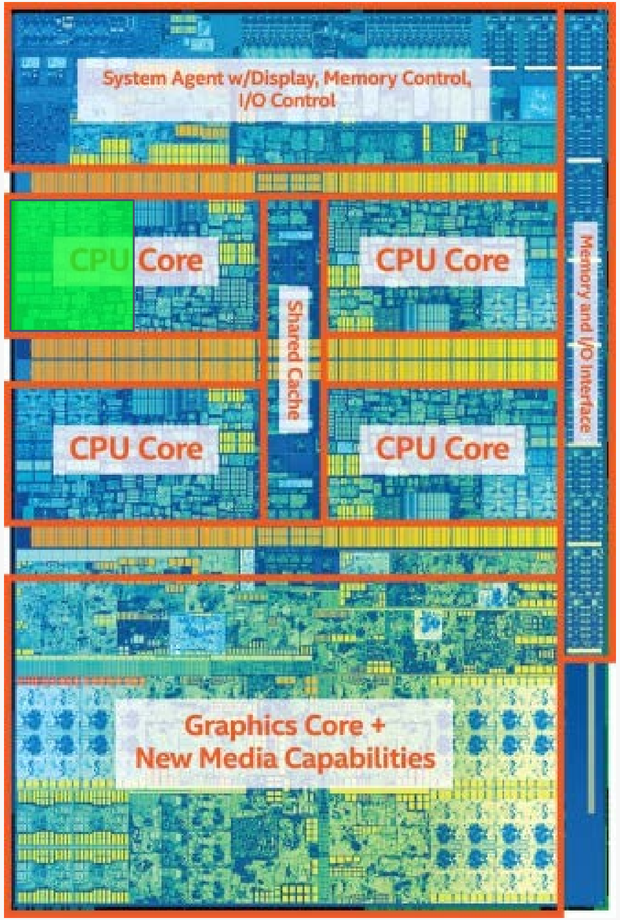


Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip


System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
Memory and I/O Interface
CPU Core
CPU Core
Graphics Core + New Media Capabilities

Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip

Using vectorisation allows you to fully utilise the resources of a single hyperthread


System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
Memory and I/O Interface
CPU Core
CPU Core
Graphics Core + New Media Capabilities

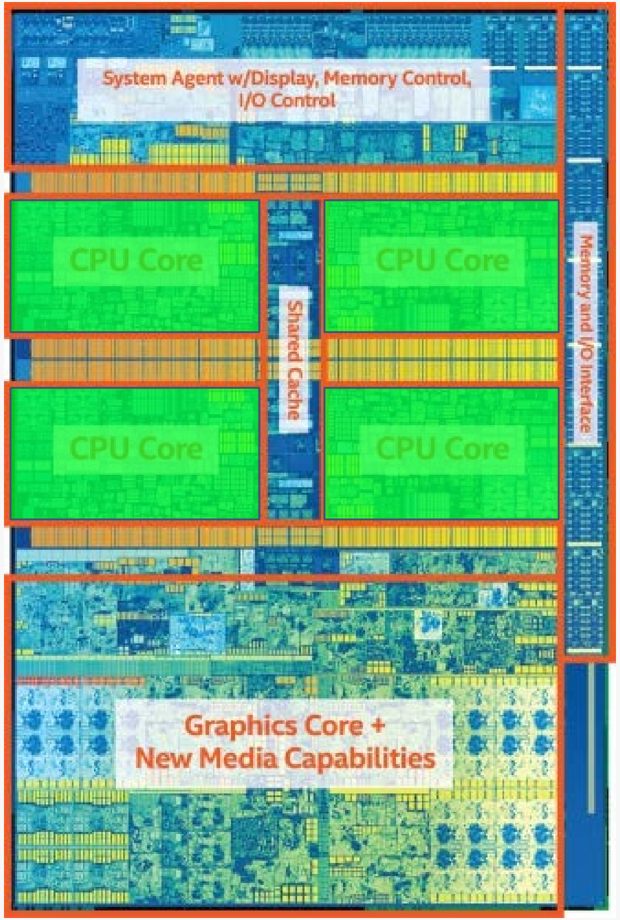
Use the Proper Abstraction with C++

Abstraction	How is it supported
Cores	C++11/14/17 threads, async
HW threads	C++11/14/17 threads, async, hw_concurrency
Vectors	Parallelism TS2->
Atomic, Fences, lockfree, futures, counters, transactions	C++11/14/17 atomics, Concurrency TS1-> Transactional Memory TS1
Parallel Loops	Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each
Heterogeneous offload, fpga	OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja
Distributed	HPX, MPI, UPC++
Caches	C++17 false sharing support
Numa	
TLS	
Exception handling in concurrent environment	

Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip

Using vectorisation allows you to fully utilise the resources of a single hyperthread

Using multi-threading allows you to fully utilise all CPU cores


System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
CPU Core
CPU Core
Memory and I/O Interface
Graphics Core + New Media Capabilities

Use the Proper Abstraction with C++

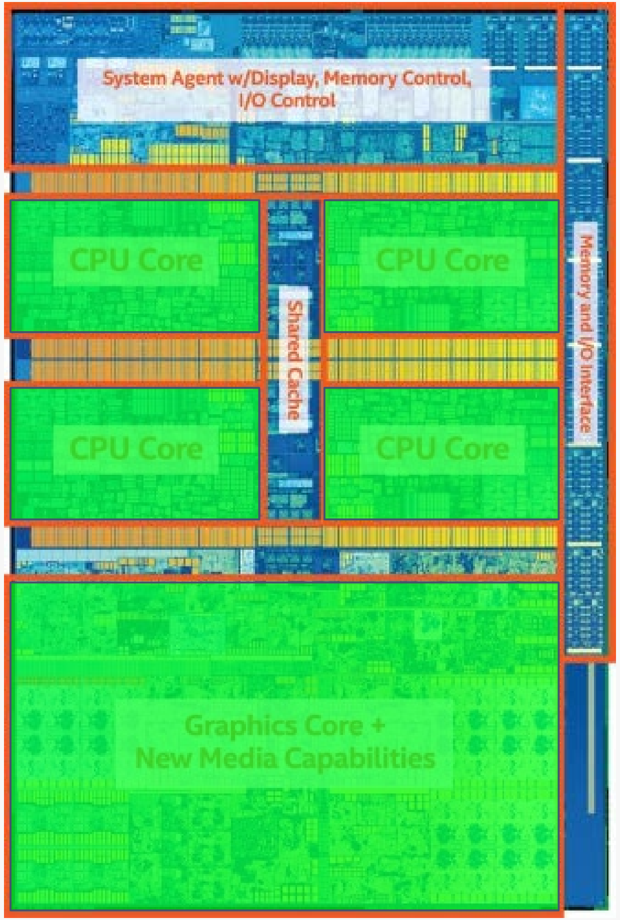
Abstraction	How is it supported
Cores	C++11/14/17 threads, async
HW threads	C++11/14/17 threads, async, hw_concurrency
Vectors	Parallelism TS2->
Atomic, Fences, lockfree, futures, counters, transactions	C++11/14/17 atomics, Concurrency TS1-> Transactional Memory TS1
Parallel Loops	Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each
Heterogeneous offload, fpga	OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja
Distributed	HPX, MPI, UPC++
Caches	C++17 false sharing support
Numa	
TLS	
Exception handling in concurrent environment	

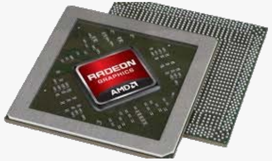
Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip

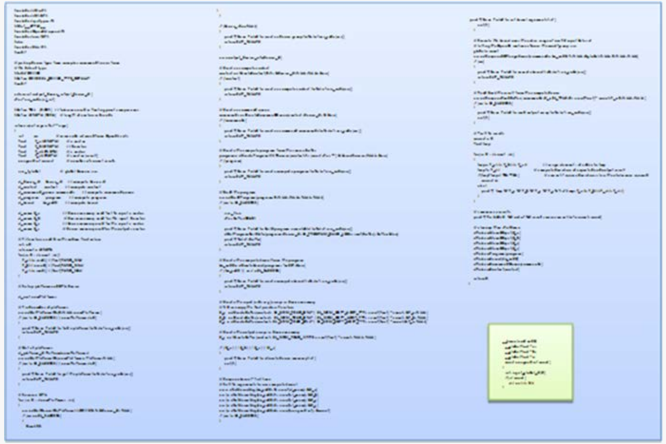
Using vectorisation allows you to fully utilise the resources of a single hyperthread

Using multi-threading allows you to fully utilise all CPU cores

Using heterogeneous dispatch allows you to fully utilise the entire chip


System Agent w/Display, Memory Control, I/O Control
CPU Core
CPU Core
Shared Cache
Memory and I/O Interface
CPU Core
CPU Core
Graphics Core + New Media Capabilities





GPGPU programming was once a niche technology

- Limited to specific domain
- Separate source solutions
- Verbose low-level APIs
- Very steep learning curve

Coverage after C++11

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	C++11: thread,lambda function, TLS	C++11: Async, packaged tasks, promises, futures, atomics	C++11: locks, memory model, mutex, condition variable, atomics, static init/term	C++11: lambda

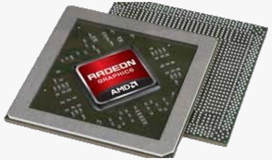
Top500 contenders

Internet of Things

- All forms of accelerators, DSP, GPU, APU, GPGPU
- Network heterogenous consumer devices
  - Kitchen appliances, drones, signal processors, medical imaging, auto, telecom, automation, not just graphics engines

This is not the case anymore

- Almost everything has a GPU now
- Single source solutions
- Modern C++ programming models
- More accessible to the average C++ developer



C++AMP

SYCL

CUDA Agency

Kokkos

HPX

Raja

invoke async parallel algorithms future::then post

defer define_task_block dispatch asynchronous operations strand<>

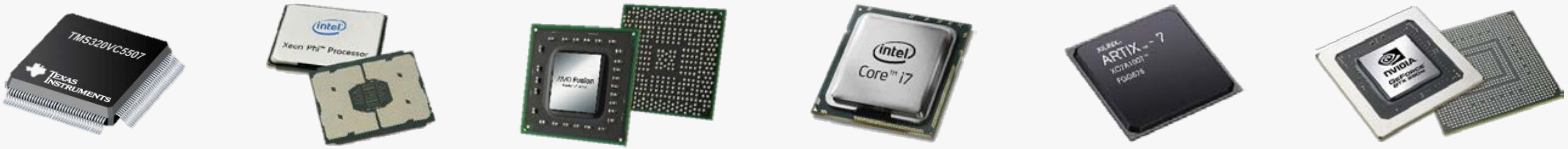
C++ Executors: Unified interface for execution

SYCL / OpenCL / CUDA / HCC

OpenMP / MPI

C++ Thread Pool

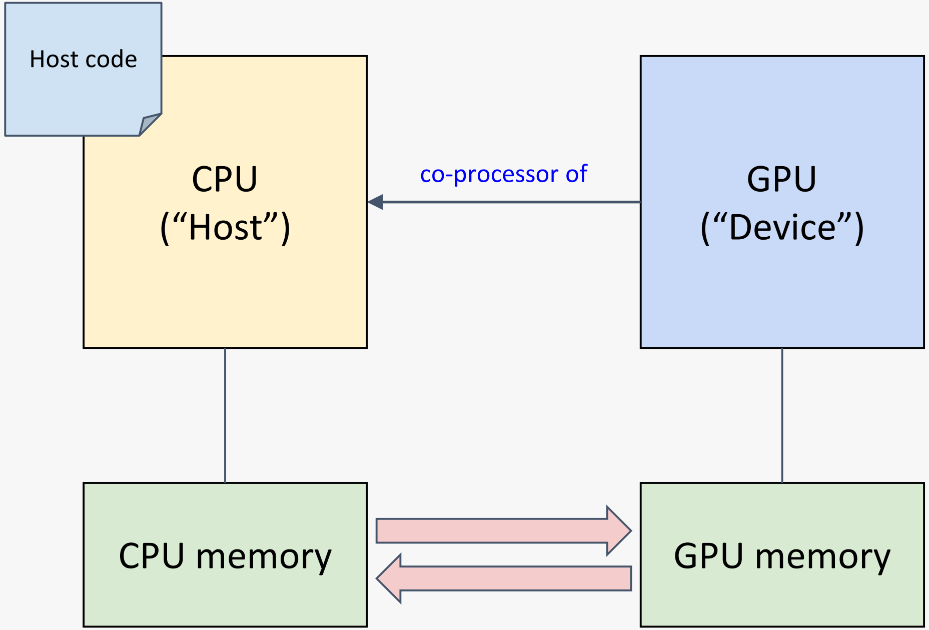
Boost.Asio / Networking TS



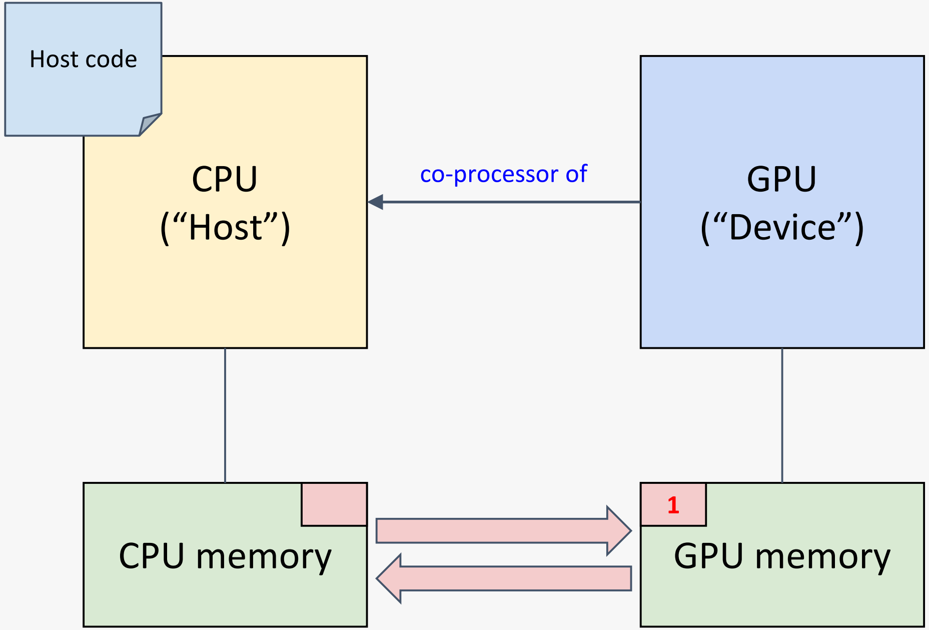
Act 2

1. What's still missing from C++?
2. What makes GPU work so fast?

The way of CPU and GPU

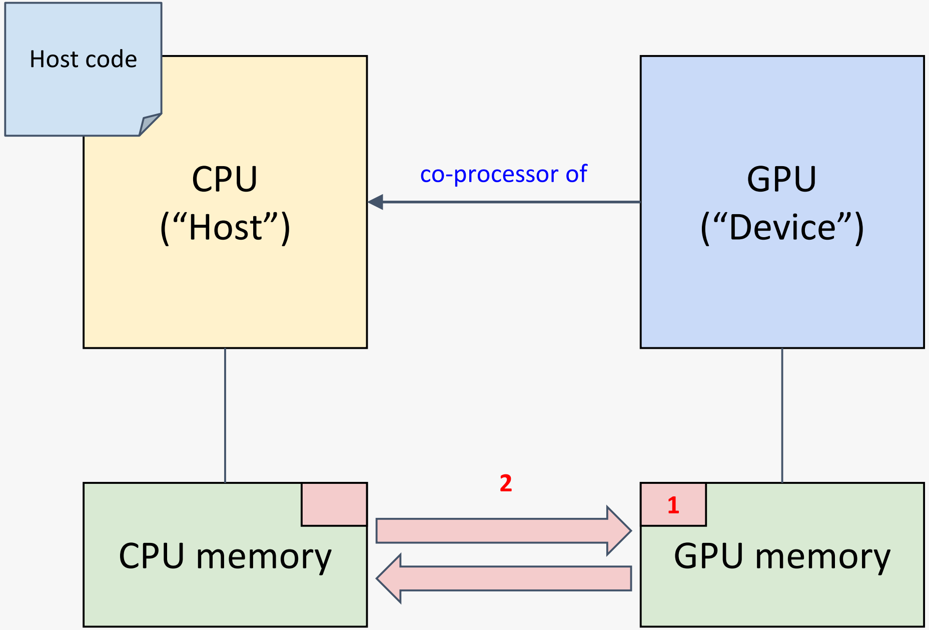


The way of CPU and GPU

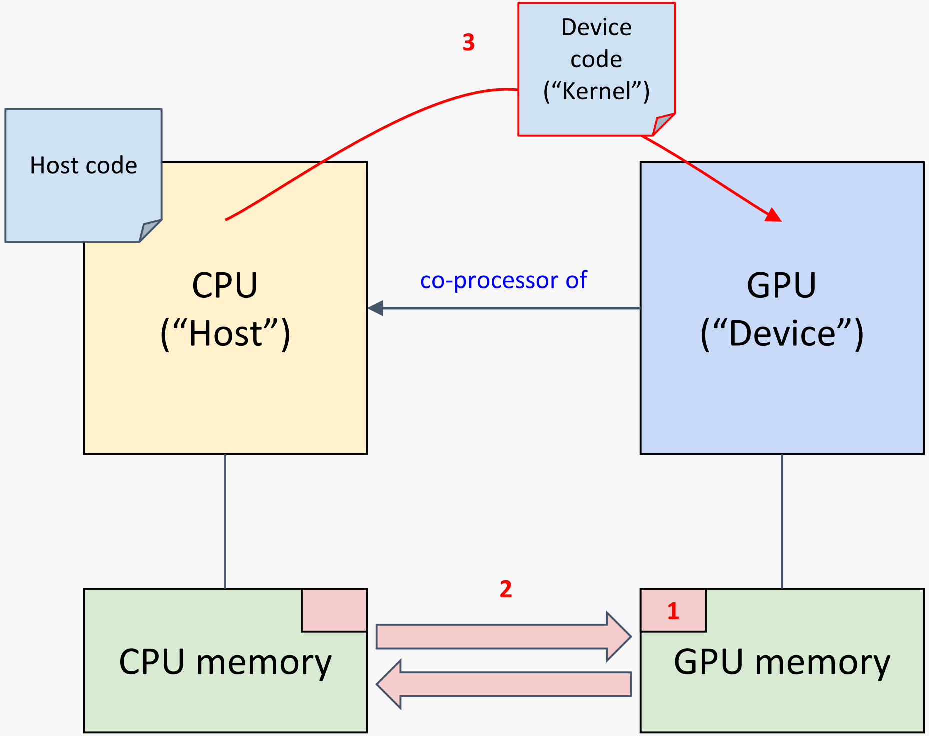


1. The CPU allocates memory on the GPU

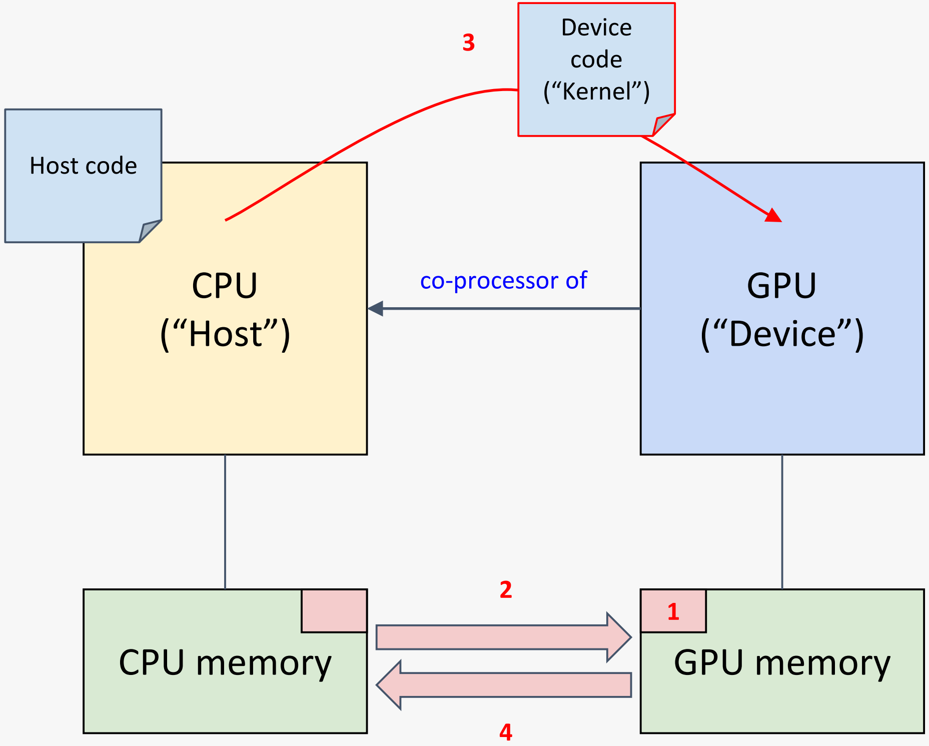
The way of CPU and GPU



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

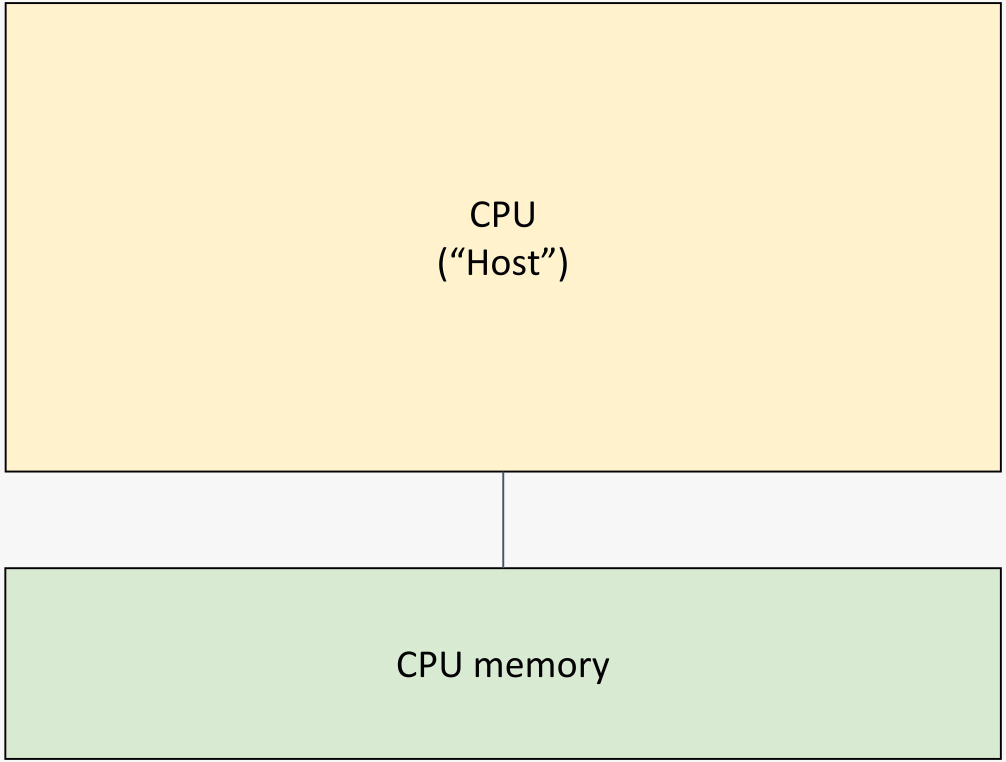


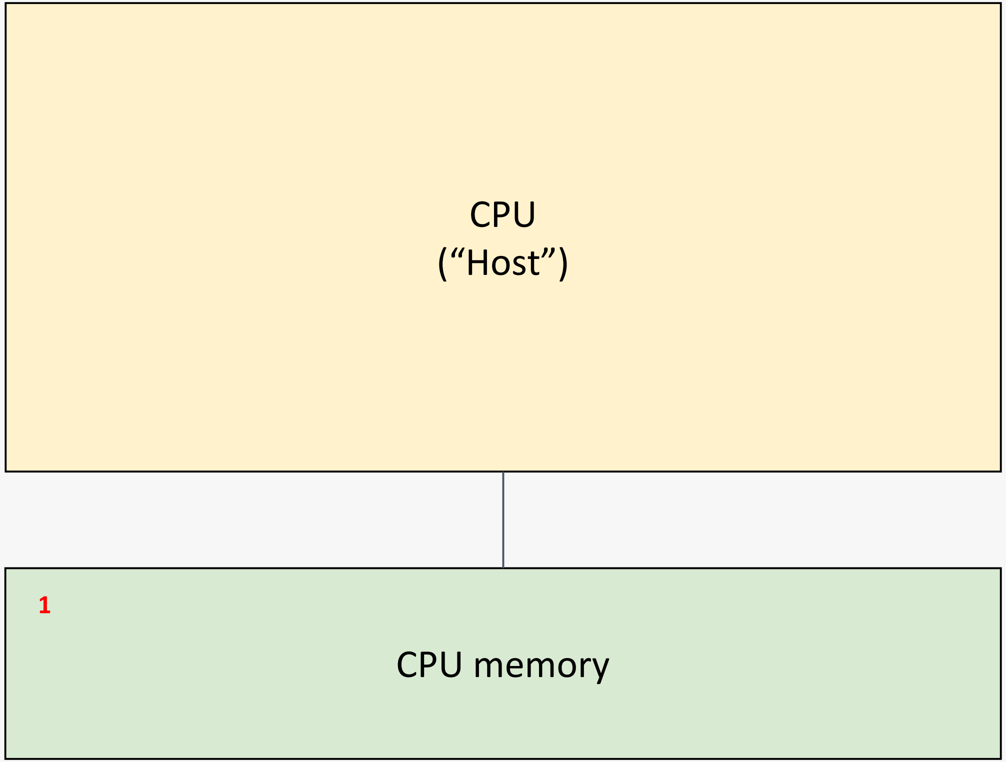
1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU



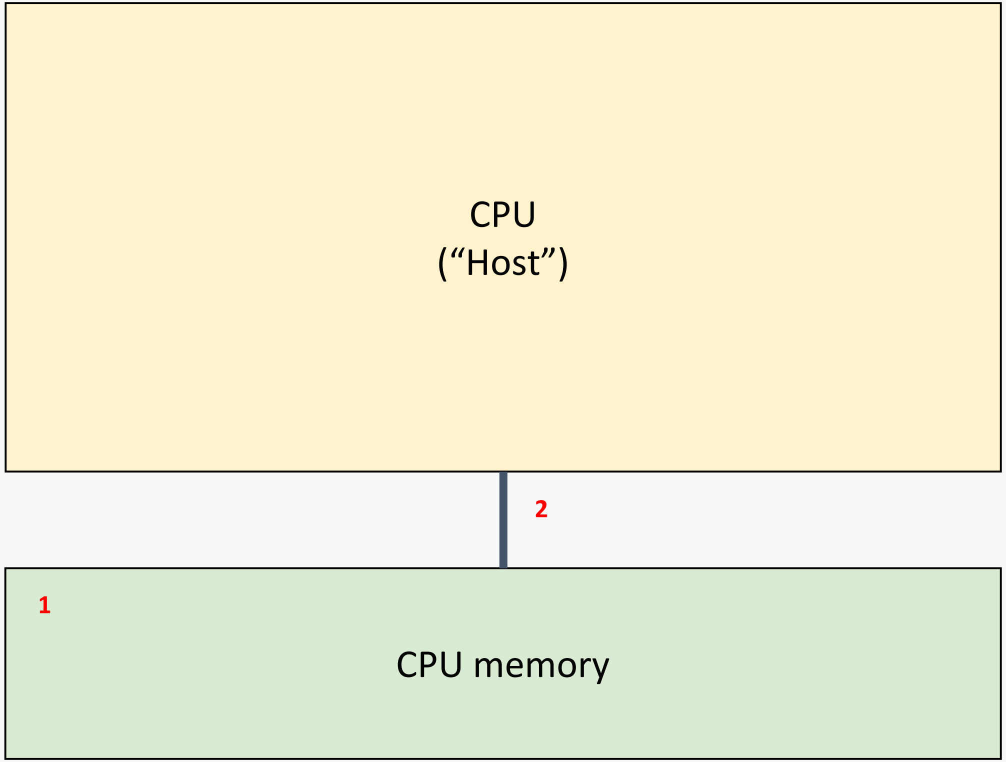
1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU
4. The CPU copies data to CPU from GPU

The CPU

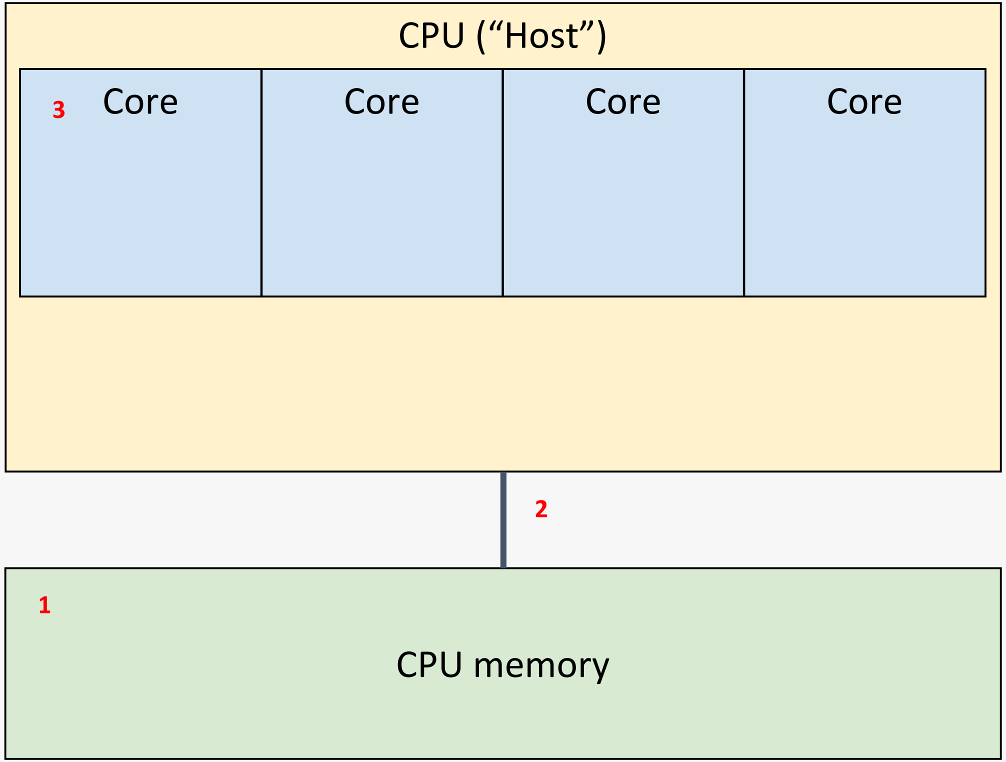




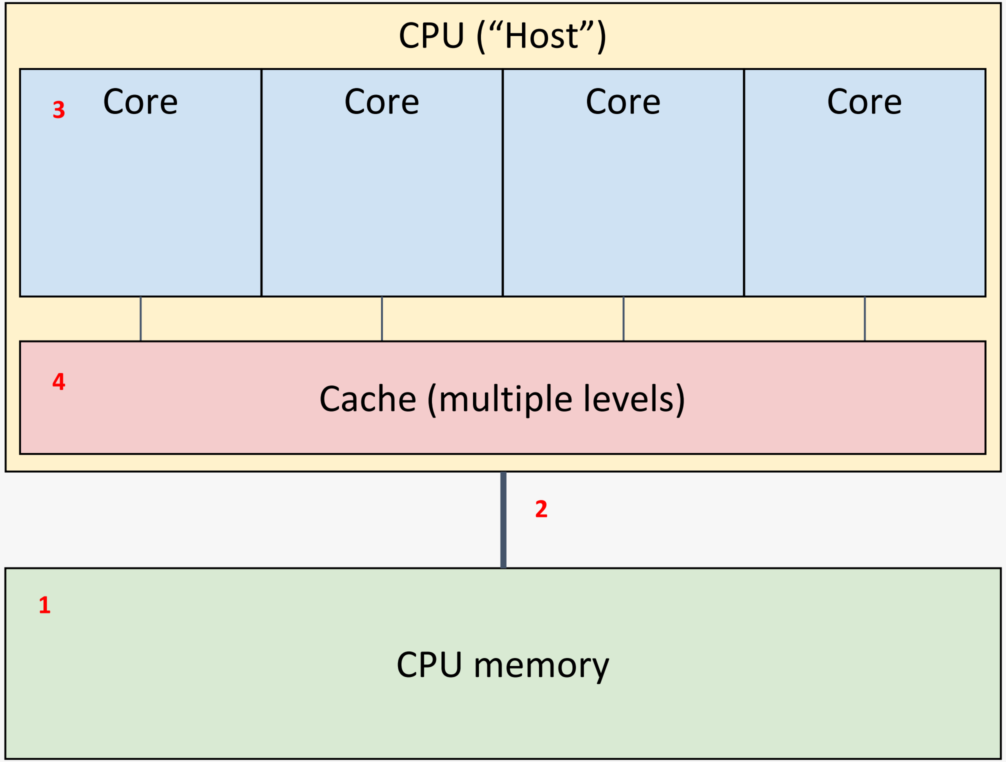
1. A CPU has a region of dedicated memory



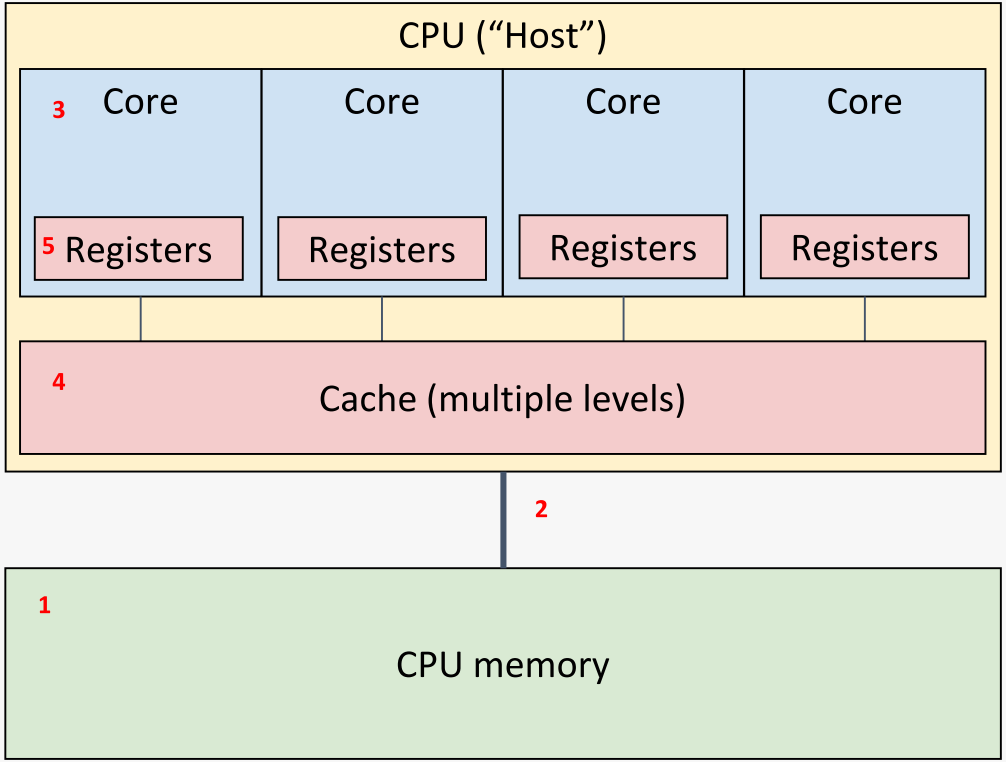
1. A CPU has a region of dedicated memory
2. CPU memory is connected to the CPU via a bus



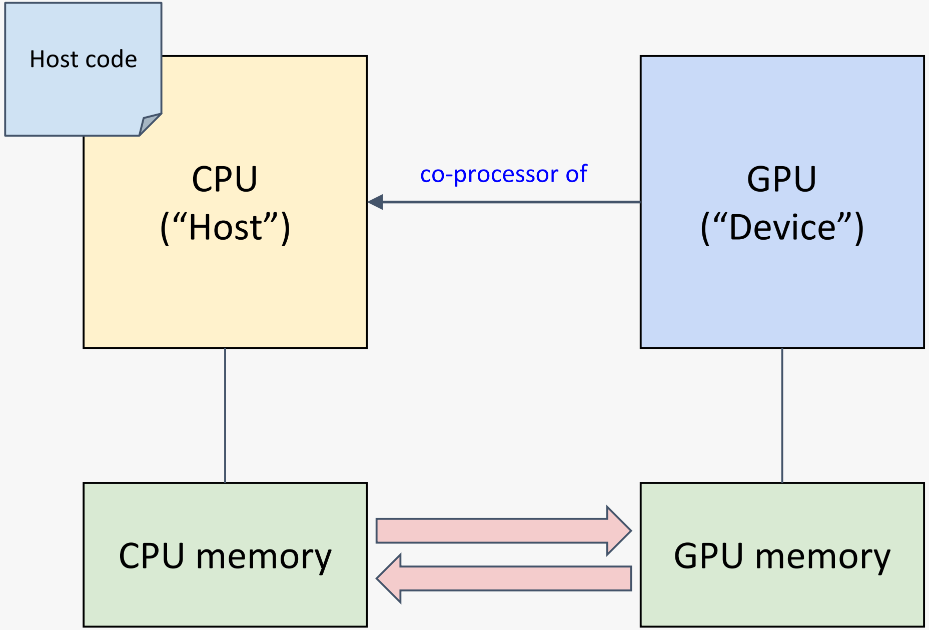
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores



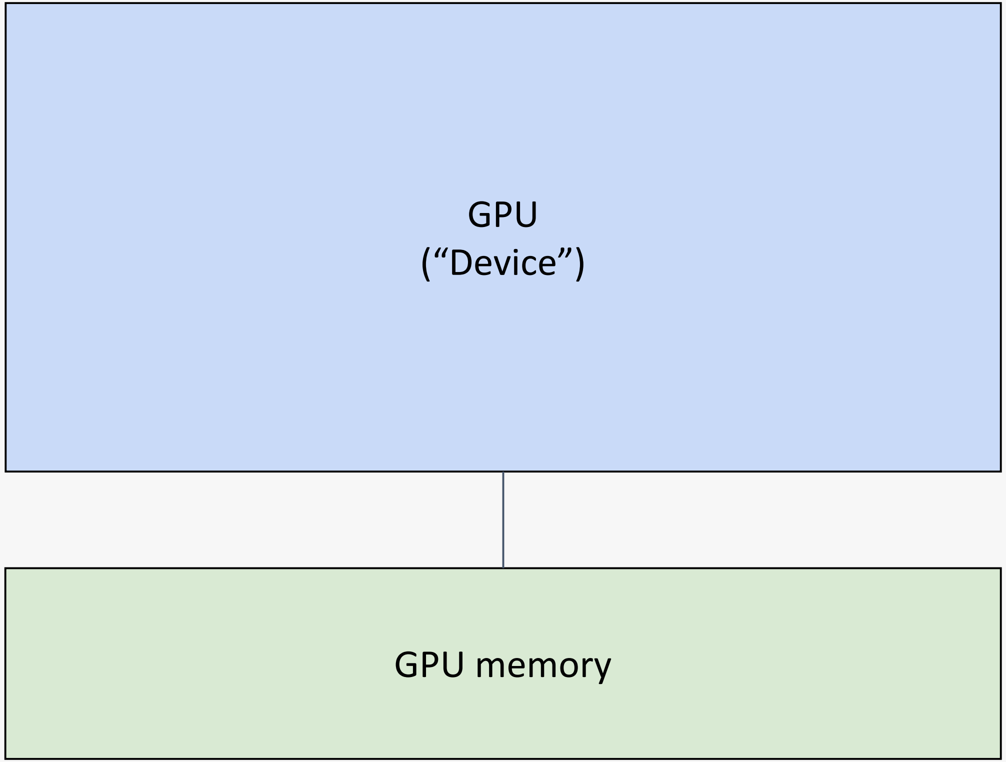
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores
4. A CPU has a number of caches of different levels

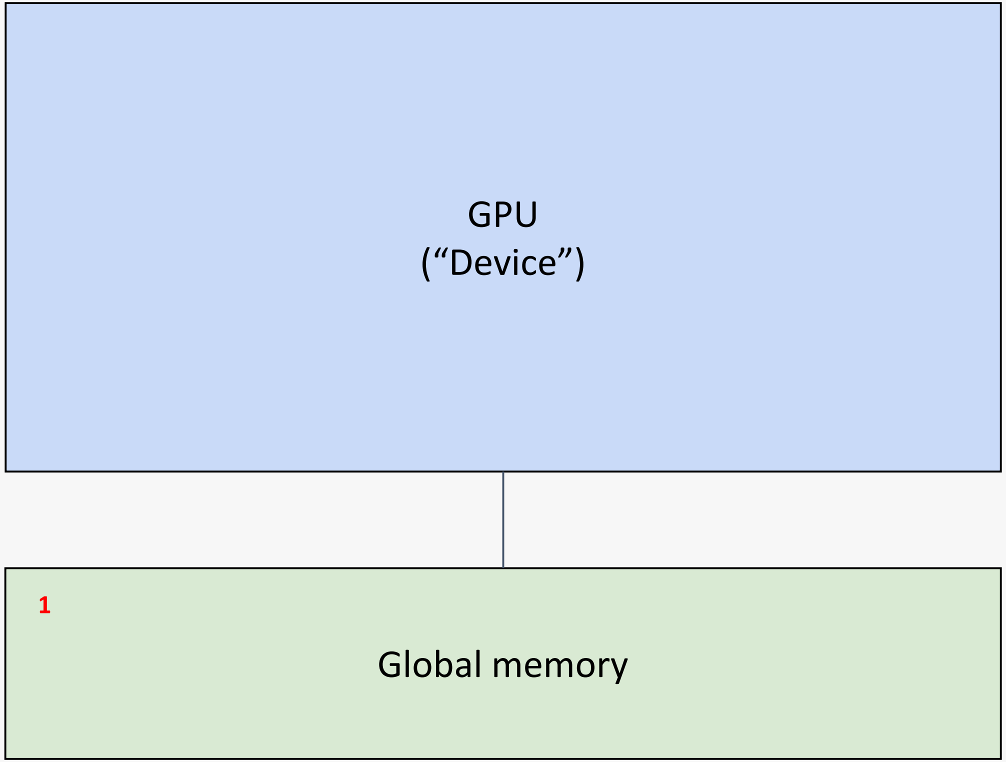


1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores
4. A CPU has a number of caches of different levels
5. Each CPU core has dedicated registers

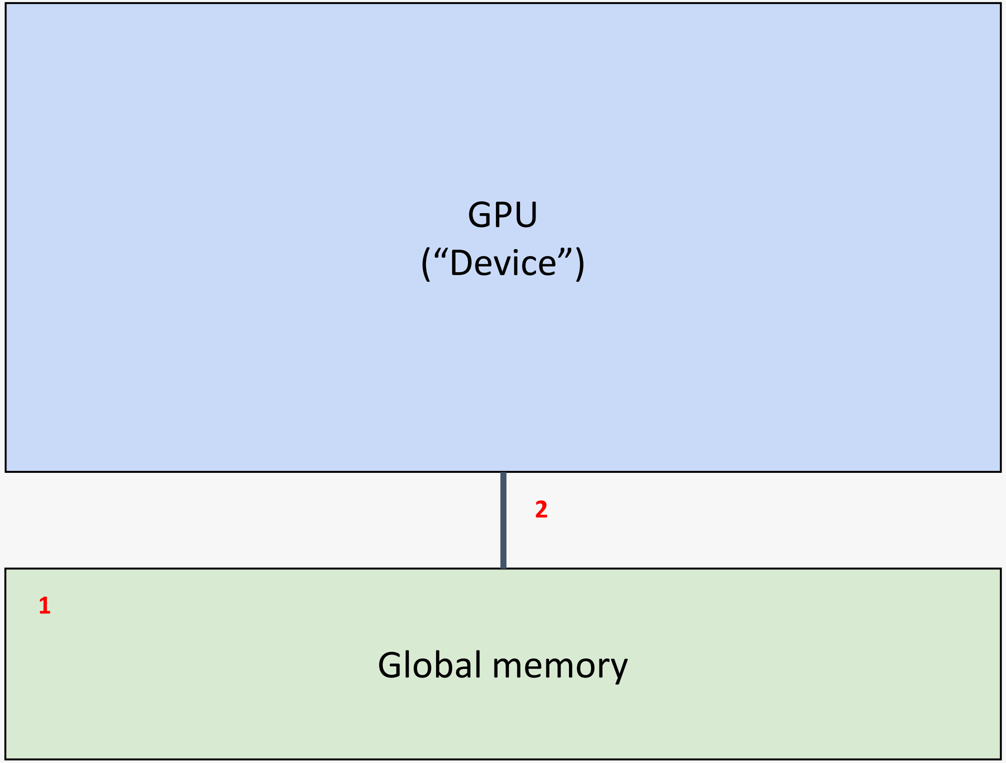

Host code
CPU
("Host")
co-processor of
GPU
("Device")
CPU memory
GPU memory

The GPU

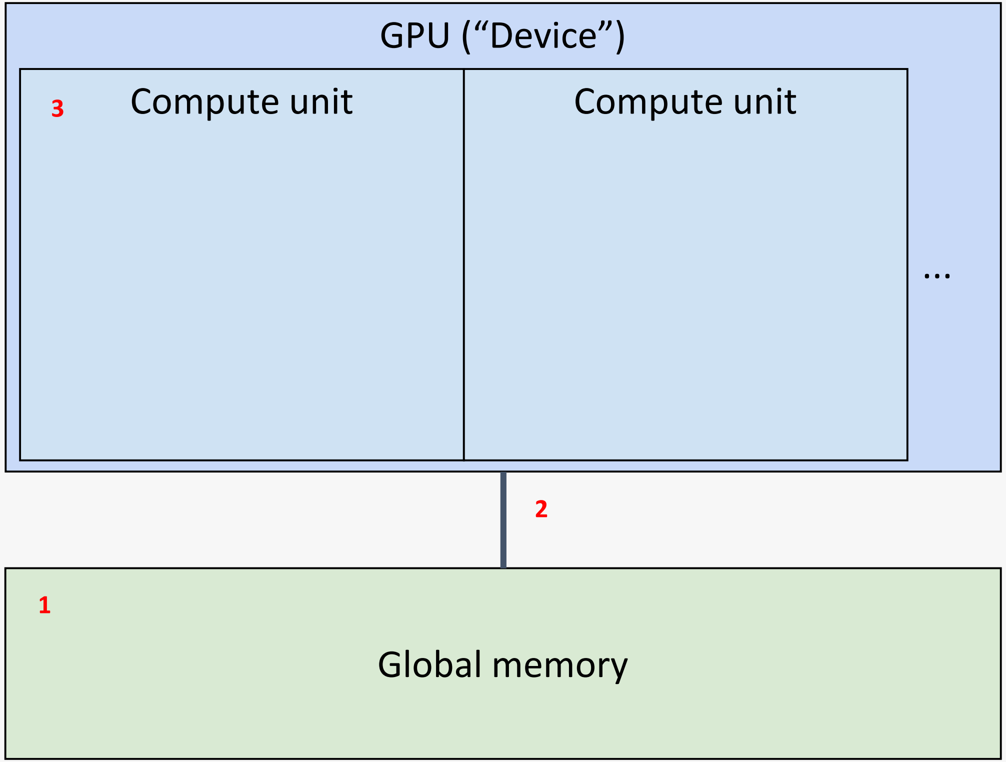




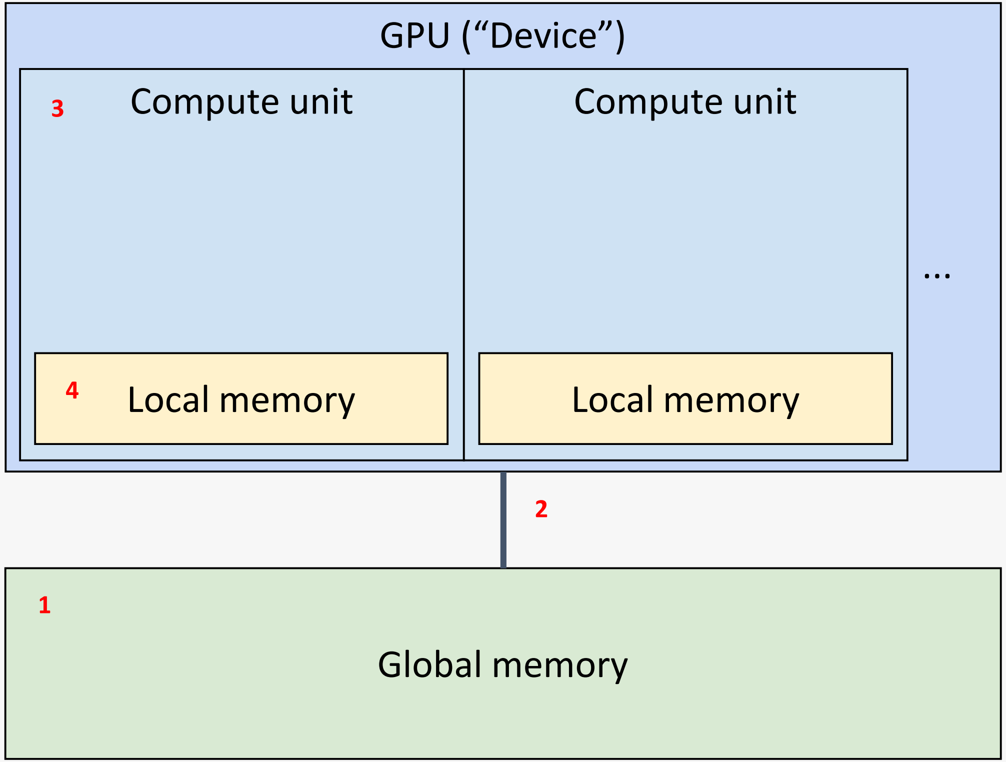
1. A GPU has a region of dedicated global memory



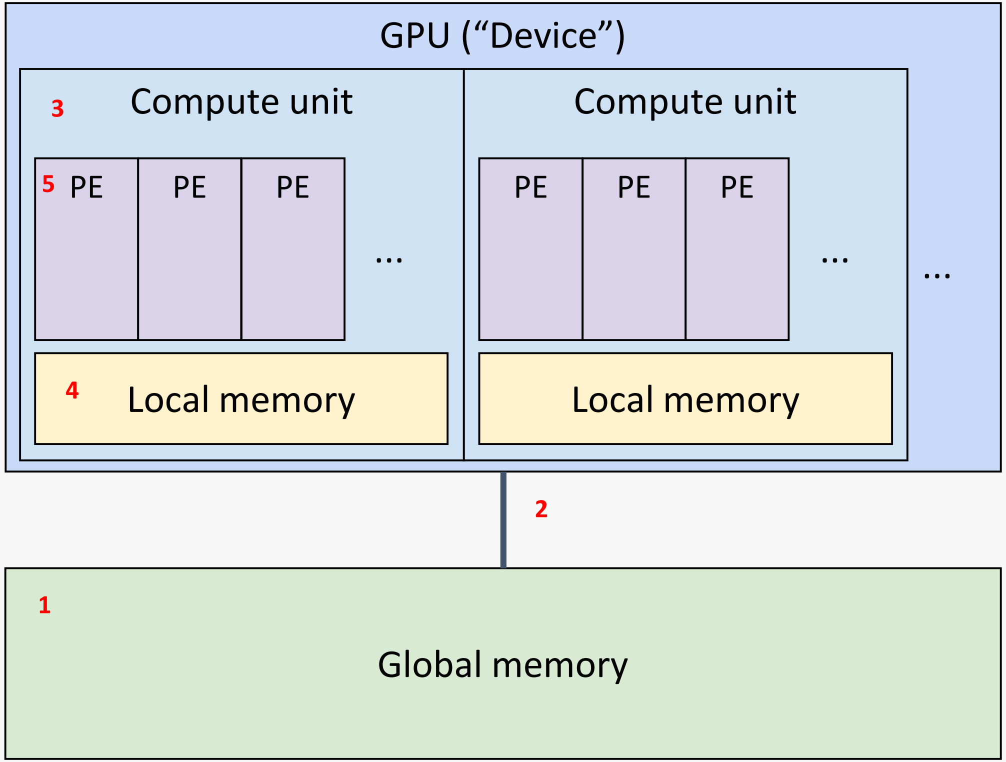
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus



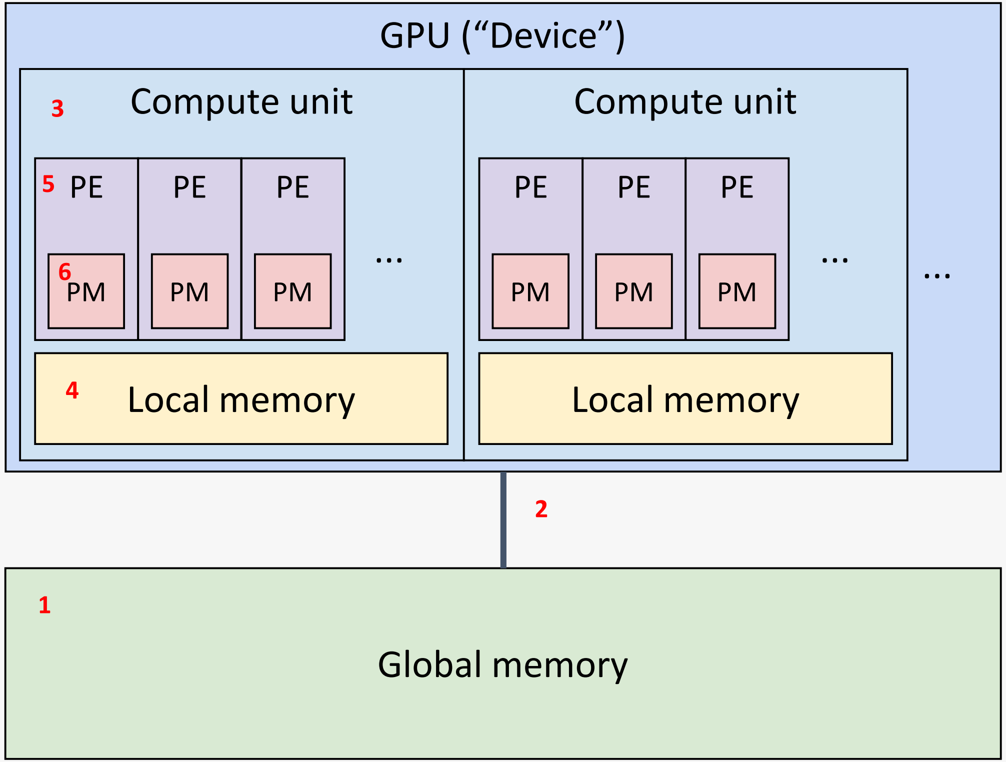
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units



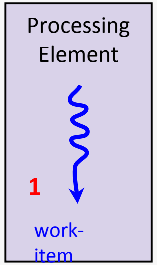
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory



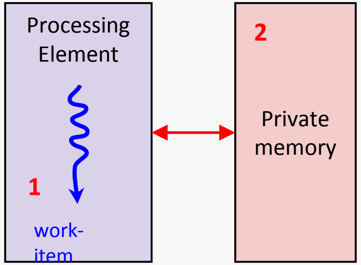
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements



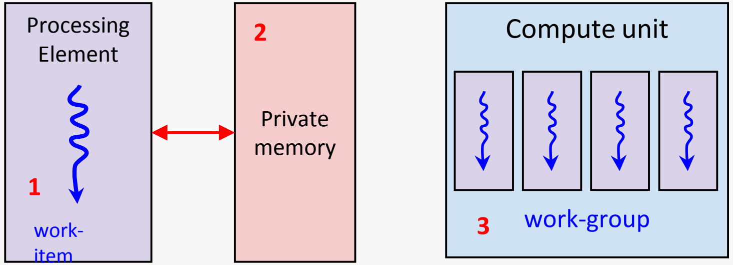
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements
6. Each processing element has dedicated private memory



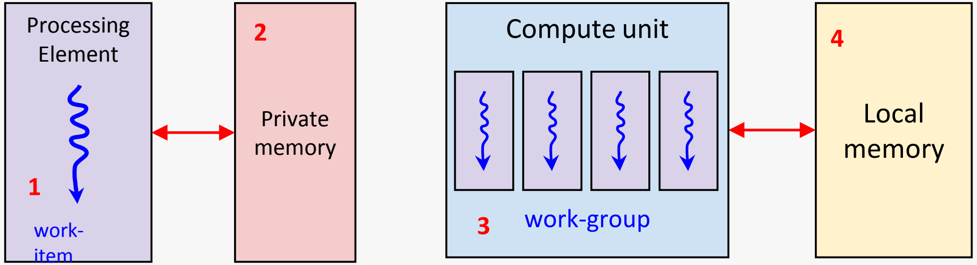
1. A processing element executes a single work-item



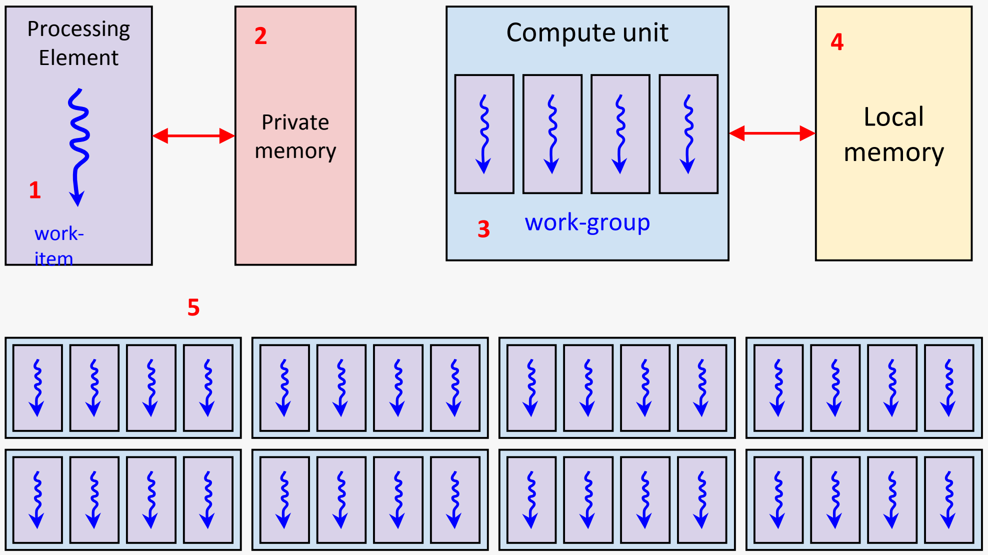
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element



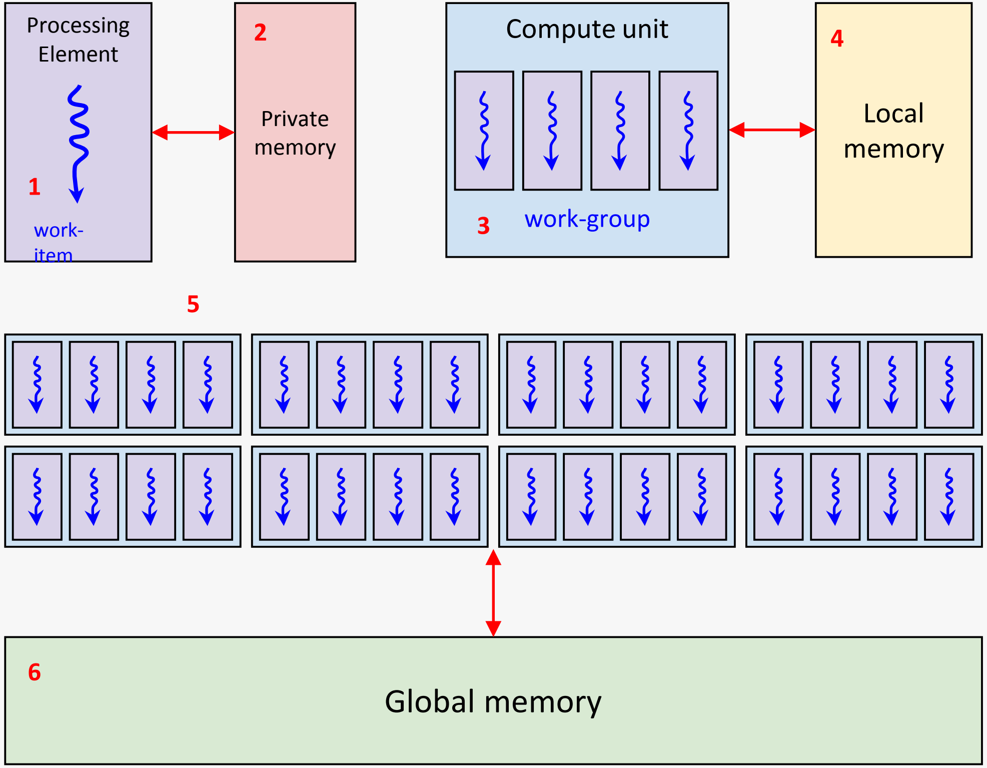
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit



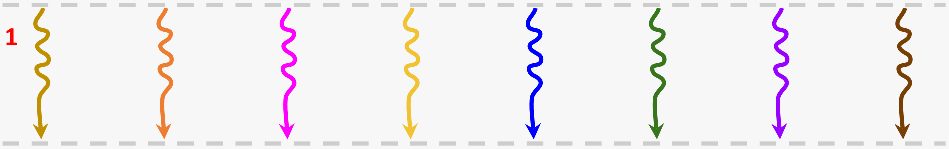
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access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



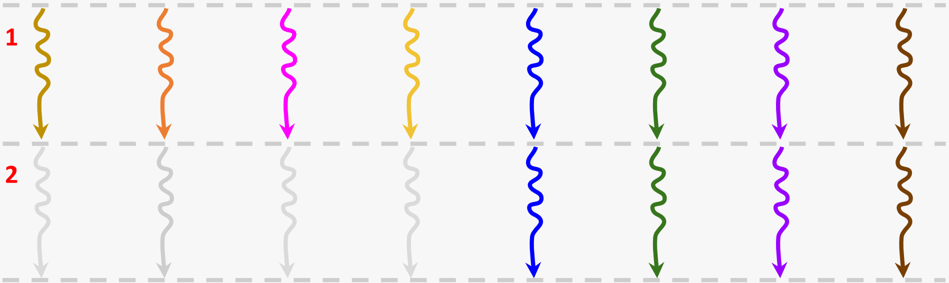
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access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



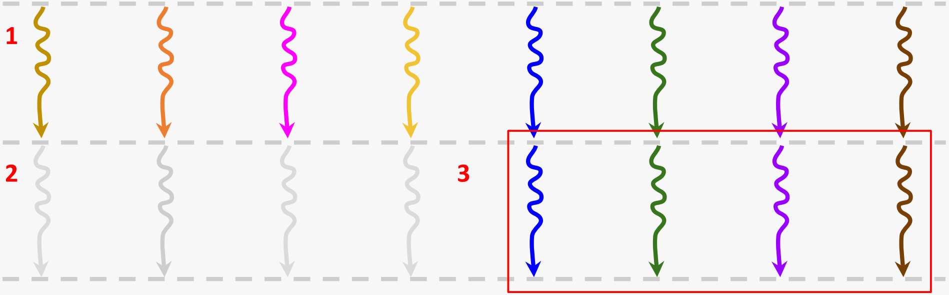
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access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



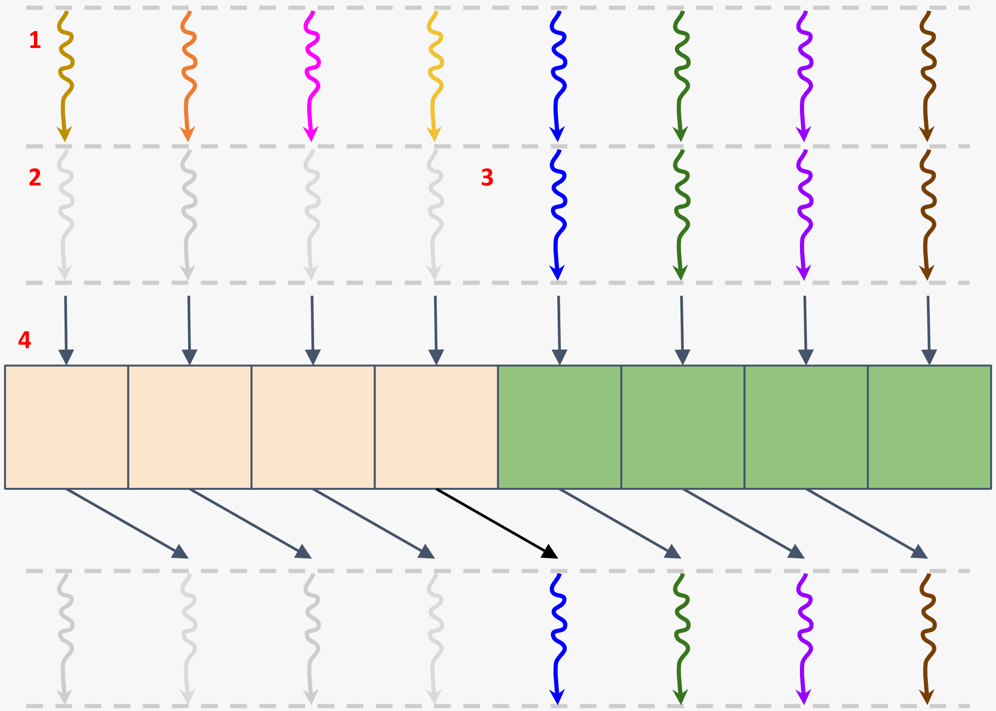
1. Multiple work-items will execute concurrently



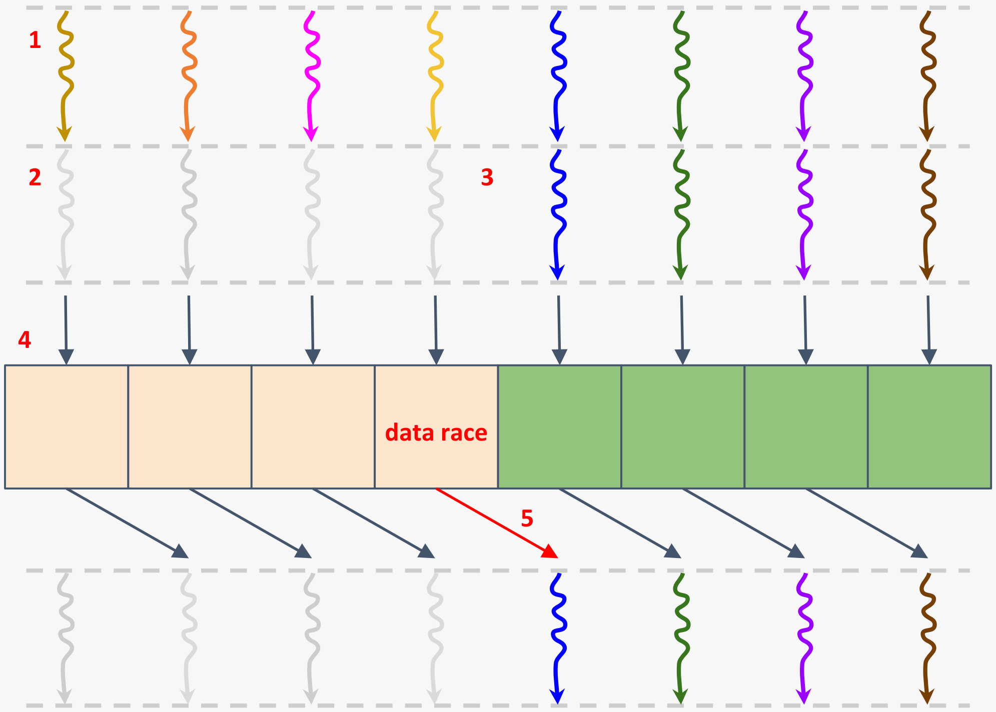
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly



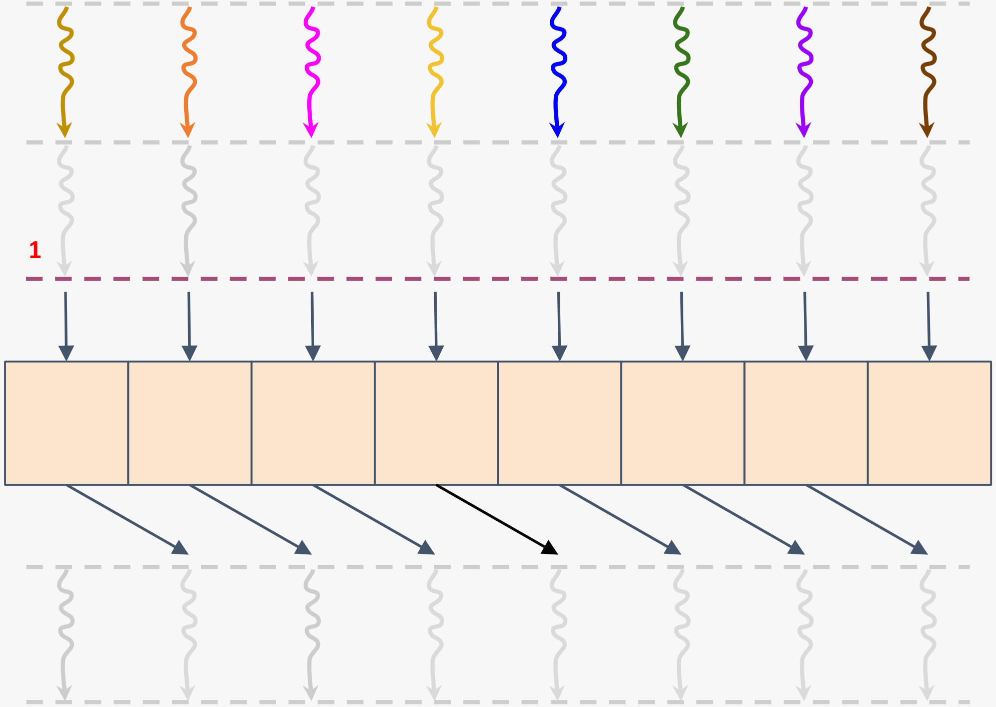
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs do execute a number of work-items uniformly (lock-step), but that number is unspecified



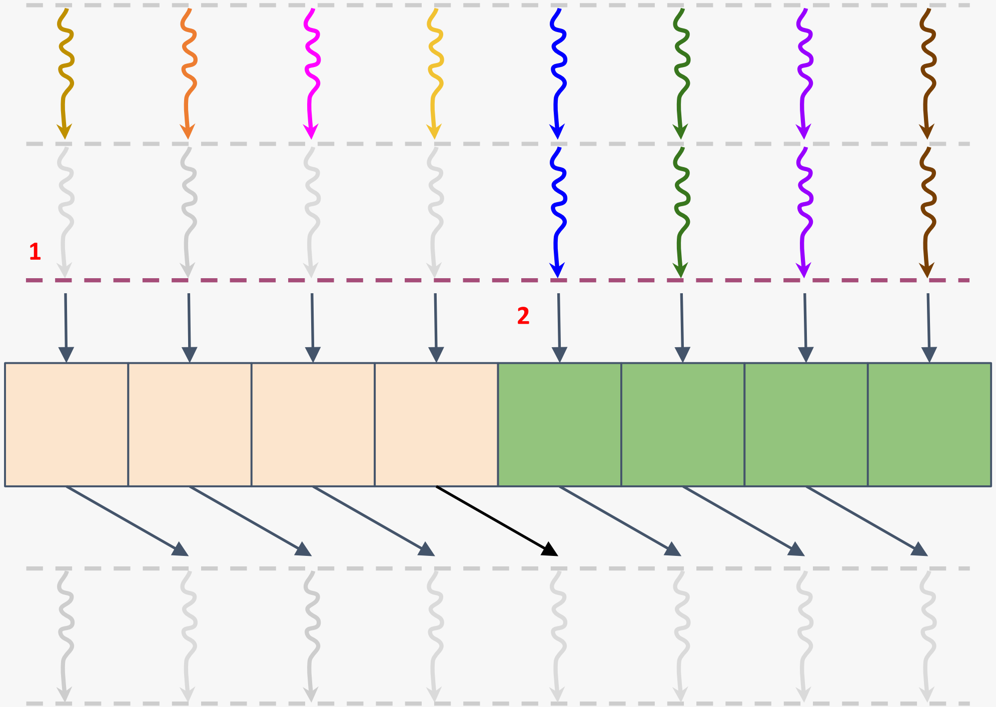
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs do execute a number of work-items uniformly (lock-step), but that number is unspecified
4. A work-item can share results with other work-items via local and global memory



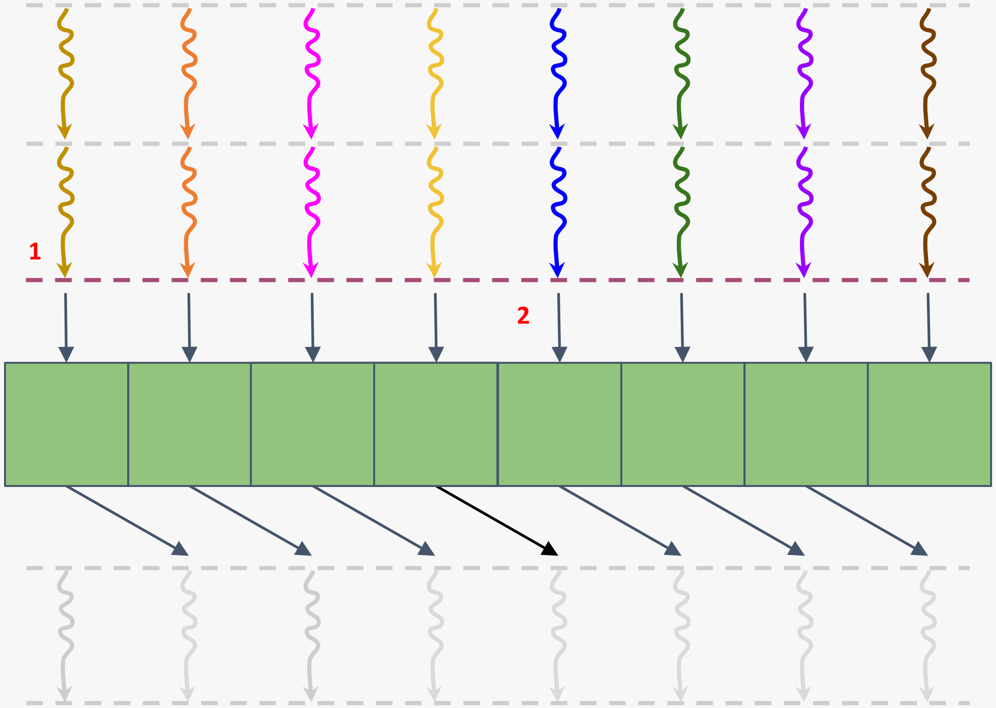
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs do execute a number of work-items uniformly (lock-step), but that number is unspecified
4. A work-item can share results with other work-items via local and global memory
5. However this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



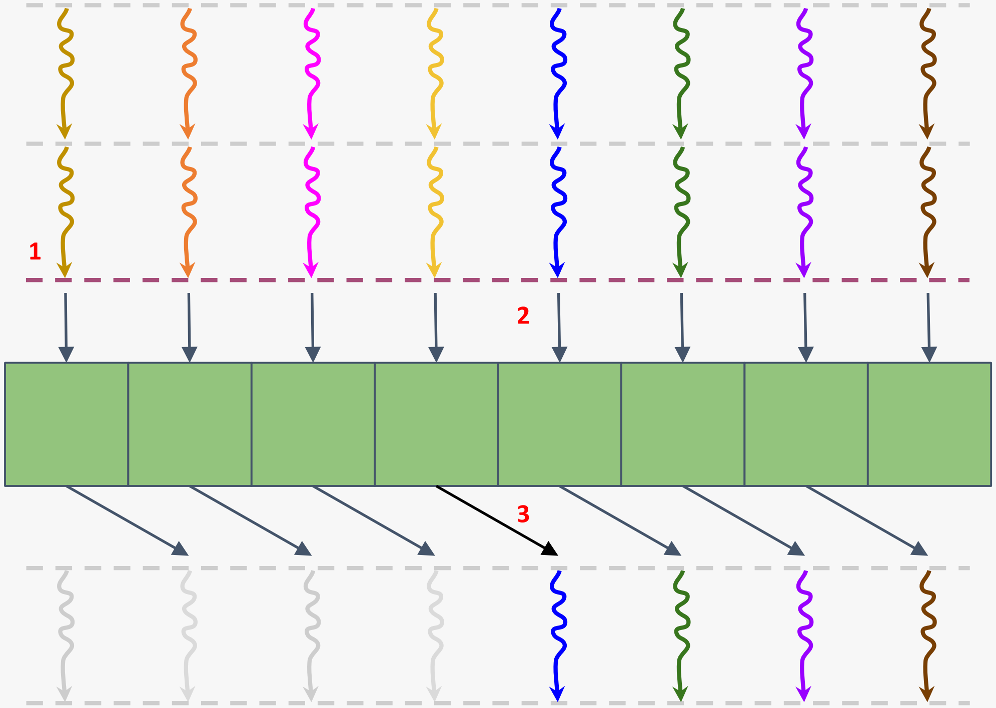
1. This problem can be solved by a synchronisation primitive called a work-group barrier



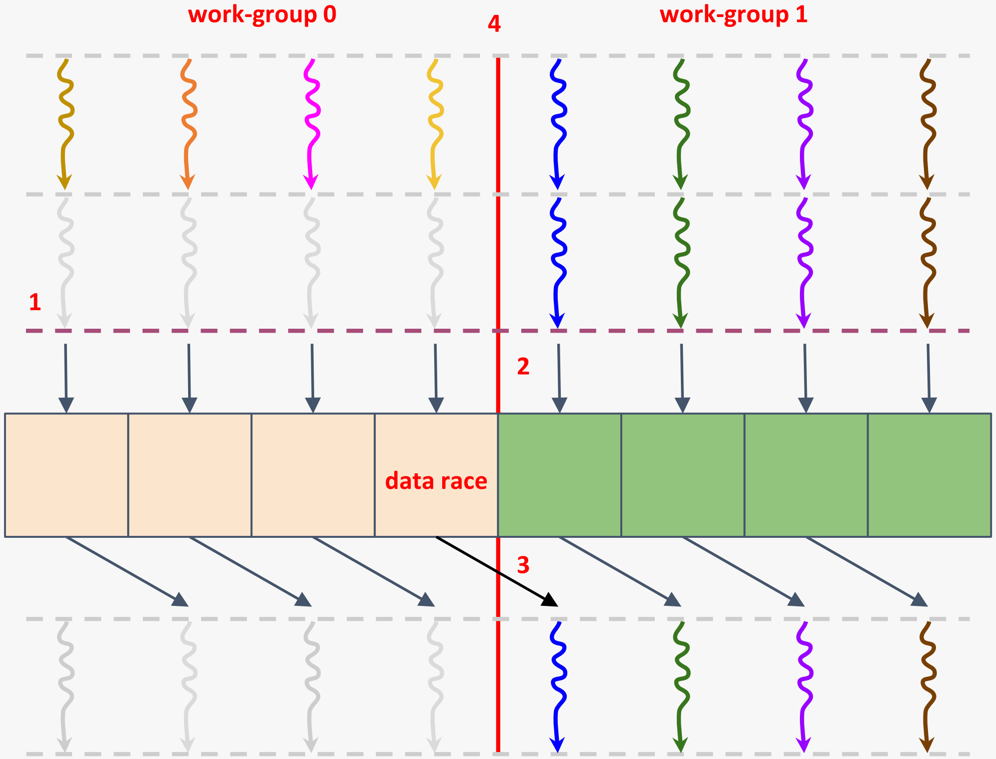
1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point



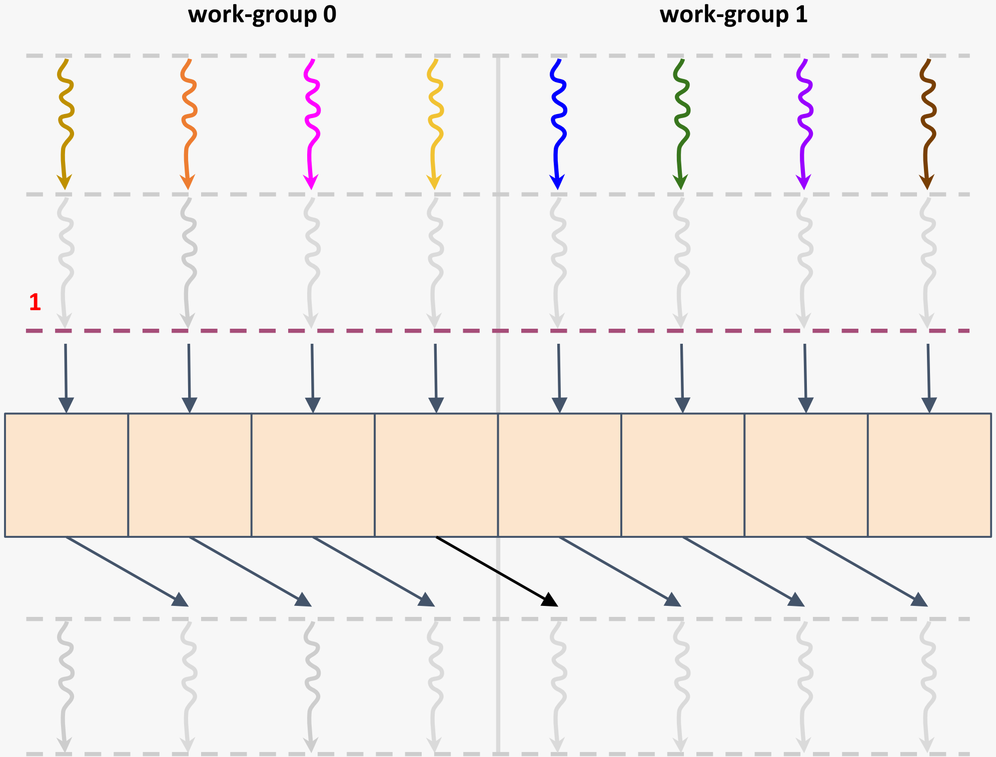
1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point



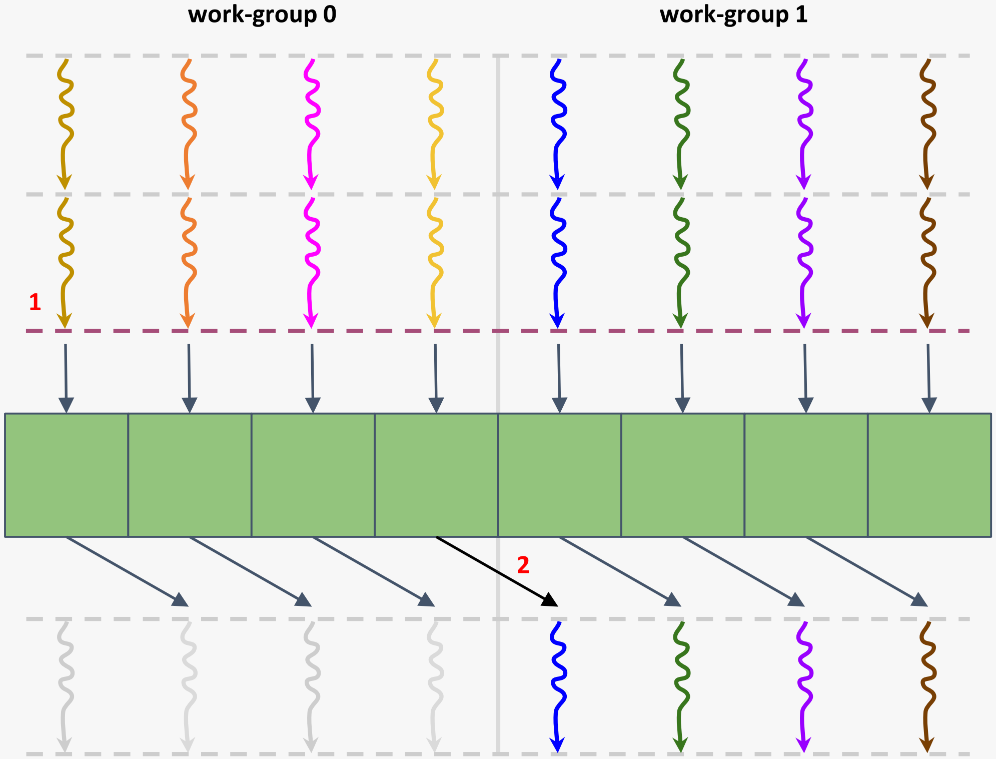
1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point
3. So now you can be sure that all of the results that you want to read from have been written to



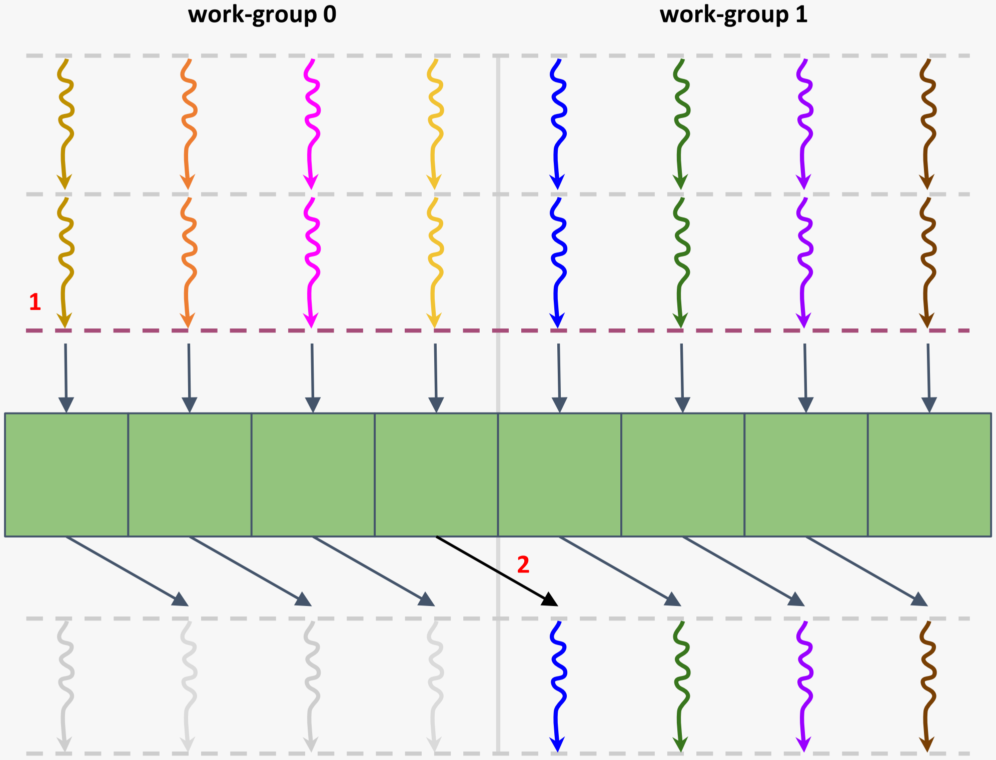
1. This problem can be solved by a synchronisation primitive called a work-group barrier
2. Work-items will block until all work-items in the work-group have reached that point
3. So now you can be sure that all of the results that you want to read from have been written to
4. However this does not apply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



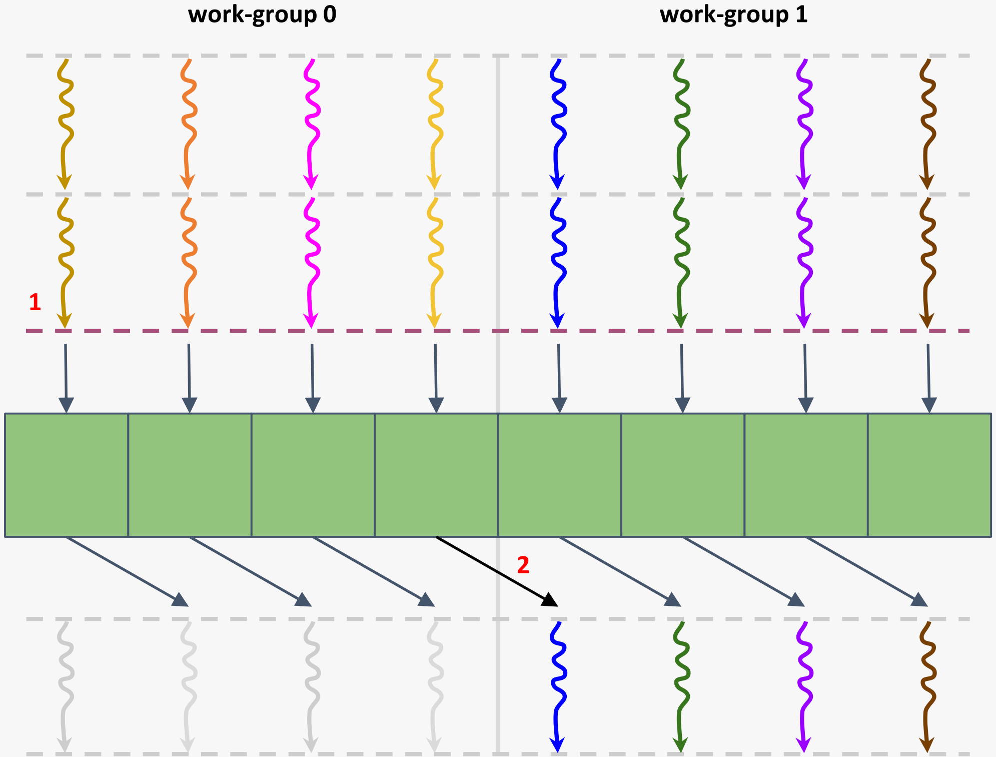
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)



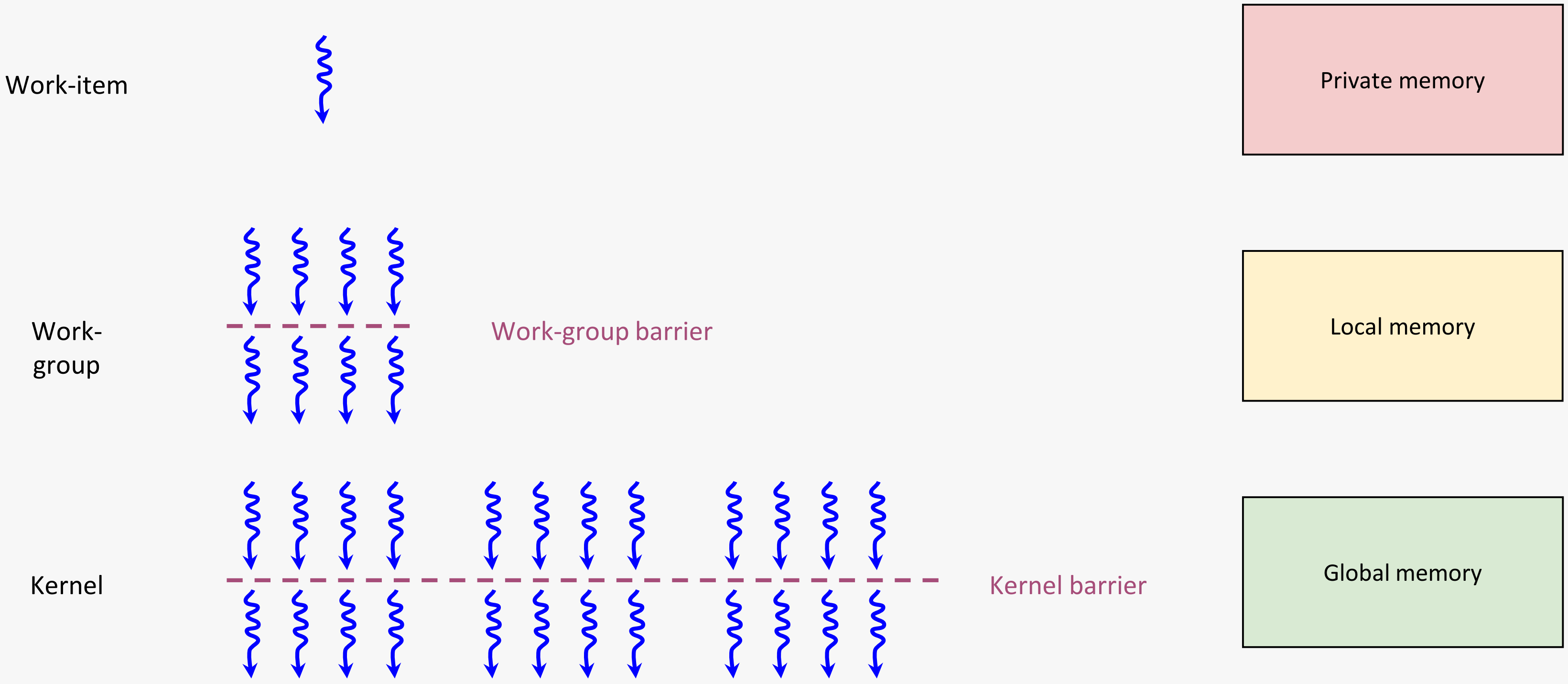
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)
2. Again you can be sure that all of the results that you want to read from have been written to



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



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



Work-item

Private memory

Work-group

Work-group barrier

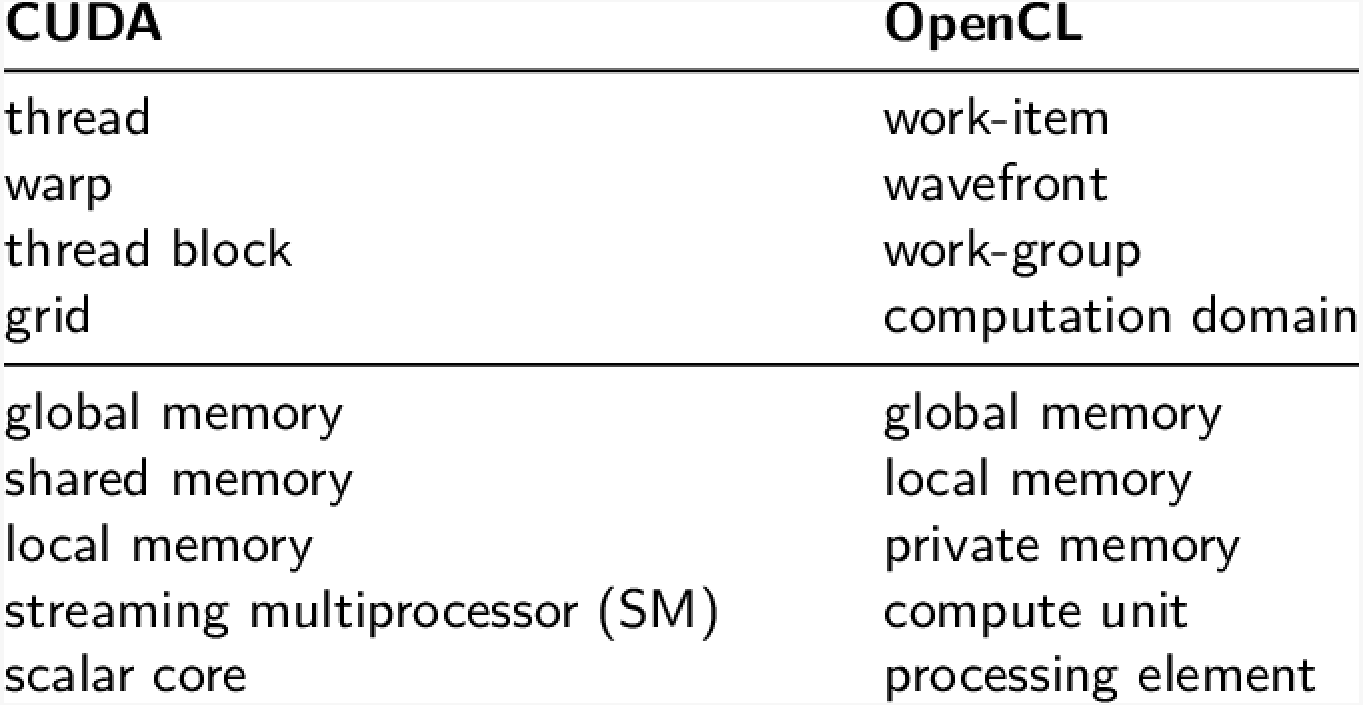
Local memory

Kernel

Kernel barrier

Global memory

CUDA vs OpenCL terminology



CUDA	OpenCL
thread	work-item
warp	wavefront
thread block	work-group
grid	computation domain
global memory	global memory
shared memory	local memory
local memory	private memory
streaming multiprocessor (SM)	compute unit
scalar core	processing element

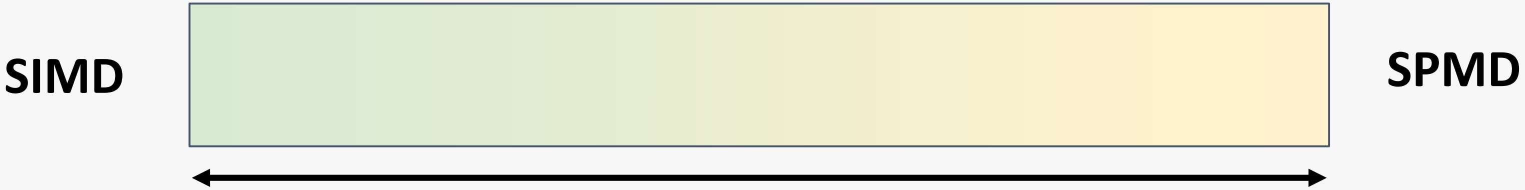
Sequential CPU code

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



calc(in, out);
```

SPMD GPU code

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


}


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

SIMD vs SPMD

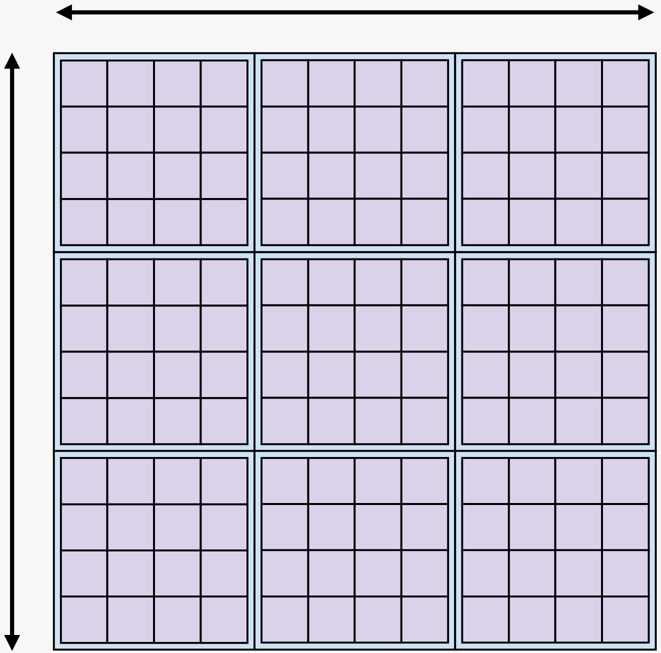


SPMD: Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data.

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

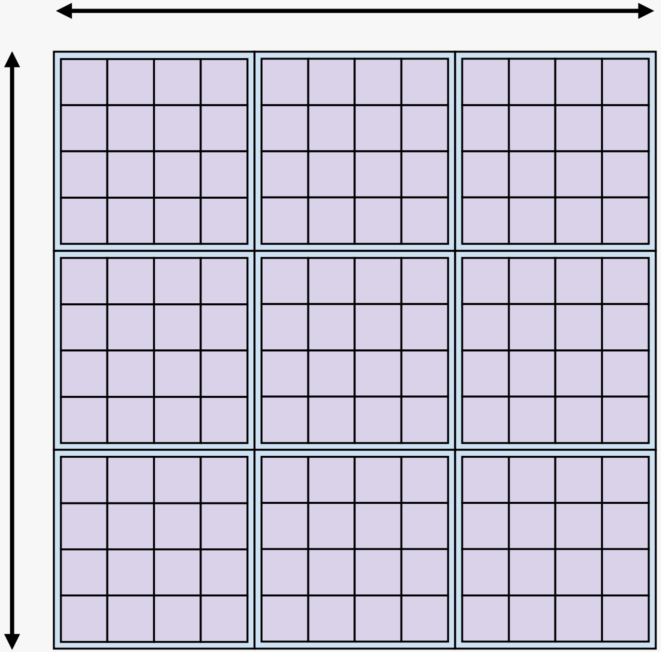
SPMD is a parallel execution model and assumes multiple cooperating processors executing a program.

`nd-range {{12, 12}, {4, 4}}`



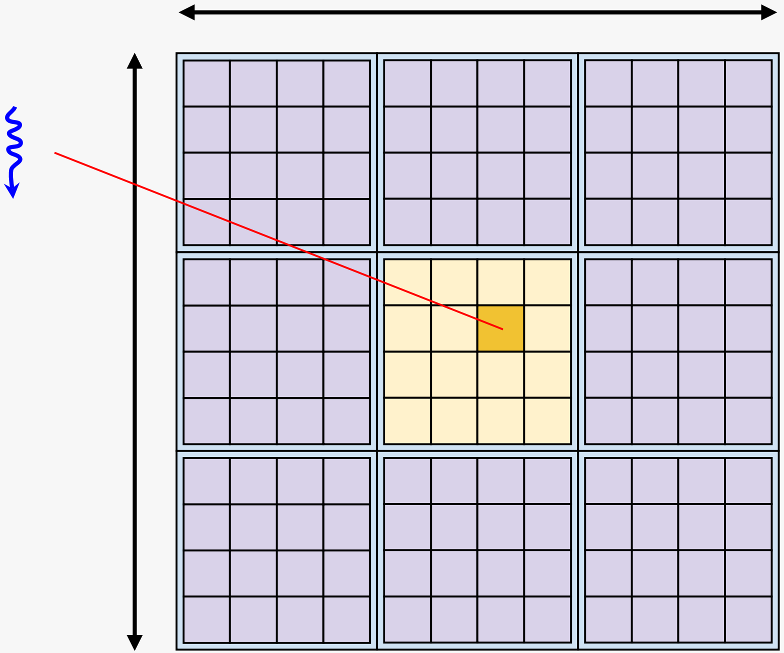
- Kernels are launched in the form of an nd-range
- An nd-range can be 1, 2 or 3 dimensions
- An nd-range describes a number of work-items divided into equally sized work-groups
- An nd-range is constructed from the total number of work-items (global range) and the number of work-items in a work-group (local range)

```
nd-range {{12, 12}, {4, 4}}
```



- An nd-range is mapped to the underlying hardware
  - Work-groups are mapped to compute units
  - Work-items are mapped to processing units

`nd-range {{12, 12}, {4, 4}}`

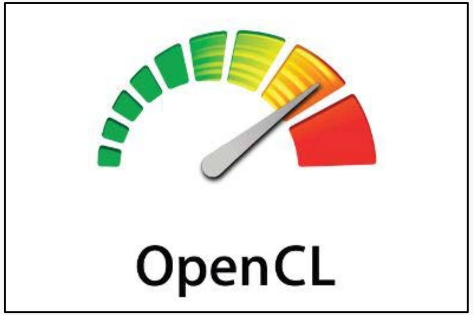


- The kernel is executed once per work-item in the nd-range
- Each work item knows it’s index within the nd-range
  a. global range {12, 12}
  b. local range {4, 4}
  c. group range {3, 3}
  d. global id {6, 5}
  e. local id {2, 1}
  f. group id {1, 1}

Act 3

1. What's still missing from C++?
2. What makes GPU work so fast?
3. What is Modern C++ that works on GPUs, CPUs, everything?

SYCL for OpenCL



Cross-platform, single-source, high-level, C++ programming layer

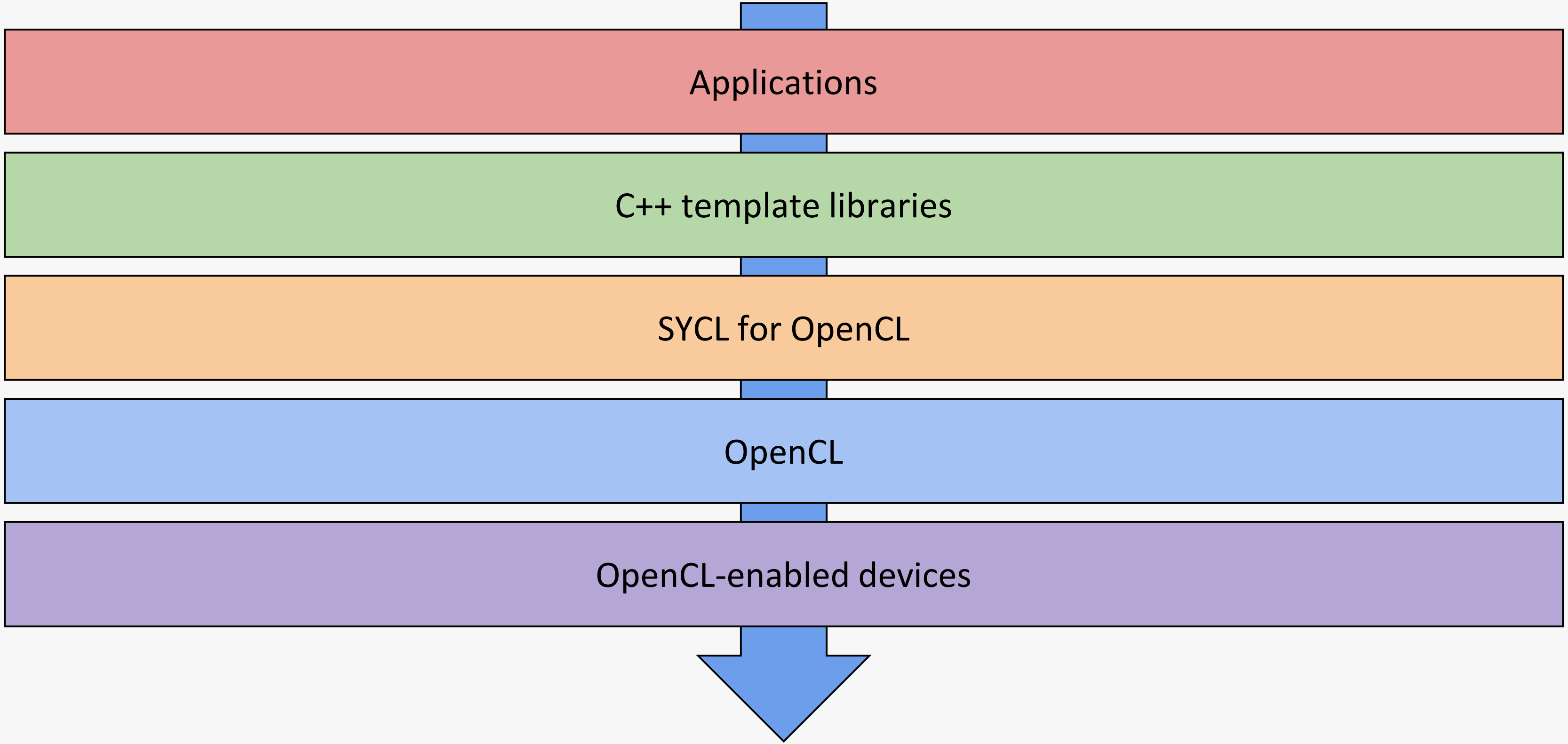
Built on top of OpenCL and based on standard C++11

Delivering a heterogeneous programming solution for C++

Why use SYCL to program a GPU?

- 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

The SYCL ecosystem



Applications

C++ template libraries

SYCL for OpenCL

OpenCL

OpenCL-enabled devices

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

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

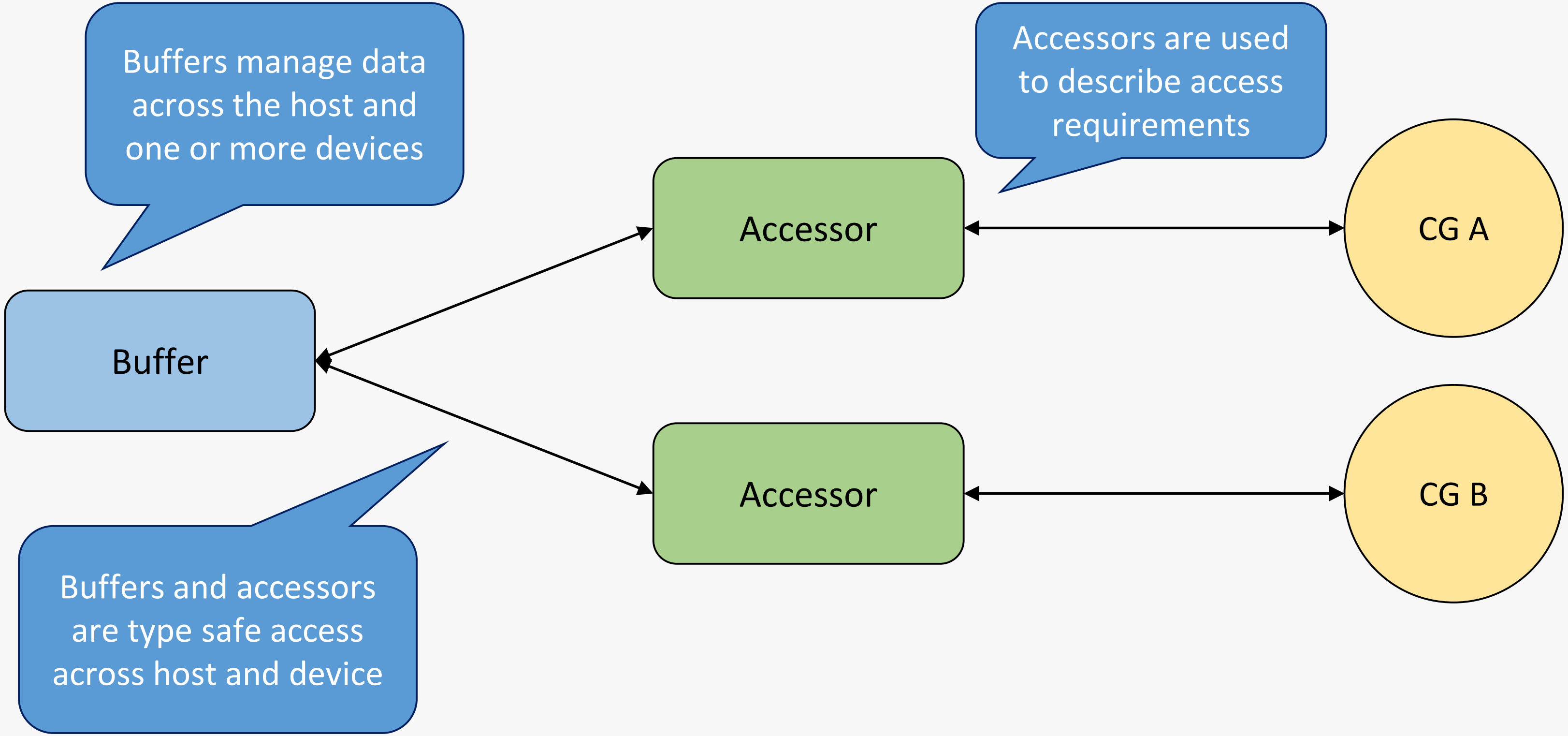
```
vector<float> a, b, c;

~~#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<vec_add>(range, [=](cl::sycl::id<2> idx) {
  c[idx] = a[idx] + c[idx];
}));
```

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



Buffers manage data across the host and one or more devices

Accessors are used to describe access requirements

Buffer

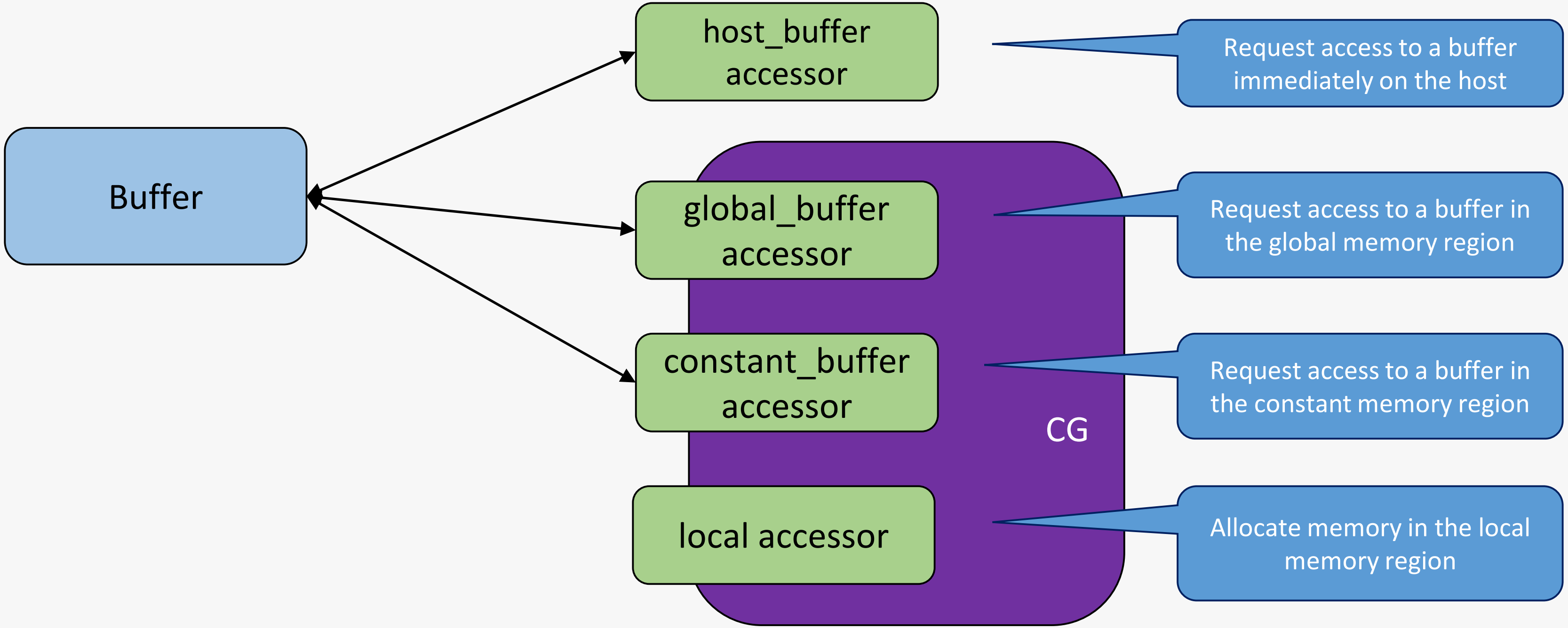
Accessor

Accessor

CG A

CG B

Buffers and accessors are type safe access across host and device

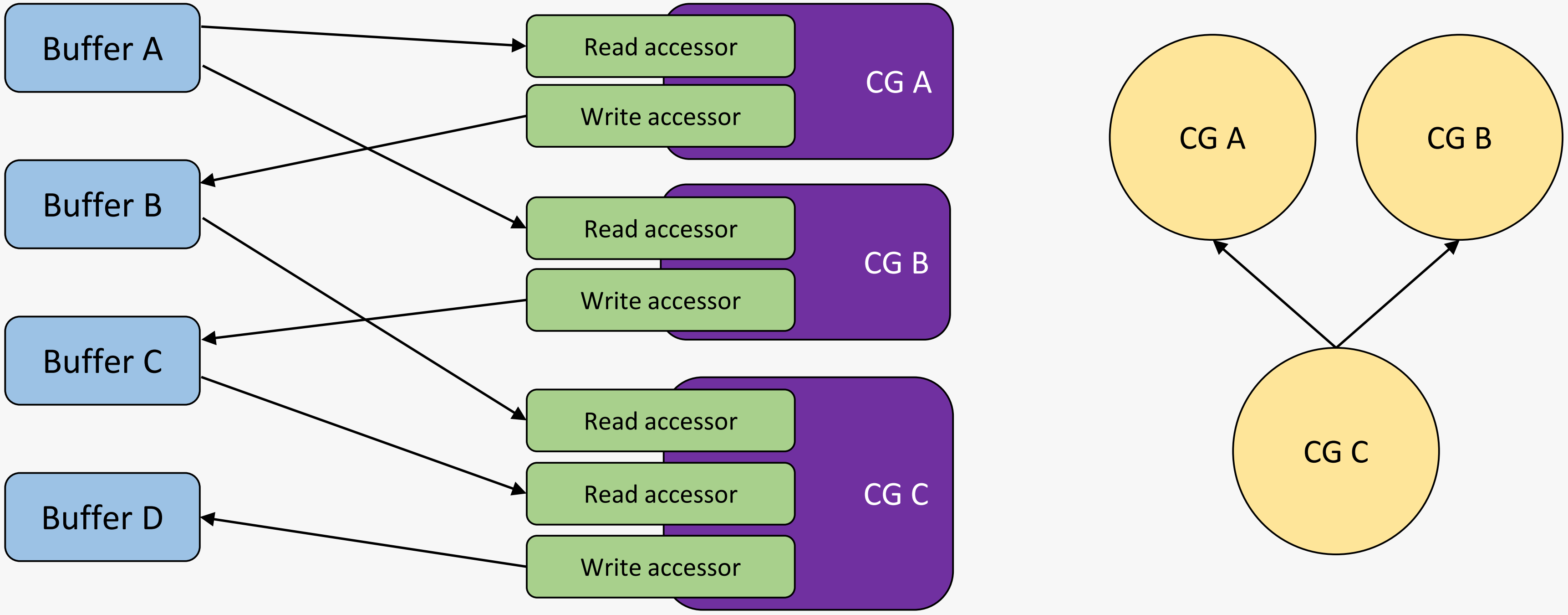


Request access to a buffer immediately on the host

Request access to a buffer in the global memory region

Request access to a buffer in the constant memory region

Allocate memory in the local memory region


Buffer A
Buffer B
Buffer C
Buffer D
Read accessor
Write accessor
CG A
Read accessor
Write accessor
CG B
Read accessor
Read accessor
Write accessor
CG C
CG A
CG B
CG C

Implicit vs Explicit Data Movement

Examples:

- SYCL, C++ AMP

Implementation:

- Data is moved to the device implicitly via cross host CPU / device data structures

Here we're using C++ AMP as an example

```
array_view<float> ptr;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx)
restrict(amp) {
  ptr[idx] *= 2.0f;
});
```

Examples:

- OpenCL, CUDA, OpenMP

Implementation:

- Data is moved to the device via explicit copy APIs

Here we're using CUDA as an example

```
float *h_a = { … }, d_a;
cudaMalloc((void **)&d_a, size);
cudaMemcpy(d_a, h_a, size,
  cudaMemcpyHostToDevice);
vec_add<<<64, 64>>>(a, b, c);
cudaMemcpy(d_a, h_a, size,
  cudaMemcpyDeviceToHost);
```

Benefits of data dependency task 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

Coverage after C++17

	Asynchronus Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
today's abstractions	C++11: thread,lambda function, TLS, async C++14: generic lambda	C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control false sharing	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence, C++ 17: scoped _lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies	C++17: , progress guarantees, TOE, execution policies

C++17 introduces a number of parallel algorithms and new execution policies which dictate how they can be parallelized

The new algorithms are unordered, allowing them to perform in parallel

Execution policies:

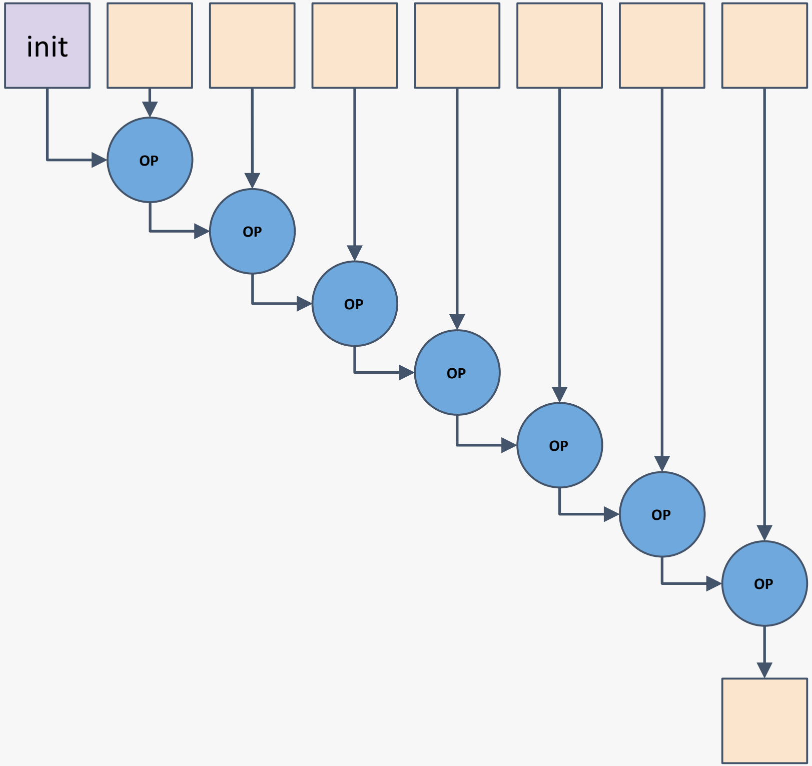
- `sequenced_execution_policy (seq)`
- `parallel_execution_policy (par)`
- `parallel_unsequenced_execution_policy (par_unseq)`

```
result accumulate(first, last,
                  init,
                  [binary_op])
```

```
first acc = init
then for each it in [first, last) in order
  acc = binary_op(acc, *it)
then return acc
```

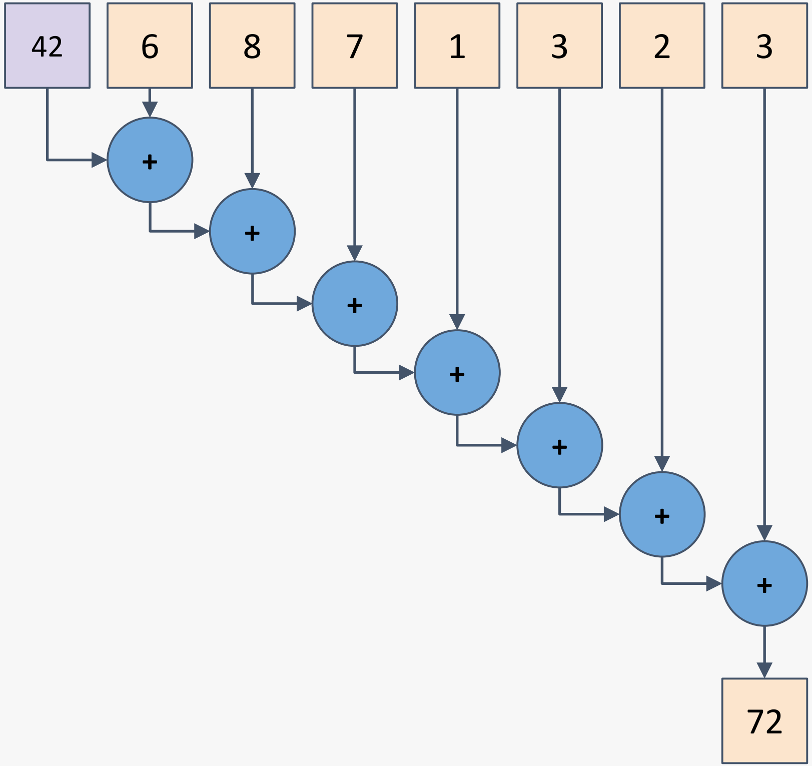
```
result accumulate(first, last,
                  init,
                  [binary_op])
```

```
first acc = init
then for each it in [first, last) in order
  acc = binary_op(acc, *it)
then return acc
```



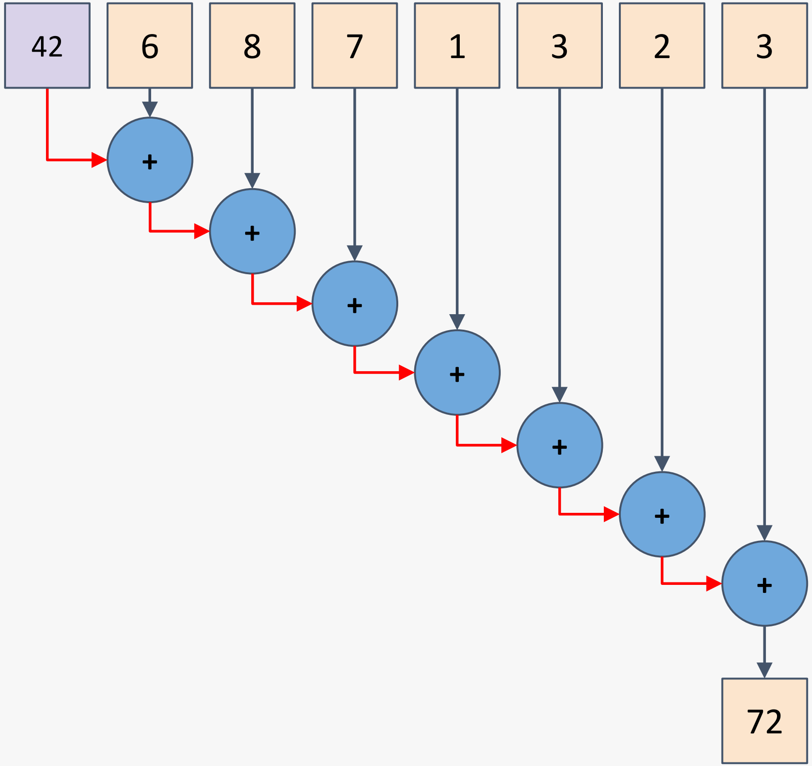
```
result accumulate(first, last,
                  init,
                  [binary_op])
```

```
first acc = init
then for each it in [first, last) in order
  acc = binary_op(acc, *it)
then return acc
```



```
result accumulate(first, last,
                  init,
                  [binary_op])
```

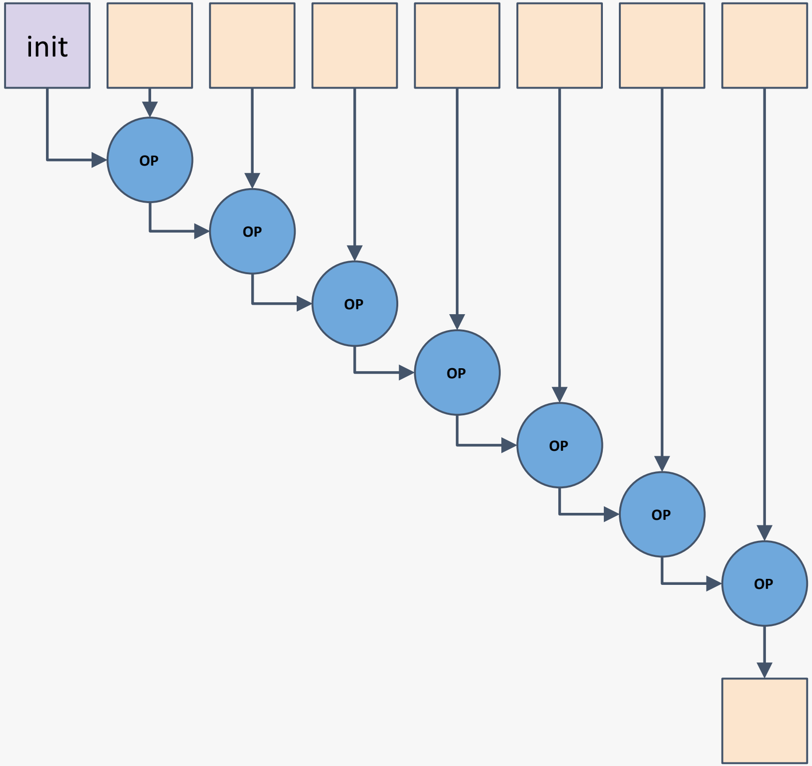
```
first  acc = init
then for each it in [first, last) in order
  acc = binary_op(acc, *it)
then return acc
```



```
result reduce([execution_policy,]
              first, last,
              init,
              [binary_op])

first acc = GSUM(binary_op, init,
                 *first, …,
                 *(last-1))
then return acc
```

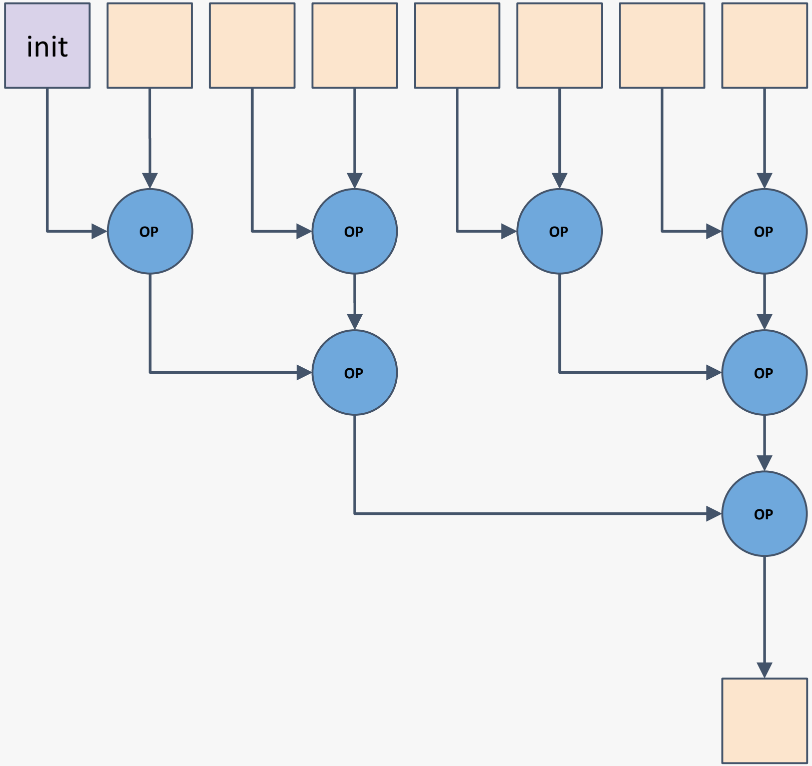
```
result reduce([execution_policy,]
              first, last,
              init,
              [binary_op])
```

```
first acc = GSUM(binary_op, init,
                  *first, …,
                  *(last-1))
then return acc
```



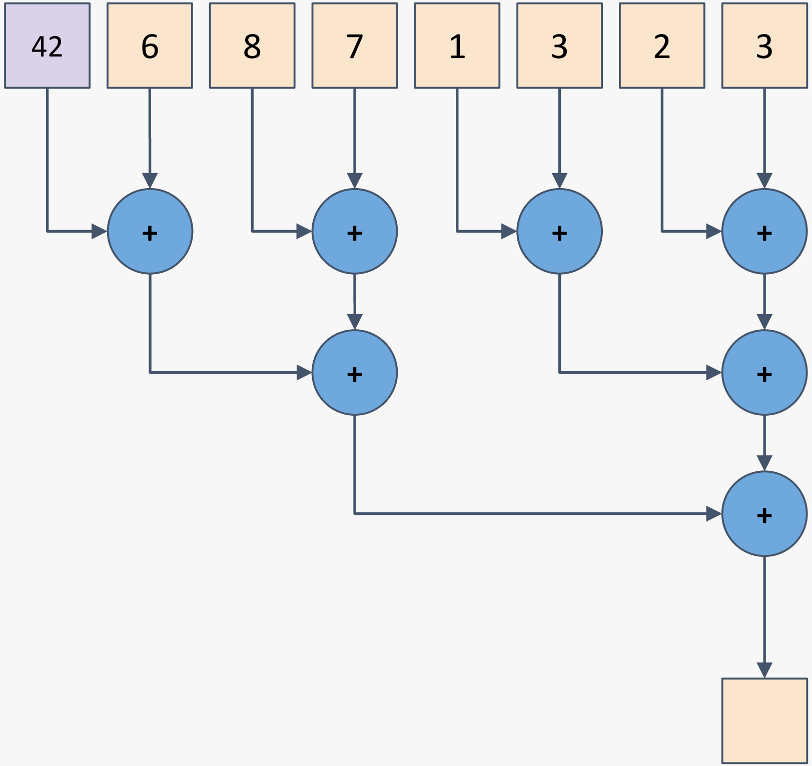
```
result reduce([execution_policy,]
              first, last,
              init,
              [binary_op])
```

```
first acc = GSUM(binary_op, init,
                  *first, …,
                  *(last-1))
then return acc
```



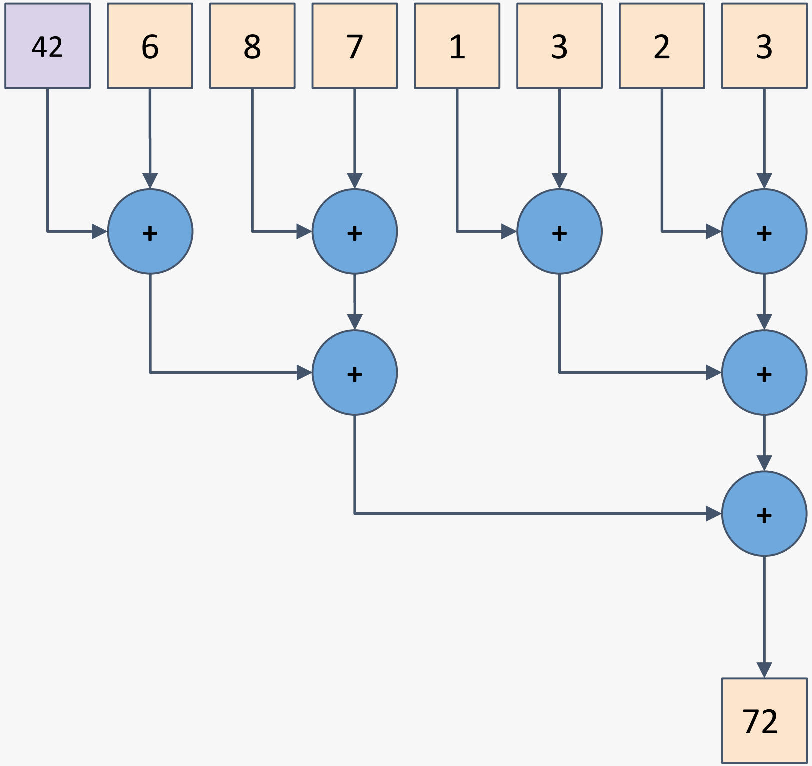
```
result reduce([execution_policy,]
              first, last,
              init,
              [binary_op])
```

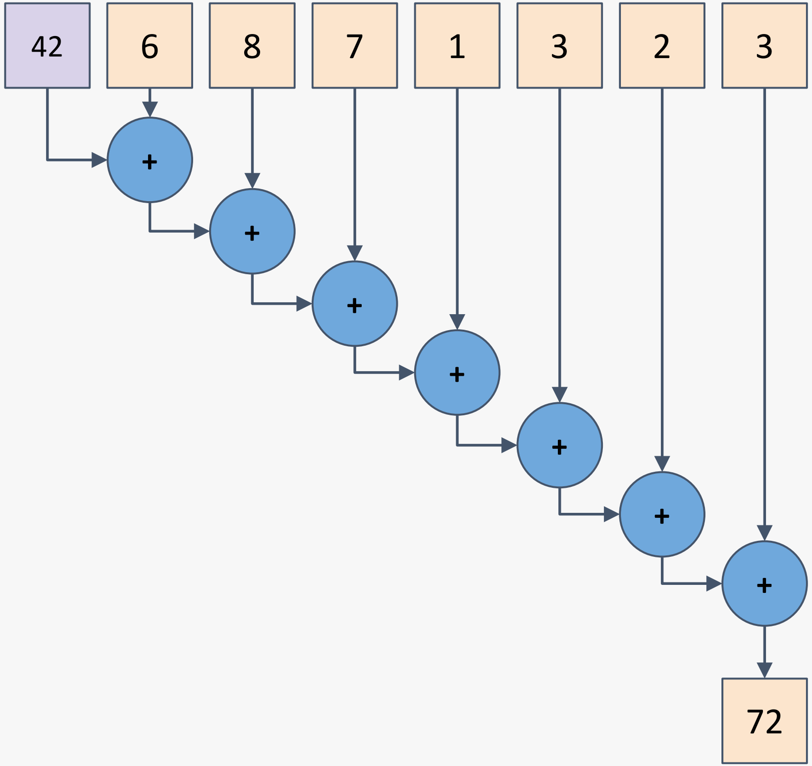
```
first acc = GSUM(binary_op, init,
                 *first, …,
                 *(last-1))
then return acc
```



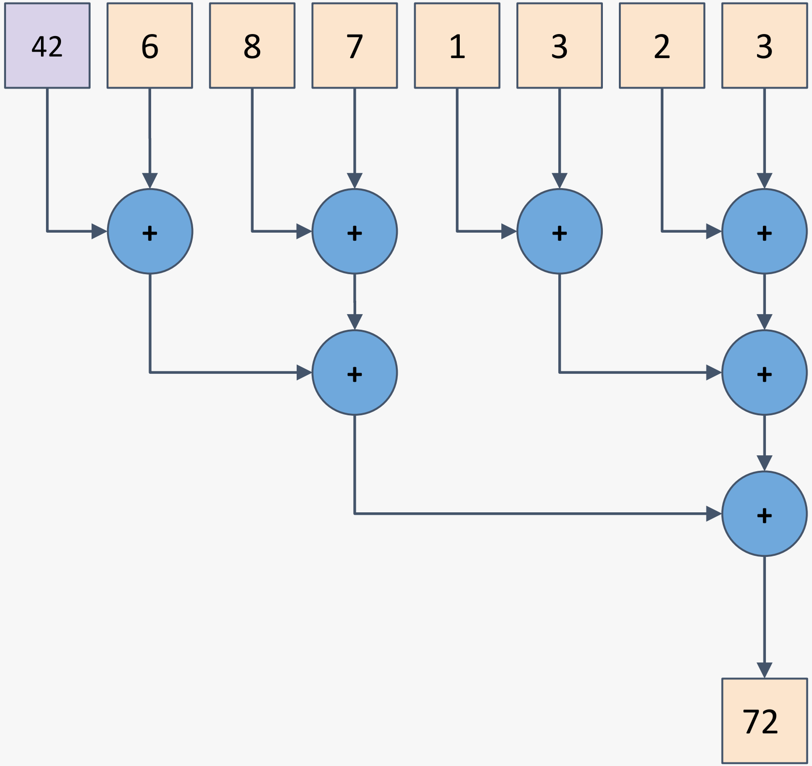
```
result reduce([execution_policy,]
              first, last,
              init,
              [binary_op])
```

```
first acc = GSUM(binary_op, init,
                  *first, …,
                  *(last-1))
then return acc
```

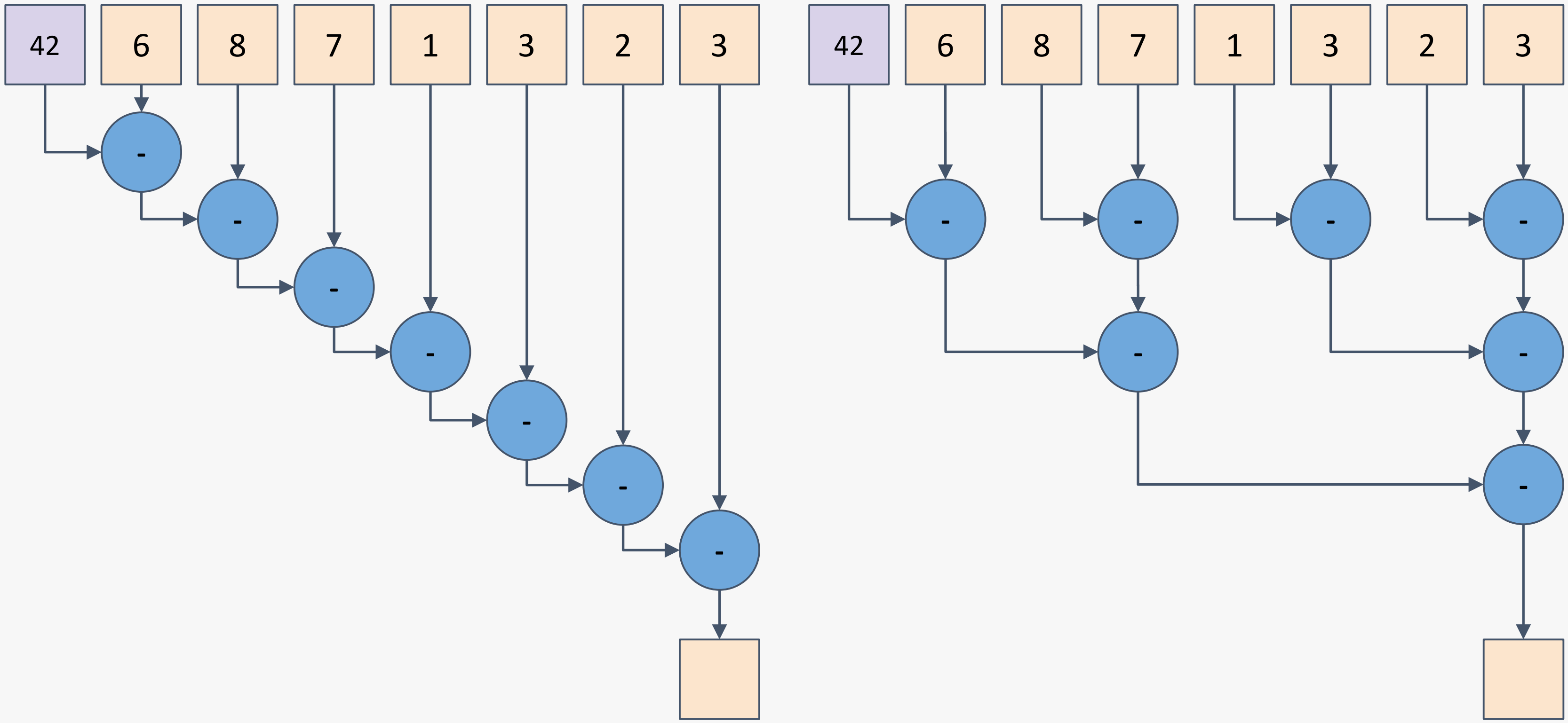


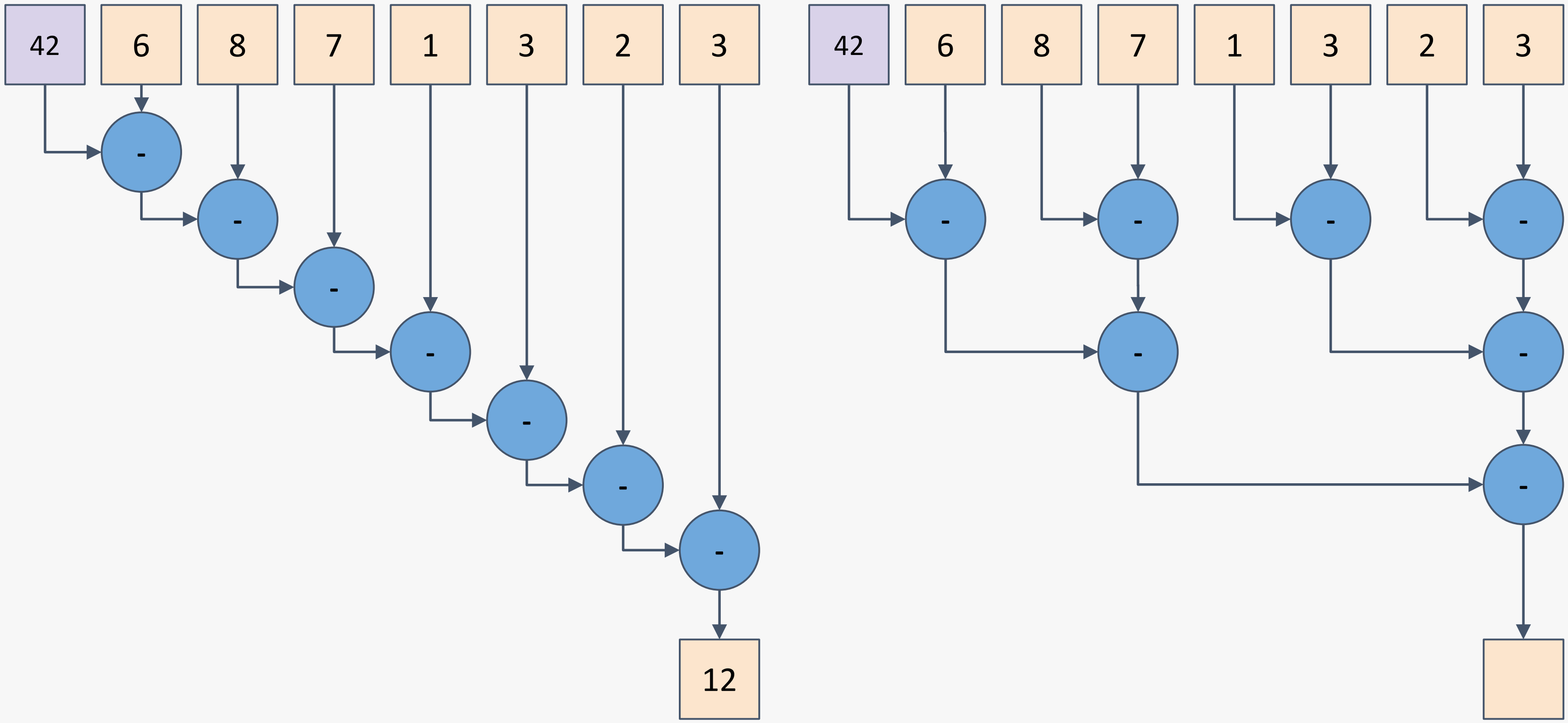


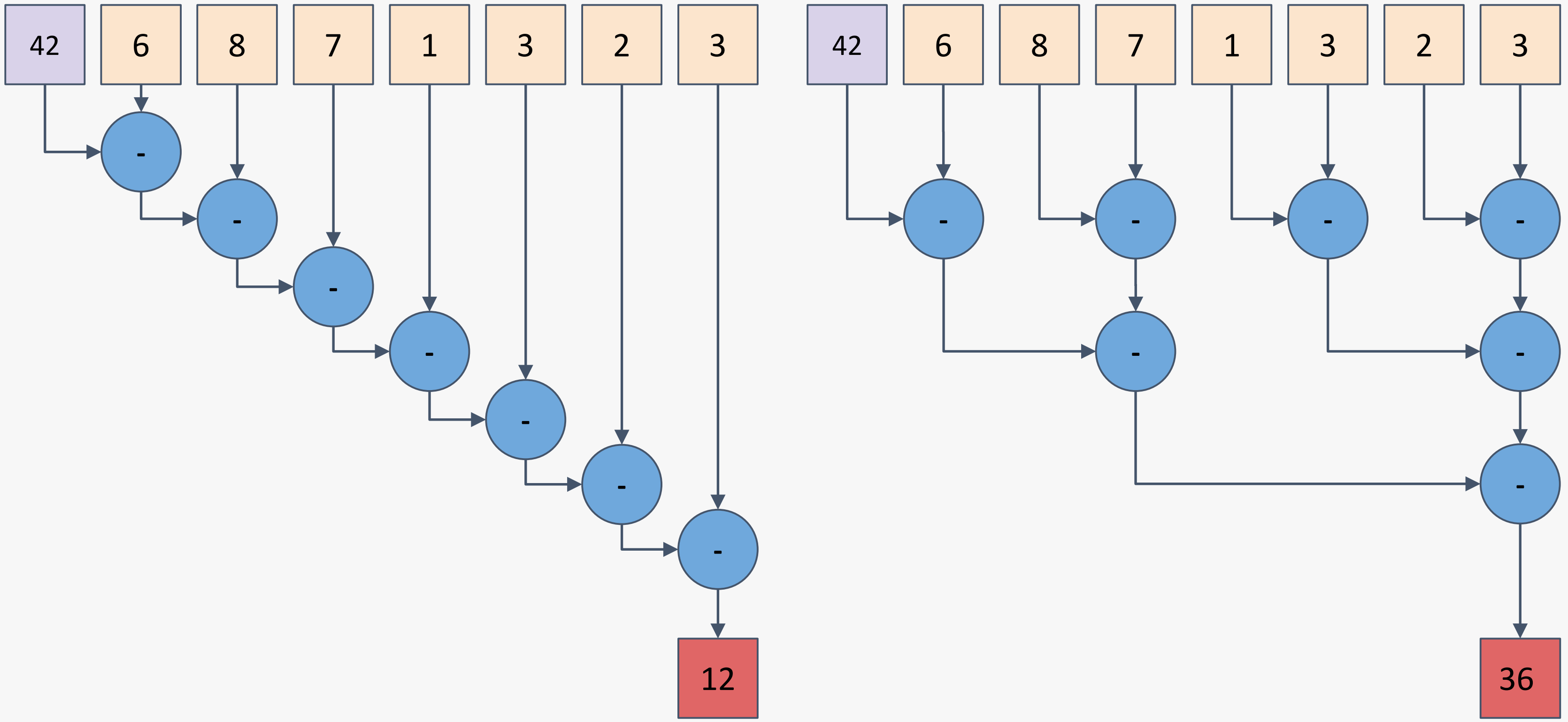
42
6
8
7
1
3
2
3
+
+
+
+
+
+
+
72



42
6
8
7
1
3
2
3
+
+
+
+
+
+
+
72


42
6
8
7
1
3
2
3
-
-
-
-
-
-
-
42
6
8
7
1
3
2
3
-
-
-
-
-
-
-


42
6
8
7
1
3
2
3
-
-
-
-
-
-
-
12
42
6
8
7
1
3
2
3
-
-
-
-
-
-
-


42
6
8
7
1
3
2
3
-
-
-
-
-
-
-
12
42
6
8
7
1
3
2
3
-
-
-
-
-
-
-
36

Due to the requirements of `GSUM` `reduce` is allowed to be unordered

However this means that `binary_op` is required to be both **commutative** and **associative**

Commutativity means changing the order of operations does not change the result

Integer operations

x + y == y + x

x * y == y * x

x - y != y - x

x / y != y / x

Floating-point operations

x + y == y + x

x * y == y * x

x - y != y - x

x / y != y / x

Associativity means changing the grouping of operations does not change the result

Integer operations

(x + y) + z == x + (y + z)

(x * y) * z == x * (y * z)

(x - y) - z != x - (y - z)

(x / y) / z != x / (y / z)

Floating-point operations

(x + y) + z != x + (y + z)

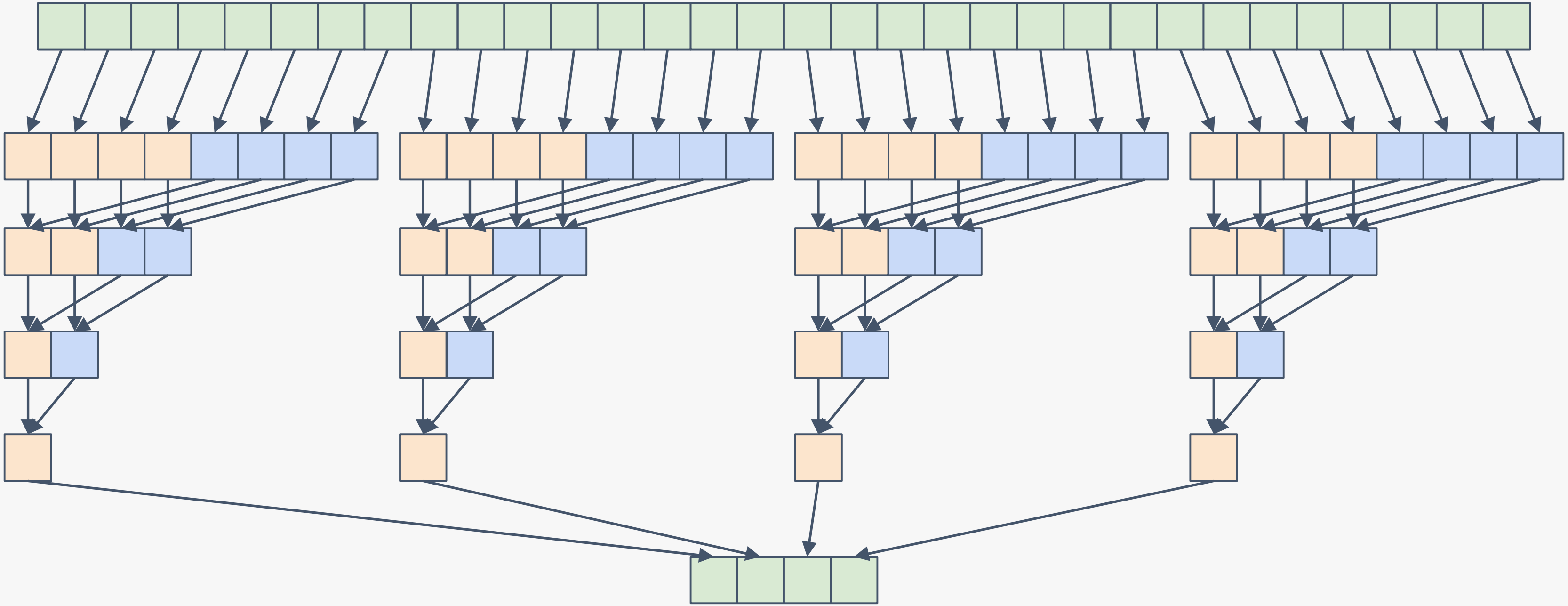
(x * y) * z != x * (y * z)

(x - y) - z != x - (y - z)

(x / y) / z != x / (y / z)

So how do we parallelise this 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



```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {


















}
```

Here we have the standard prototype for the reduce parallel algorithm, taking a SYCL execution policy

There is an assumption here that the iterators are contiguous

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
}
```

SYCL separates memory storage and access using buffers and accessors

Buffers manage a region of memory across host and one or more devices

Accessors represent an instance of access to a particular buffer

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);

}
```

We create a buffer to manage the input data

We call set_final_data with nullptr in order to tell the runtime not to copy back to the original host address on destruction

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
















}
```

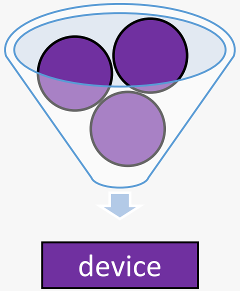
Buffers synchronise and copy their data back to the original pointer when they are destroyed

So in this case, on returning from the reduce function

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
















}
```

In SYCL devices are selected using a device selector

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



```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
















}
```

We create a queue that we can enqueue work on taking a gpu_selector, which will return a GPU to execute work on

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();









}
```

We deduce the data size of the input range and the maximum work-group size

These are important for determining how work is distributed across work-groups

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {



















    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We create a loop that will launch a SYCL kernel for each kernel invocation required for the reduction

After each iteration the data size is divided by the work-group size

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {









    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

In SYCL all work is enqueued to a queue via command groups which represent the kernel function, an nd-range and the data dependencies

We create a command group to enqueue a kernel

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));

    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);

}
```

We determine the global range to be the data size

We determine the local range to be the max work group size, providing that's smaller than the data size

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);



      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We create an accessor for the input buffer

The access mode is read_write because we want to be able to write back a result

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);











      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

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

We create a local accessor of elements of value type with the size of the local range

```
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<          >(nd_range<1>(global, local), [=](nd_item<1> it) {










      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

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

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

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {



      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);

}
```

We provide a template parameter to parallel_for to name the kernel function

This is necessary for portability between C++ compilers

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];










      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We copy each element from global memory to local memory of their respective work-group

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);









      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

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

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {





        }

      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We create a loop that will iterate over the work-items in the work-group and providing an offset to the midpoint

codeplay

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {


          }

        }

      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We branch on the first half of the work-items per loop by only executing work-items before the offset

codeplay

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                    scratch[it.get_local_id(0) + offset]);

          }

        }

      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We call the binary_op with the elements in local memory of the current work-item and the respective work-item on the other side of the offset and assign the result to the element in local memory of the current work-item

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                    scratch[it.get_local_id(0) + offset]);
          }
          it.barrier(access::fence_space::local_space);
        }

      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

We insert a barrier to ensure all work-items in the current loop have performed their operation

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                    scratch[it.get_local_id(0) + offset]);
          }
          it.barrier(access::fence_space::local_space);
        }
        if (it.get_local_id(0) == 0) { inputAcc[it.get_group(0)] = scratch[it.get_local_id(0)]; }
      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);


}
```

Once the loop has complete there will be a single value for each work-group in local memory for the first work-item

We copy this value into an element in global memory for the current work group

codeplay

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                    scratch[it.get_local_id(0) + offset]);
          }
          it.barrier(access::fence_space::local_space);
        }
        if (it.get_local_id(0) == 0) { inputAcc[it.get_group(0)] = scratch[it.get_local_id(0)]; }
      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);
  auto accH = bufI.template get_access<access::mode::read>();

}
```

A host accessor provides immediate access to data maintained by a buffer

We create a host accessor to retrieve the final result of the reduction

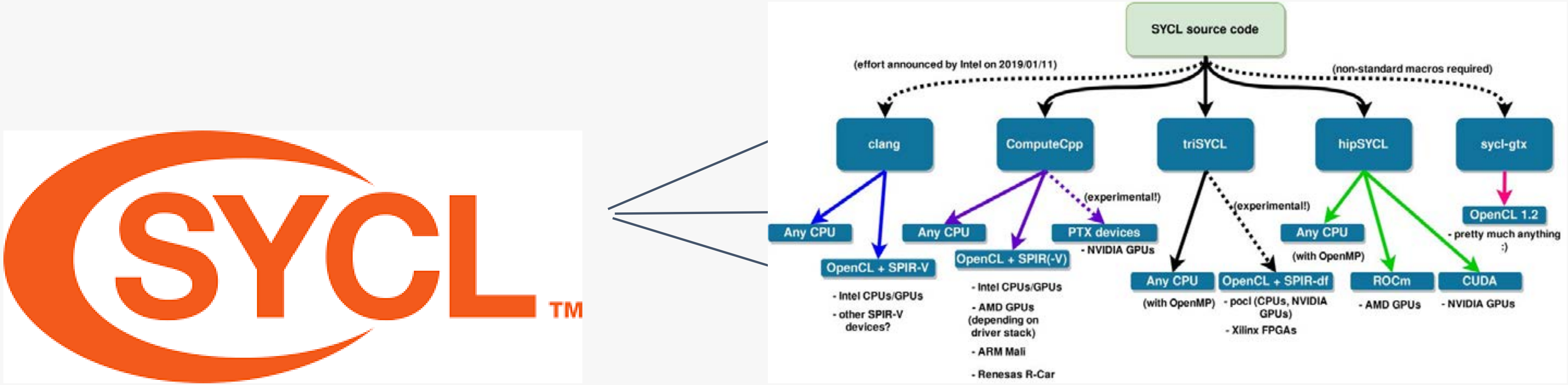
```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local> scratch(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                    scratch[it.get_local_id(0) + offset]);
          }
          it.barrier(access::fence_space::local_space);
        }
        if (it.get_local_id(0) == 0) { inputAcc[it.get_group(0)] = scratch[it.get_local_id(0)]; }
      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);
  auto accH = bufI.template get_access<access::mode::read>();
  return binary_op(init, accH[0]);
}
```

Once the data size has been reduced to 1 this means the reduction is complete and we can return the result

We call binary_op with init and the result of the reduction and then we return the result

```
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                    scratch[it.get_local_id(0) + offset]);
          }
          it.barrier(access::fence_space::local_space);
        }
        if (it.get_local_id(0) == 0) { inputAcc[it.get_group(0)] = scratch[it.get_local_id(0)]; }
      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);
  auto accH = bufI.template get_access<access::mode::read>();
  return binary_op(init, accH[0]);
}
```

C++
Ignite your applications with XL Compilers for C, C++, & Fortran
IBM Power
AIX
Linux
GCC
Visual C++
Edison Design Group



SYCL source code

(effort announced by Intel on 2019/01/11)

(non-standard macros required)

clang

ComputeCpp

triSYCL

hipSYCL

sycl-gtx

Any CPU

OpenCL + SPIR-V
- Intel CPUs/GPUs
- other SPIR-V devices?

Any CPU

OpenCL + SPIR(-V)
- Intel CPUs/GPUs
- AMD GPUs (depending on driver stack)
- ARM Mali
- Renesas R-Car

(experimental!)

PTX devices
- NVIDIA GPUs

Any CPU
(with OpenMP)

(experimental!)

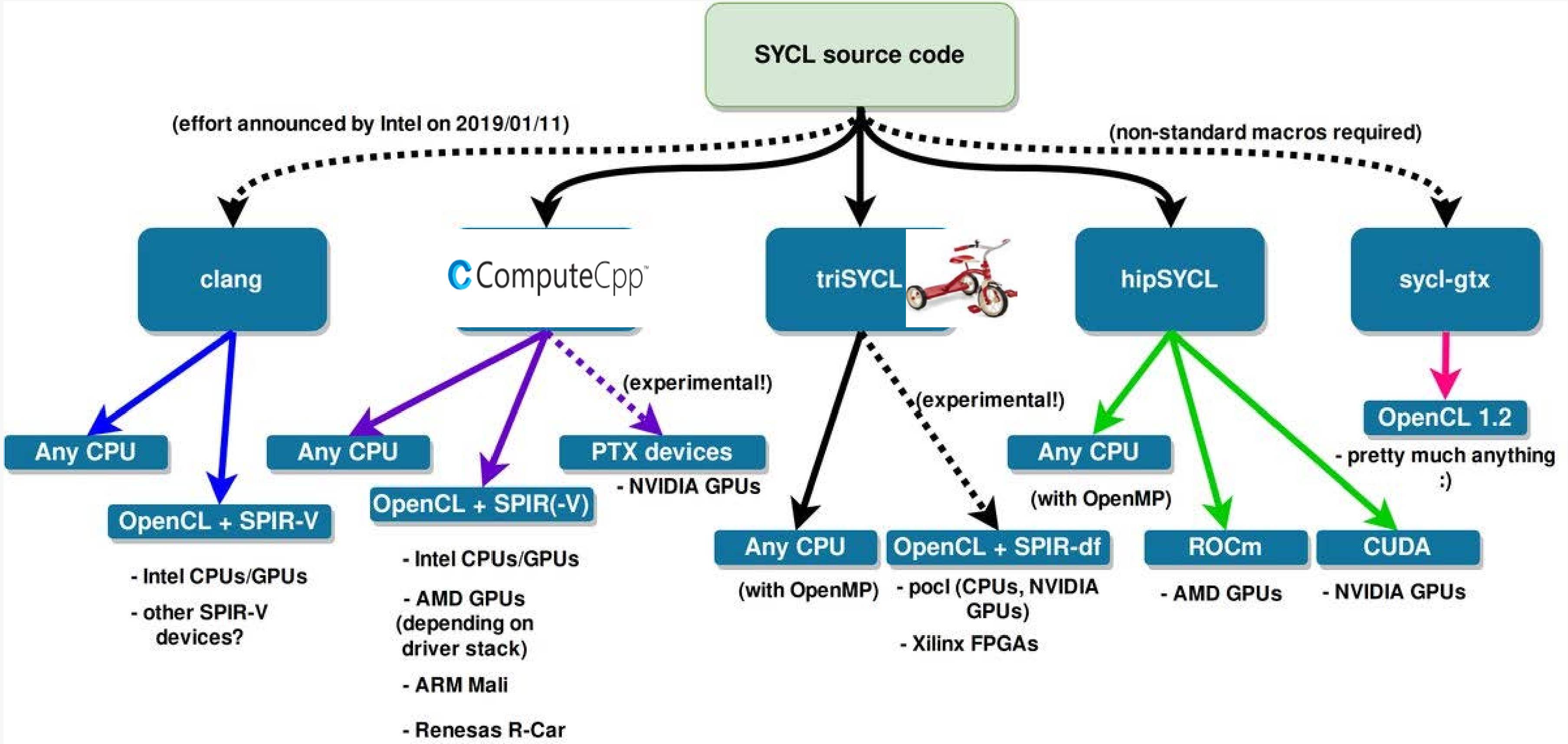
OpenCL + SPIR-df
- pocl (CPUs, NVIDIA GPUs)
- Xilinx FPGAs

Any CPU
(with OpenMP)

ROCm
- AMD GPUs

CUDA
- NVIDIA GPUs

OpenCL 1.2
- pretty much anything :)


SYCL source code
(effort announced by Intel on 2019/01/11)
(non-standard macros required)
clang
ComputeCpp
triSYCL
hipSYCL
sycl-gtx
Any CPU
OpenCL + SPIR-V
- Intel CPUs/GPUs
- other SPIR-V devices?
Any CPU
OpenCL + SPIR(-V)
- Intel CPUs/GPUs
- AMD GPUs (depending on driver stack)
- ARM Mali
- Renesas R-Car
(experimental!)
PTX devices
- NVIDIA GPUs
(experimental!)
Any CPU
(with OpenMP)
OpenCL + SPIR-df
- pocl (CPUs, NVIDIA GPUs)
- Xilinx FPGAs
Any CPU
(with OpenMP)
ROCm
- AMD GPUs
CUDA
- NVIDIA GPUs
OpenCL 1.2
- pretty much anything :)

Conclusion

We looked at how to write a reduction for the GPU in C++ using SYCL

We looked at how the SYCL programming model allows us to do this

We looked at how this applies to the GPU architecture

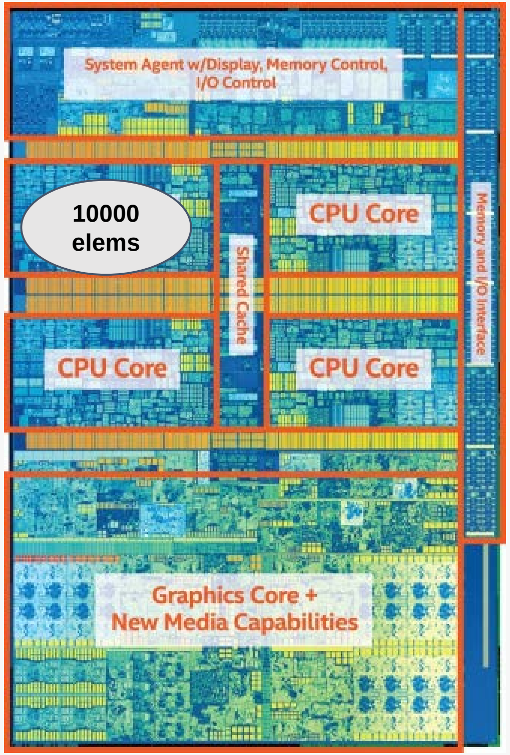
We looked at why this is so important in modern C++

Use the Proper Abstraction with C++

Abstraction	How is it supported
Cores	C++11/14/17 threads, async
HW threads	C++11/14/17 threads, async
Vectors	Parallelism TS2->C++20
Atomic, Fences, lockfree, futures, counters, transactions	C++11/14/17 atomics, Concurrency TS1->C++20, Transactional Memory TS1
Parallel Loops	Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each
Heterogeneous offload, fpga	OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja P0796 on affinity
Distributed	HPX, MPI, UPC++ P0796 on affinity
Caches	C++17 false sharing support
Numa	Executors, Execution Context, Affinity, P0443->Executor TS
TLS	EALS, P0772
Exception handling in concurrent environment	EH reduction properties P0797

Oh, and one more thing

What can I do with a Parallel For Each?

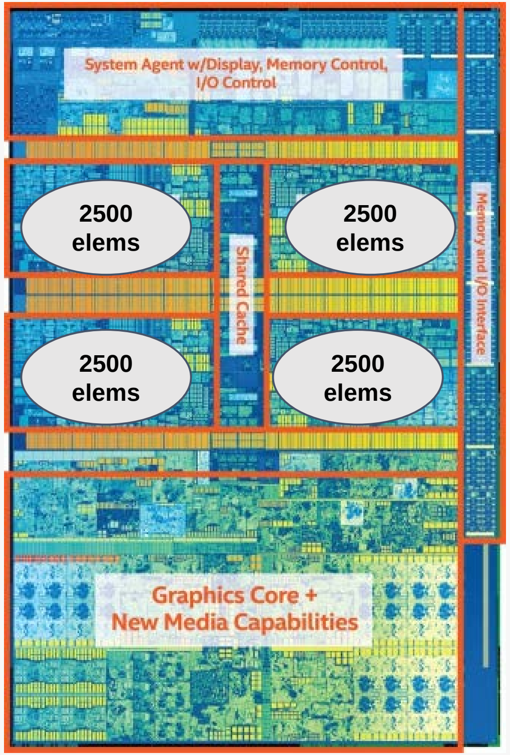

System Agent w/Display, Memory Control, I/O Control
10000 elems
CPU Core
Shared Cache
CPU Core
CPU Core
Memory and I/O Interface
Graphics Core + New Media Capabilities

Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::begin(v1), nElems, 1);

std::for_each(std::begin(v), std::end(v),
                          [=](float f) { f * f + f });
```

Traditional for each uses only one core, rest of the die is unutilized!

What can I do with a Parallel For Each?


System Agent w/Display, Memory Control, I/O Control
2500 elems
2500 elems
2500 elems
2500 elems
Shared Cache
Memory and I/O Interface
Graphics Core + New Media Capabilities

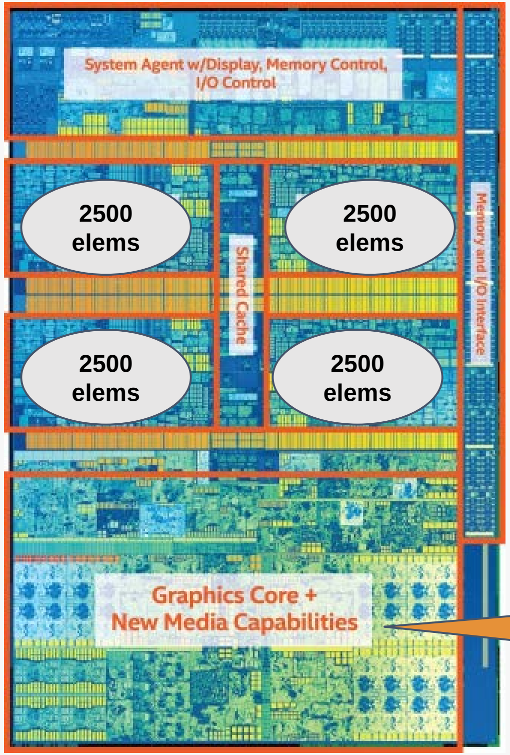
Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::execution_policy::par,
            std::begin(v1), nElems, 1);

std::for_each(std::execution_policy::par,
              std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?


System Agent w/Display, Memory Control, I/O Control
2500 elems
2500 elems
2500 elems
2500 elems
Shared Cache
Memory and I/O Interface
Graphics Core + New Media Capabilities

Intel Core i7 7th generation

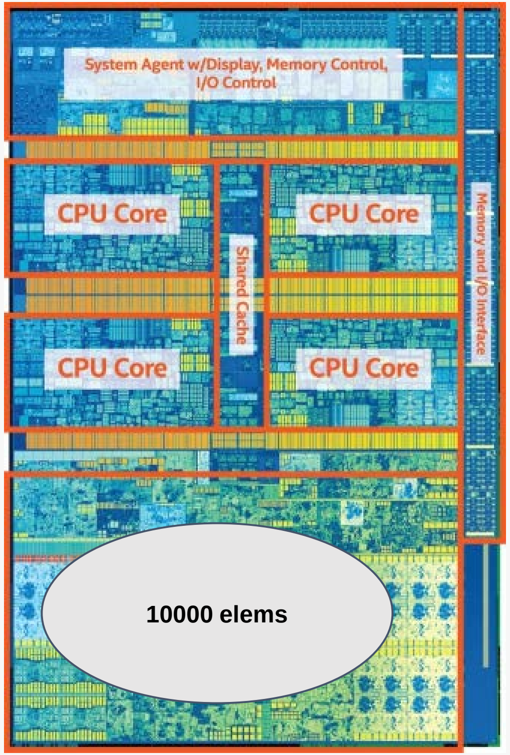
```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(std::execution_policy::par,
            std::begin(v1), nElems, 1);

std::for_each(std::execution_policy::par,
              std::begin(v), std::end(v),
               [=](float f) { f * f + f });
```

What about this part?

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?



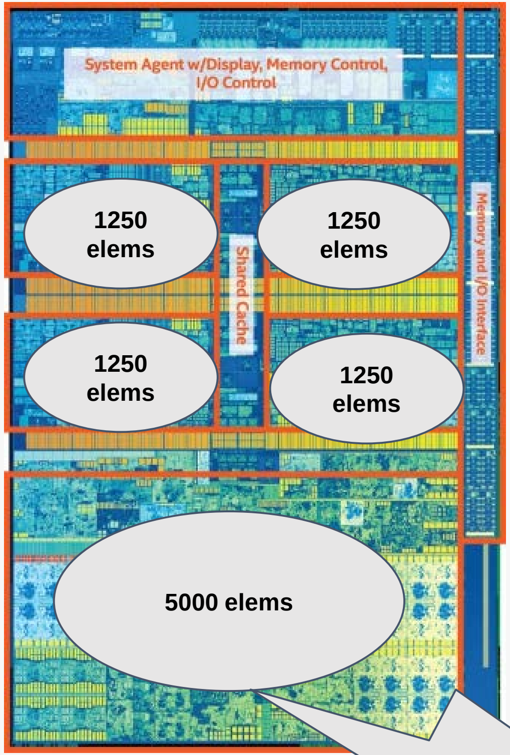
Intel Core i7 7th generation

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(sycl_policy,
            std::begin(v1), nElems, 1);

std::for_each(sycl_named_policy
                 <class KernelName>,
              std::begin(v), std::end(v),
              [=](float f) { f * f + f });
```

Workload is distributed on the GPU cores

(mileage may vary, implementation-specific behaviour)

What can I do with a Parallel For Each?



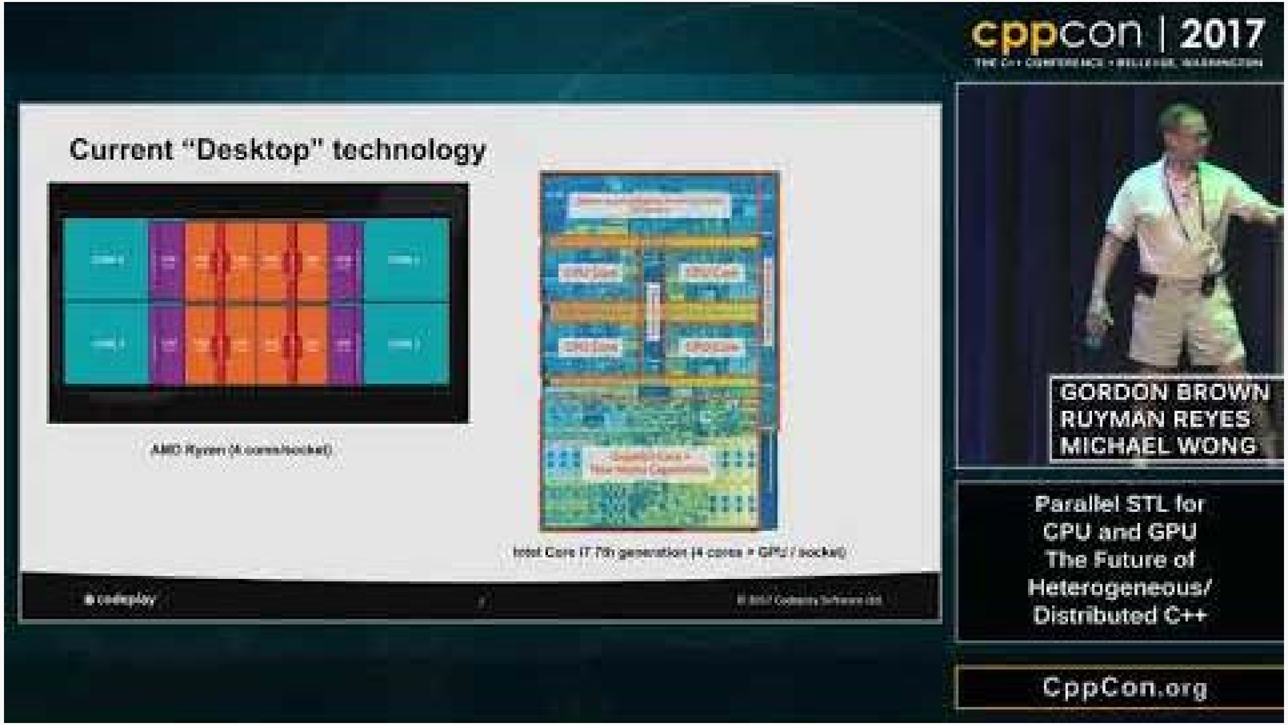
Intel Core i7 7th

```
size_t nElems = 1000u;
std::vector<float> nums(nElems);

std::fill_n(sycl_heter_policy(cpu, gpu, 0.5),
                std::begin(v1), nElems, 1);

std::for_each(sycl_heter_policy<class kName>
                     (cpu, gpu, 0.5),
                     std::begin(v), std::end(v),
                      [=](float f) { f * f + f });
```

Workload is distributed on all cores!

Experimental!

(mileage may vary, implementation-specific behaviour)



Current "Desktop" technology

AMD Ryzen (4 cores/socket)

Intel Core i7 7th generation (4 cores + GPU / socket)

codeplay

© 2017 Codeplay Software Ltd.

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
GORDON BROWN
RUYMAN REYES
MICHAEL WONG
Parallel STL for CPU and GPU The Future of Heterogeneous/ Distributed C++
CppCon.org

Demo Results - Running std::sort

(Running on Intel i7 6600 CPU & Intel HD Graphics 520)

size	**2^16**	**2^17**	**2^18**	***2^19***
std::seq	0.27031s	0.620068s	0.669628s	***1.48918s***
std::par	0.259486s	0.478032s	0.444422s	***1.83599s***
std::unseq	0.24258s	0.413909s	0.456224s	***1.01958s***
sycl_execution_policy	0.273724s	0.269804s	0.277747s	***0.399634s***

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

Eigen Linear Algebra Library

- SYCL backend in mainline
- Focused on Tensor support, providing support for machine learning/CNNs
- Equivalent coverage to CUDA
- Working on optimization for various hardware architectures (CPU, desktop and mobile GPUs)
- https://bitbucket.org/eigen/eigen/

TensorFlow

- SYCL backend support for all major CNN operations
- Complete coverage for major image recognition networks
  - GoogLeNet, Inception-v2, Inception-v3, ResNet, ….
- Ongoing work to reach 100% operator coverage and optimization for various hardware architectures (CPU, desktop and mobile GPUs)
- https://github.com/tensorflow/tensorflow

TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

SYCL Ecosystem

- Single-source heterogeneous programming using STANDARD C++
  - Use C++ templates and lambda functions for host & device code
  - Layered over OpenCL
- Fast and powerful path for bring C++ apps and libraries to OpenCL
  - C++ Kernel Fusion - better performance on complex software than hand-coding
  - Halide, Eigen, Boost.Compute, SYCLBLAS, SYCL Eigen, SYCL TensorFlow, SYCL GTX
  - Clang, triSYCL, ComputeCpp, VisionCpp, ComputeCpp SDK …
- More information at http://sycl.tech

Developer Choice

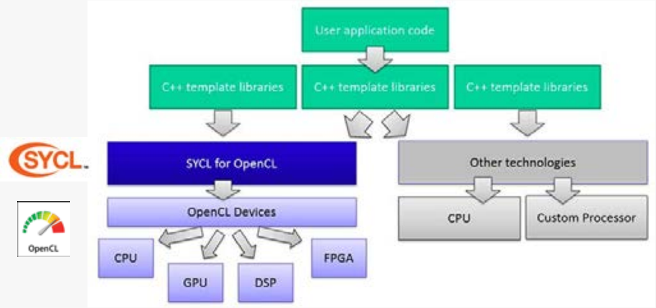
The development of the two specifications are aligned so code can be easily shared between the two approaches

C++ Kernel Language
Low Level Control
'GPGPU'-style separation of device-side kernel source code and host code

OpenCL

Single-source C++
Programmer Familiarity
Approach also taken by C++ AMP and OpenMP

SYCL™



Codeplay

Standards bodies

- HSA Foundation: Chair of software group, spec editor of runtime and debugging
- Khronos: chair & spec editor of SYCL. Contributors to OpenCL, Safety Critical, Vulkan
- ISO C++: Chair of Low Latency, Embedded WG; Editor of SG1 Concurrency TS
- EEMBC: members

Research

- Members of EU research consortiums: PEPPHER, LPGPU, LPGPU2, CARP
- Sponsorship of PhDs and EngDs for heterogeneous programming: HSA, FPGAs, ray-tracing
- Collaborations with academics
- Members of HiPEAC

Open source

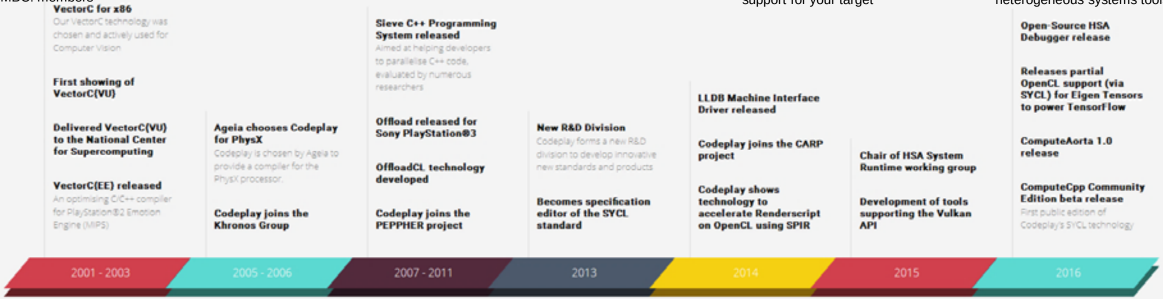
- HSA LLDB Debugger
- SPIR-V tools
- RenderScript debugger in AOSP
- LLDB for Qualcomm Hexagon
- TensorFlow for OpenCL
- C++ 17 Parallel STL for SYCL
- VisionCpp: C++ performance-portable programming model for vision

Presentations

- Building an LLVM back-end
- Creating an SPMD Vectorizer for OpenCL with LLVM
- Challenges of Mixed-Width Vector Code Gen & Scheduling in LLVM
- C++ on Accelerators: Supporting Single-Source SYCL and HSA
- LLDB Tutorial: Adding debugger support for your target

Company

- Based in Edinburgh, Scotland
- 57 staff, mostly engineering
- License and customize technologies for semiconductor companies
- ComputeAorta and ComputeCpp: implementations of OpenCL, Vulkan and SYCL
- 15+ years of experience in heterogeneous systems tools



2001 - 2003

VectorC for x86
Our VectorC technology was chosen and actively used for Computer Vision

First showing of VectorC(VU)

Delivered VectorC(VU) to the National Center for Supercomputing

VectorC(EE) released
An optimising C/C++ compiler for PlayStation®2 Emotion Engine (MIPS)

2005 - 2006

Ageia chooses Codeplay for PhysX
Codeplay is chosen by Ageia to provide a compiler for the PhysX processor.

Codeplay joins the Khronos Group

2007 - 2011

Sieve C++ Programming System released
Aimed at helping developers to parallelise C++ code, evaluated by numerous researchers

Offload released for Sony PlayStation®3

OffloadCL technology developed

Codeplay joins the PEPPHER project

2013

New R&D Division
Codeplay forms a new R&D division to develop innovative new standards and products

Becomes specification editor of the SYCL standard

2014

LLDB Machine Interface Driver released

Codeplay joins the CARP project

Codeplay shows technology to accelerate Renderscript on OpenCL using SPIR

2015

Chair of HSA System Runtime working group

Development of tools supporting the Vulkan API

2016

Open-Source HSA Debugger release

Releases partial OpenCL support (via SYCL) for Eigen Tensors to power TensorFlow

ComputeAorta 1.0 release

ComputeCpp Community Edition beta release
First public edition of Codeplay's SYCL technology

Codeplay build the software platforms that deliver massive performance

What our ComputeCpp users say about us

Benoit Steiner – Google TensorFlow engineer

“We at Google have been working closely with Luke and his Codeplay colleagues on this project for almost 12 months now. Codeplay's contribution to this effort has been tremendous, so we felt that we should let them take the lead when it comes down to communicating updates related to OpenCL. … we are planning to merge the work that has been done so far… we want to put together a comprehensive test infrastructure”

ONERA

“We work with royalty-free SYCL because it is hardware vendor agnostic, single-source C++ programming model without platform specific keywords. This will allow us to easily work with any heterogeneous processor solutions using OpenCL to develop our complex algorithms and ensure future compatibility”

Hartmut Kaiser - HPX

“My team and I are working with Codeplay's ComputeCpp for almost a year now and they have resolved every issue in a timely manner, while demonstrating that this technology can work with the most complex C++ template code. I am happy to say that the combination of Codeplay's SYCL implementation with our HPX runtime system has turned out to be a very capable basis for Building a Heterogeneous Computing Model for the C++ Standard using high-level abstractions.”

WIGNER Research Centre for Physics

It was a great pleasure this week for us, that Codeplay released the ComputeCpp project for the wider audience. We've been waiting for this moment and keeping our colleagues and students in constant rally and excitement. We'd like to build on this opportunity to increase the awareness of this technology by providing sample codes and talks to potential users. We're going to give a lecture series on modern scientific programming providing field specific examples.“

Further information

- OpenCL https://www.khronos.org/opencl/
- OpenVX https://www.khronos.org/openvx/
- HSA http://www.hsafoundation.com/
- SYCL http://sycl.tech
- OpenCV http://opencv.org/
- Halide http://halide-lang.org/
- VisionCpp https://github.com/codeplaysoftware/visioncpp

SYCL™

ComputeCpp™

Community Edition

Available now for free!

Visit:

computecpp.codeplay.com

- Open source SYCL projects:
  - ComputeCpp SDK - Collection of sample code and integration tools
  - SYCL ParallelSTL – SYCL based implementation of the parallel algorithms
  - VisionCpp – Compile-time embedded DSL for image processing
  - Eigen C++ Template Library – Compile-time library for machine learning

All of this and more at: http://sycl.tech

We're Hiring!
codeplay.com/careers/

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you for listening

@codeplaysoft

/codeplaysoft

codeplay.com