



Parallelism in Modern C++; from CPU to GPU



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

CppCon 2019

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.

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 feedback to form part of this talk.

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

But we claim all credit for errors, and stupid mistakes. **These are ours, all ours!**

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

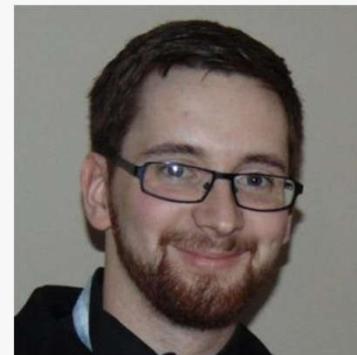
Customers



Instructors



Michael Wong



Gordon Brown

Who are we?

Michael Wong

Current Khronos SYCL chair,
WG21 SG14, SG19 chair, Directions
Group, Director of ISO C++
Head of Canada for Programming
Languages
Lead ISO C++ future Heterogeneous
VP of R & D
Past Compiler Technical Team lead for
IBM's C++ 11/14, clang update for
OpenMP, OpenMP CEO leading to
accelerators

Gordon Brown

Developer with Codeplay Software for 7
years, background in C++ programming
models for heterogeneous systems

Worked on ComputeCpp (SYCL) since it's
inception and contributor to the Khronos
SYCL standard for 7 years and to ISO C++
executors and heterogeneity for 3 years

But more importantly who are you?

- Hobbyist
- Programmer
- Manager
- Designer
- Architect
- Teacher

What do you want to get out of the class?

Maybe we can tailor future content to what you need.

What will you learn in this class?

This class will cover the following topics:

- Fundamentals of parallelism
- C++ threading library
- Synchronisation, Mutex, async, promises and packaged tasks
- Atomics and Memory model
- CPU & GPU architecture
- SYCL programming model
- Parallel Algorithm

This class is interactive

If you have a question just put your hand up

- You can stop us at any time to ask a question or clarify something we've said

We have a number of exercises throughout the class

- Opportunities to what you have learnt into practice
- These are optional, do as much or as little as you want
- We encourage you to experiment, break the code and just see what happens

Examples

There will be a number examples used during lectures

The examples can be found here:

<https://github.com/AerialMantis/cppcon-parallelism-class>

For each example there will be a source file in the **examples** directory that you can build and run

We encourage you to try them out and experiment with them

Exercises

There will also be a number of SYCL exercises throughout the day

The exercises can be found here:

<https://github.com/AerialMantis/cppcon-parallelism-class>

For each exercise there will be a source file in the **source** directory and a solution in the **solutions** directory

Setting up ComputeCpp

The exercises will require you to have ComputeCpp SYCL set up on your laptop

- 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
- But don't worry this will be the first exercise

All of the instructions for this are in the Github README

Using ComputeCpp

ComputeCpp has support for:

- Intel GPU
- AMD GPU (if SPIR is supported)
- Nvidia GPU (experimental support on Linux)

If you don't have a supported GPU you can also use:

- Intel CPU
- Host device (emulated device running in native C++)

Docker Image

If you are unable to install the OpenCL drivers for your GPU we provide a Ubuntu 16.04 docker image as an alternative

Dockerhub: [Aerialmantis/computecpp_ubuntu1604](https://hub.docker.com/r/aerialmantis/computecpp_ubuntu1604)

Wifi

CppCon
Password is: stepanov

Schedule

Day 1	Day 2
Welcome	
Chapter 1: Importance of Parallelism Chapter 2: Performance Portability	Chapter 8: Data Parallelism Chapter 9: Launching a Kernel SYCL Exercise 2: Hello World
Break	
Chapter 3: C++ Multi-threading Chapter 4a: C++ Synchronization,, Futures, Promises and Packaged Tasks Chapter 4b: C++ Atomics & Memory Model	Chapter 10: Managing Data SYCL Exercise 3: Vector add Chapter 11: Fundamentals of Parallelism
Lunch	
Chapter 5: Intro to SYCL SYCL Exercise 0: Setting up ComputeCpp	Chapter 12: GPU Optimization (part 1) SYCL Exercise 4: Image grayscale Chapter 13: Optimization (part 2)
Break	
Chapter 6: Understanding CPU & GPU Architecture Chapter 7: Configuring a Queue SYCL Exercise 1: Configuring a Queue	SYCL Exercise 5: Matrix transpose Chapter 14: Parallel Algorithms



Chapter 8: Data Parallelism

Michael Wong

CppCon 2019 – Sep 2019

- Learning objectives:

- Learn the differences between task and data parallelism
- Learn about Flynn's taxonomy
- Learn the differences between multi-core CPU and many-core CPU
- Learn about the current state of SIMD programming
- Learn about auto-vectorization

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.



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

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

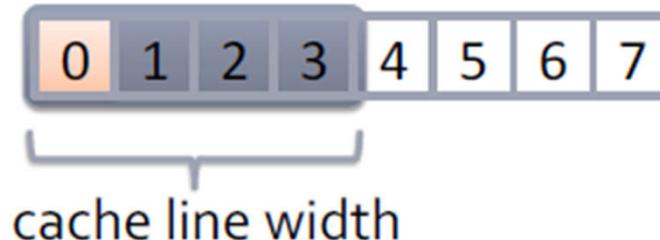
- Many core gpu is a device for turning a compute bound problem into a memory bound problem



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

Memory is SIMD too

- Virtually all processors have SIMD memory subsystems



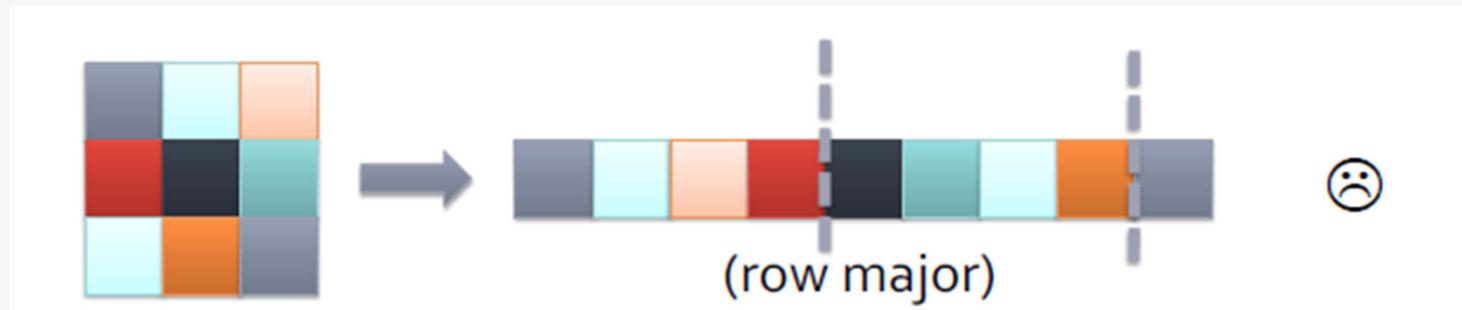
- This has 2 effects
 - Sparse access wastes bandwidth



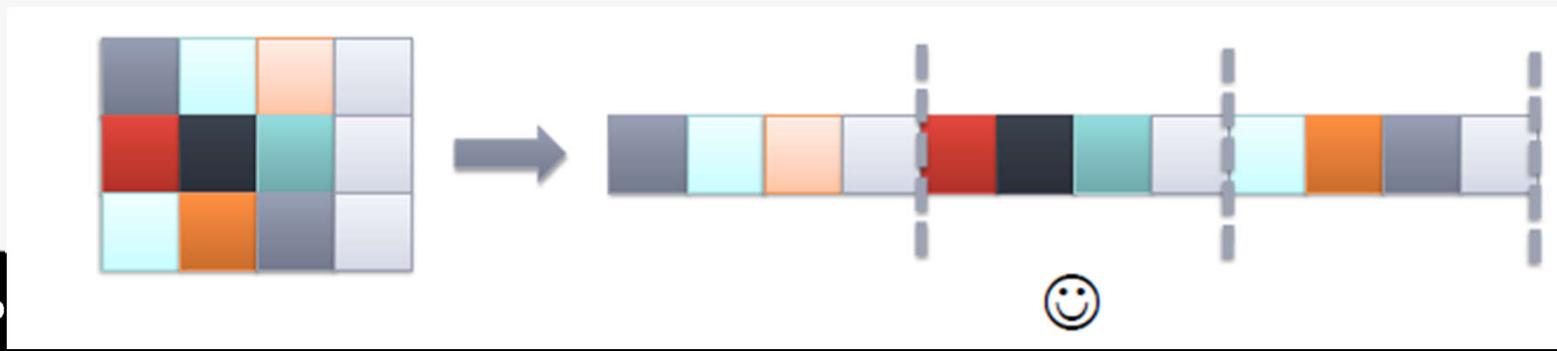
- Unaligned access wastes bandwidth



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

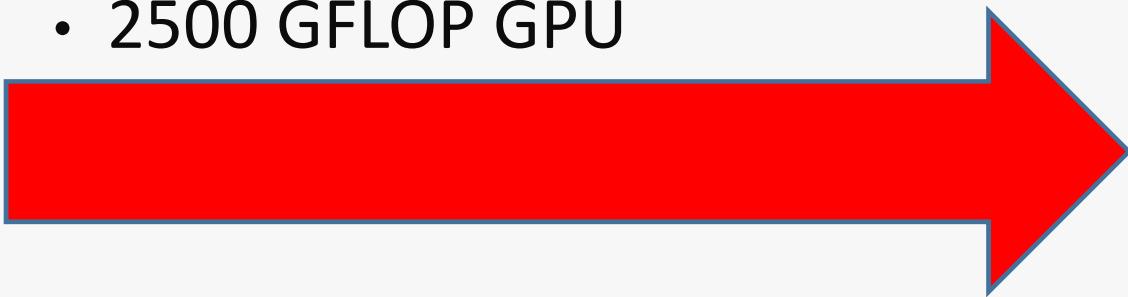
- 1998, when C++ 98 was released
 - Intel Pentium II: 0.45 GFLOPS
 - No SIMD: SSE came in Pentium III
 - No GPUs: GPU came out a year later
- 2011: when C++11 was released
 - Intel Core-i7: 80 GFLOPS
 - AVX: 8 DP flops/HZ*4 cores *4.4 GHz= 140 GFlops
 - GTX 670: 2500 GFLOPS
- Computers have gotten so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

In 1998, a typical machine had the following flops

- .45 GFLOP, 1 core
- Single threaded C++98/C99/Fortran dominated this picture

In 2011, a typical machine had the following flops

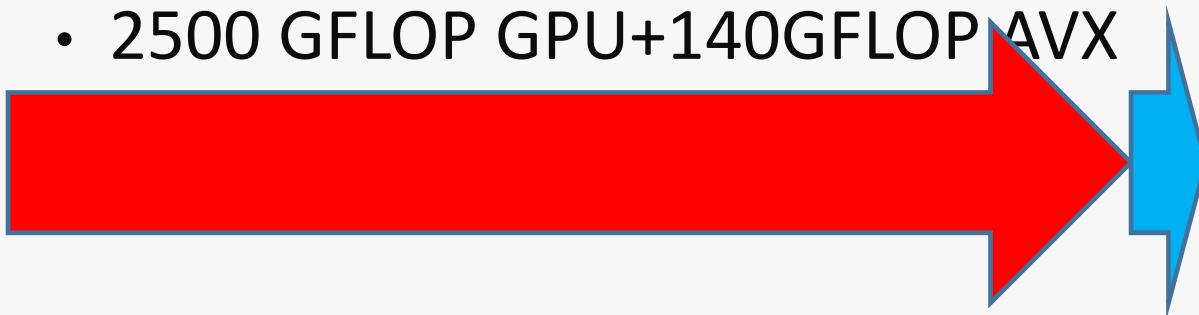
- 2500 GFLOP GPU



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

In 2011, a typical machine had the following flops

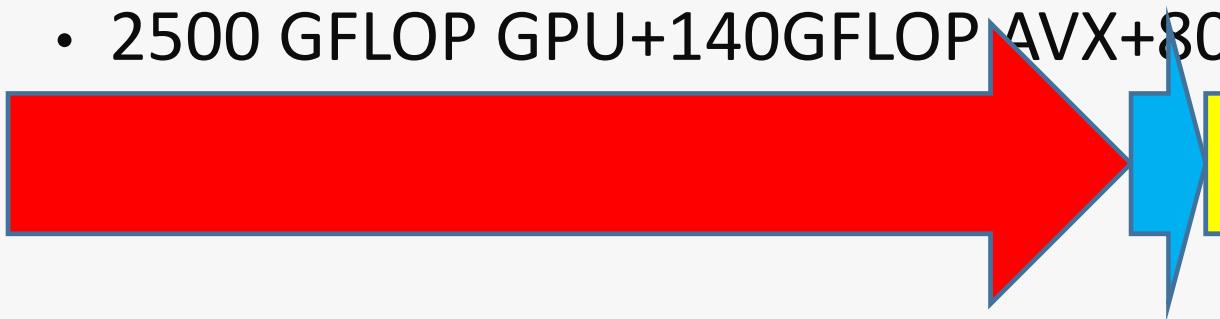
- 2500 GFLOP GPU+140GFLOP AVX



- 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

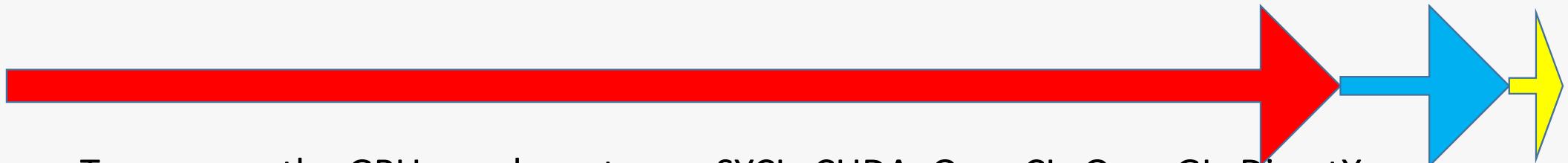
- 2500 GFLOP GPU+140GFLOP AVX+80GFLOP 4 cores



- 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

Parallel/concurrency aiming for C++20

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous/Distributed
today's abstractions	C++11: thread,lambda function, TLS, async C++ 20: Jthreads +interrupt_token, coroutines	C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control_falsesharing C++ 20: is_ready(), make_ready_future() , simd<T>, Vec execution policy, Algorithm unsequenced policy, span	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++20: atomic_ref, Latches and barriers, atomic<shared_ptr> Atomics & padding bits Simplified atomic init Atomic C/C++ compatibility Semaphores and waiting Fixed gaps in memory model , Improved atomic flags, Repair memory model	C++17: , progress guarantees, TOE, execution policies C++20: atomic_ref, simd<T>

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, SSE, 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 than figuring out what the code is doing
- Support required vendor-specific extensions

C++ 20 Parallelism

- 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

But it still only covers CPUs. There are a lot more SIMD lanes in GPUs.



Questions?



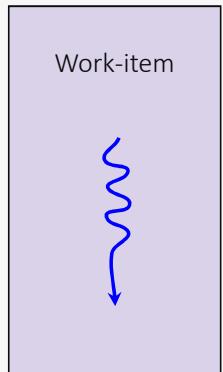
Chapter 9: Launching a Kernel

Gordon Brown

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about the SPMD model of describing parallelism
 - Learn about the SYCL execution model
 - Learn how to declare a SYCL kernel function and invoke over an nd-range
 - Learn about the rules and restrictions on SYCL kernel functions
 - Learn how to define a SYCL kernel function as a lambda or a function object
 - Learn how to pre-compile a SYCL kernel function

SYCL execution model

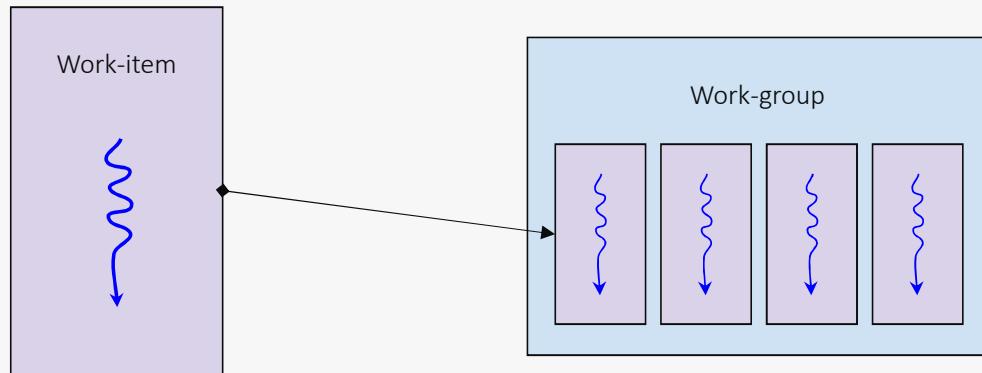


SYCL kernel functions are executed by **work-items**

You can think of a work-item as like a thread of execution

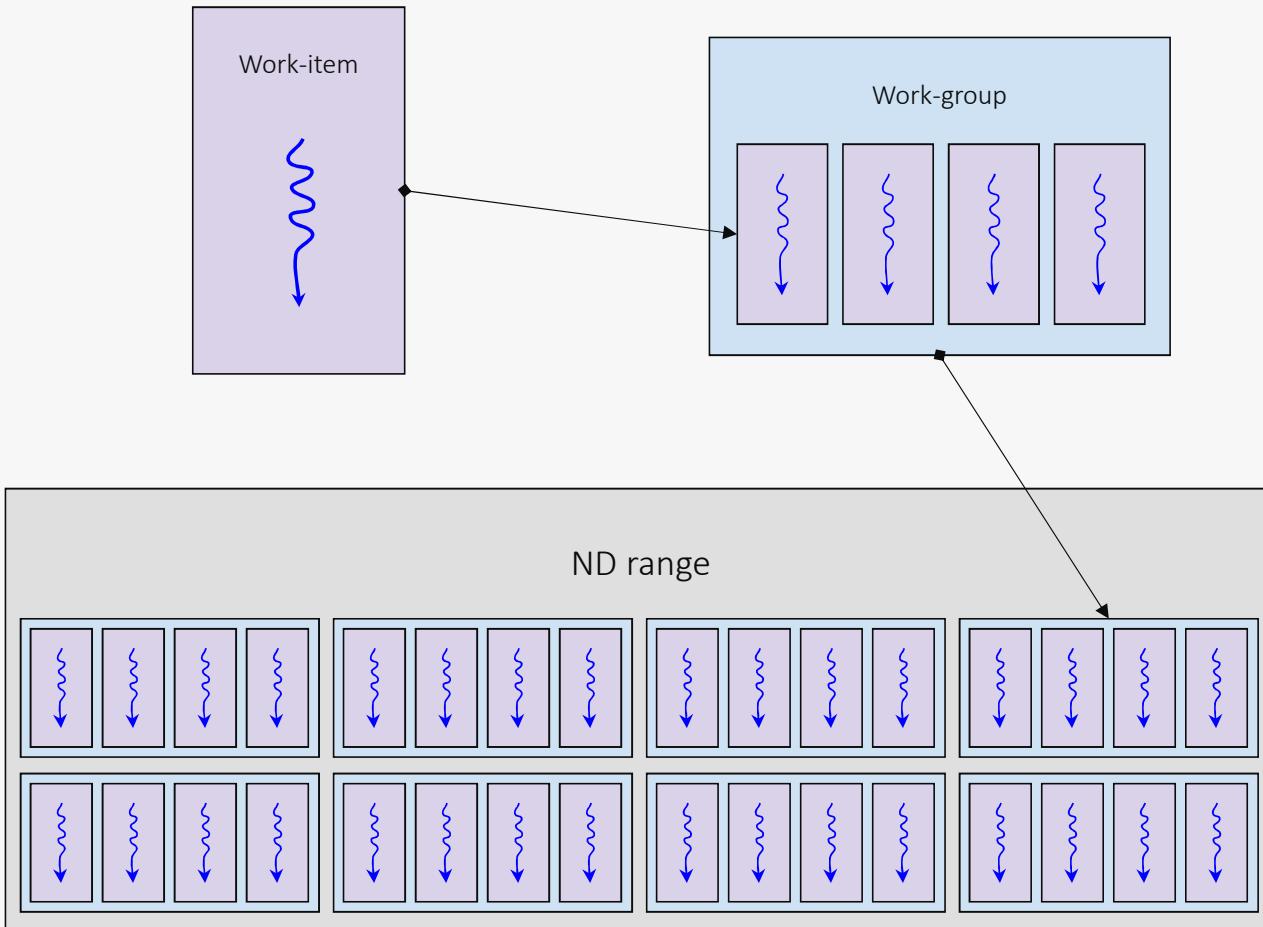
Though the way work-items are executed in SYCL is different to how threads are executed in C++

Underneath these work-items can be run on OS threads, GPU processing elements, etc, depending on the device



Work-items are collected together into **work-groups**

The size of work-groups is generally relative to what is optimal on the device being targeted

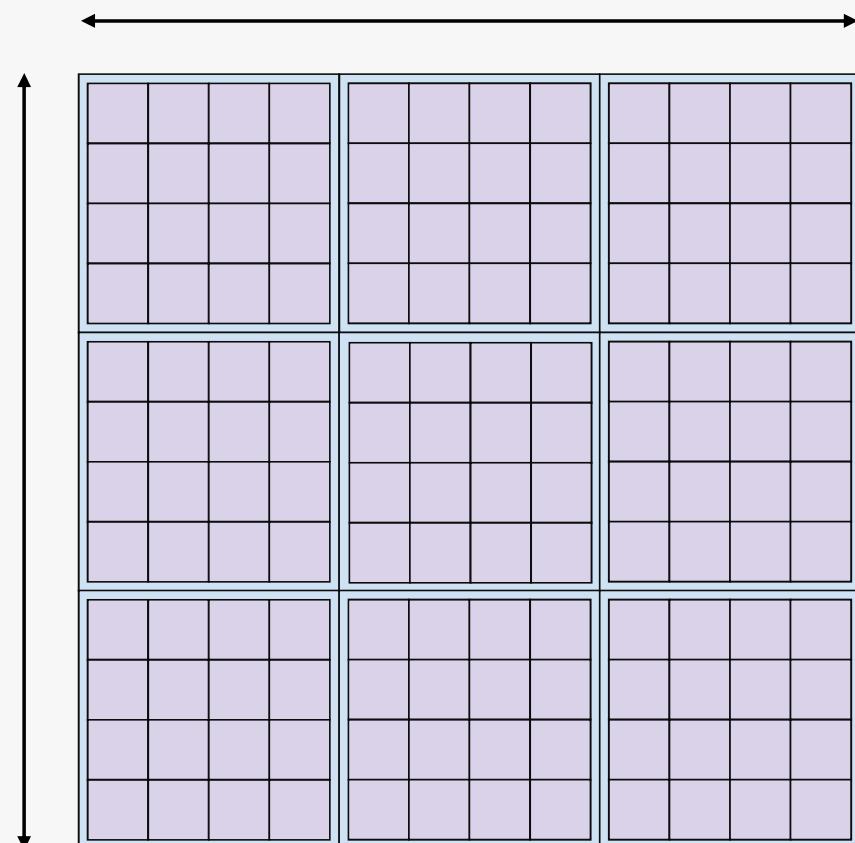


SYCL kernel functions are invoked within an **nd-range**

An nd-range has a number of work-groups and subsequently a number of work-items

Work-groups always have the same number of work-items

nd-range $\{\{12, 12\}, \{4, 4\}\}$



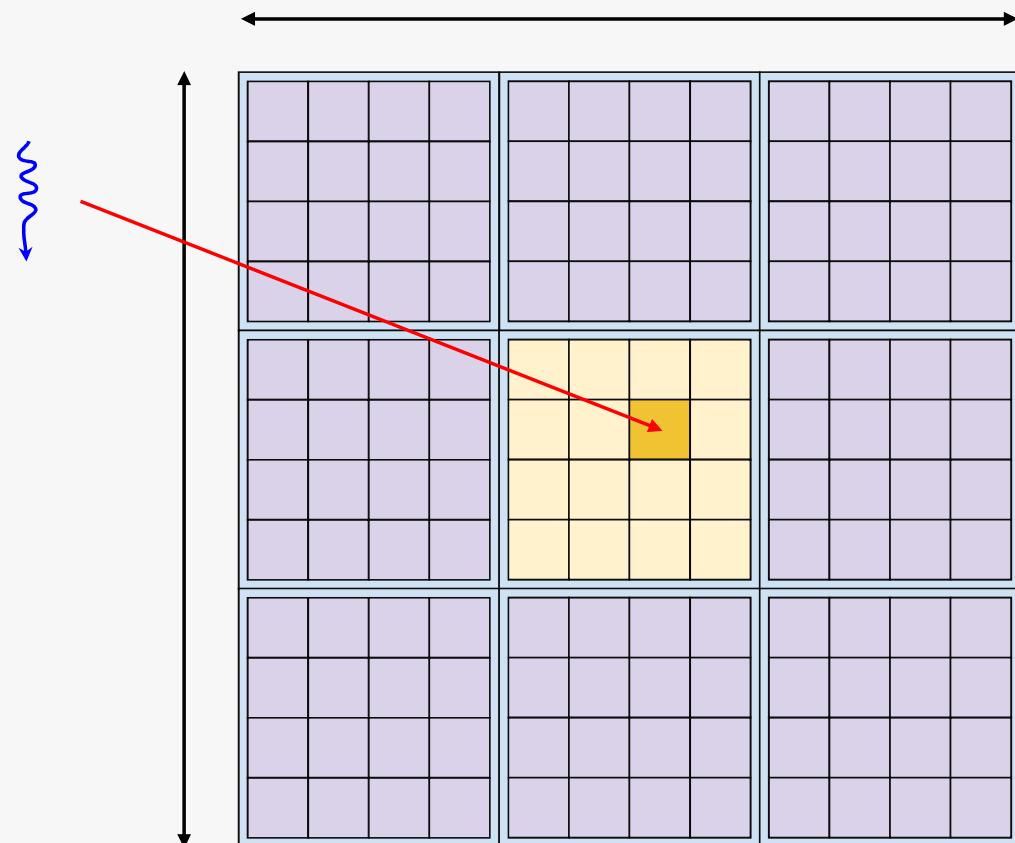
The nd-range describes an **iteration space**; how the work-items and work-groups are composed

An nd-range can be 1, 2 or 3 dimensions

An nd-range has two components

- The **global range** describes the total number of work-items in each dimension
- The **local range** describes the number of work-items in a work-group in dimension

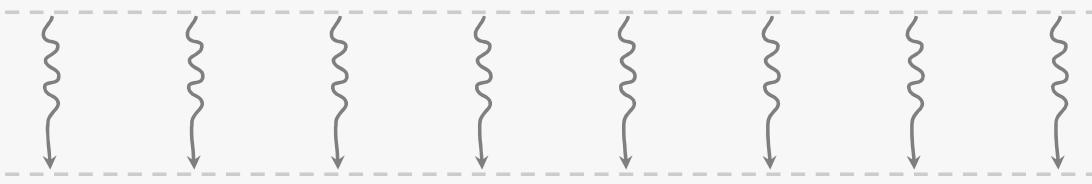
nd-range $\{\{12, 12\}, \{4, 4\}\}$



Each invocation in the iteration space of an nd-range is a work-item

Each invocation knows which work-item it is and can query certain information about its position in the iteration space

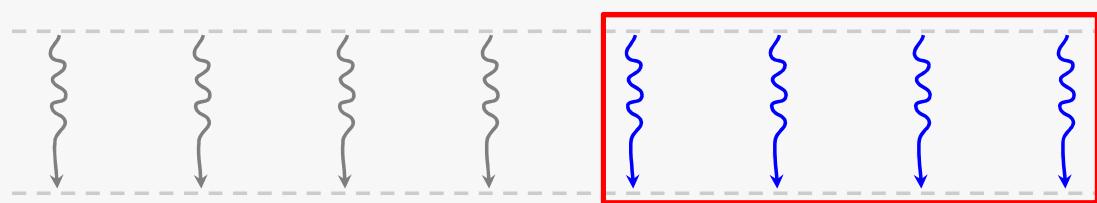
- **Global range:** $\{12, 12\}$
- **Global id:** $\{6, 5\}$
- **Group range:** $\{3, 3\}$
- **Group id:** $\{1, 1\}$
- **Local range:** $\{4, 4\}$
- **Local id:** $\{2, 1\}$



Typically an `nd-range` invocation SYCL will execute the SYCL kernel function on a very large number of work-items, often in the thousands

Multiple work-items will generally execute concurrently

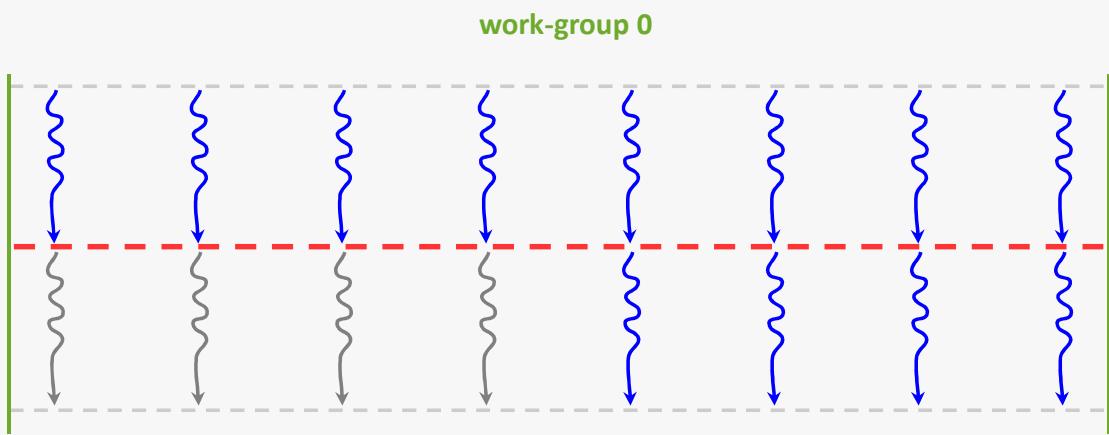
On vector hardware this done in **lock-step**, which means the same hardware instructions



The number of work-items that will execute concurrently can vary from one device to another

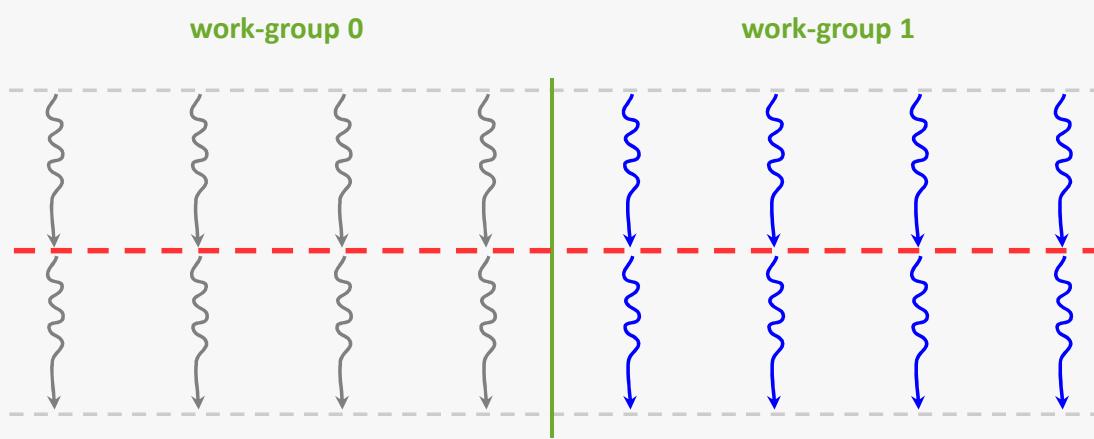
Work-items will be batched along with other work-items in the same work-group

The order work-items and work-groups are executed in is implementation defined



Work-items in a work-group can be synchronised using a work-group barrier

This means all work-items within a work-group must reach the barrier before any can continue on



SYCL does not support synchronising across all work-items in the nd-range

The only way to do this is to split the computation into separate SYCL kernel functions

SYCL is an SPMD programming model

Sequential CPU code

```
void calc(int *in, int *out) {  
    // all iterations are run in the same  
    // thread in a loop  
    for (int i = 0; i < 1024; i++) {  
        out[i] = in[i] * in[i];  
    }  
}
```

```
// calc is invoked just once and all  
// iterations are performed inline  
calc(in, out);
```

Parallel SPMD code

```
void calc(int *in, int *out, int id) {  
    // function is described in terms of  
    // a single iteration  
    out[id] = in[id] * in[id];  
}
```

```
// parallel_for invokes calc multiple  
// times in parallel  
parallel_for(calc, in, out, 1024);
```

Invoking SYCL kernel functions

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

class add;

int main(int argc, char *argv[]) {
    queue gpuQueue(gpu_selector{});

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

        cgh.parallel_for<add>(range<1>(1024),
            [=] (id<1> i) {
                // kernel code
            });
    });
    gpuQueue.wait();
}
```

SYCL kernel functions are defined and invoked using one of the kernel function invoke APIs provided by the **handler** class

These add a SYCL kernel function command to the command group

There can only be one SYCL kernel function command in a command group

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

class add;

int main(int argc, char *argv[]) {
    queue gpuQueue(gpu_selector{});
    gpuQueue.submit([&] (handler &cgh) {

        cgh.parallel_for<add>(range<1>(1024),
            [=] (id<1> i) {
                // kernel code
            });
    });
    gpuQueue.wait();
}
```

The lambda or function object represents the SYCL device function

This is the part that is compiled for the device

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

class add;

int main(int argc, char *argv[]) {
    queue gpuQueue(gpu_selector{});

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

        cgh.parallel_for<add>(range<1>(1024),
            [=] (id<1> i) {
                // kernel code
            });
    });
    gpuQueue.wait();
}
```

There are a number of different APIs for expressing different forms of parallelism, complexity and functionality

Each takes some representation of the nd-range and expects a certain parameter type that describes the current index into the iteration space

These types have a number of member functions for retrieving different index and range information about the current iteration

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

class add;

int main(int argc, char *argv[]) {
    queue gpuQueue(gpu_selector{});

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

        cgh.parallel_for<add>(range<1>(dA.size()) ,
            [=] (id<1> i) {
                // kernel code
            });
    });
    gpuQueue.wait();
}
```

There are a number of ways to wait on a SYCL kernel function invocation to complete

Here we trigger a wait on the queue which will wait for all work submitted to it to complete

You can also trigger a wait on an event which is returned from a command group submission

Or you can rely on a buffer's synchronisation on-destruction to ensure the invocation is complete

```
cgh.single_task<T>([=] () {  
    // SYCL kernel function is executed  
    // once on a single work-item  
});
```

```
cgh.parallel_for<T>(range<2>(64, 64),  
                      [=](id<2> idx) {  
    // SYCL kernel function is executed  
    // on an nd-range of work-items  
});
```

```
cgh.parallel_for_work_group(range<2>(64, 64),  
                           [=](group<2> gp) {  
    // SYCL kernel function is executed once per  
    // work-group  
  
    parallel_for_work_item(gp, [=](h_item<2> it) {  
        // SYCL kernel function is executed once per  
        // work-item  
  
    });  
});
```

```
cgh.parallel_for<kernel>(range<1>(1024),  
    [=] (id<1> idx) {  
        // kernel code  
    });
```

Overload taking a **range** object specifies the global range, runtime decides local range

An **id** parameter represents the index within the global range

```
cgh.parallel_for<kernel>(range<1>(1024),  
    [=] (item<1> item) {  
        // kernel code  
    });
```

Overload taking a **range** object specifies the global range, runtime decides local range

An **item** parameter represents the global range and the index within the global range

```
cgh.parallel_for<kernel>(nd_range<1>(range<1>(1024),  
    range<1>(32)), [=] (nd_item<1> ndItem) {  
        // kernel code  
    });
```

Overload taking an **nd_range** object specifies the global and local range

An **nd_item** parameter represents the global and local range and index

```
cgh.parallel_for<kernel>(range<1>(1024), id<1>(512),  
    [=](id<1> idx) {  
        // kernel code  
    });
```

```
cgh.parallel_for<kernel>(range<1>(1024), id<1>(512),  
    [=](item<1> item) {  
        // kernel code  
    });
```

```
cgh.parallel_for<kernel>(nd_range<1>(range<1>(1024),  
    range<1>(32), id<1>(512)), [=](nd_item<1> ndItem) {  
        // kernel code  
    });
```

All overloads of `parallel_for` also allow you to optionally specify an offset

The offset, if used will increment each index into the global index by the specified value

E.g. with a range of 1024 and an offset of 512 the indexes would become [512, 1536)

Rules and restrictions

- Rules for SYCL kernel functions:
 - SYCL kernel functions must be defined using a C++ lambda or function object, they cannot be a function pointer or std::function
 - SYCL kernel functions must always capture or store members by-value
 - SYCL kernel function captures or stored members must be standard-layout
- Rules for naming SYCL kernel functions:
 - SYCL kernel functions declared with a lambda must be named using a forward declarable C++ type, declared in global scope
 - SYCL kernel function names follow C++ ODR rules, which means you cannot have two kernels with the same name

- Language restrictions for SYCL kernel functions:
 - No dynamic allocation
 - No dynamic polymorphism
 - No recursion
 - No function pointers
 - No exception handling
 - No RTTI

SYCL kernels as function objects

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

All the examples of SYCL kernel functions up until now have been defined using lambdas

```
struct add {
    using read_accessor_t =
        accessor<float, 1,
        access::mode::read,
        access::target::global_buffer>;
    using write_accessor_t =
        accessor<float, 1
        access::mode::write,
        access::target::global_buffer>;

    read_accessor_t inA_, inB_;
    write_accessor_t out_;

    void operator()(id<1> i){
        out_[i] = inA_[i] + inB_[i];
    }
};
```

As well as defining SYCL kernels using lambdas you can also define a SYCL kernel using a regular C++ function object

Where the accessors are stored as members and the function call operator takes the appropriate parameter

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for(range<1>(dA.size()), add{inA, inB, out});

        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

```

struct add {
    using read_accessor_t =
        accessor<float, 1,
        access::mode::read,
        access::target::global_buffer>;
    using write_accessor_t =
        accessor<float, 1
        access::mode::write,
        access::target::global_buffer>;

    read_accessor_t inA_, inB_;
    write_accessor_t out_;

    void operator()(id<1> i) {
        out[i] = inA[i] + inB[i];
    }
};

```

To use a C++ function object you simply construct an instance of the type initialising the accessors and pass it to parallel_for

Notice you no longer need to name the SYCL kernel

Pre-compiling SYCL kernel functions

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });

        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

When you invoke a SYCL kernel function the runtime has to just-in-time compile the kernel for the device it's being executed on

This means the first time a kernel function is invoked will take longer than subsequent invocations

However you can avoid this by pre-compiling the kernel

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        program addProgram(gpuQueue.get_context());

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

To pre-compile a kernel you need to create a program that is associated with the context you are executing the kernel on

A program is represented by the **program** class

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        program addProgram(gpuQueue.get_context());
        addProgram.compile_with_kernel_type<add>();

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

The program can be then be compiled from a SYCL kernel function name by calling **compile_with_kernel_type** and specifying the kernel name as a template parameter

If the kernel function was defined as a function object then the name would be the function object type

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        program addProgram(gpuQueue.get_context());
        addProgram.compile_with_kernel_type<add>();
        auto addKernel = addProgram.get_kernel<add>();

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

Once the program is compiled you can retrieve the kernel

A kernel is represented by the **kernel** class

This is retrieved from a program object by calling the **get_kernel** member function and specifying the kernel name as a template parameter

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue(gpu_selector{}, async_handler{});

        program addProgram(gpuQueue.get_context());
        addProgram.compile_with_kernel_type<add>();
        auto addKernel = addProgram.get_kernel<add>();

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(addKernel, range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

Finally the kernel object can then be passed to parallel_for in order to specify that the invocation should use the pre-compiled kernel

Key takeaways

SYCL executes a number of work-items

Work-items are split up into a number of equally sized work-groups

SYCL provides a number of APIs for defining kernel functions depending on the level of functionality you require

You can manually pre-build kernels using the **program** class



Questions?

Exercise 2:

Hello world

- How to create and submit a command group
- How to define a SYCL kernel function
- How to stream output from a SYCL kernel function



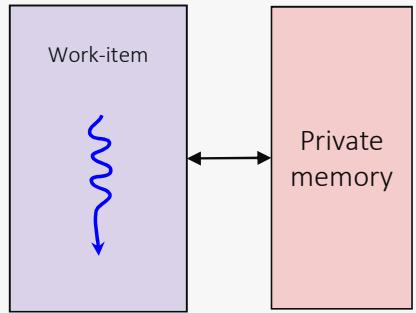
Chapter 10: Managing Data

Gordon Brown

CppCon 2019 – Sep 2019

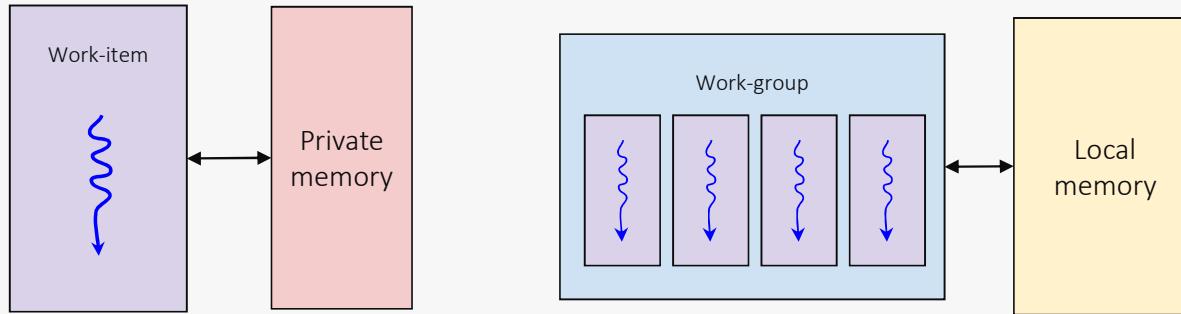
- Learning objectives:
 - Learn about the SYCL memory model
 - Learn how to manage and access data using buffers and accessors
 - Learn how to access memory in different address spaces
 - Learn about different ways to access the data represented by accessors
 - Learn about how the SYCL runtime orders execution using data dependencies
 - Learn about how SYCL synchronises data

SYCL memory model



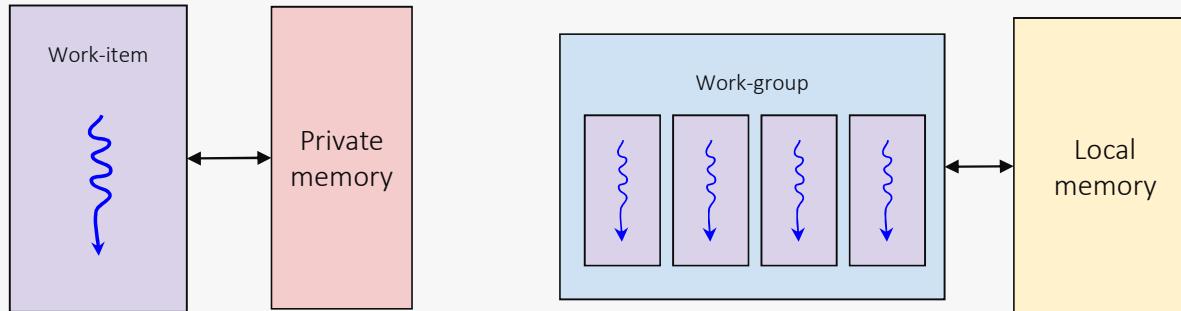
Each work-item can access a
dedicate region of **private
memory**

A work-item cannot access the
private memory of another
work-item

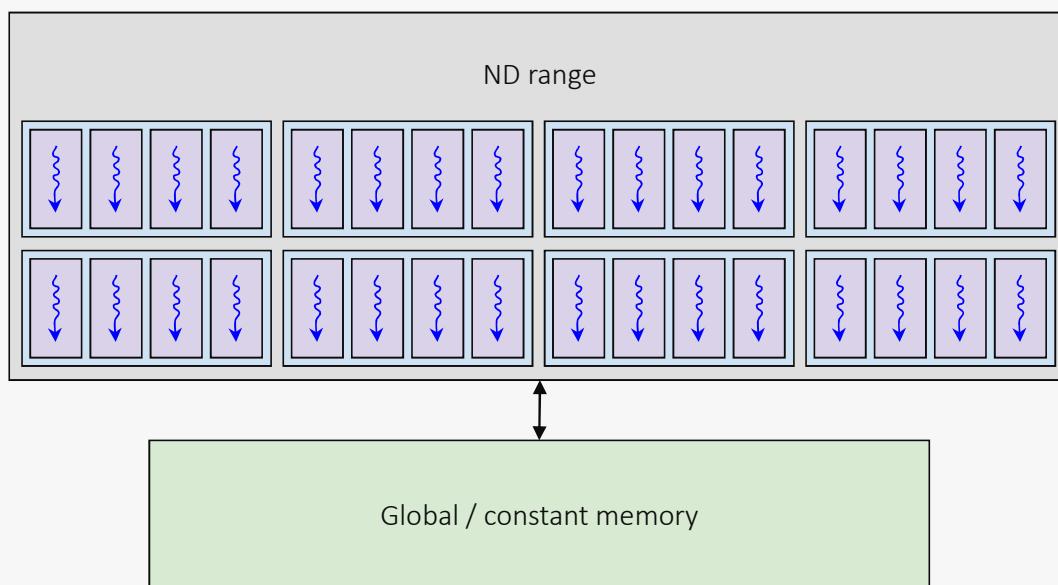


Each work-item can access a dedicated region of **local memory** accessible to all work-items in a work-group

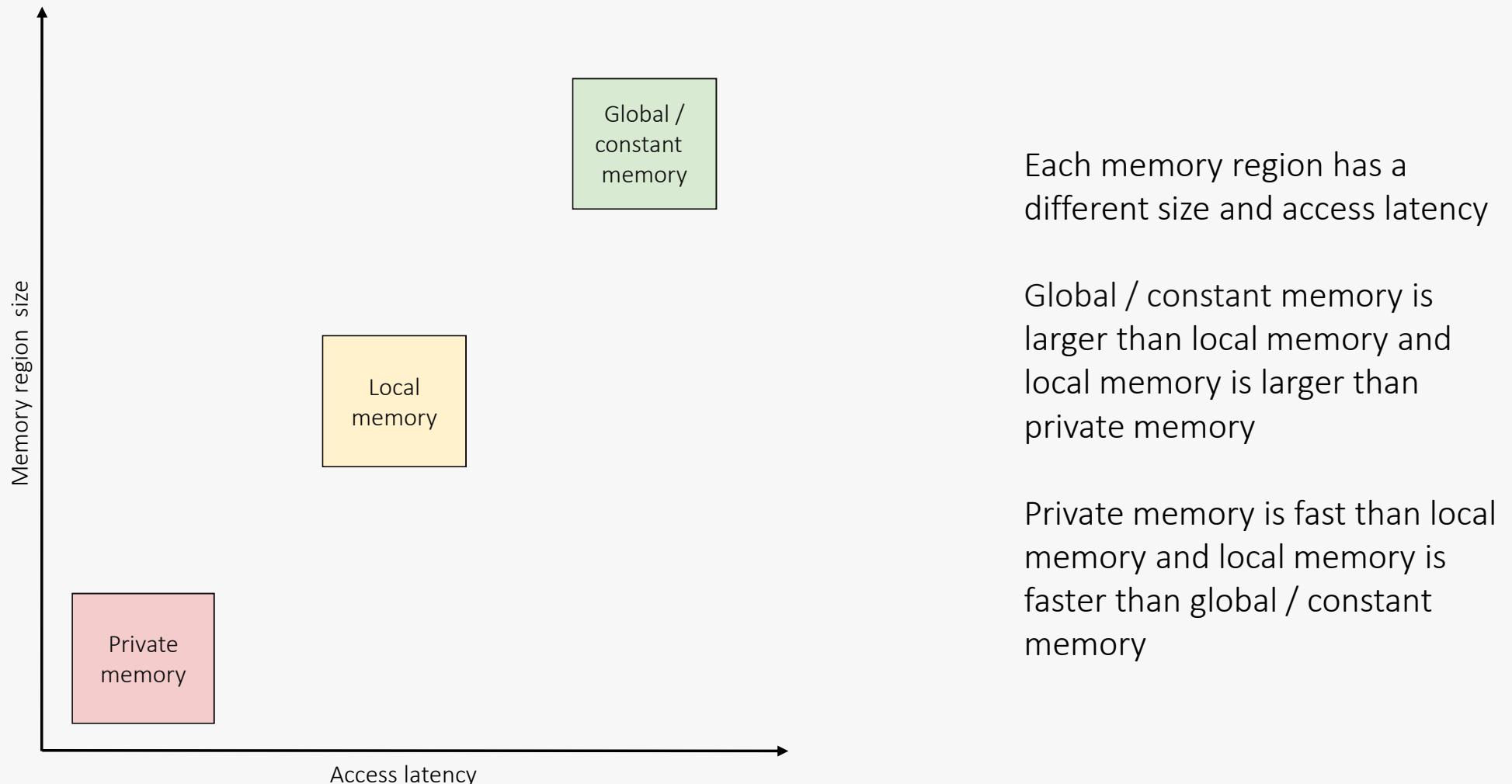
A work-item cannot access the local memory of another work-group



Each work-item can access a single region of **global memory** that's accessible to all work-items in an nd-range



Each work-item can also access a region of global memory reserved as **constant memory**, which is read-only



Buffers & accessors

- SYCL separates the storage and access of data
 - A SYCL buffer manages data across the host and any number of devices
 - A SYCL accessor requests access to data on the host or on a device for a specific SYCL kernel function
- Accessors are also used to access data within SYCL kernel function
 - This means they are declared in the host code but captured by and then accessed within a SYCL kernel function

```
buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

    auto inA = bufA.get_access<access::mode::read>(cgh);
    auto inB = bufB.get_access<access::mode::read>(cgh);
    auto out = bufO.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ out[i] = inA[i] + inB[i]; });
});
```

In order to achieve this the host compiler and the SYCL device compiler interpret accessors differently

The host compiler interprets accessors as host objects which instruct the SYCL runtime of data dependencies for a SYCL kernel function

The SYCL device compiler interprets accessors as a wrapper on the argument to the SYCL kernel function (a pointer to device memory) that is used to access data on the device



A SYCL buffer can be constructed with a pointer to host memory

For the lifetime of the buffer this memory is owned by the SYCL runtime

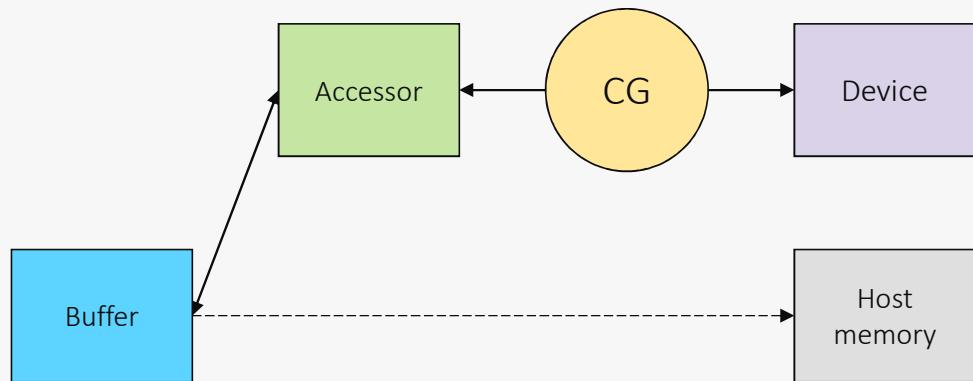
When a buffer object is constructed it will not allocate or copy to device memory at first

This will only happen once the SYCL runtime knows the data needs to be accessed and where it needs to be accessed



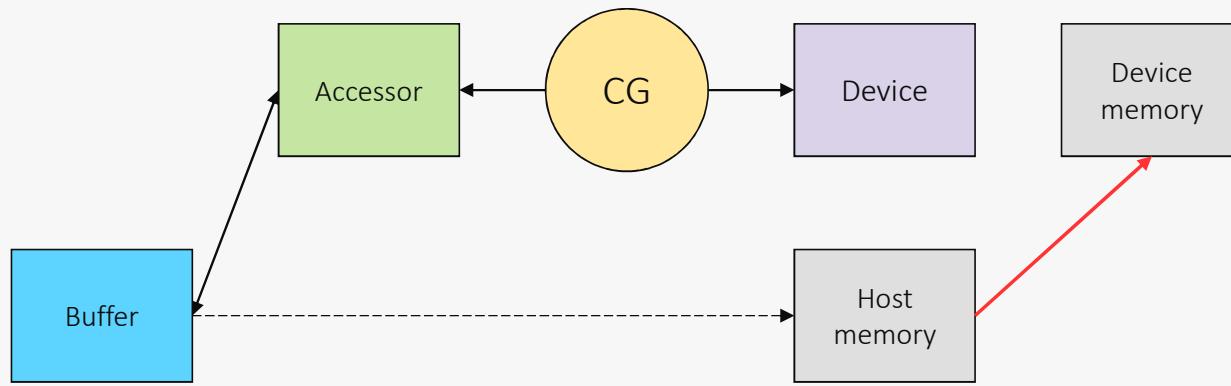
Constructing an accessor specifies a request to access the data managed by the buffer

There are a range of different types of accessor which provide different ways to access data



When an accessor is constructed it is associated with a command group via the handler object

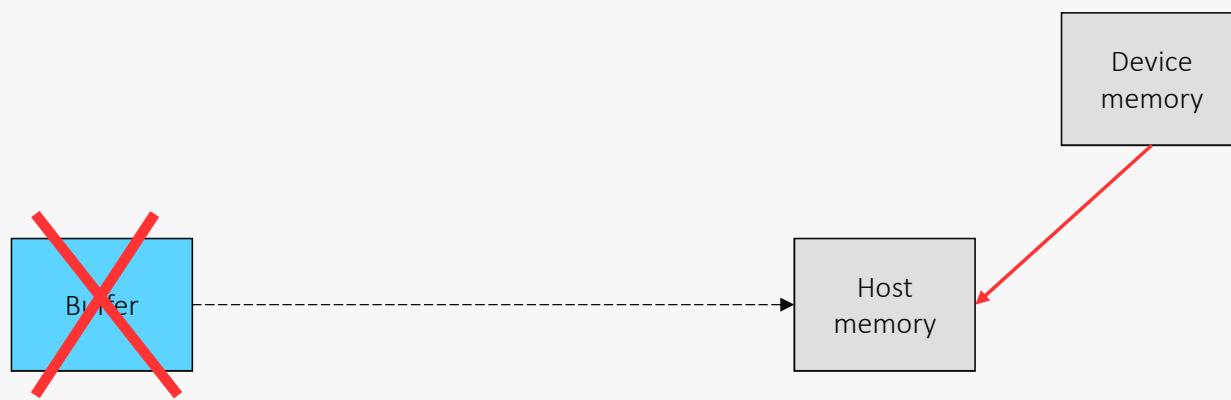
This connects the buffer that is being accessed, the way in which it's being accessed and the device that the command group is being submitted to



Once the SYCL scheduler selects the command group to be executed it must first satisfy its data dependencies

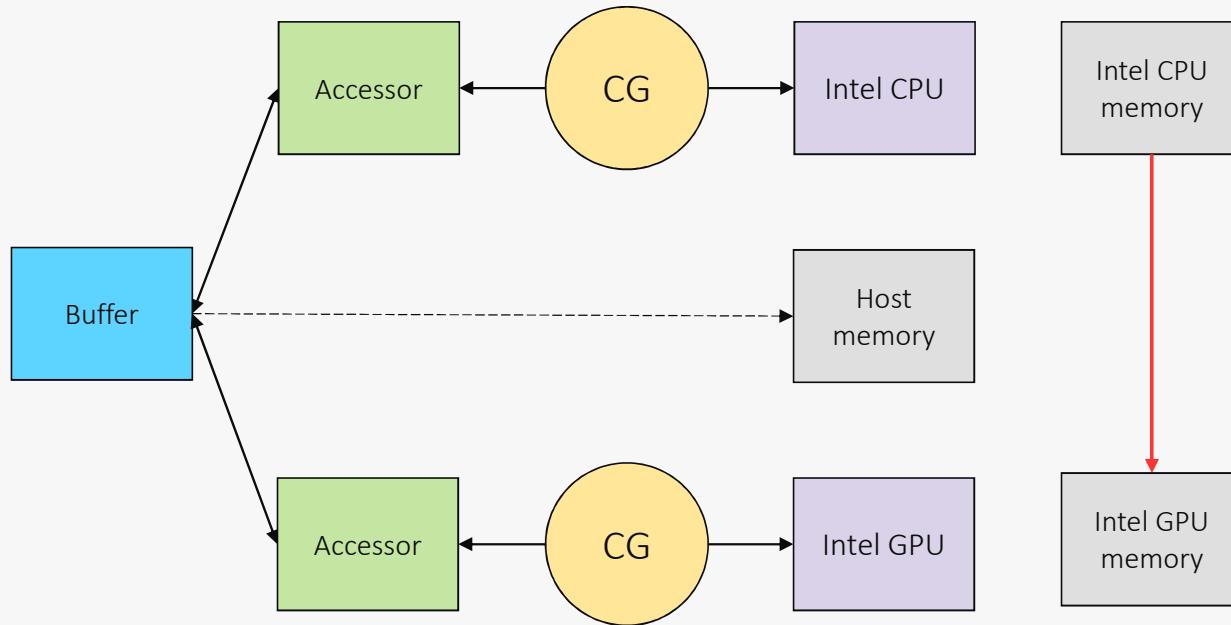
This means allocating and copying data to the device the data is being accessed on if necessary

If the most recent copy of the data is already on the device then the runtime will not copy again



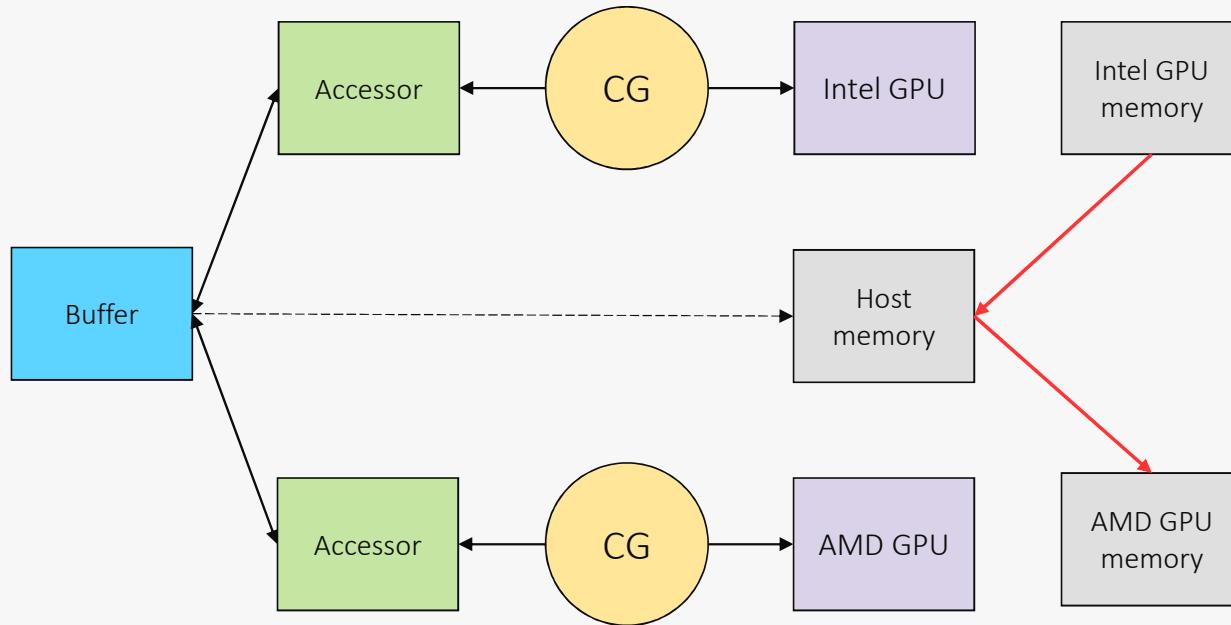
Data will remain in device memory after kernels finish executing until another command group requests access in a different device or on the host

When the buffer object is destroyed it will wait for any outstanding work that is accessing the data to complete and then copy back to the original host memory



If a buffer is accessed on one device when the latest copy of the data is on another device, the data will be copied

If the two devices are of the same context the data can be copied directly

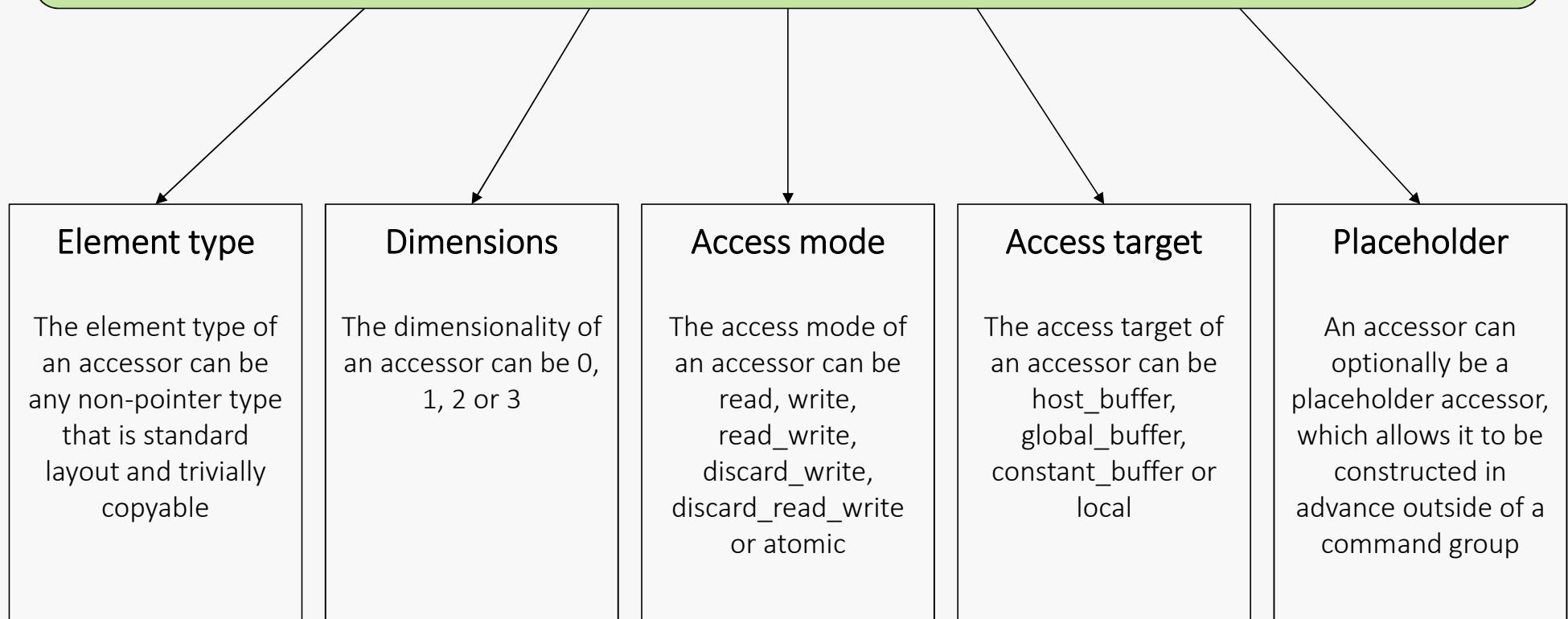


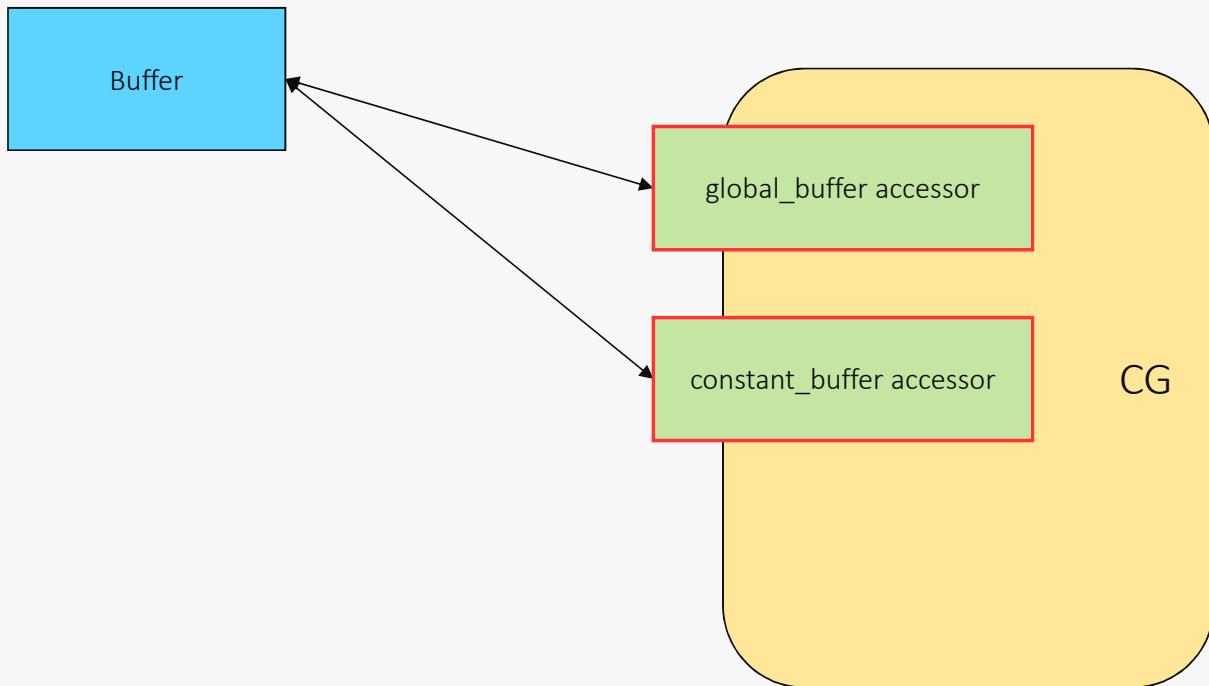
If the devices are of different contexts the data must be copied via host memory

It's important to consider this as it could incur further overhead when copying between devices

Different kinds of accessors

```
accessor<elementT, dimensions, access::mode, access::target,  
access::placeholder>
```



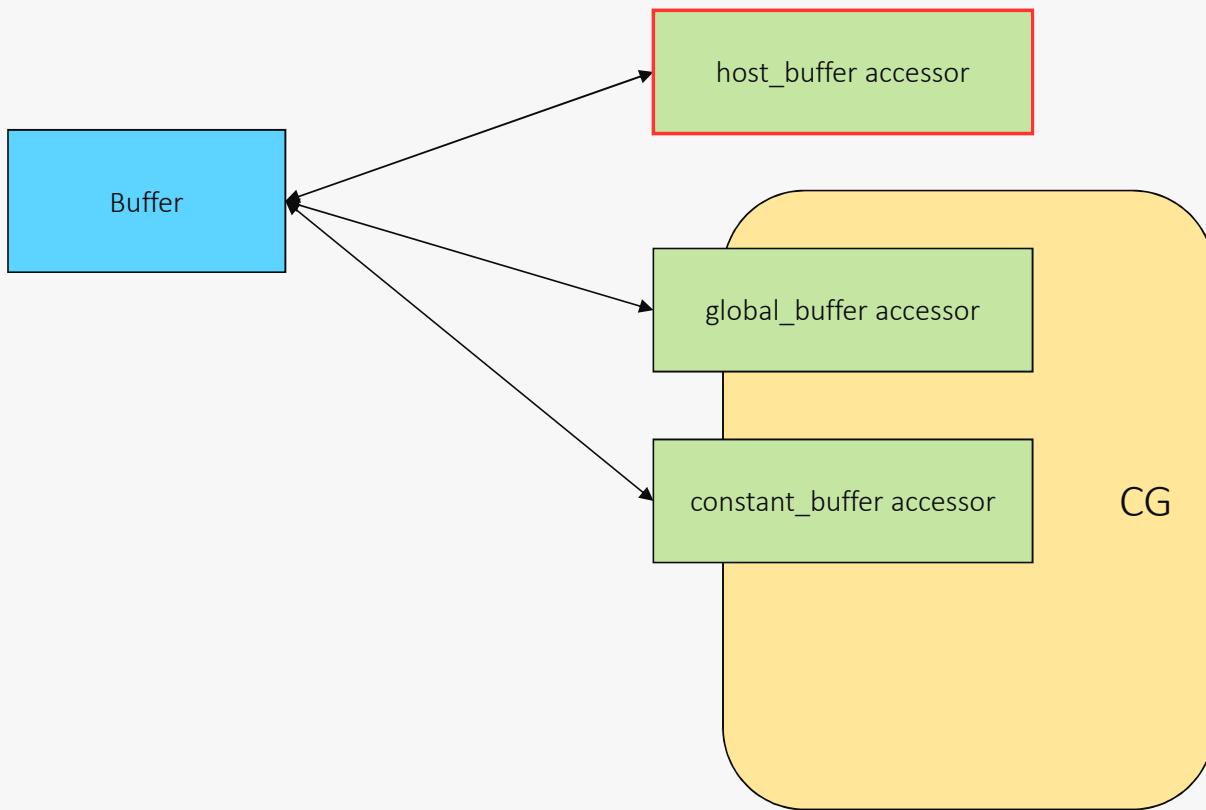


Accessors using the `access::target` **`global_buffer`** and **`constant_buffer`** will allocate memory on the device in the **`global`** and **`constant`** address space respectively

These access targets must be constructed using a buffer inside a command group

A useful shortcut to construct an accessor with the `access::target` **`global_buffer`** is to use the **`get_access`** member function of the buffer

Element type must be the same

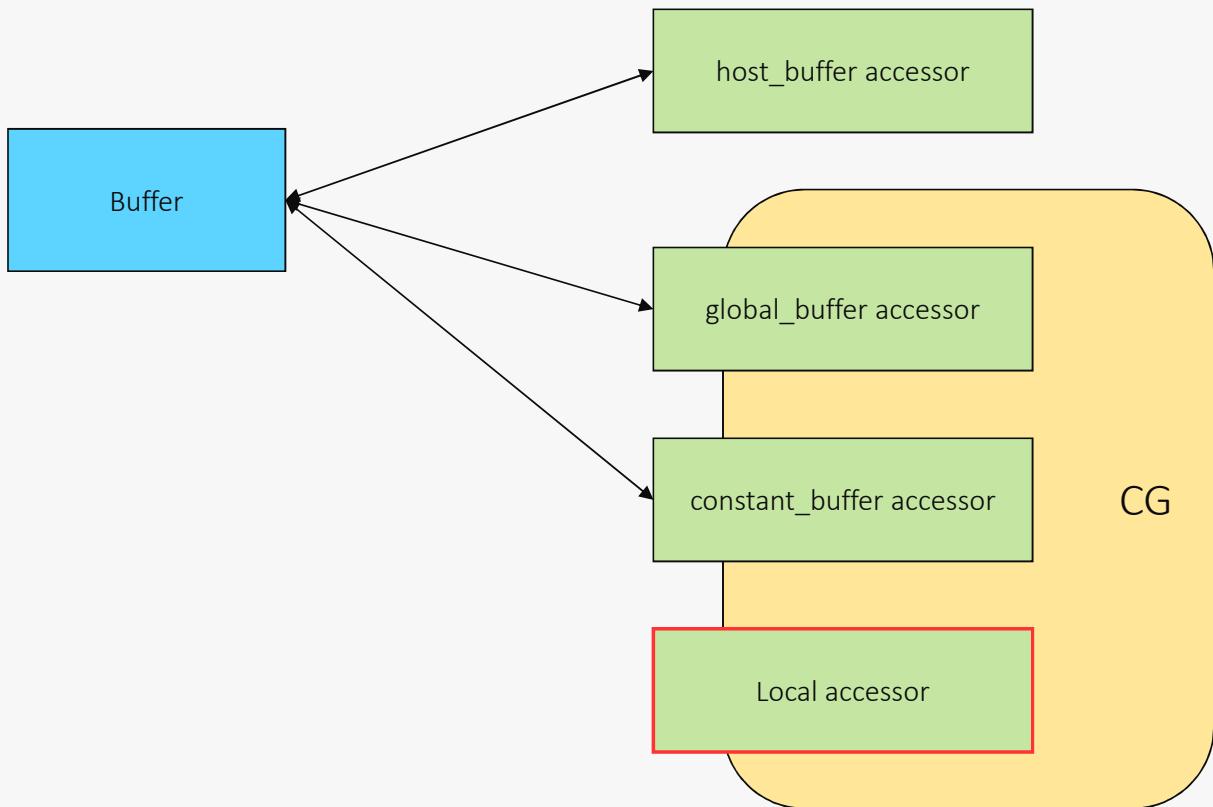


Accessors using the `access::target host_buffer` will make the data available immediately on the host

This access targets must be constructed using a buffer outside of a command group

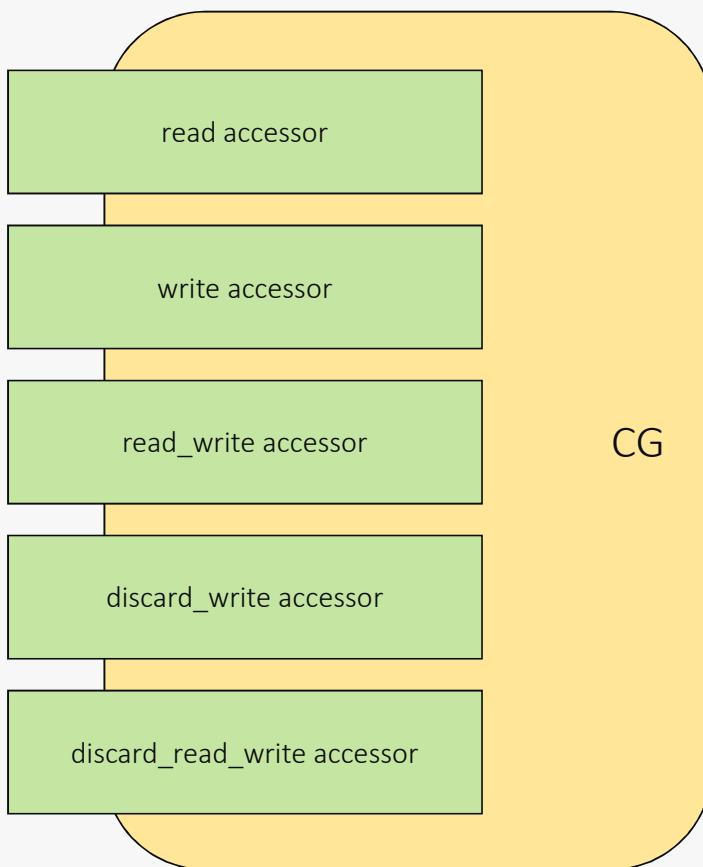
A useful shortcut to construct an accessor with the `access::target global_buffer` is to use the `get_access` member function of the buffer

Note that a host accessor will block other accessors until it's destroyed



Accessors using the `access::target local` will allocate memory in the `local` address space per work-group

This access targets does not require a buffer to be constructed as it's temporary memory allocation for the duration of a SYCL kernel function invocation



A **read** accessor instructs the SYCL runtime that the SYCL kernel function will read the data – cannot be written to within a SYCL kernel function

A **write** accessor instructs the SYCL runtime that the SYCL kernel function will modify the data – creating a dependency for future command groups

A **discard_*** accessor instructs the SYCL runtime that the SYCL kernel function does not need the initial values of the data – removing the dependency on previous command groups

Accessor resolution

- If a command group has accessors to the same buffer with conflicting access modes they are resolved into one
 - read & write => read_write
 - read_write & discard_write => discard_read_write
- Within the SYCL kernel function there are still multiple accessors, but they alias to the same memory address

```
buffer<float, 1> bufI(dB.data(), range<1>(dB.size()));
buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

    auto inA = bufI.get_access<access::mode::read>(cgh);
    auto inB = bufI.get_access<access::mode::write>(cgh);
    auto out = bufO.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ inB[i] = 10; out[i] = inA[i]; });
});
```

Here **inA** and **inA** both point to **bufI** but one is `access::mode::read` and one is `access::mode::write`

So the SYCL runtime will treat them both as `access::mode::read_write`

Both will point to a single allocation of global memory on the device

This optimization can be useful for generic programming

Accessing accessors

- There are a few different ways to access the data represented by an accessor
 - The subscript operator can take an `id`
 - Must be the same dimensionality of the accessor
 - For dimensions > 1, linear address is calculated in row major
 - Nested subscript operators can be called for each dimension taking a `size_t`
 - E.g. a 3-dimensional accessor: `acc[x][y][z] = ...`
 - A pointer to the underlying memory can be retrieved by calling `get_pointer`
 - This returns a `global_ptr`, which is a wrapper class for pointers with an address space

```
buffer<float, 3> bufA(dA, rng);
buffer<float, 3> bufB(dB, rng);
buffer<float, 3> bufO(dO, rng);

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

    auto inA = bufA.get_access<access::mode::read>(cgh);
    auto inB = bufB.get_access<access::mode::read>(cgh);
    auto out = bufO.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(rng,
        [=](id<3> i){
            out[i] = inA[i] + inB[i];
        });
}) ;
```

Here we access the data of the accessor by passing in the id object passed to the SYCL kernel function

```

buffer<float, 3> bufA(dA, rng);
buffer<float, 3> bufB(dB, rng);
buffer<float, 3> bufO(dO, rng);

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

    auto inA = bufA.get_access<access::mode::read>(cgh);
    auto inB = bufB.get_access<access::mode::read>(cgh);
    auto out = bufO.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(rng,
        [=](id<3> i) {
            auto x = i[0];
            auto y = i[1];
            auto z = i[2];

            out[z][y][x] = inA[z][y][x] + inB[z][y][x];
        });
    });
}

```

Here we access the data of the accessor by passing in the index at each dimension in nested subscript operators

```

buffer<float, 3> bufA(dA, rng);
buffer<float, 3> bufB(dB, rng);
buffer<float, 3> bufO(dO, rng);

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

    auto inA = bufA.get_access<access::mode::read>(cgh);
    auto inB = bufB.get_access<access::mode::read>(cgh);
    auto out = bufO.get_access<access::mode::write>(cgh);

    cgh.parallel_for<add>(rng,
        [=](item<3> it) {
            auto ptrA = inA.get_pointer();
            auto ptrB = inB.get_pointer();
            auto ptrO = out.get_pointer();
            auto linearId = it.get_linear_id();

            ptrA[linearId] = ptrB[linearId] + ptrO[linearId];
        });
    });
});

```

Here we retrieve the underlying pointe of each of the accessors

We then access the pointer using the linearized id by calling the `get_linear_id` member function on the `item` class

Again this linearization is calculated in row major

SYCL data dependency analysis

- When a command group is submitted to a SYCL queue the runtime performs dependency analysis
 - If the command group requests access to a buffer this creates a pre-requisite that the data must be available before the SYCL kernel function executes
 - The SYCL scheduler uses these pre-requisites to order the execution of SYCL kernel functions
- Data is copied when required or when explicitly requested
 - Data will stay on the device to avoid unnecessary copies back to the host

```

queue cpuQueue(cpu_selector{}, async_handler{});

cpuQueue.submit([&] (handler &cgh){ // CG A
    auto in = bufA.get_access<access::mode::read>(cgh);
    auto out = bufB.get_access<access::mode::write>(cgh);

    cgh.parallel_for<A>(range<1>(dA.size()), func(in, out));
});

cpuQueue.submit([&] (handler &cgh){ // CG B
    auto in = bufB.get_access<access::mode::read>(cgh);
    auto out = bufC.get_access<access::mode::write>(cgh);

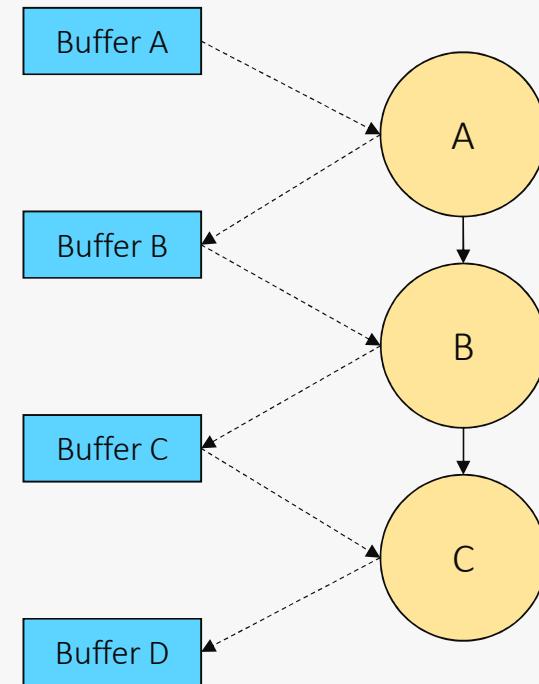
    cgh.parallel_for<B>(range<1>(dA.size()), func(in, out));
});

cpuQueue.submit([&] (handler &cgh){ // CG C
    auto in = bufC.get_access<access::mode::read>(cgh);
    auto out = bufD.get_access<access::mode::write>(cgh);

    cgh.parallel_for<C>(range<1>(dA.size()), func(in, out));
});

cpuQueue.wait_and_throw();

```



Each command group has a data dependency on the previous forcing a sequential execution order

```

queue cpuQueue(cpu_selector{}, async_handler{});

cpuQueue.submit([&] (handler &cgh){ // CG A
    auto in = bufA.get_access<access::mode::read>(cgh);
    auto out = bufB.get_access<access::mode::write>(cgh);

    cgh.parallel_for<A>(range<1>(dA.size()), func(in, out));
});

cpuQueue.submit([&] (handler &cgh){ // CG B
    auto in = bufB.get_access<access::mode::read>(cgh);
    auto out = bufC.get_access<access::mode::write>(cgh);

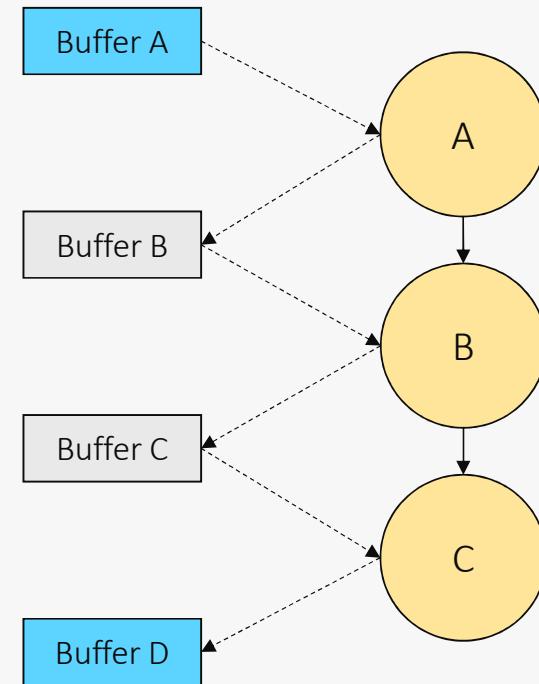
    cgh.parallel_for<B>(range<1>(dA.size()), func(in, out));
});

cpuQueue.submit([&] (handler &cgh){ // CG C
    auto in = bufC.get_access<access::mode::read>(cgh);
    auto out = bufD.get_access<access::mode::write>(cgh);

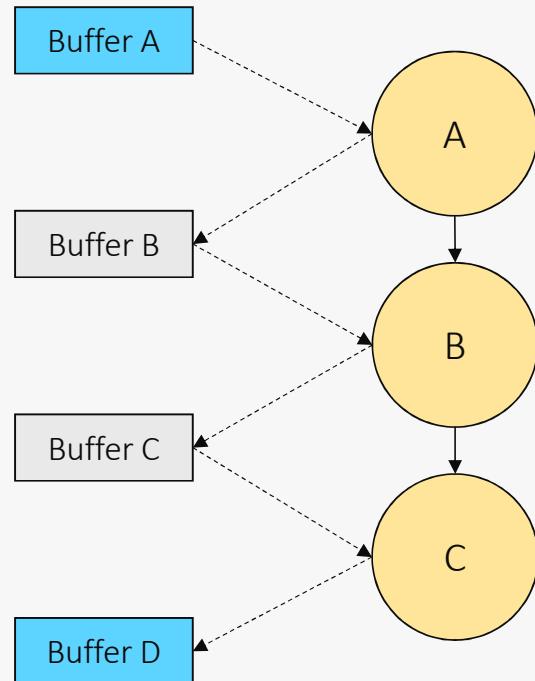
    cgh.parallel_for<C>(range<1>(dA.size()), func(in, out));
});

cpuQueue.wait_and_throw();

```



Buffers B and C are not accessed on the host so they can be optimized to remain on the device



As these commands are required to execute in sequence, they are enqueued to OpenCL with events between each one

There are no copies required for buffers B and C as they remain on the device



```

queue cpuQueue(cpu_selector{}, async_handler{});

cpuQueue.submit([&] (handler &cgh) { // CG A
    auto in = bufA.get_access<access::mode::read>(cgh);
    auto out = bufB.get_access<access::mode::write>(cgh);

    cgh.parallel_for<A>(range<1>(dA.size()), func(in, out));
});

cpuQueue.submit([&] (handler &cgh) { // CG B
    auto in = bufA.get_access<access::mode::read>(cgh);
    auto out = bufC.get_access<access::mode::write>(cgh);

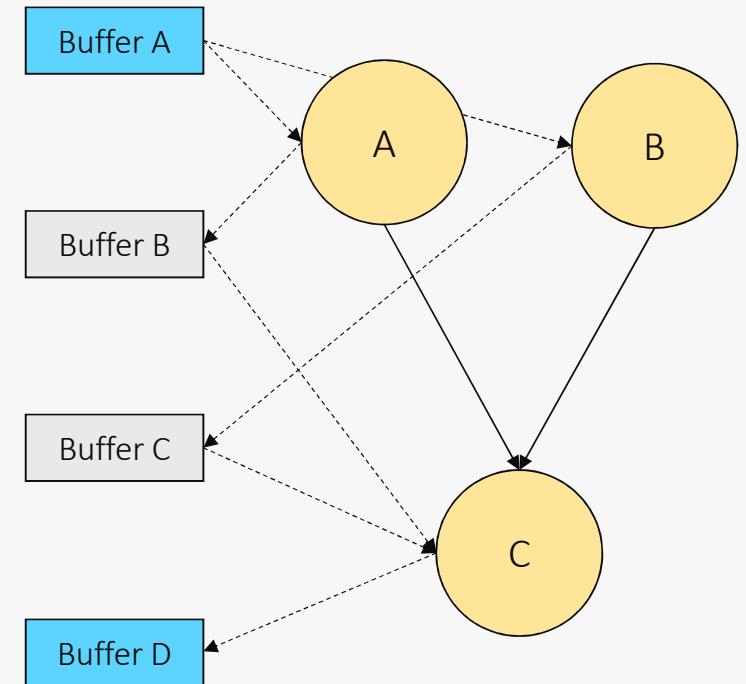
    cgh.parallel_for<B>(range<1>(dA.size()), func(in, out));
});

cpuQueue.submit([&] (handler &cgh) { // CG C
    auto inA = bufB.get_access<access::mode::read>(cgh);
    auto inB = bufC.get_access<access::mode::read>(cgh);
    auto out = bufD.get_access<access::mode::write>(cgh);

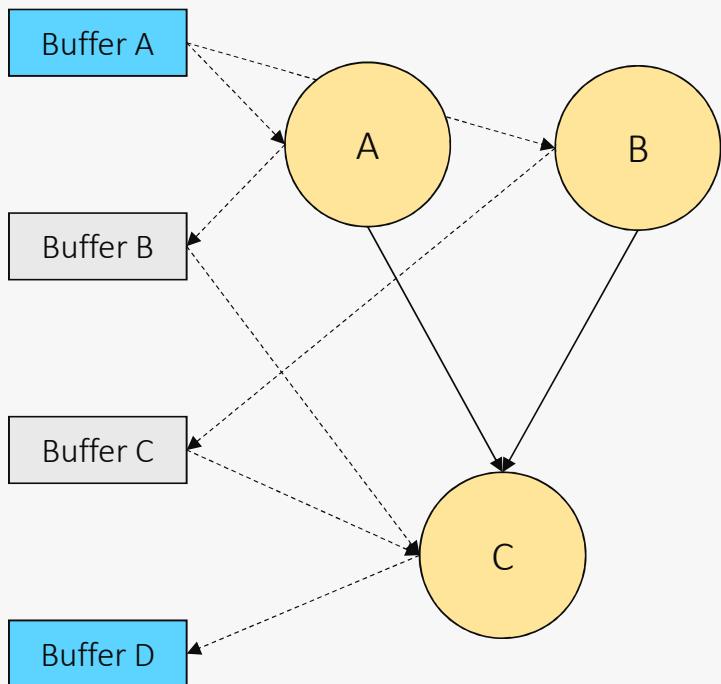
    cgh.parallel_for<C>(range<1>(dA.size()), func(inA, inB, out));
});

cpuQueue.wait_and_throw();

```



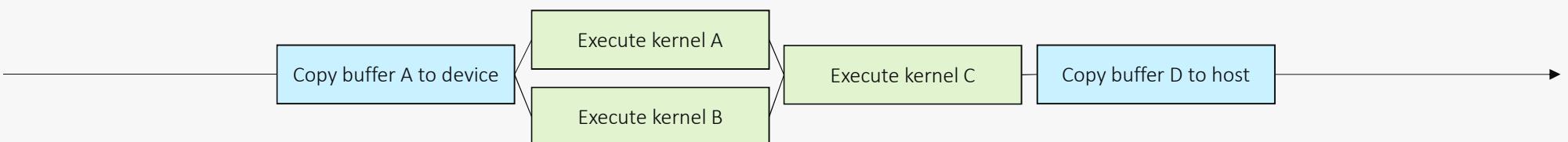
The third command group has data dependencies on both of the previous allowing the first two command groups to run concurrently



As command groups A and B are only reading from buffer A, they can both access it concurrently

As there are no dependencies between command groups A and B they can be run in parallel if the hardware allows it

Again, there are no copies required for buffers B and C as they remain on the device



Explicit copy commands

- As well as SYCL kernel functions a command group can also contain explicit copy commands
 - These commands enqueue a copy operation to the SYCL scheduler with the same data dependency analysis
 - This can be used to perform double buffering of copy and compute

```

queue cpuQueue(cpu_selector{}, async_handler{});

cpuQueue.submit([&] (handler &cgh) { // Copy
    auto ptr = bufA.get_access<access::mode::read>(cgh);

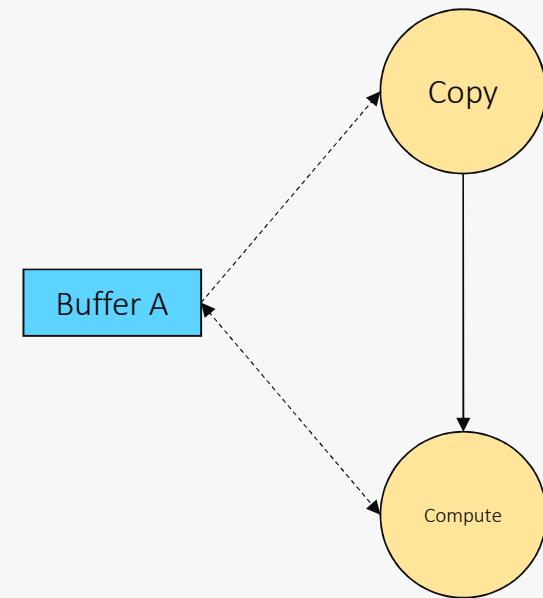
    cgh.copy(data, ptr);
});

cpuQueue.submit([&] (handler &cgh) { // Compute
    auto ptr = bufA.get_access<access::mode::read_write>(cgh);

    cgh.parallel_for<A>(range<1>(dA.size()), func(ptr));
});

cpuQueue.wait_and_throw();

```



The command group performing the copy must complete before the command group performing the computation

```

queue cpuQueue(cpu_selector{}, async_handler{});

cpuQueue.submit([&] (handler &cgh){ // Copy A
    auto ptr = bufA.get_access<access::mode::read>(cgh);
    cgh.copy(data, ptr);
});

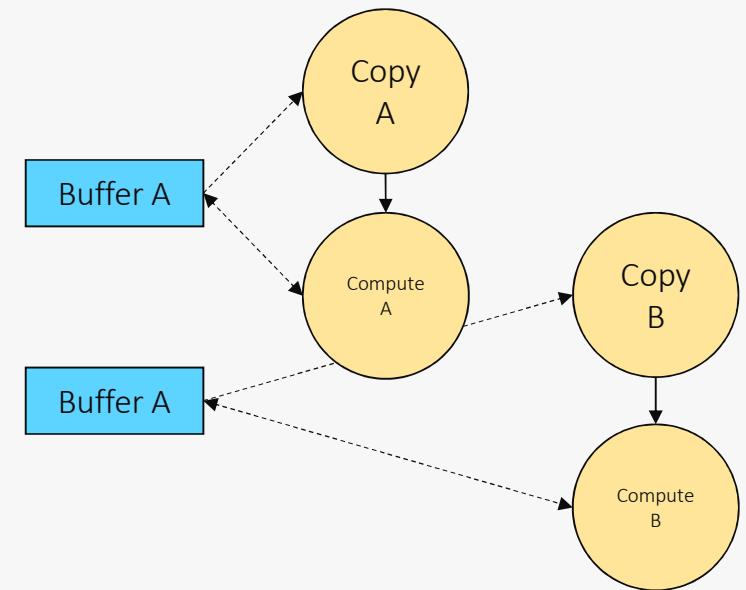
cpuQueue.submit([&] (handler &cgh){ // Compute A
    auto ptr = bufA.get_access<access::mode::read_write>(cgh);
    cgh.parallel_for<A>(range<1>(dA.size()), func(ptr));
});

cpuQueue.submit([&] (handler &cgh){ // Copy B
    auto ptr = bufB.get_access<access::mode::read>(cgh);
    cgh.copy(data, ptr);
});

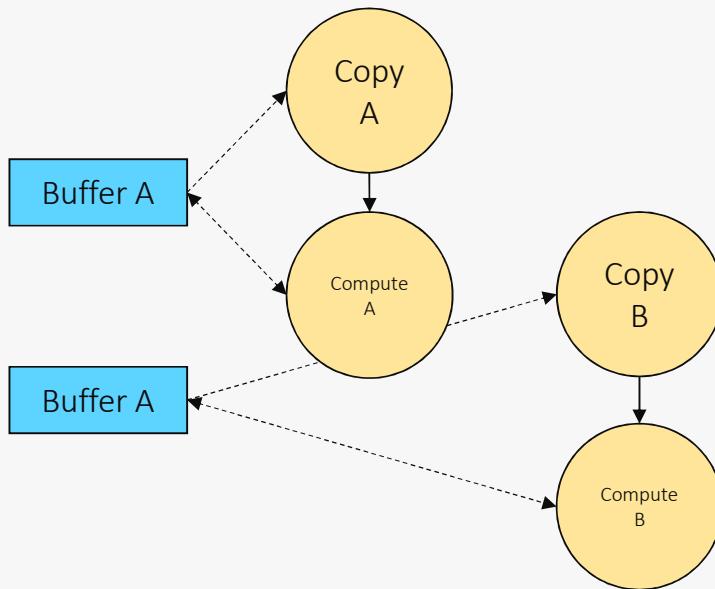
cpuQueue.submit([&] (handler &cgh){ // Compute B
    auto ptr = bufB.get_access<access::mode::read_write>(cgh);
    cgh.parallel_for<B>(range<1>(dB.size()), func(ptr));
});

cpuQueue.wait_and_throw();

```

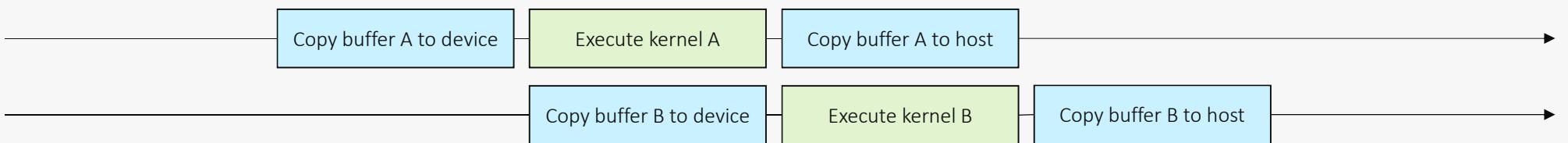


The command group copying buffer B can execute concurrently with the compute of buffer A



The copy and compute on buffer A and buffer B are independent so they have separate chains of events

This means that they can be run in parallel, double buffering copy and compute



Ranged accessors

- By default accessors access the entire buffer, however it's possible to access only a region of a buffer
 - Only the region of the buffer that you are accessing is copied
 - This is particularly useful for tiling larger data

```

queue cpuQueue(cpu_selector{}, async_handler{});

cpuQueue.submit([&] (handler &cgh) { // Copy first half
    auto ptr = bufA.get_access<access::mode::read>(cgh, halfSize,
origin);

    cgh.copy(data, ptr);
});

cpuQueue.submit([&] (handler &cgh) { // Compute first half
    auto ptr
    = bufA.get_access<access::mode::read_write>(cgh, halfSize, origin);

    cgh.parallel_for<A>(range<1>(dA.size()), func(ptr));
});

cpuQueue.submit([&] (handler &cgh) { // Copy second half
    auto ptr =
        bufB.get_access<access::mode::read>(cgh, halfSize, halfSize);

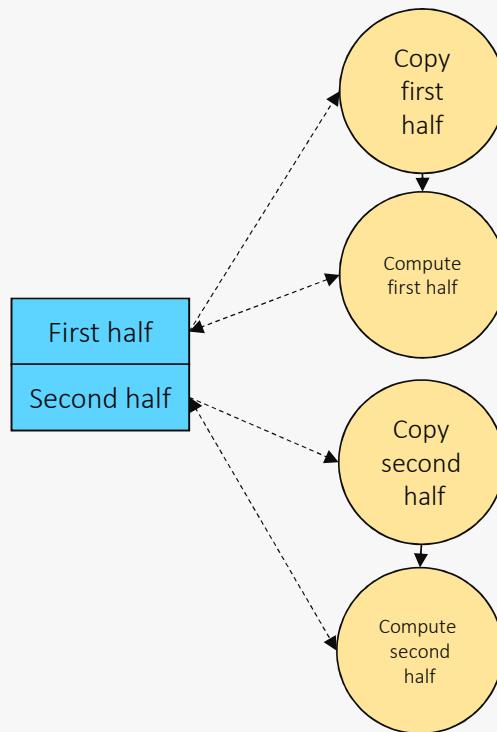
    cgh.copy(data, ptr);
});

cpuQueue.submit([&] (handler &cgh) { // Compute second half
    auto ptr =
        bufB.get_access<access::mode::read_write>(cgh, halfSize, halfSize);

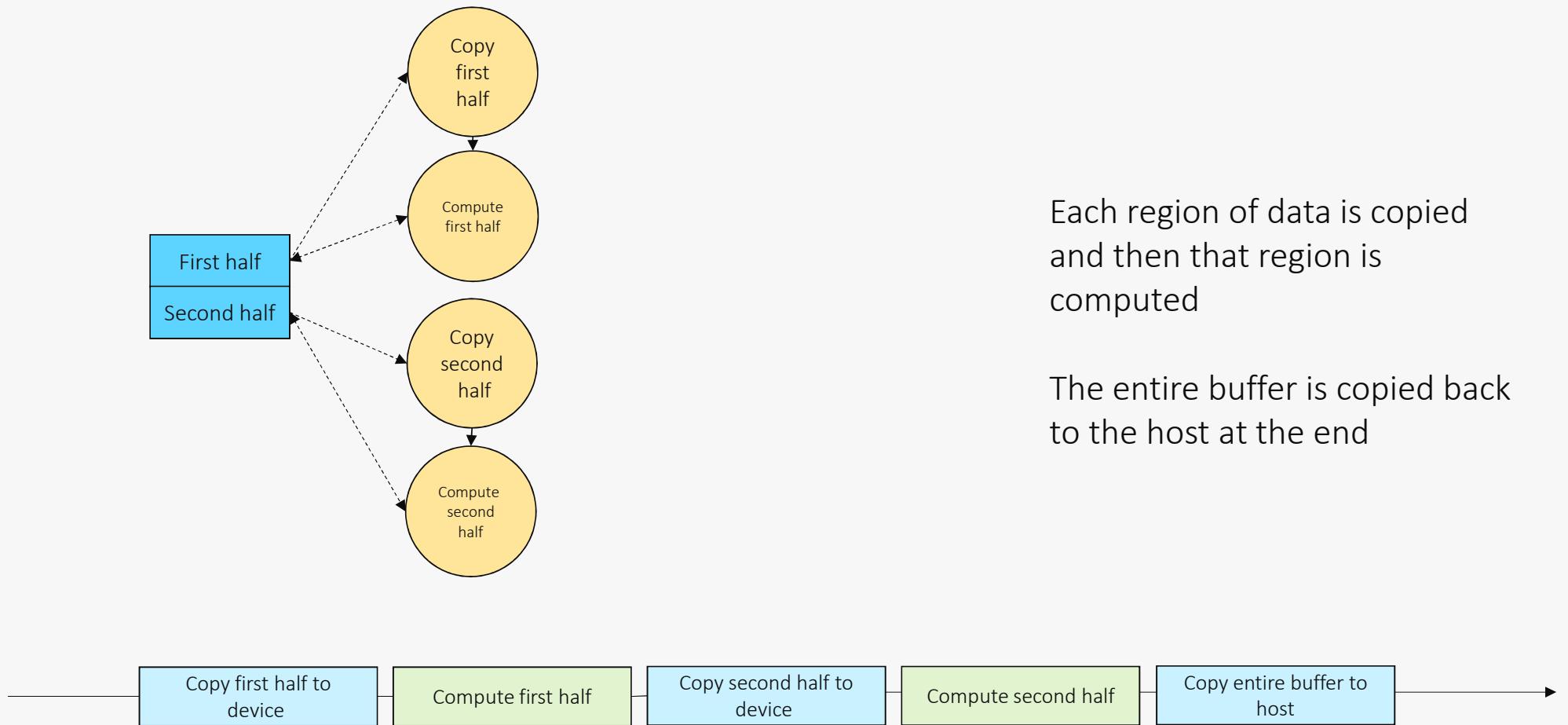
    cgh.parallel_for<B>(range<1>(dA.size()), func(ptr));
});

cpuQueue.wait_and_throw();

```



The first copy and compute operate on the first half of the buffer and the other copy and compute operate on the second half



Key takeaways

SYCL has a hierarchy of address spaces with varying sizes and latency

SYCL **buffers** manage data and SYCL **accessors** provide access to data

The SYCL runtime will optimise kernel scheduling and move data using data dependencies specified by accessors



Questions?

Exercise 3:

Vector add

- How to manage data using buffers
- How to access data using accessors
- How to define nd-range SYCL kernel function



Chapter 11: Fundamentals of Parallelism

Gordon Brown

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about communication patterns
 - Learn about reordering algorithms
 - Learn about handling dependencies
 - Learn about work distribution

Communication patterns

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors working together to solve a problem

Communication patterns

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors **working together** to solve a problem

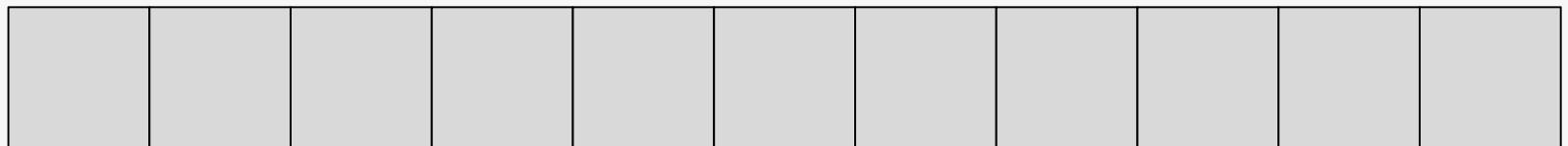
This requires communication, and in parallel computing this is done via memory

Communication patterns

Parallel computing is about breaking up a problem into smaller tasks and having multiple processors **working together** to solve a problem

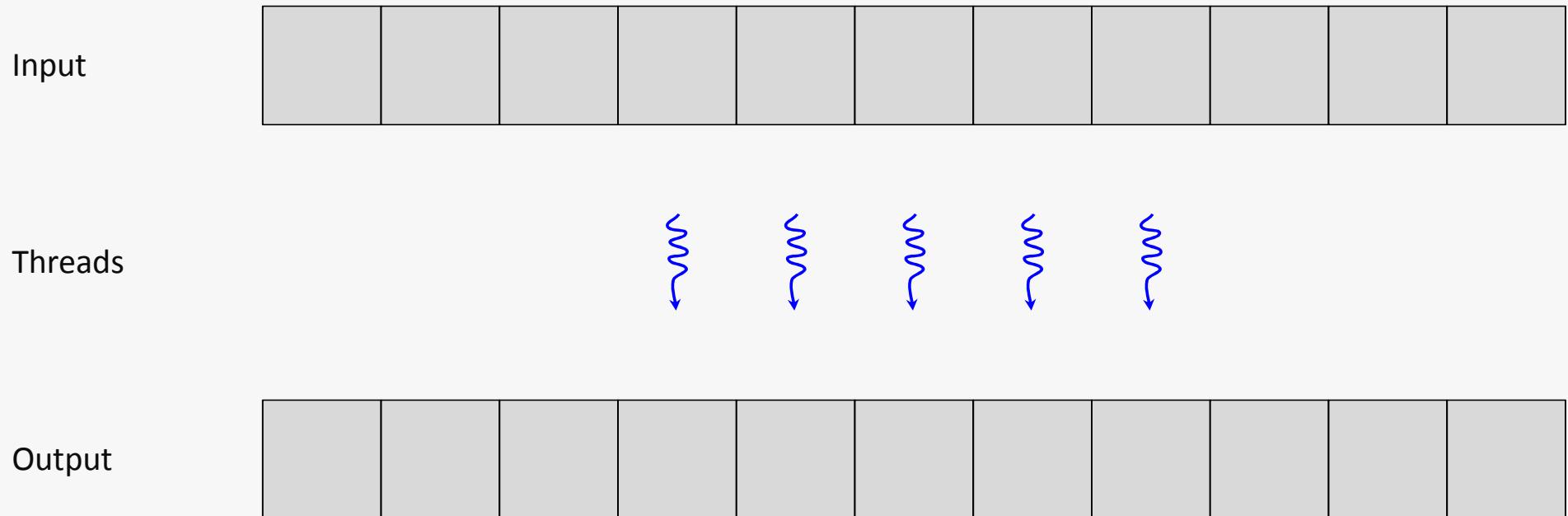
This requires communication, and in parallel computing this is done via memory

Memory



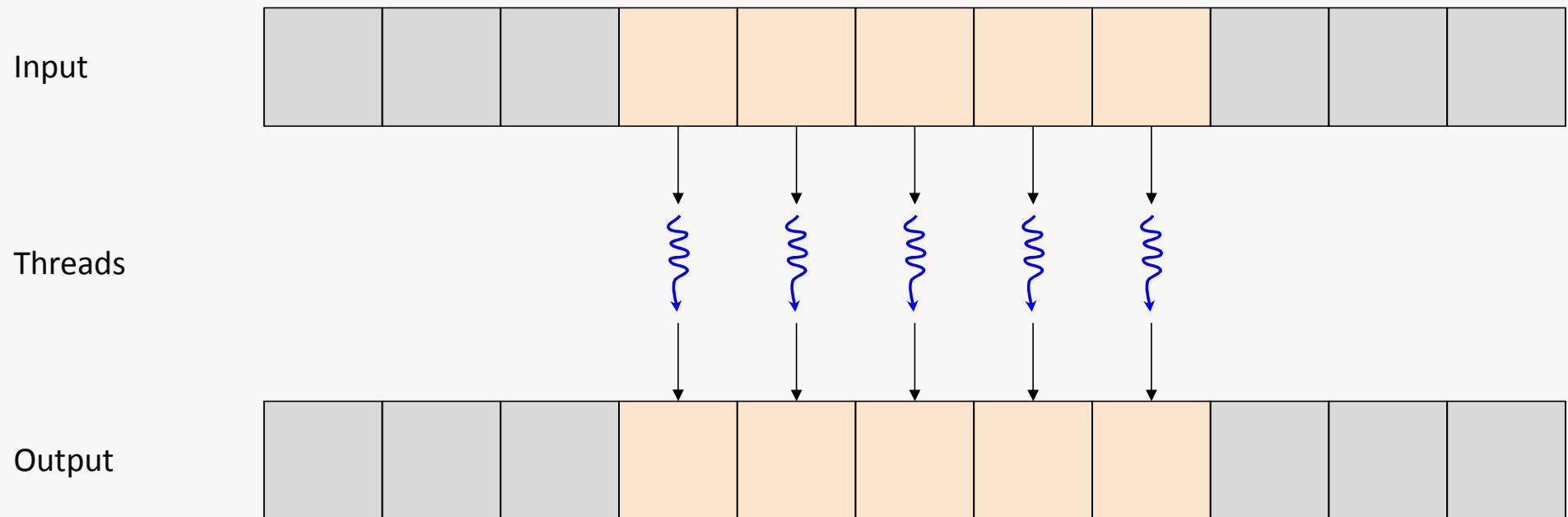
Communication patterns

Communication patterns are used to describe the relationship between threads and the data they read from and write to



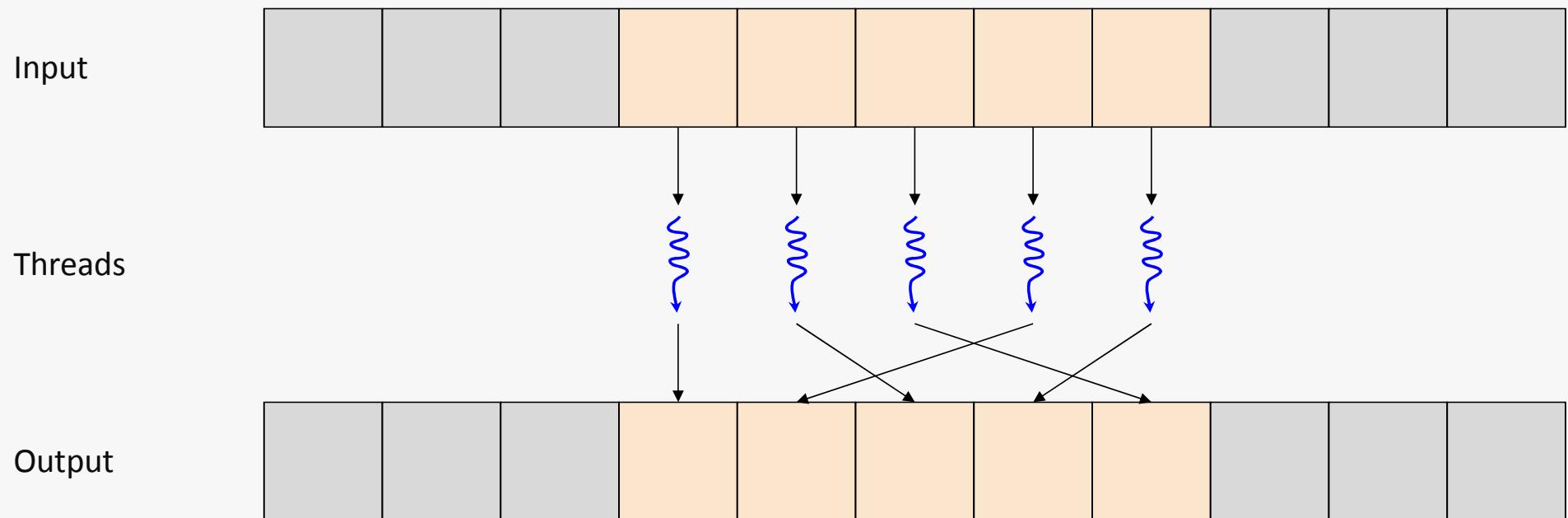
Map pattern

A map pattern is any operation in which each element of the input range maps to the same element of the output range.



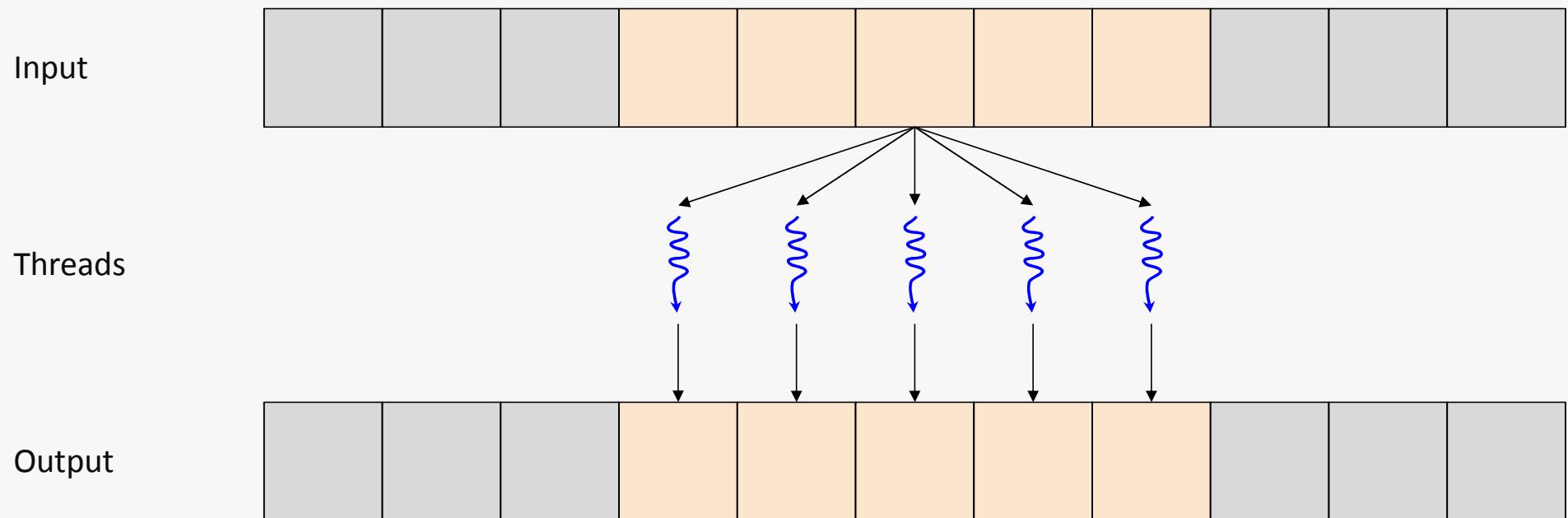
Transpose pattern

A transpose pattern is any operation in which each element of the input range maps to a different element of the output range.



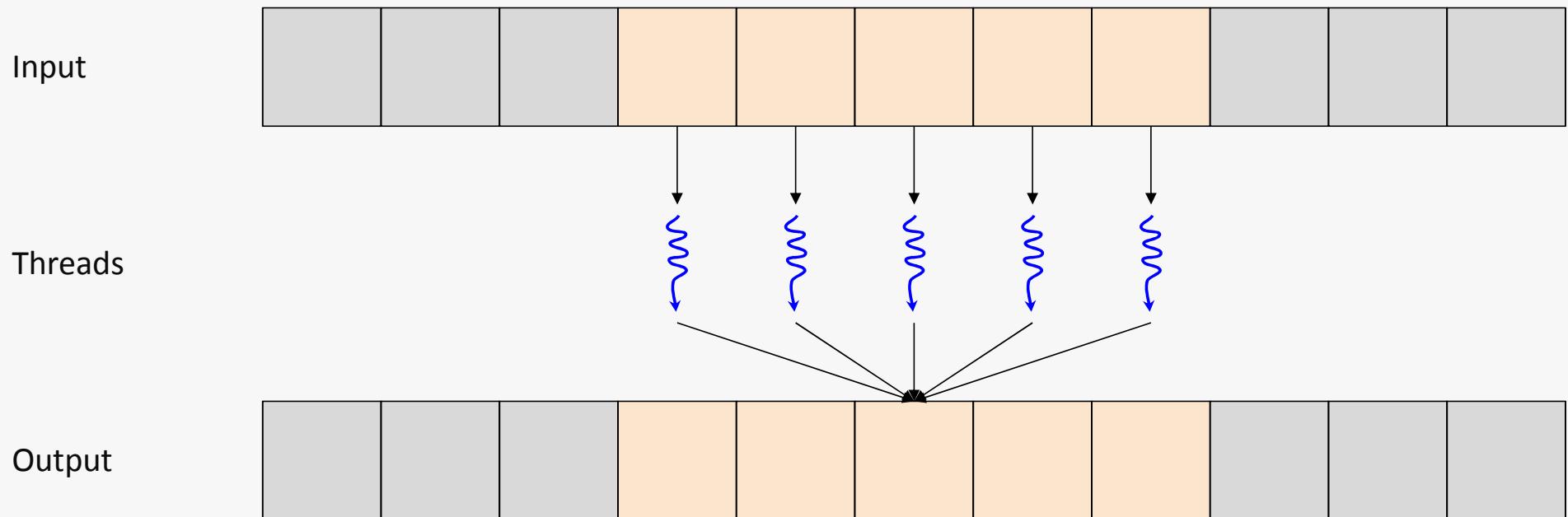
Scatter pattern

A scatter pattern is any operation in which a single element of the input range maps to multiple elements of the output range.



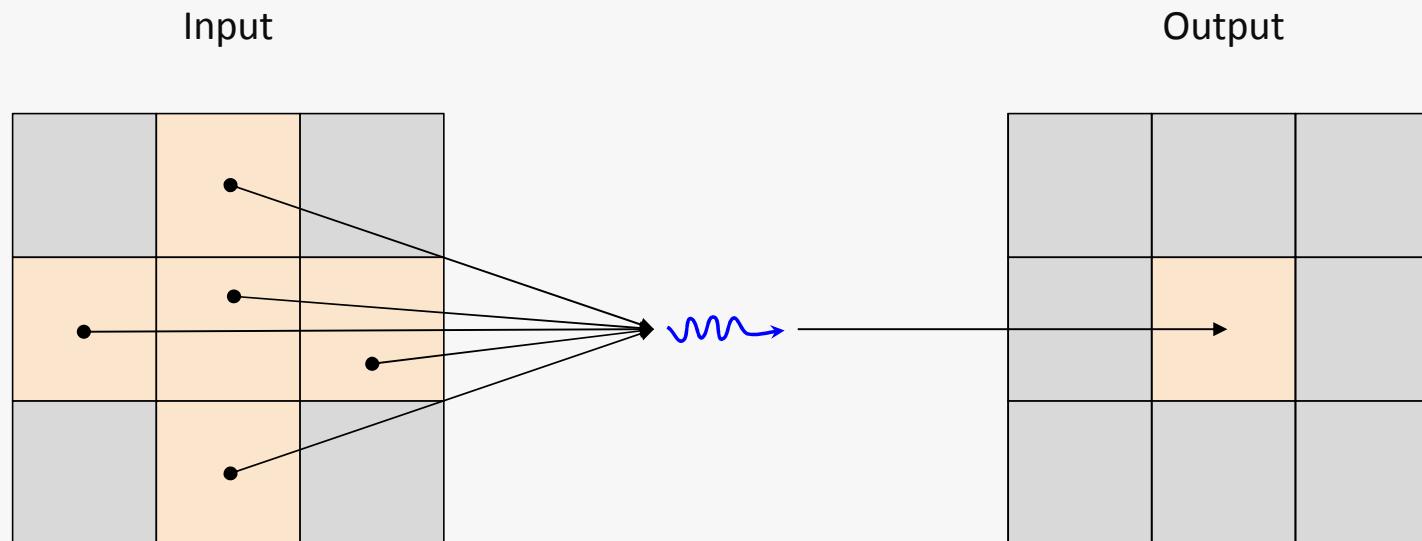
Gather pattern

A gather pattern is any operation in which multiple elements of the input range maps to a single element of the output range.



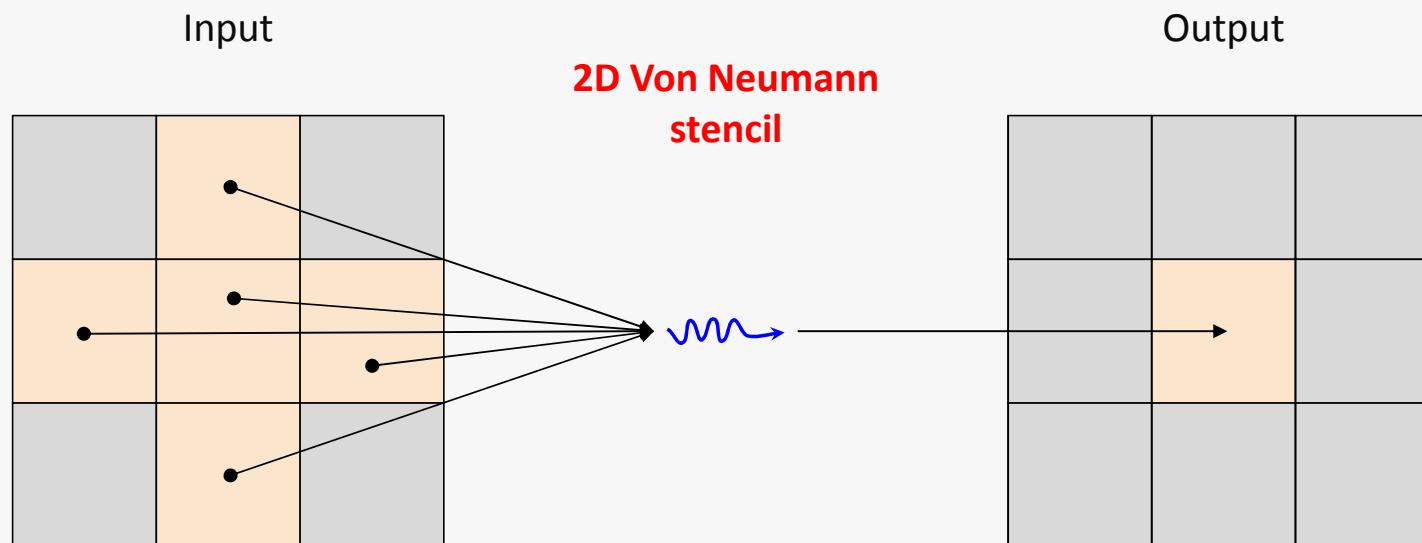
Stencil pattern

A stencil pattern is a special case of the gather pattern where elements are arranged a multi-dimensional space in which a grouping of elements of the input range maps to a single element of the output range.



Stencil pattern

A stencil pattern is a special case of the gather pattern where elements are arranged a multi-dimensional space in which a grouping of elements of the input range maps to a single element of the output range.



What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[128 - index];  
3. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[128 - index];  
3. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[index];  
3. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     out[index] = pi * in[index];  
3. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index] = (in[index] + in[index - 1] + in[index + 1]) / 3;  
4.     }  
5. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index] = (in[index] + in[index - 1] + in[index + 1]) / 3;  
4.     }  
5. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index - 1] = in[index] / 2;  
4.         out[index + 1] = in[index] / 2;  
5.     }  
6. }
```

Map

Transpose

Scatter

Gather

Stencil

What kind of communication pattern does this code embody?

```
1. void foo(int *in, int *out, int index) {  
2.     if(index % 2) {  
3.         out[index - 1] = in[index] / 2;  
4.         out[index + 1] = in[index] / 2;  
5.     }  
6. }
```

Map

Transpose

Scatter

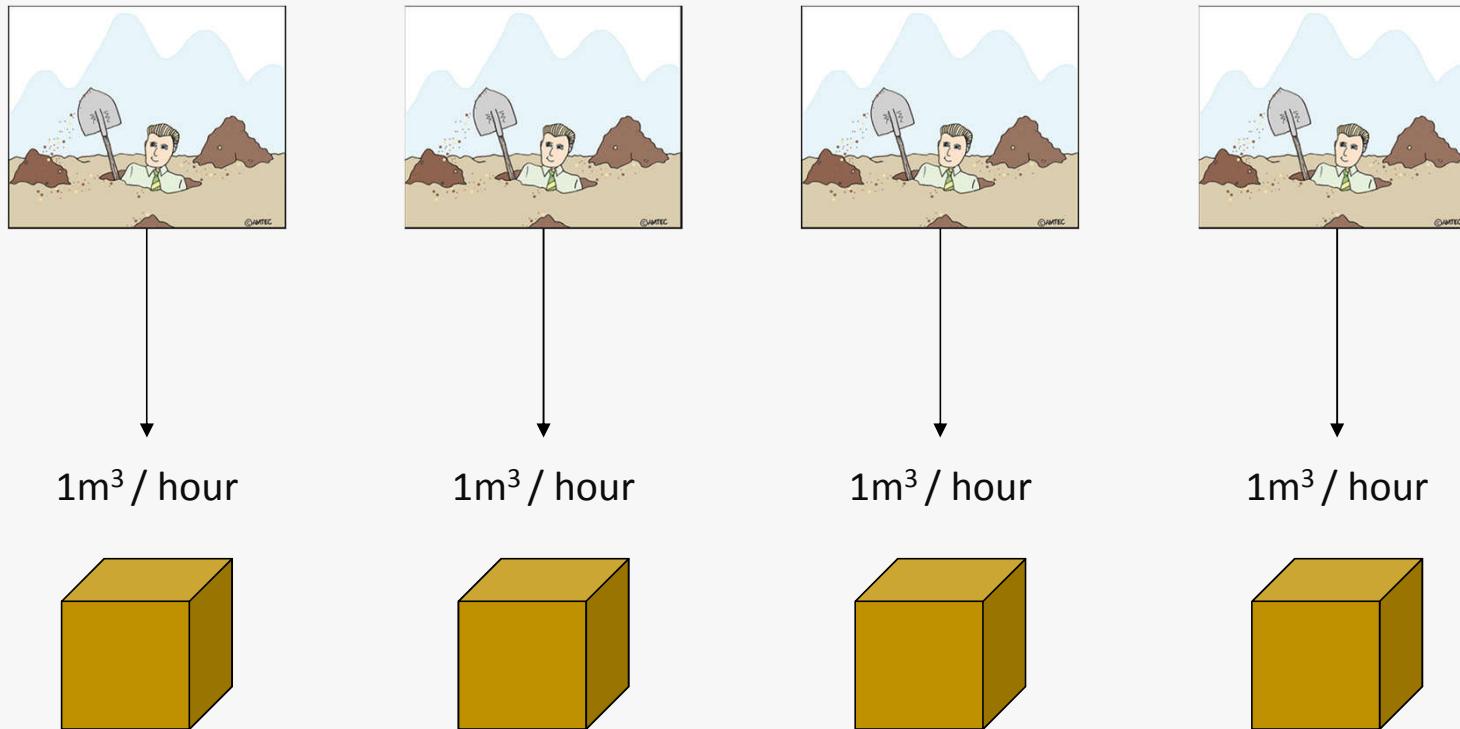
Gather

Stencil

Let's go back to the holes analogy...



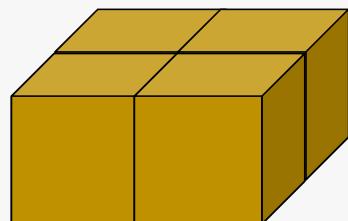
Say you now have four diggers...



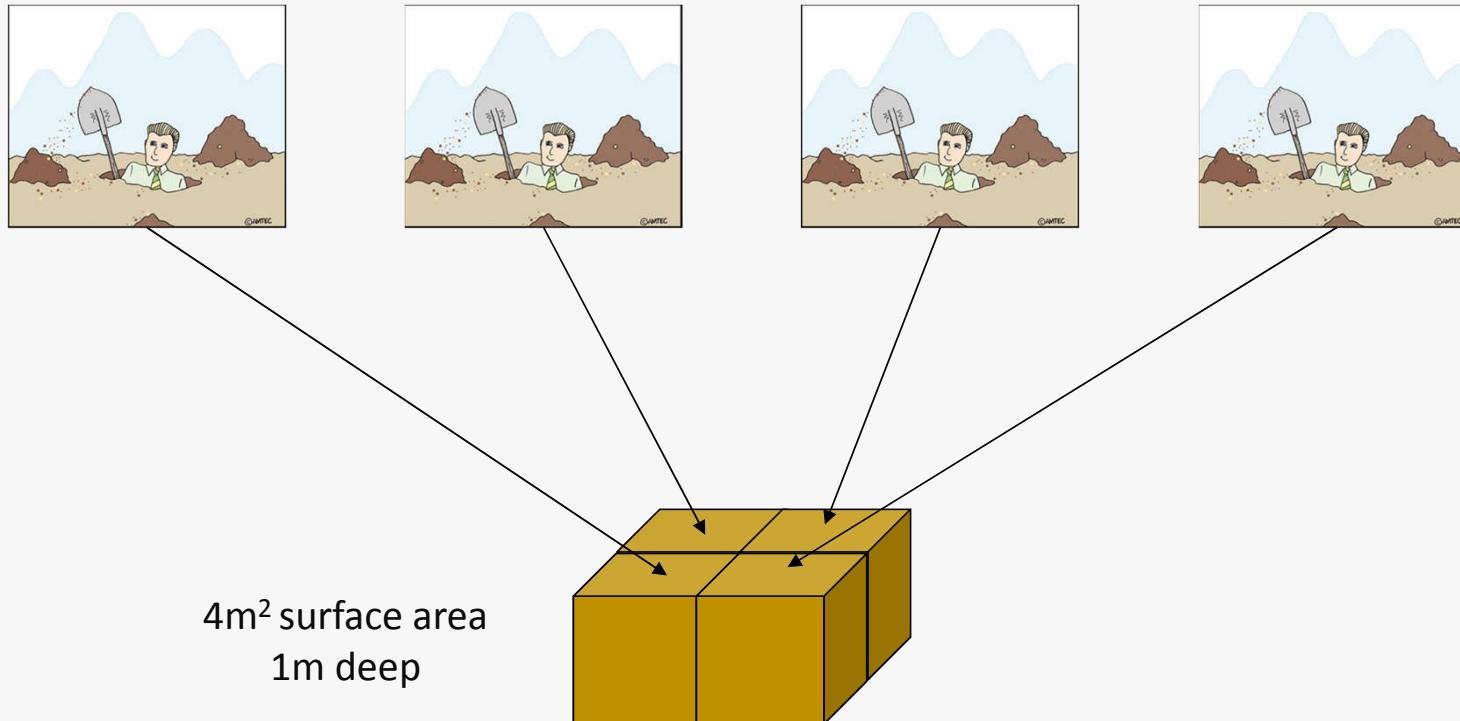
Say you want a hole with a 4m^2 surface area



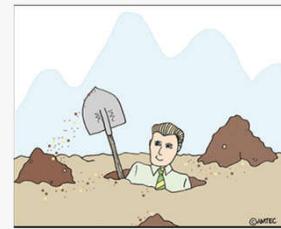
4m^2 surface area
1m deep



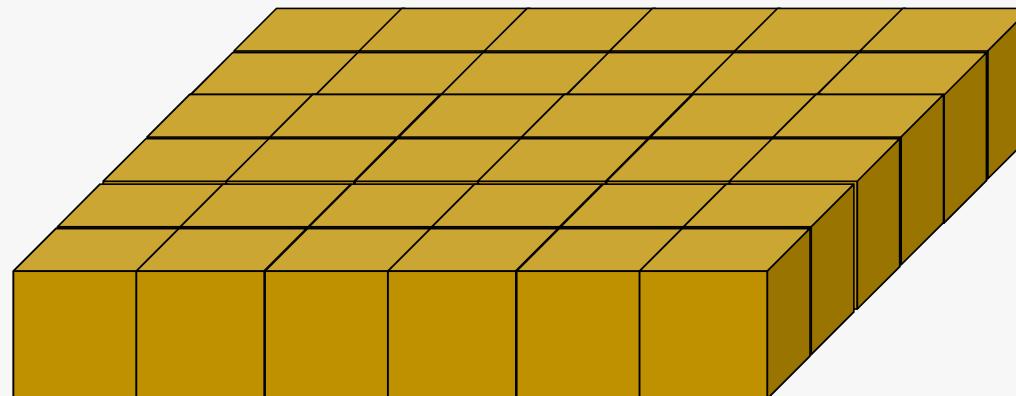
Each digger digs a part each



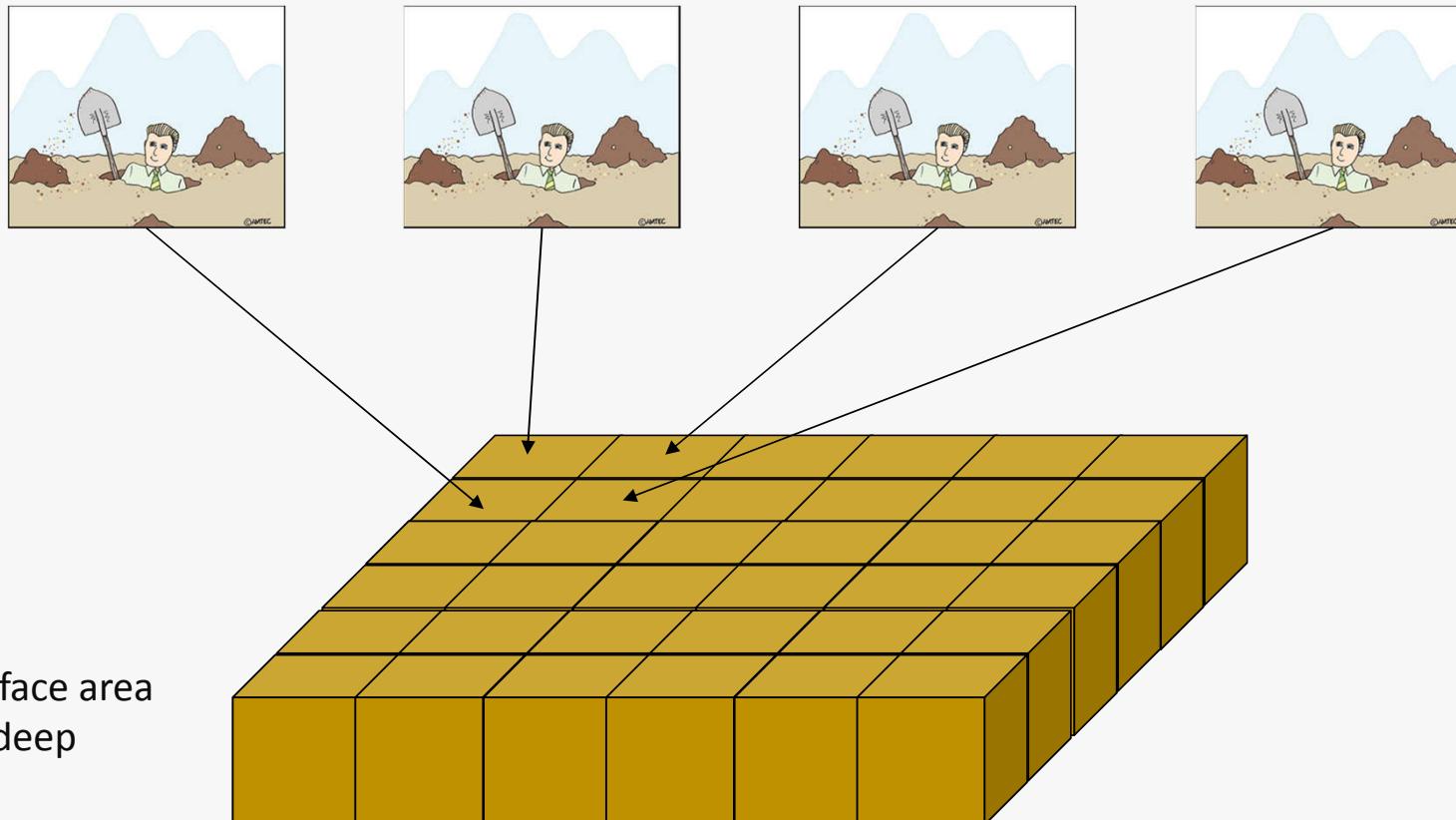
Say you want a hole with a 36m^2 surface area



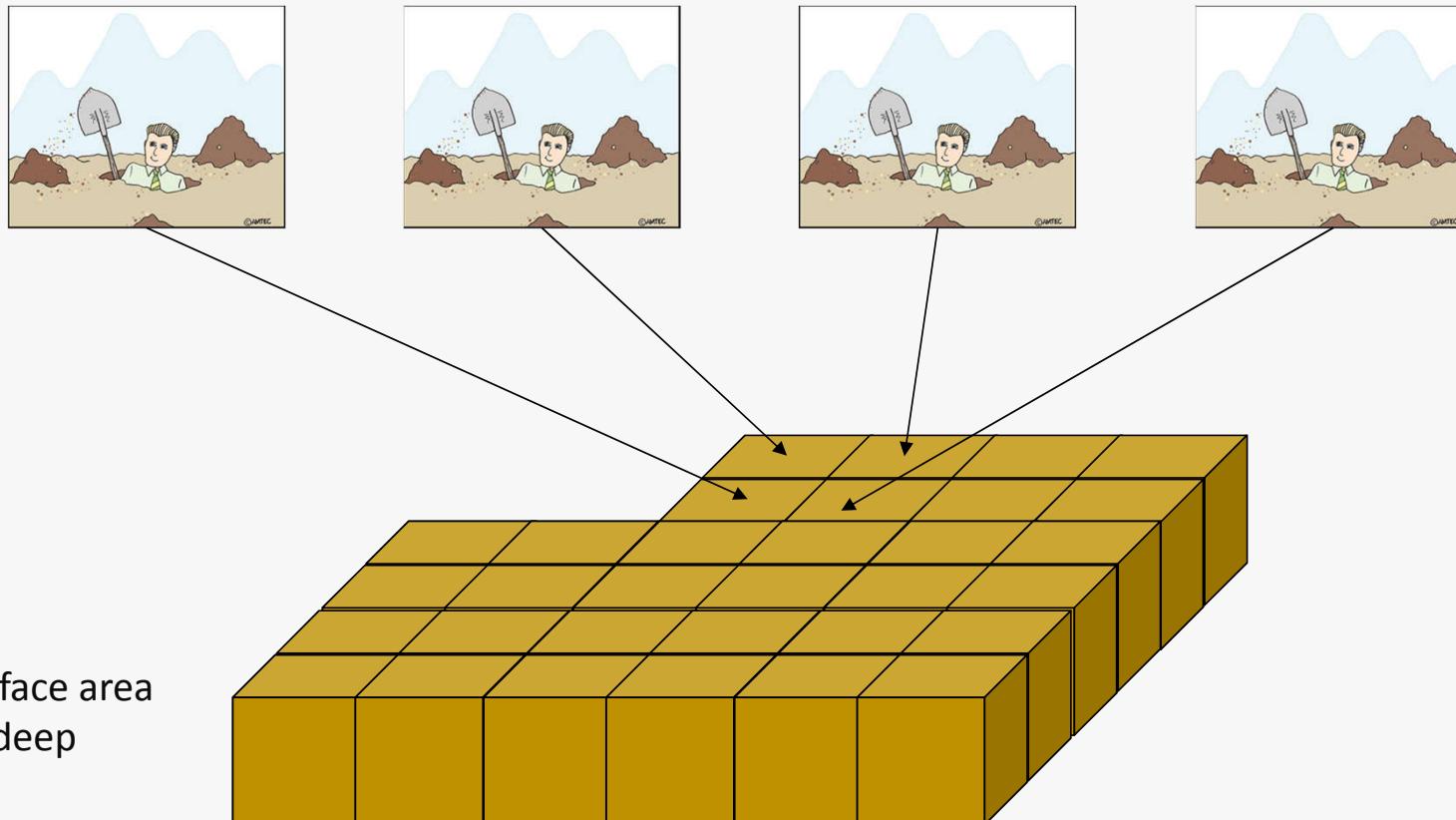
36m^2 surface area
1m deep



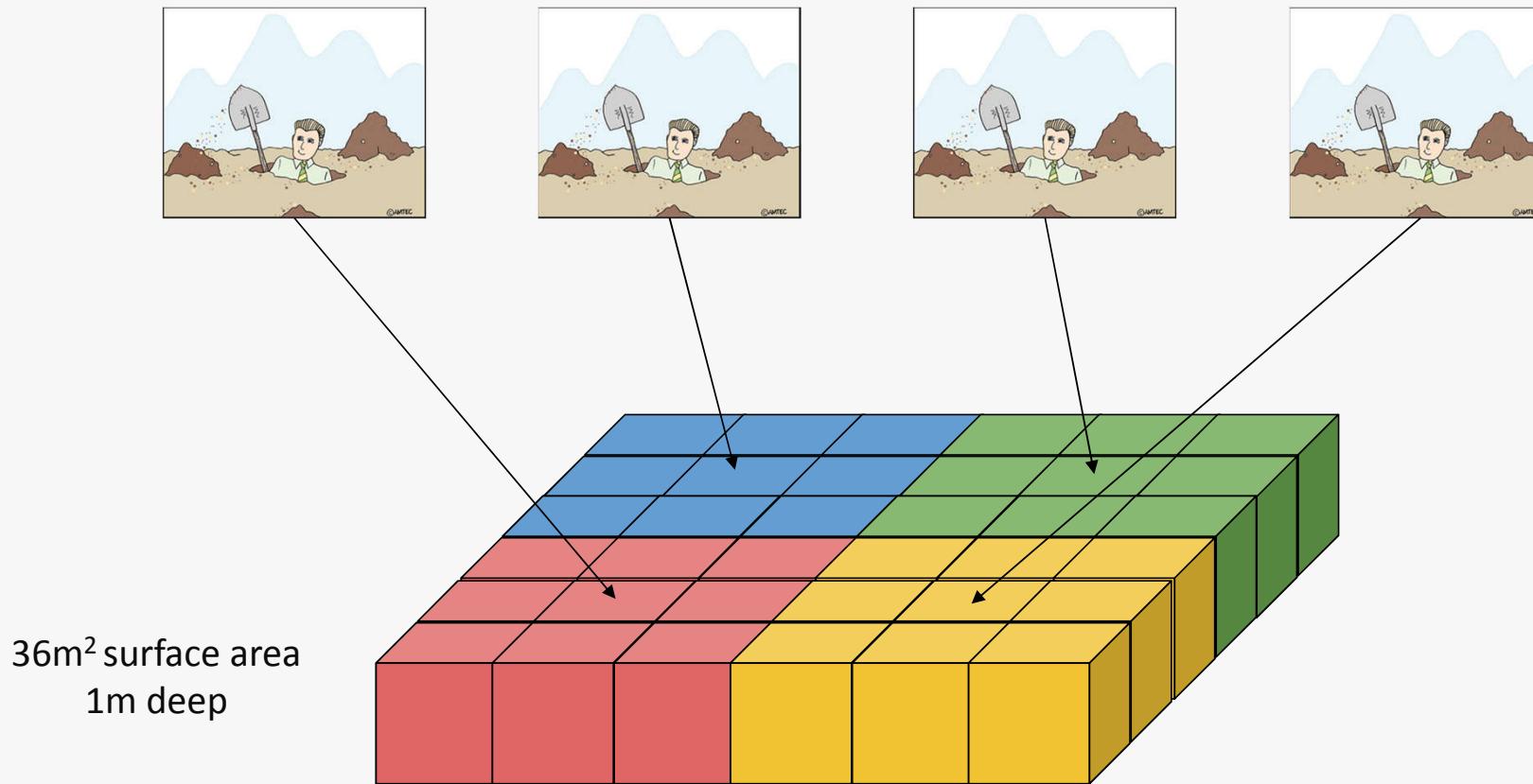
You can share the work between the diggers



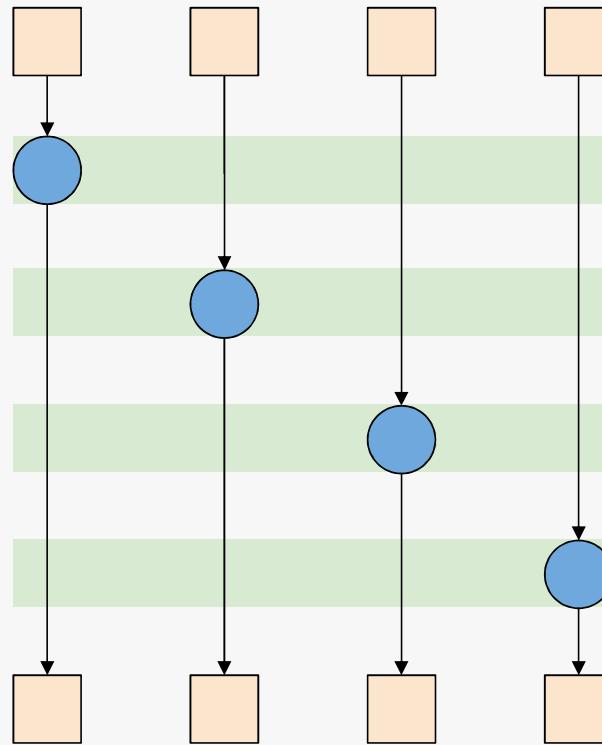
You can share the work between the diggers



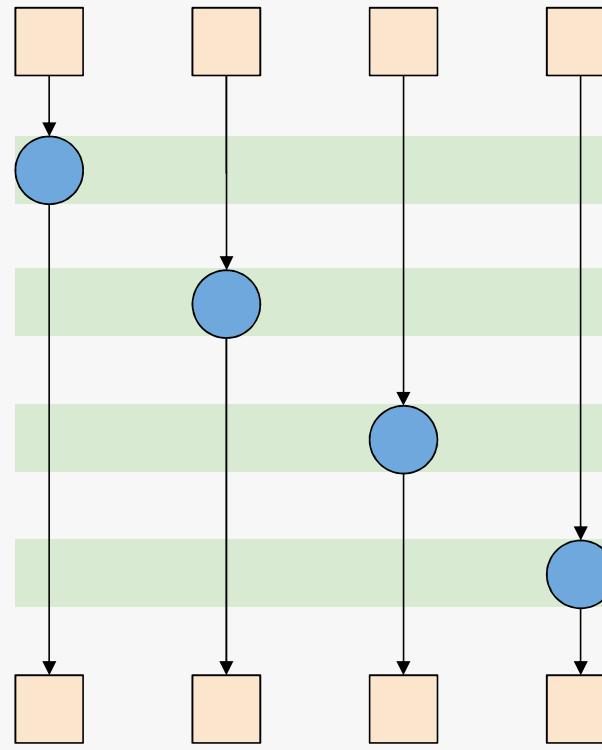
You can distribute work across the diggers



This applies to a transform algorithm

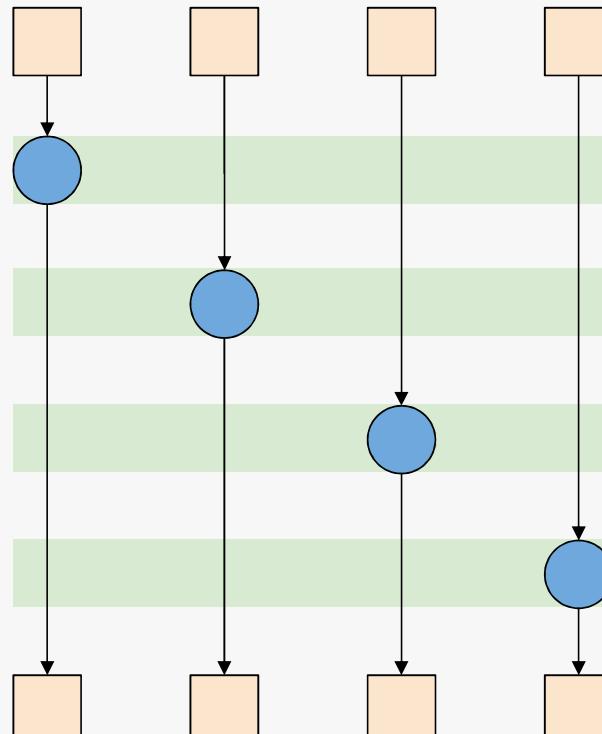


Let's look at a serial transform...



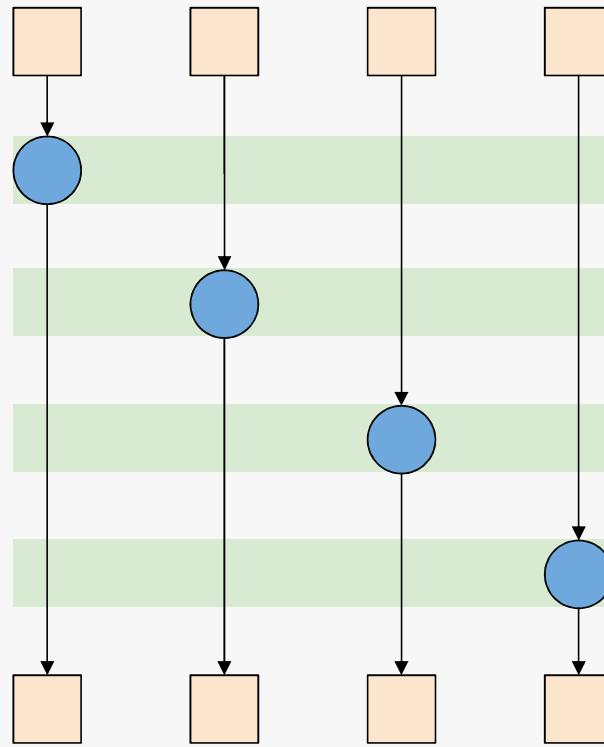
4 elements

Let's look at a serial transform...



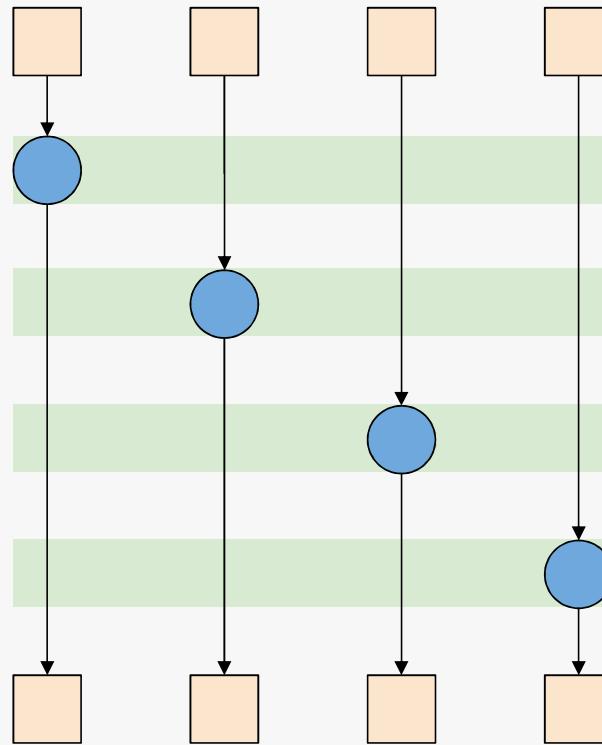
4 elements | 4 Operations

Let's look at a serial transform...



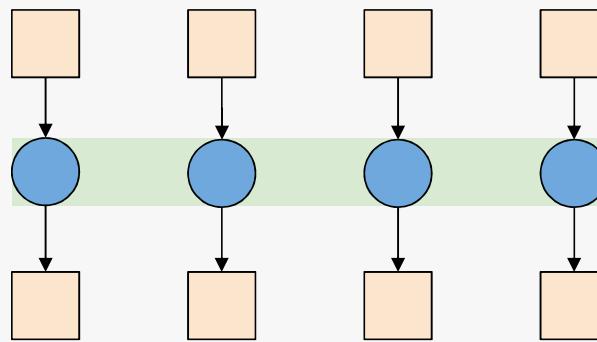
4 elements | 4 Operations | 4 steps

Let's look at a serial transform...



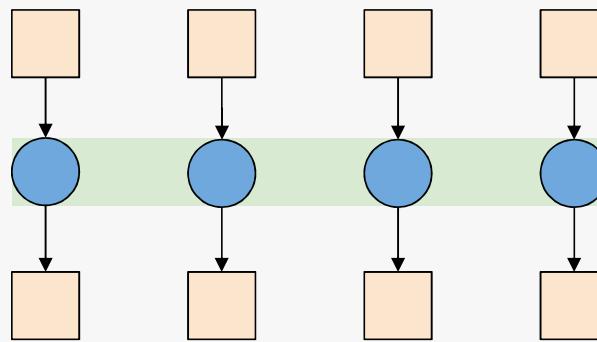
4 elements | 4 Operations| 4 steps | 1 operations / step

Now let's look at a parallel transform...



4 elements | 4 Operations| **1 step** | **4 operations / step**

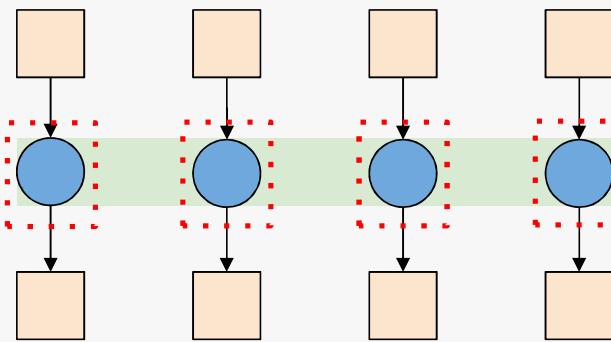
Now let's look at a parallel transform...



Brent's theorem

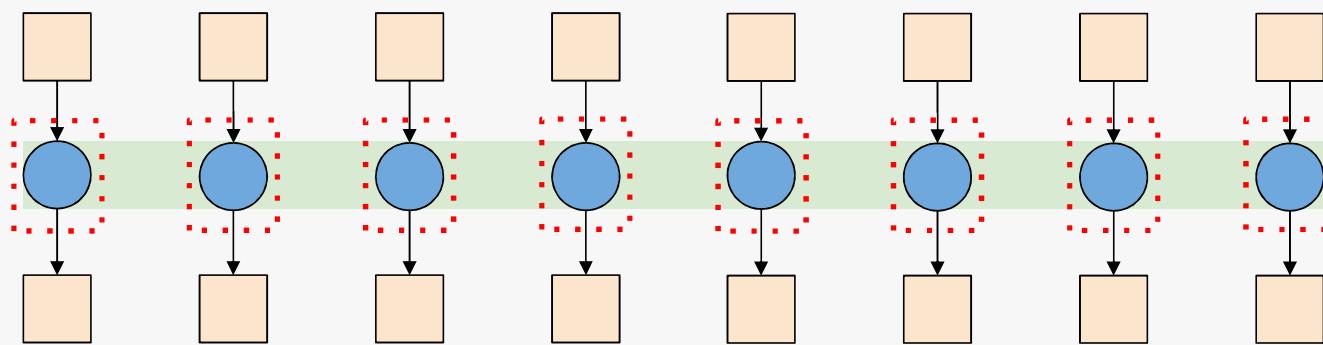
4 elements | 4 Operations | 1 step | 4 operations / step

In order to do this you need parallel workers



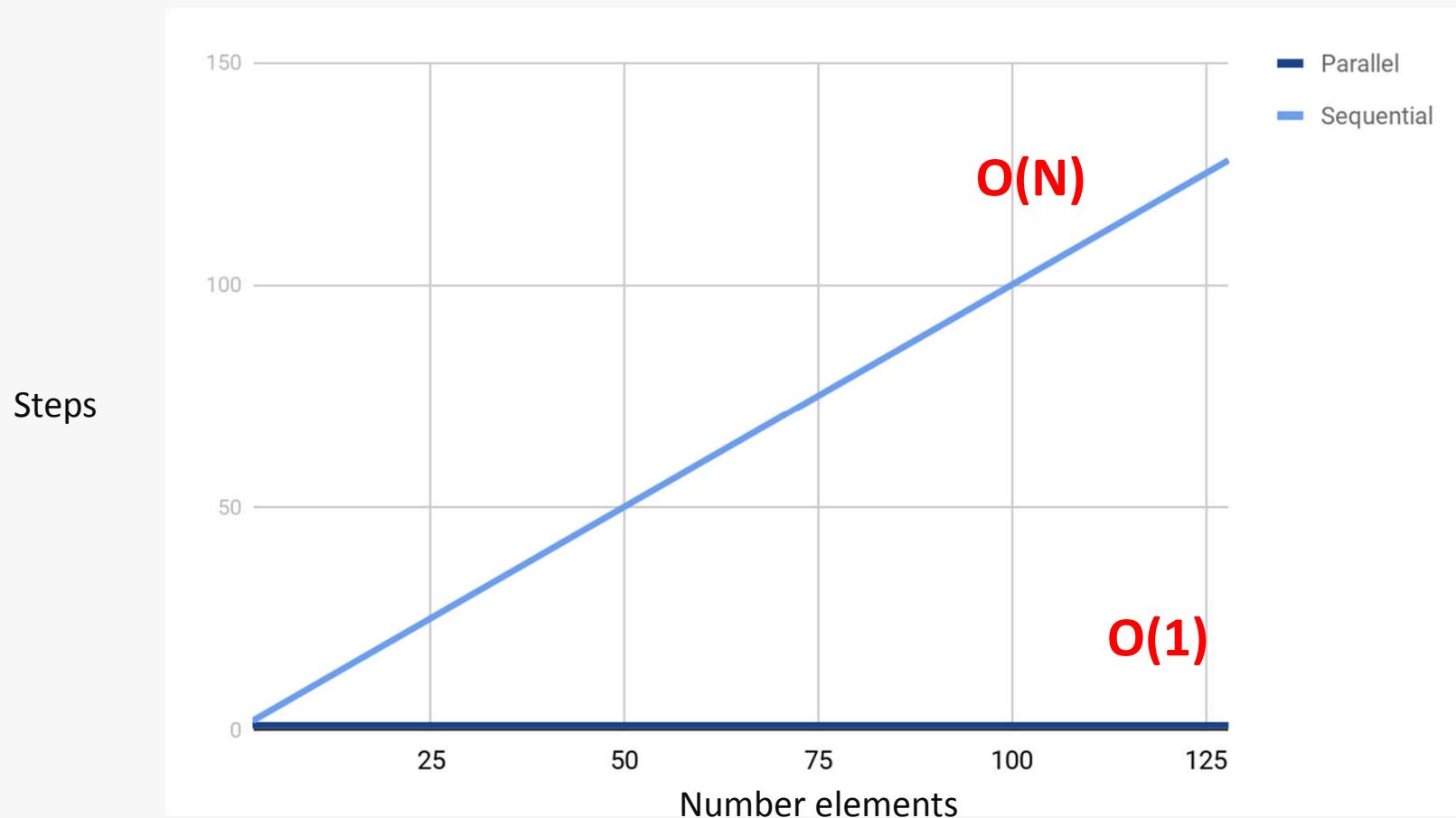
4 elements | 4 Operations| **4 workers** | 1 steps | 4 operations / step

Now let's scale this up...

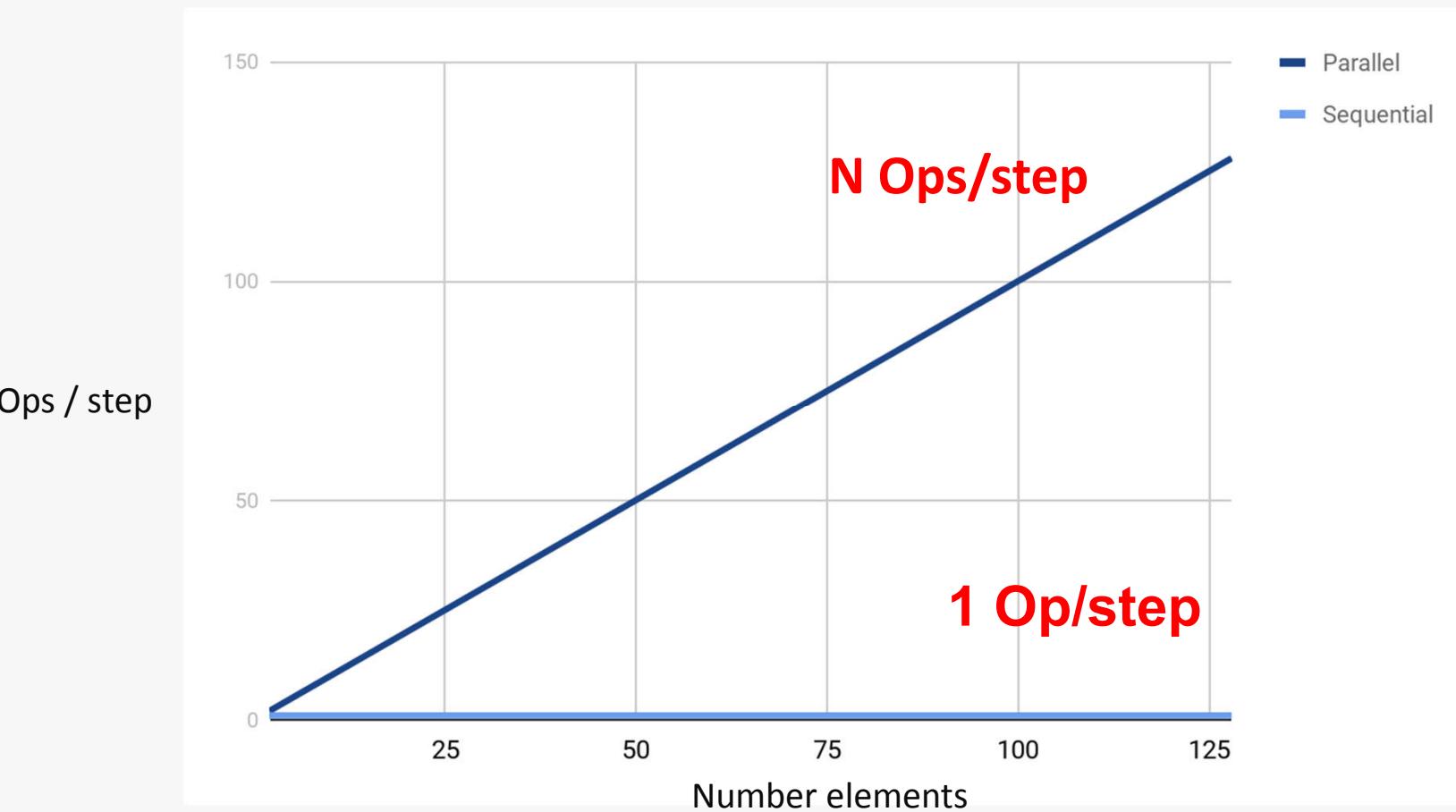


8 elements | 8 Operations | 8 workers | 1 step | 8 operations / step

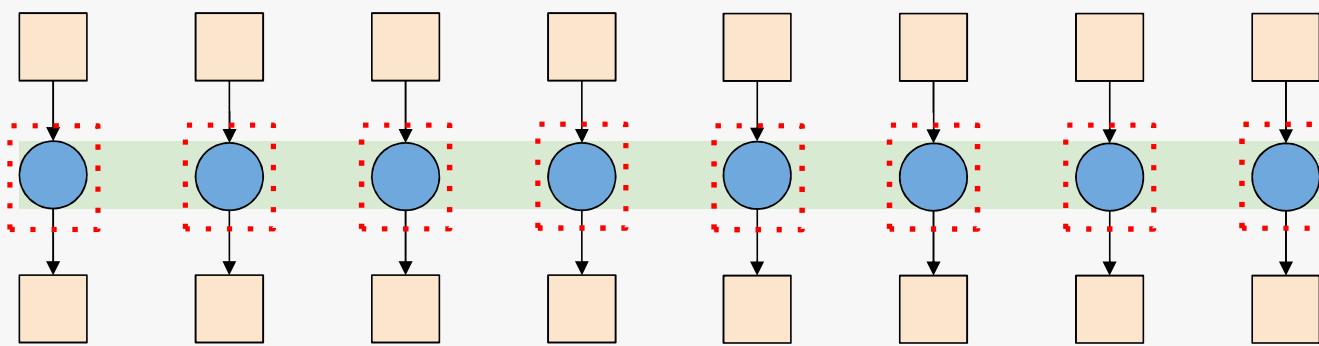
Step complexity of transform



Theoretical operations per step

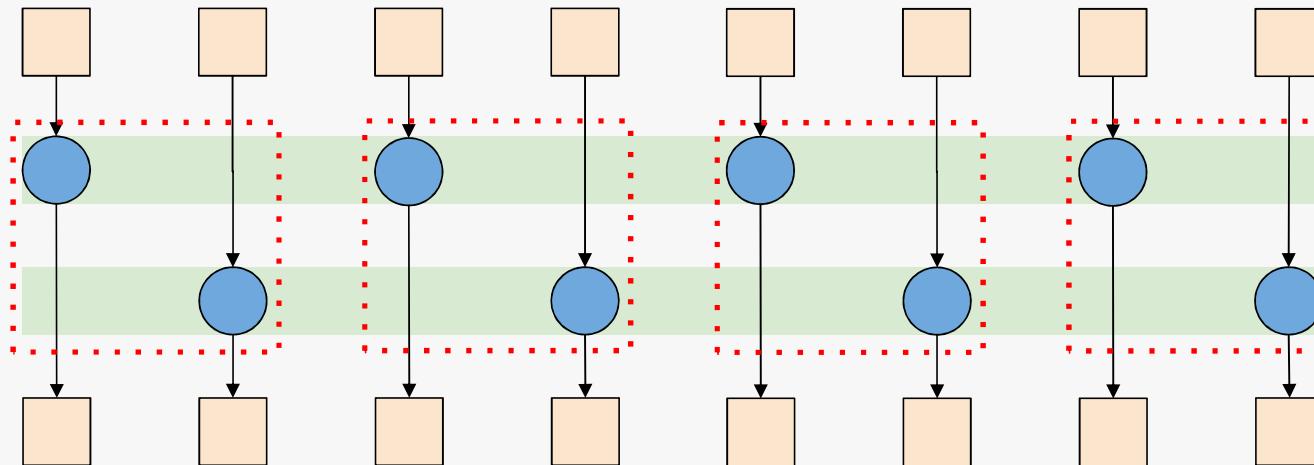


What happens if you only have 4 workers?



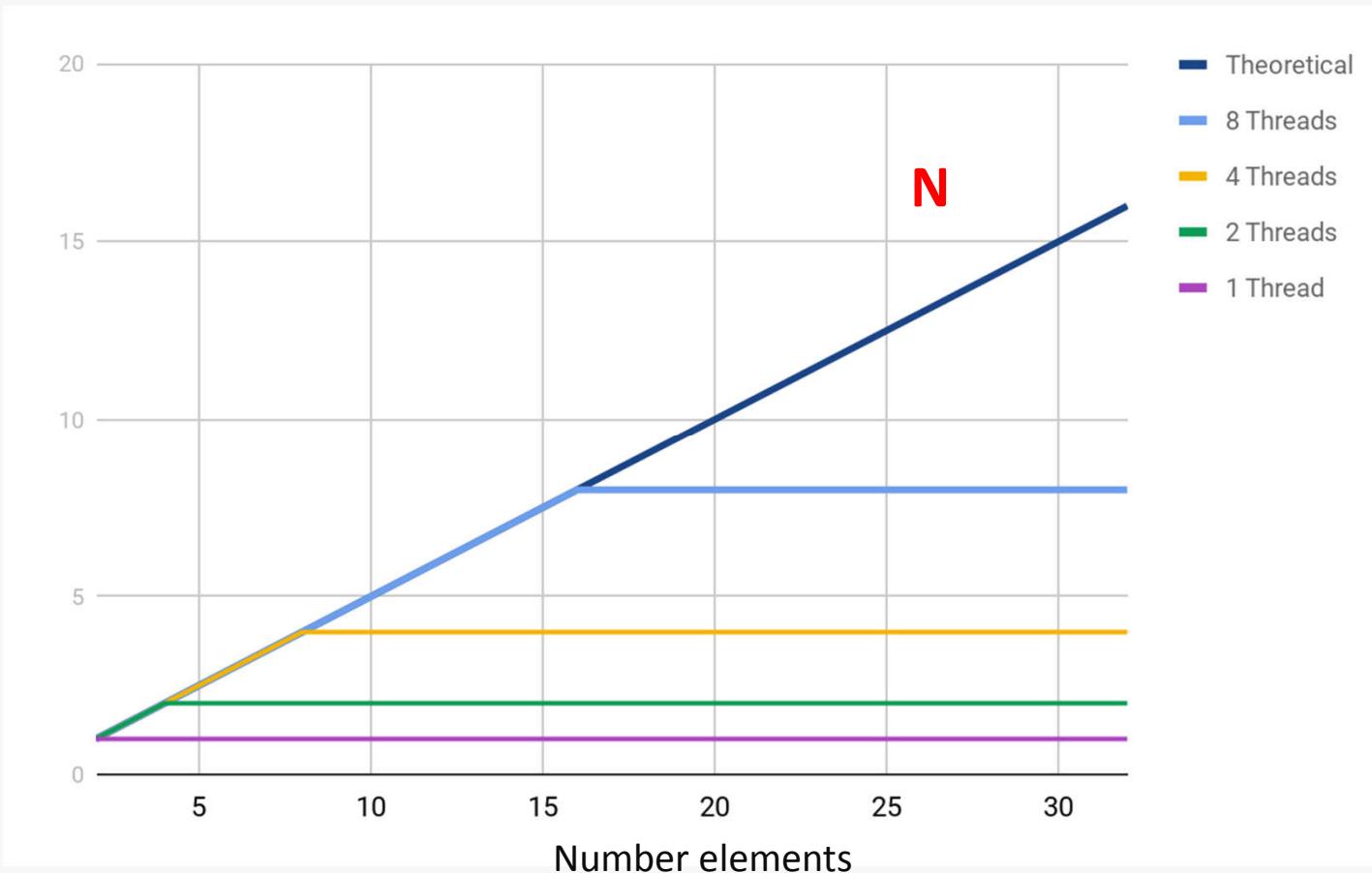
8 elements | 8 Operations | 8 workers | 1 step | 8 operations / step

You have to batch work together



8 elements | 8 Operations | **4 workers** | **2 steps** | **4 operations / step**

Actual operations per step



Maximizing throughput

The theoretical operations / step is always limited by the available workers

Maximising the actual operations / step will provide optimal throughout

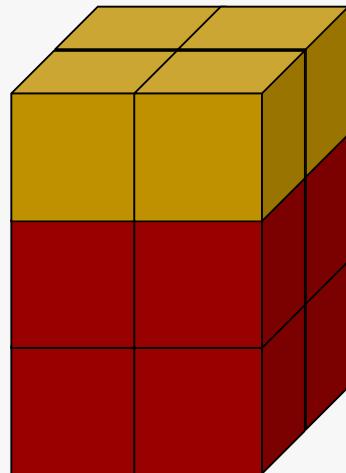
You will most often have a much larger number of operations to perform than available workers

How you perform this batching may differ depending on the architecture you are executing on

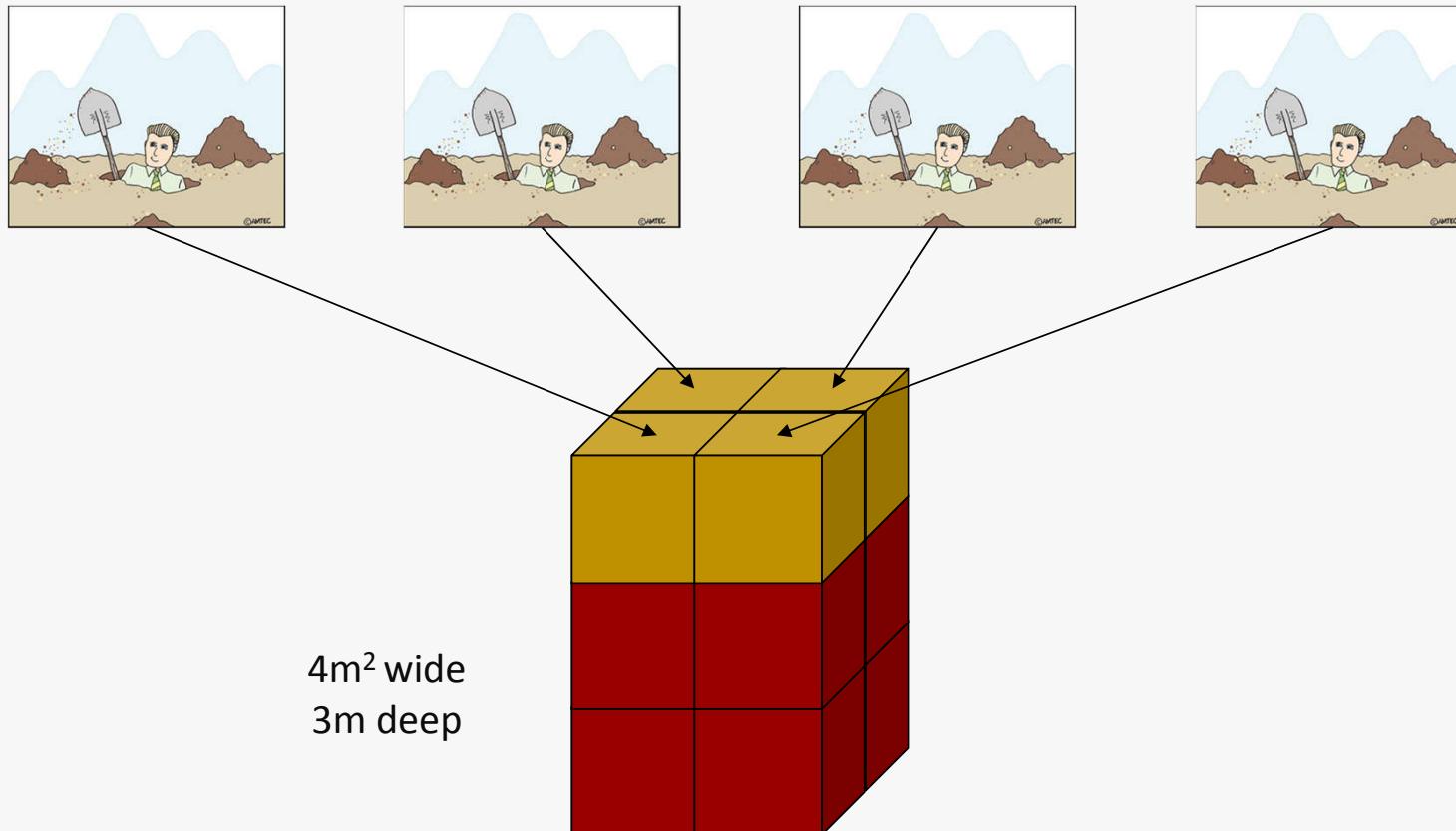
Say you want to dig a hole 3m deep



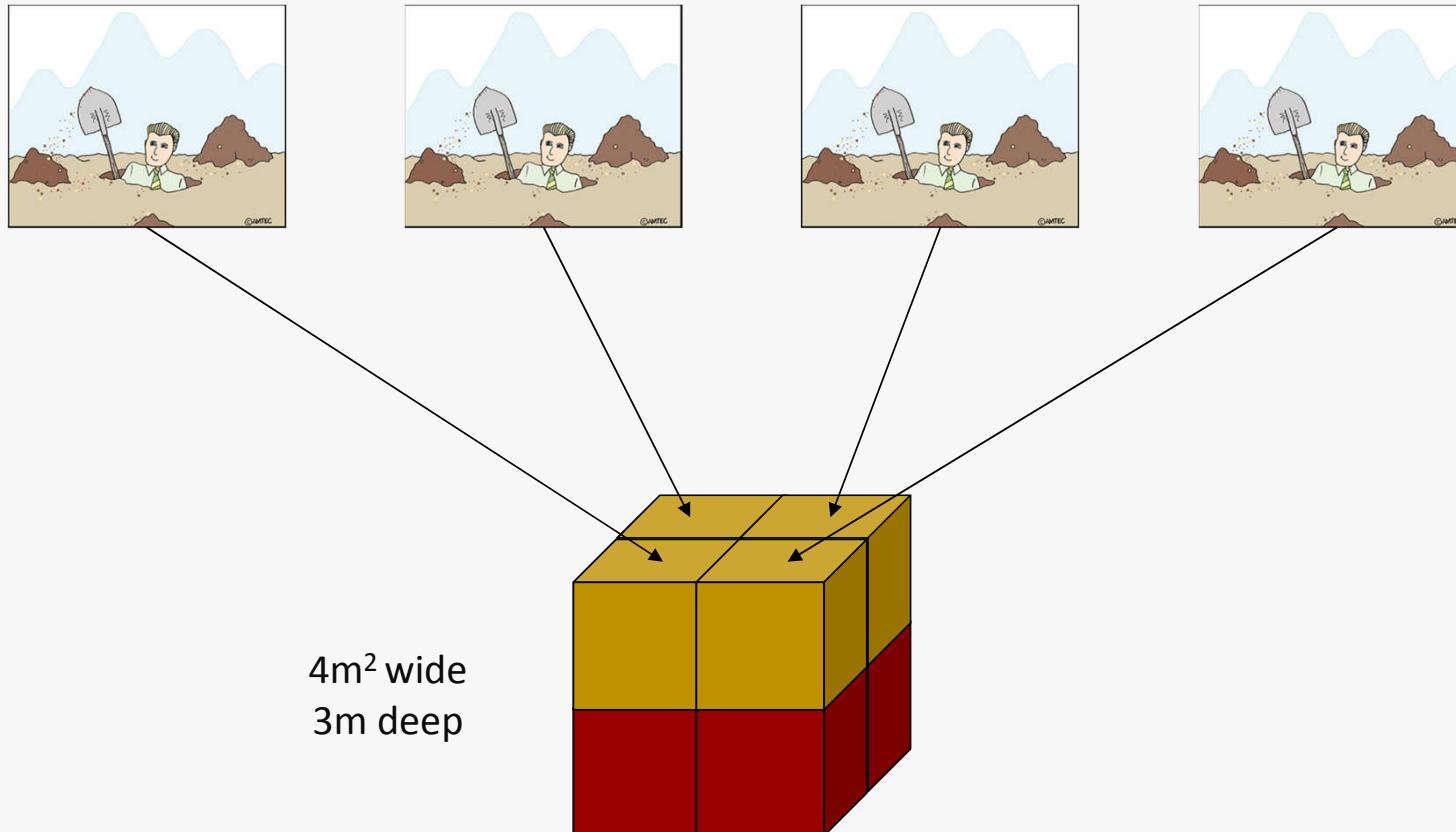
4m² wide
3m deep



All four diggers have work to do



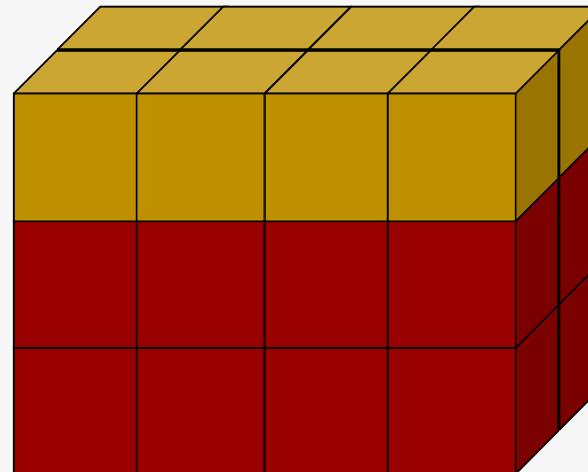
All four diggers have work to do



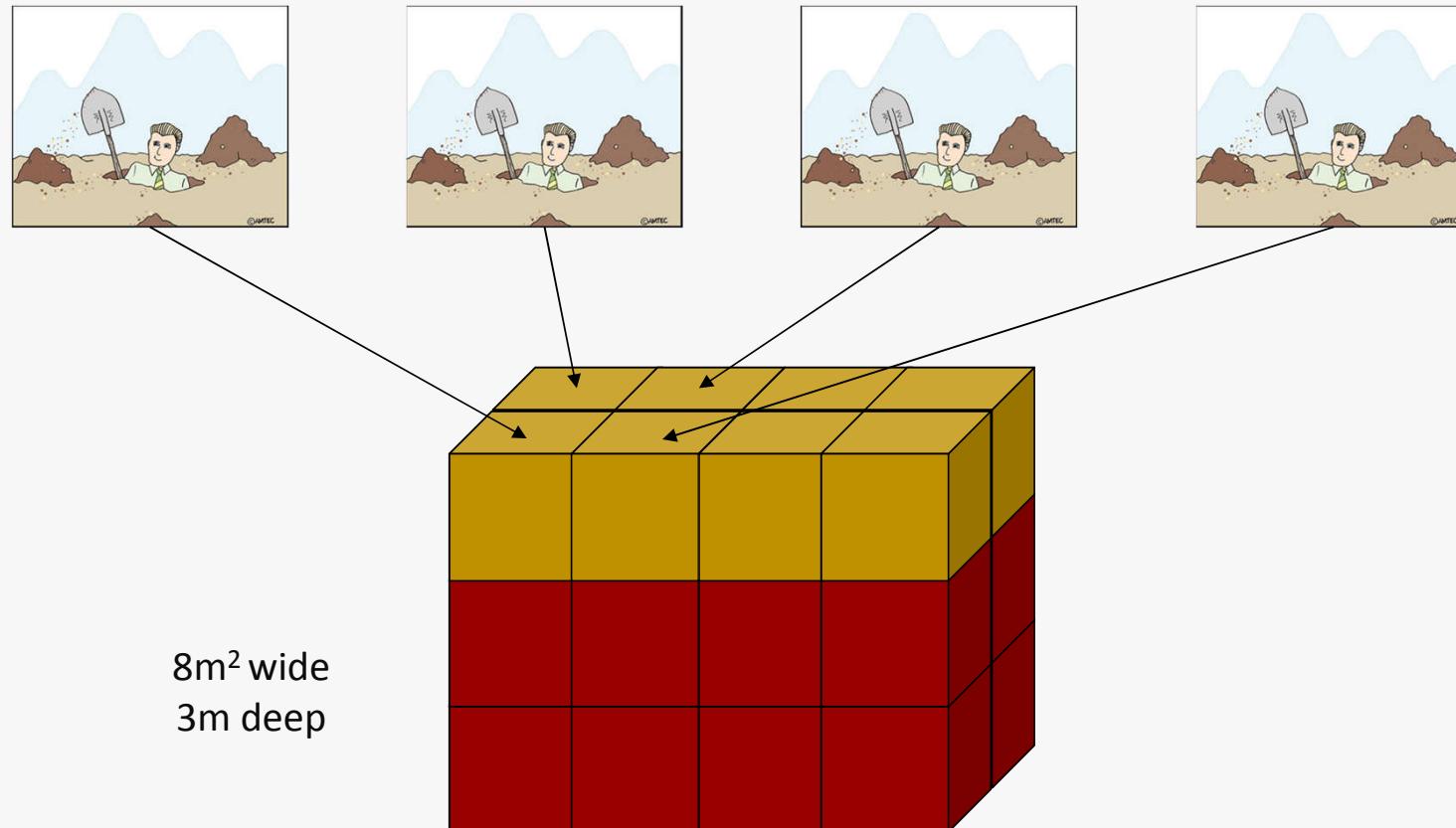
Now say the hole is 8m^2 wide



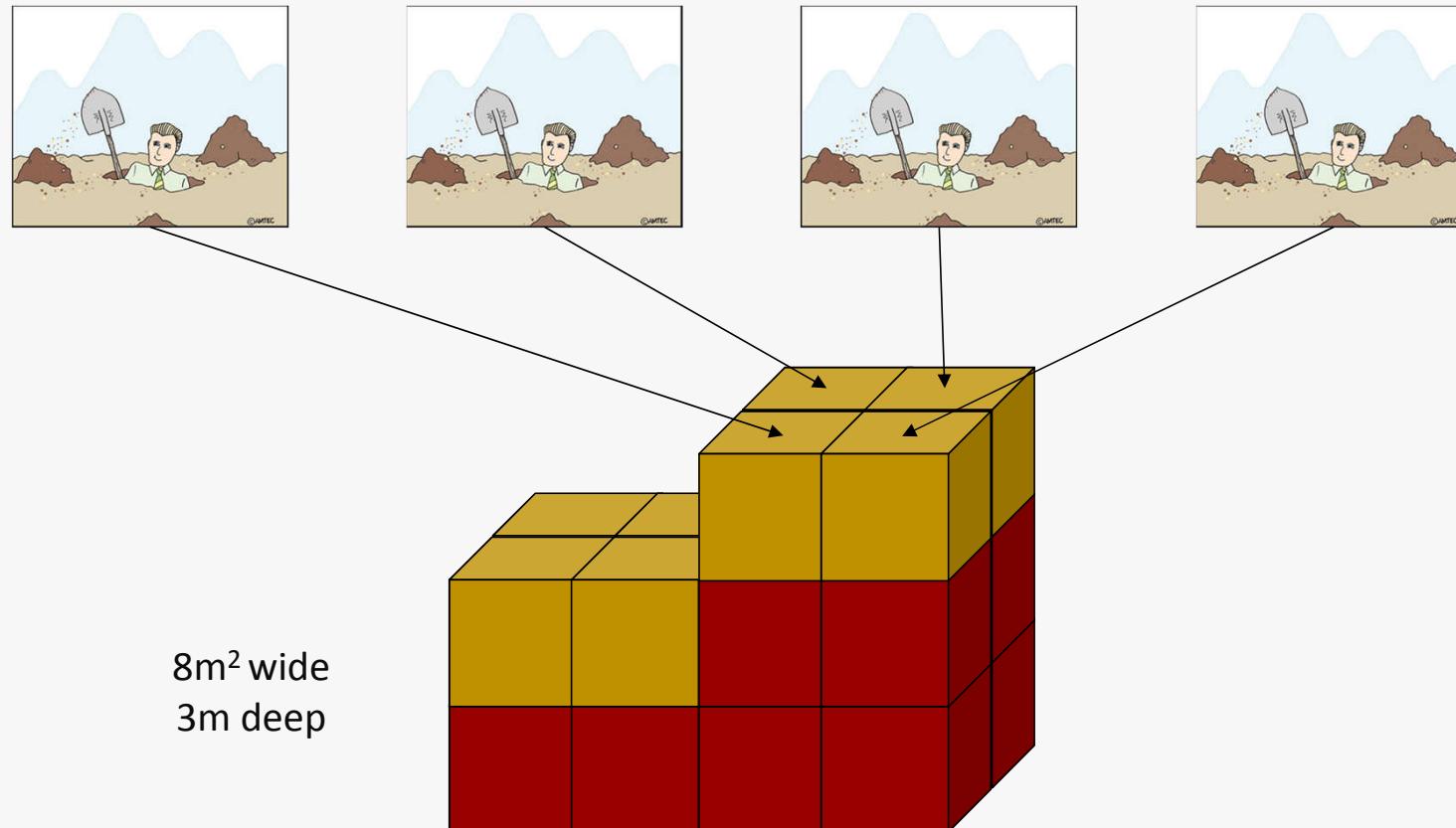
8m^2 wide
 3m deep



Again can share the work



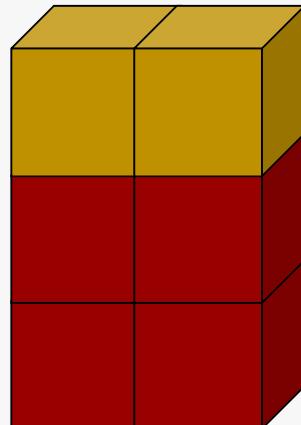
Again can share the work



Now say the hole 2m² wide



2m² wide
3m deep



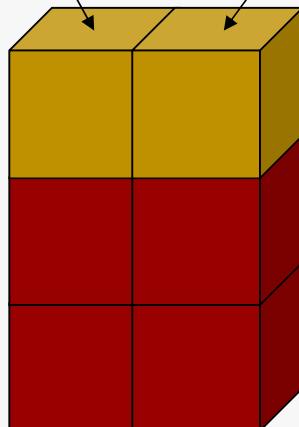
Now you have diggers with no work to do



on break

on break

2m² wide
3m deep



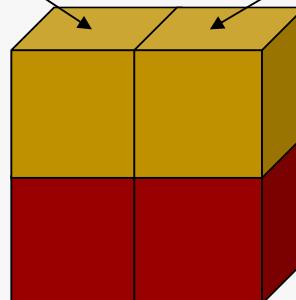
Now you have diggers with no work to do



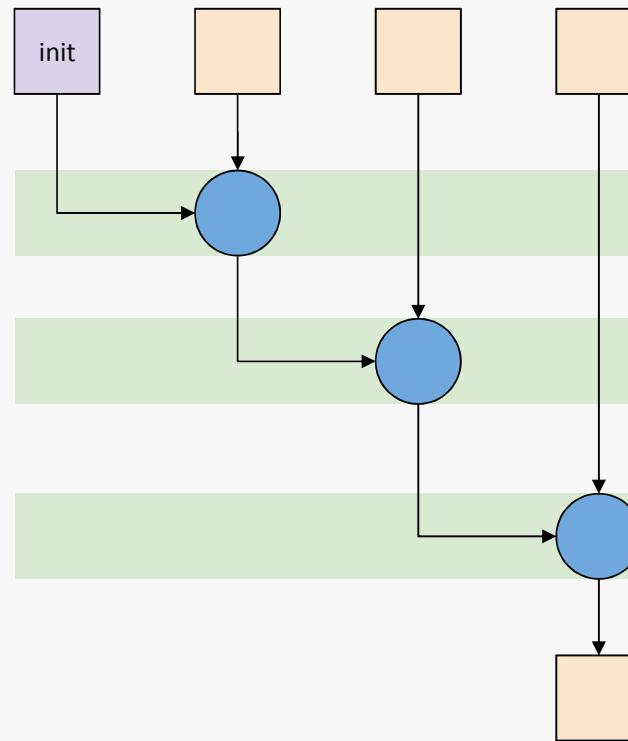
on break

on break

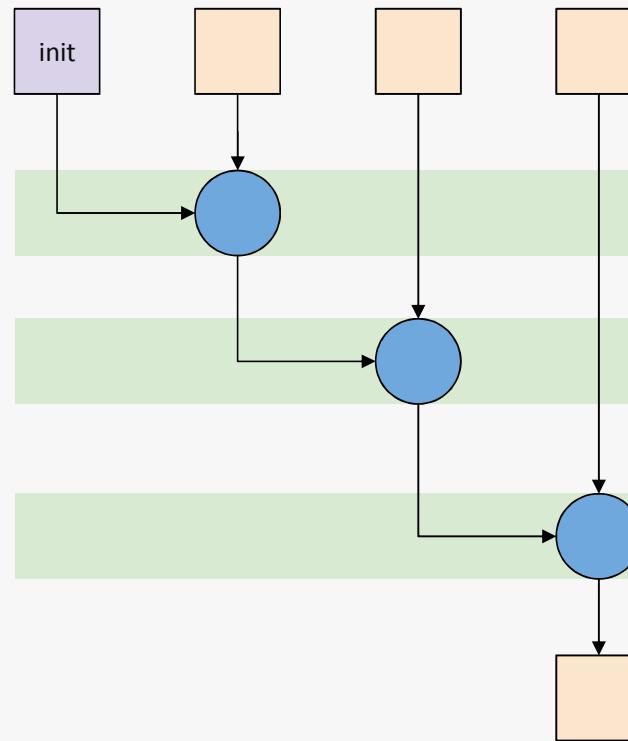
2m² wide
3m deep



This applies to a reduction algorithm

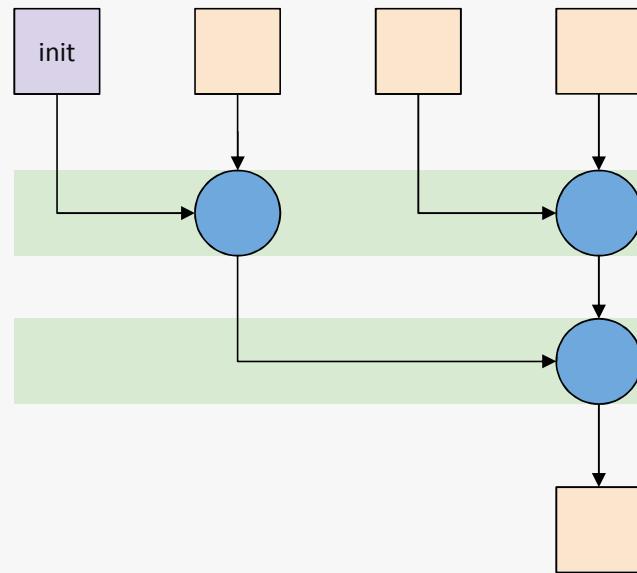


Let's look at a serial reduction...



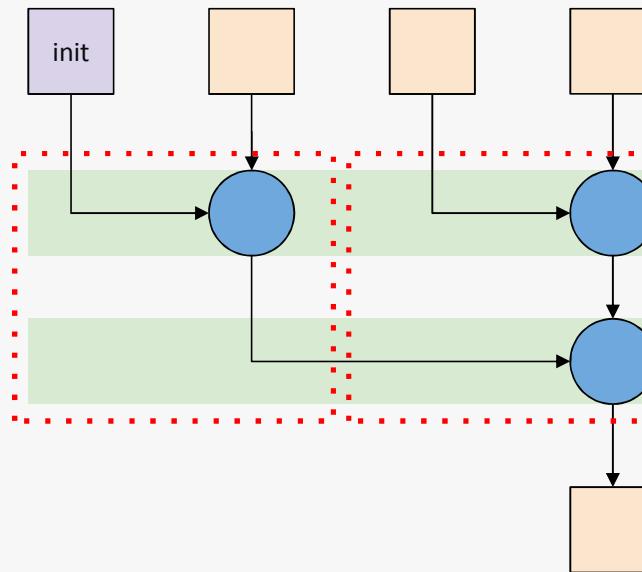
3 elements | 3 Operations| 3 steps | 1 operations / step

Now let's look at a parallel reduction...



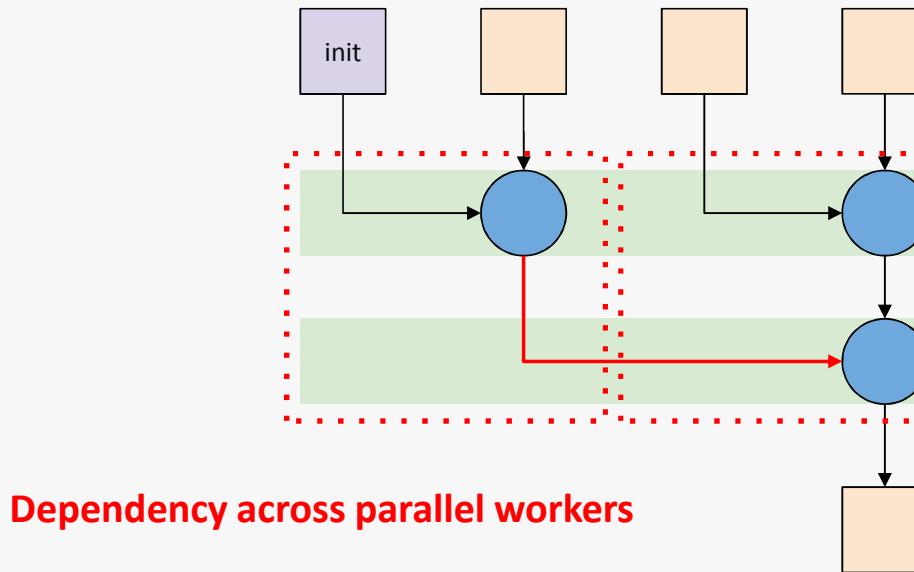
3 elements | 3 Operations | **2 steps** | **1.5 operations / step**

Let's try to distribute this work



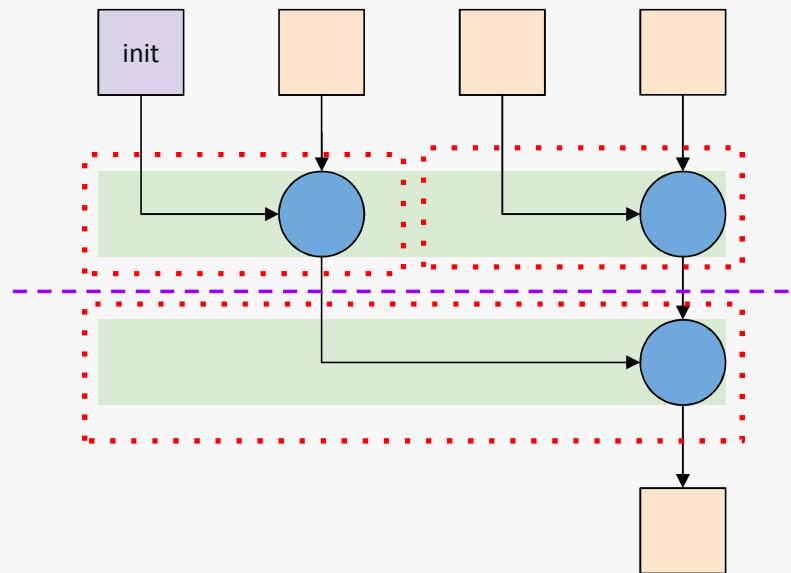
3 elements | 3 Operations| 2 workers | 2 steps | 1.5 operations / step

This creates a dependency between workers



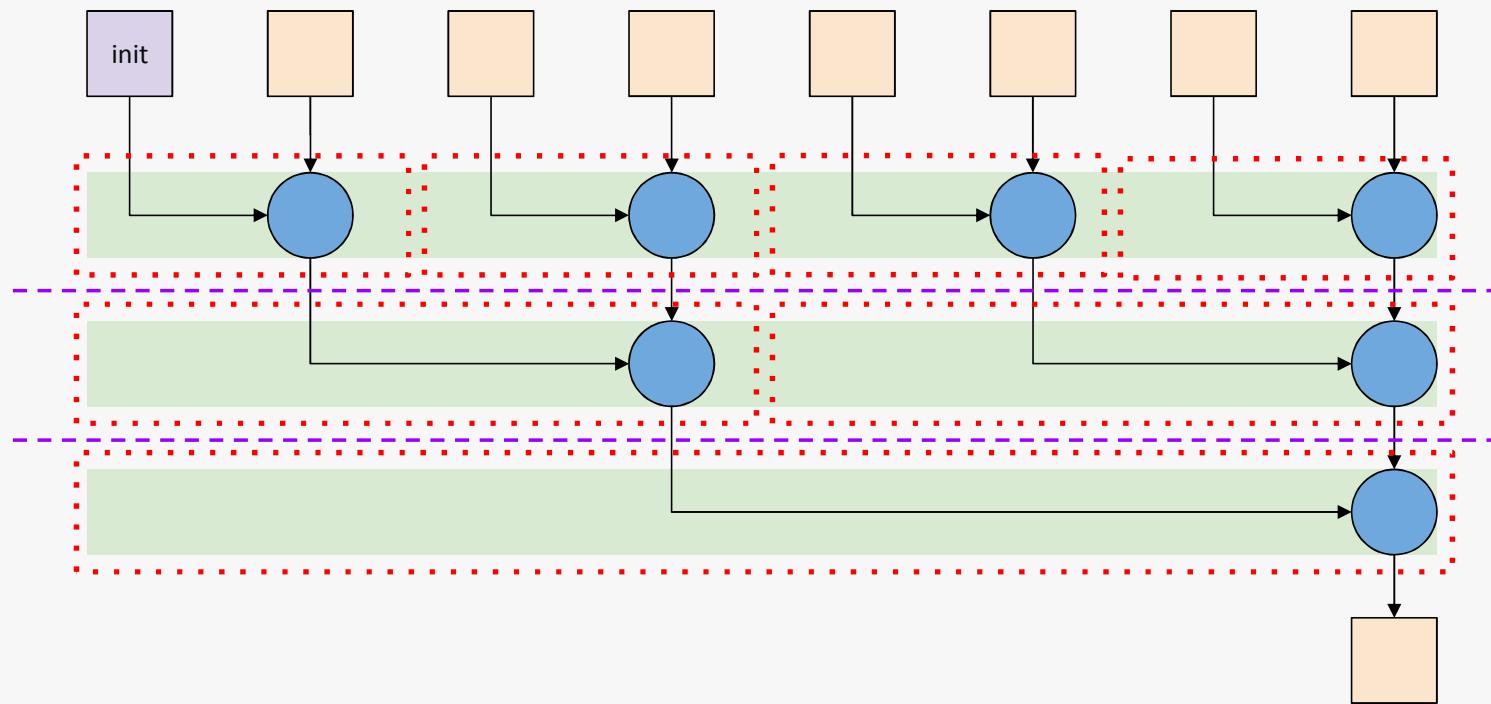
3 elements | 3 Operations | 2 workers | 2 steps | 1.5 operations / step

This creates a dependency between workers



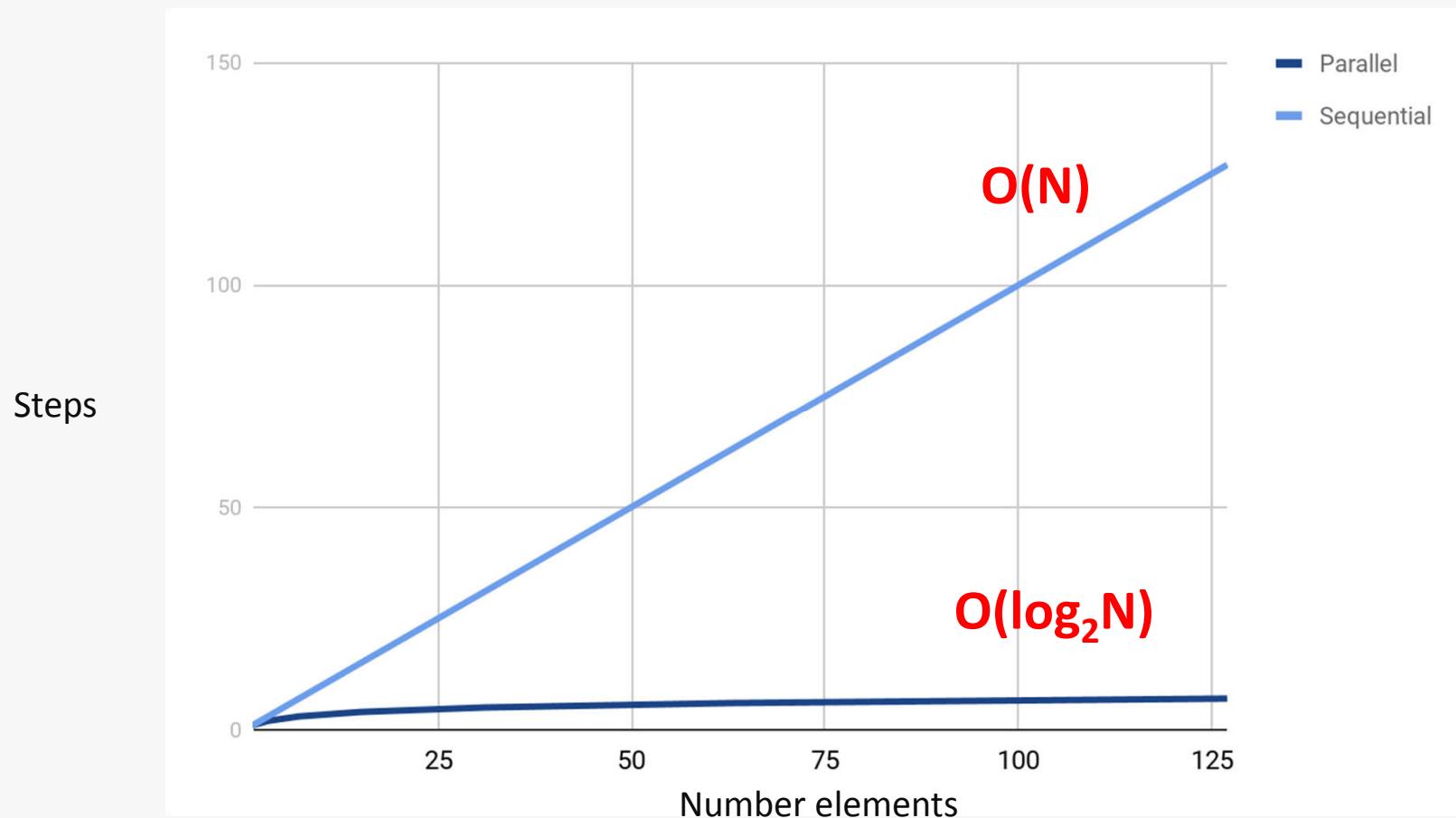
3 elements | 3 Operations| 2 workers | 2 steps | 1.5 operations / step

Now let's scale this up for 4 workers

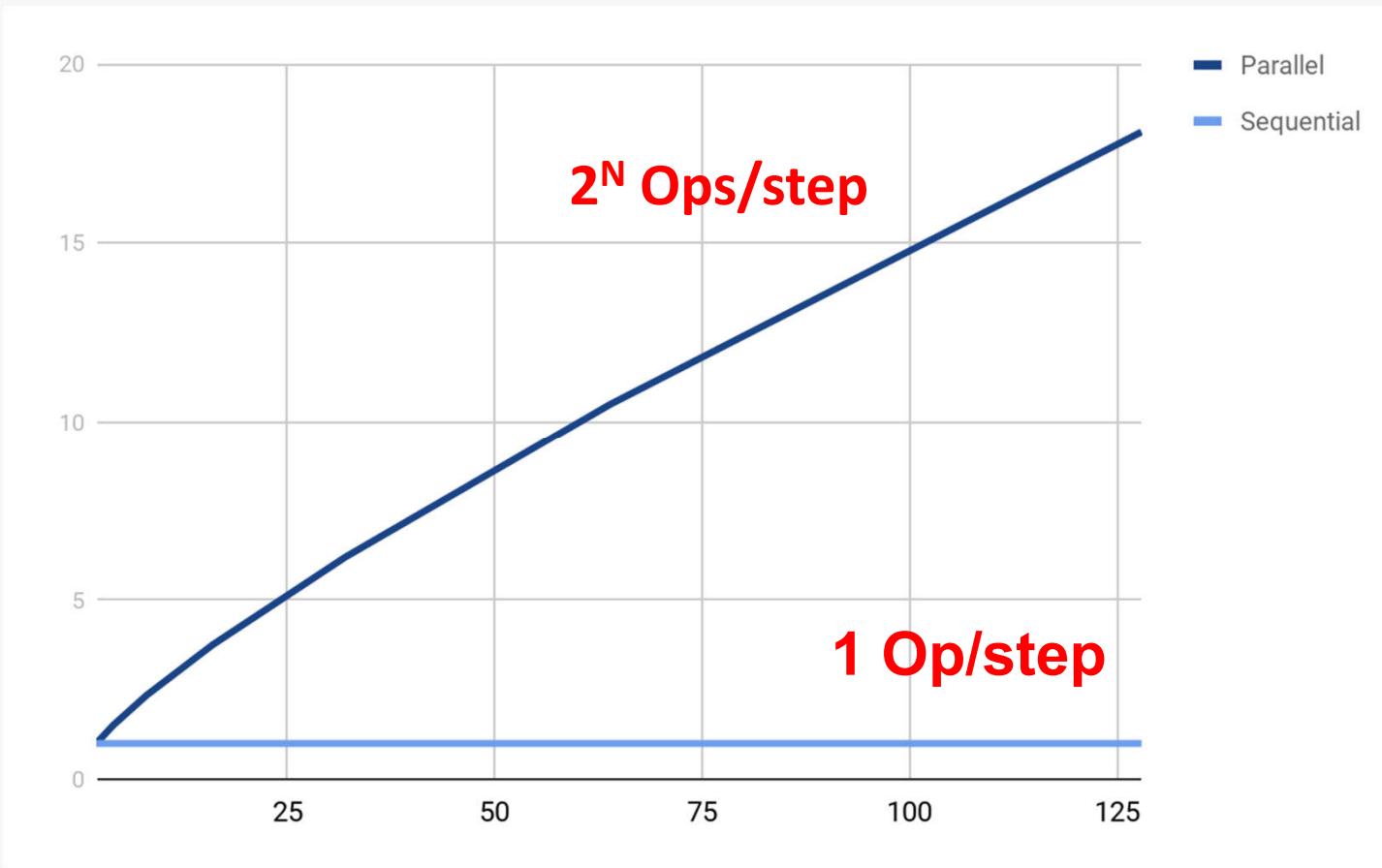


7 elements | 7 Operations| 4 workers | 3 steps | 2.3 operations / step

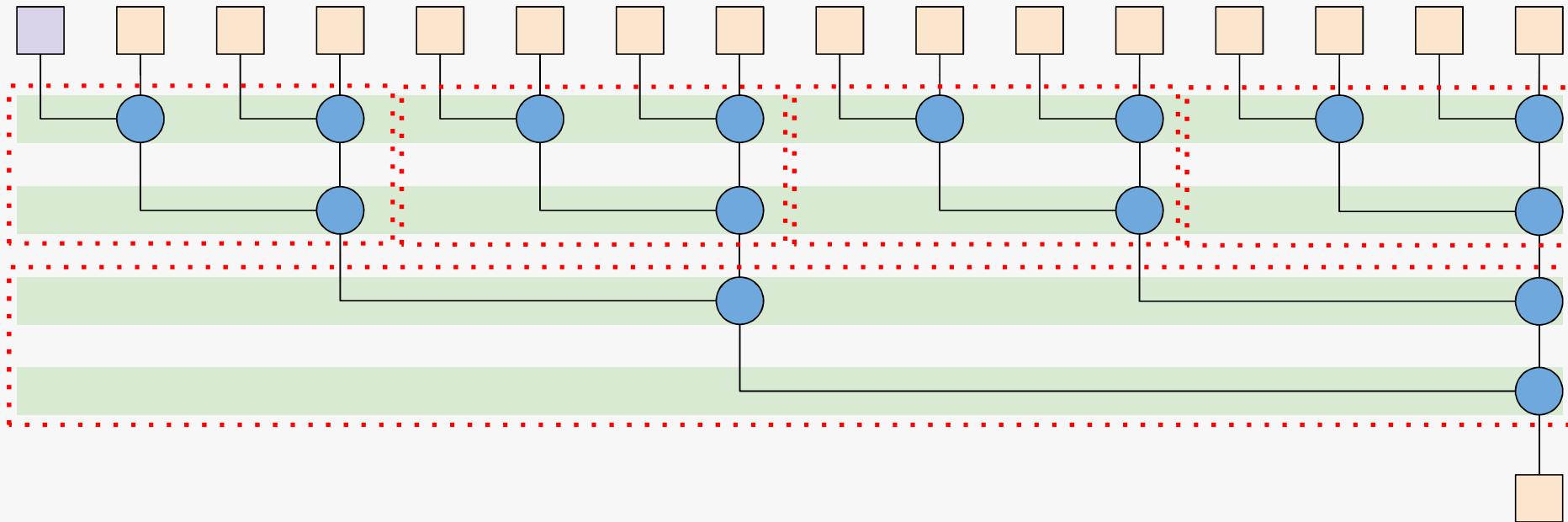
Step complexity of reduce



Theoretical operations per step

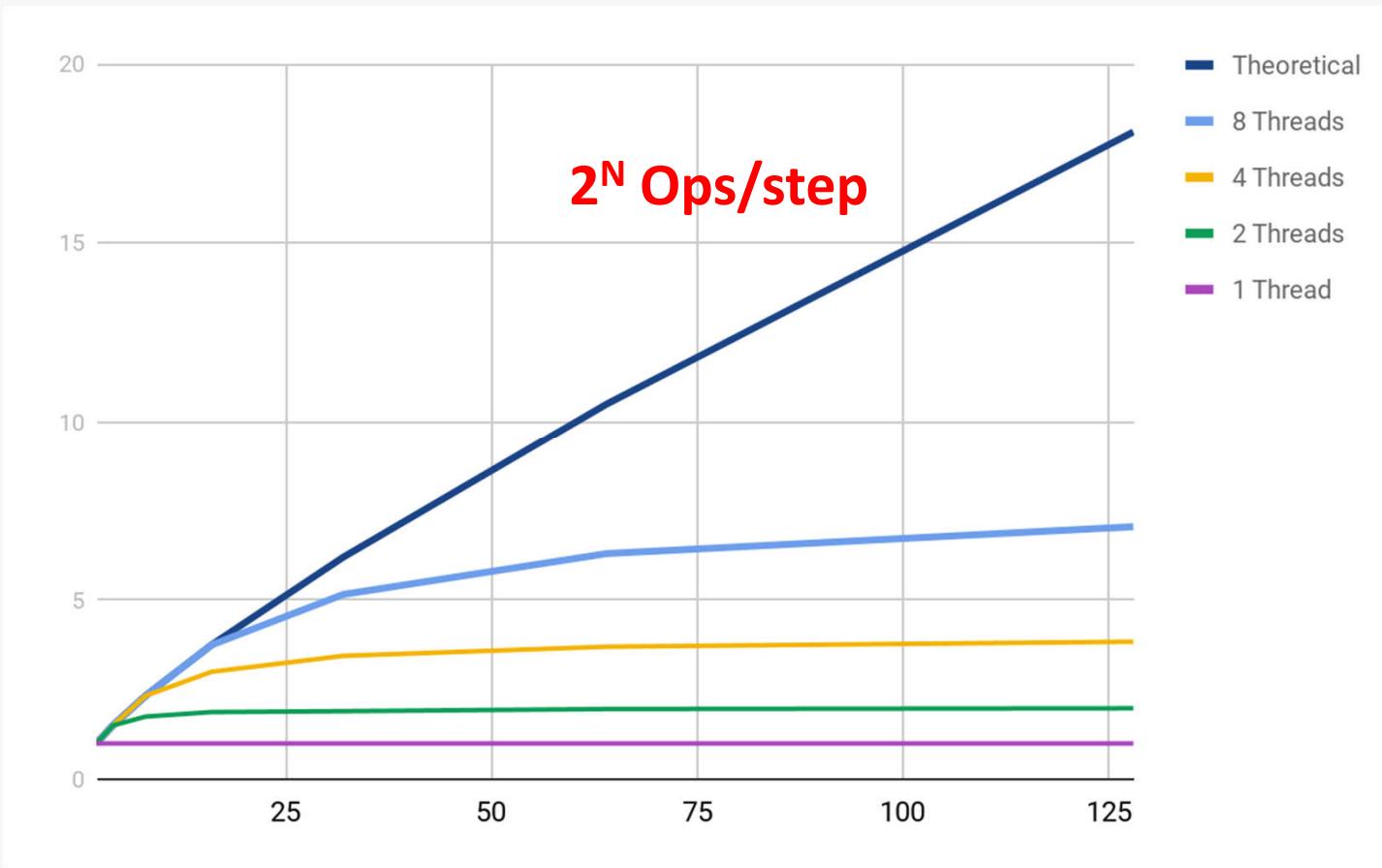


Now let's scale this up for 4 workers



15 elements | 15 operations | 4 workers | 4 steps | 3.8 operations / step

Actual operations per step



Handling dependencies

You should structure your algorithm to distribute dependencies

You should avoid dependencies across workers in the same step

When you have dependencies between operations this creates dependencies between workers

These dependencies often have different implications depending on the architecture you are executing on

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

Key takeaways

Designing your algorithm to maximise the operations per step will improve throughput and utilisation of the hardware

Designing your algorithm to distribute dependencies between operations will reduce increase operations per step and improve throughput

How you batch work together on workers and how you handle dependencies will vary depending on the architecture you are executing on



Questions?



Chapter 12: GPU Optimization (part 1)

Gordon Brown

CppCon 2019 – Sep 2019

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

There are different levels of optimizations you can apply

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

There are different levels of optimizations you can apply

- Choosing the right algorithm
 - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
 - *Such as coalescing global memory access or using local memory*

- Architecture specific optimisations
 - *Optimising for register usage or avoiding bank conflicts*

- Micro-optimisations
 - *Such as floating point dnorm hacks*

This class will focus on these two

Choosing the right algorithm

What to parallelise on a GPU

- Find hotspots in your code base
 - *Looks for areas of your codebase that are bottlenecks that are taking up a lot of execution time*
- Look for areas of your codebase that are a good fit for GPU parallelism
 - *Or areas of your codebase that could be adapted for parallelism*

Follow good optimisation practice

- Follow an adaptive optimisation approach such as APOD
 - *Analyse your code base to find opportunities for parallelism*
 - *Parallelise a piece of code on the GPU*
 - *Optimise the algorithm to be more efficient on the GPU*
 - *Deploy the codebase to evaluate the performance with different inputs*
- Avoid over-optimisation
 - *You may reach a point where optimisations provide diminishing returns*
 - *It may be more efficient to look for another bottleneck in your codebase*

What to look for in an algorithm

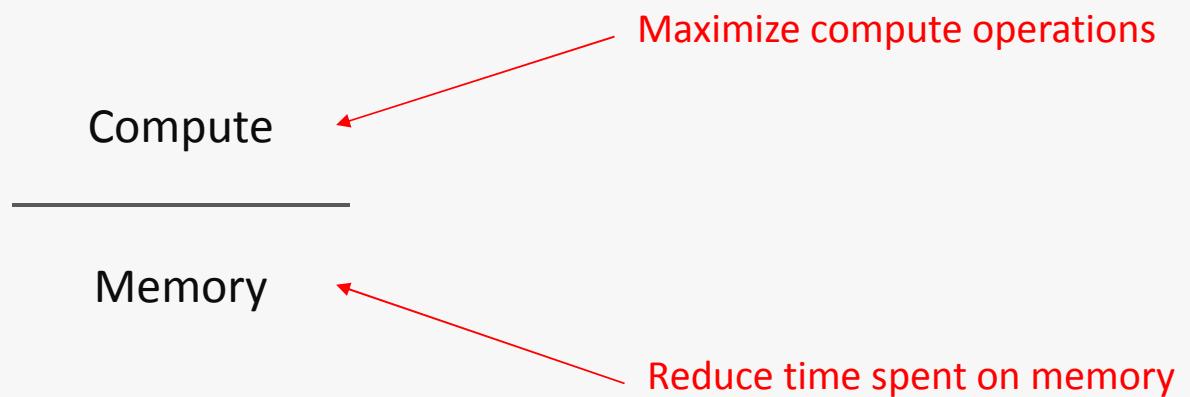
- Naturally data parallel
 - *Performing the same operation on multiple items in the computation*
- Large problem
 - *Enough work to utilise the GPU's processing elements*
- Independent progress
 - *Little or no dependencies between items in the computation*
- Non-divergent control flow
 - *Little or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

Embarrassingly parallel algorithms

- Some problems are considered “embarrassingly parallel”
 - *The problem is naturally parallel*
 - *The problem has no communication between items in the computation*
- These kind of problems are perfect for the GPU

Basic GPU programming principles

Optimizing GPU programs means maximizing throughput



Optimizing GPU programs means maximizing throughput

Compute

Memory

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

Optimizing GPU programs means maximizing throughput

Compute

Memory

Most GPU problems
are memory bound

- Compute bound
 - *Problems that are bound by the computation time*

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

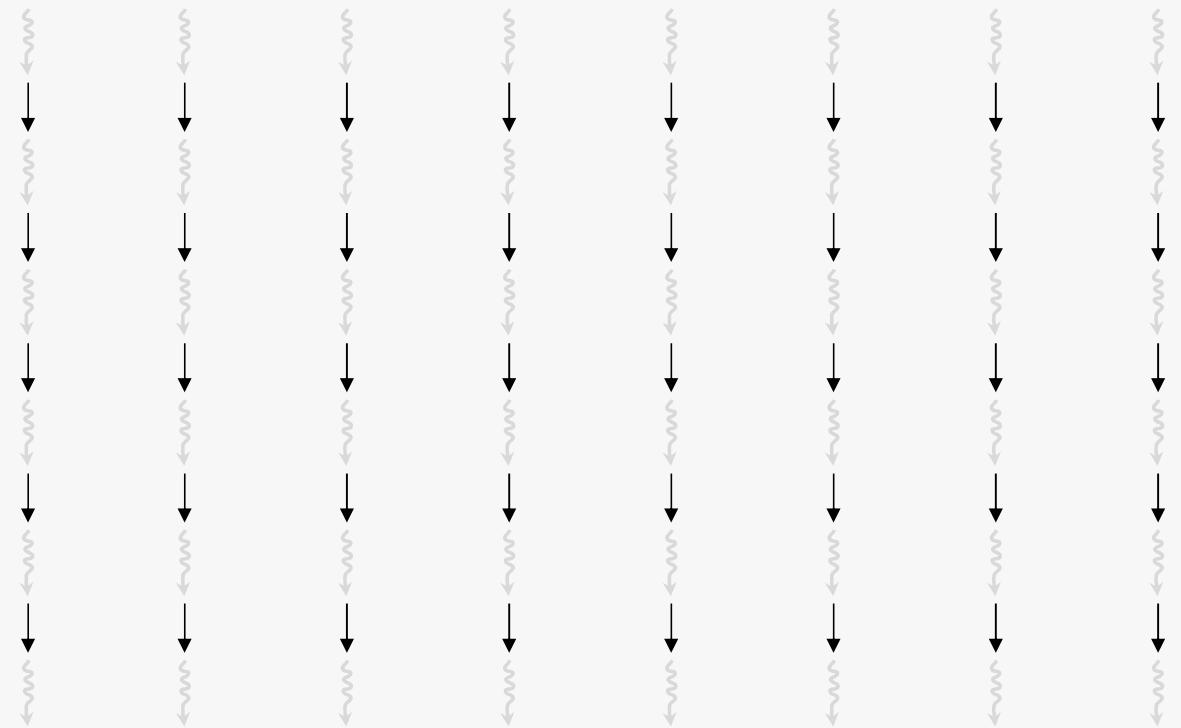
Optimizing GPU programs means maximizing throughput

- Maximise compute operations per cycle
 - *Make effective utilisation of the GPU's hardware*
- Reduce time spent on memory operations
 - *Reduce latency of memory access*

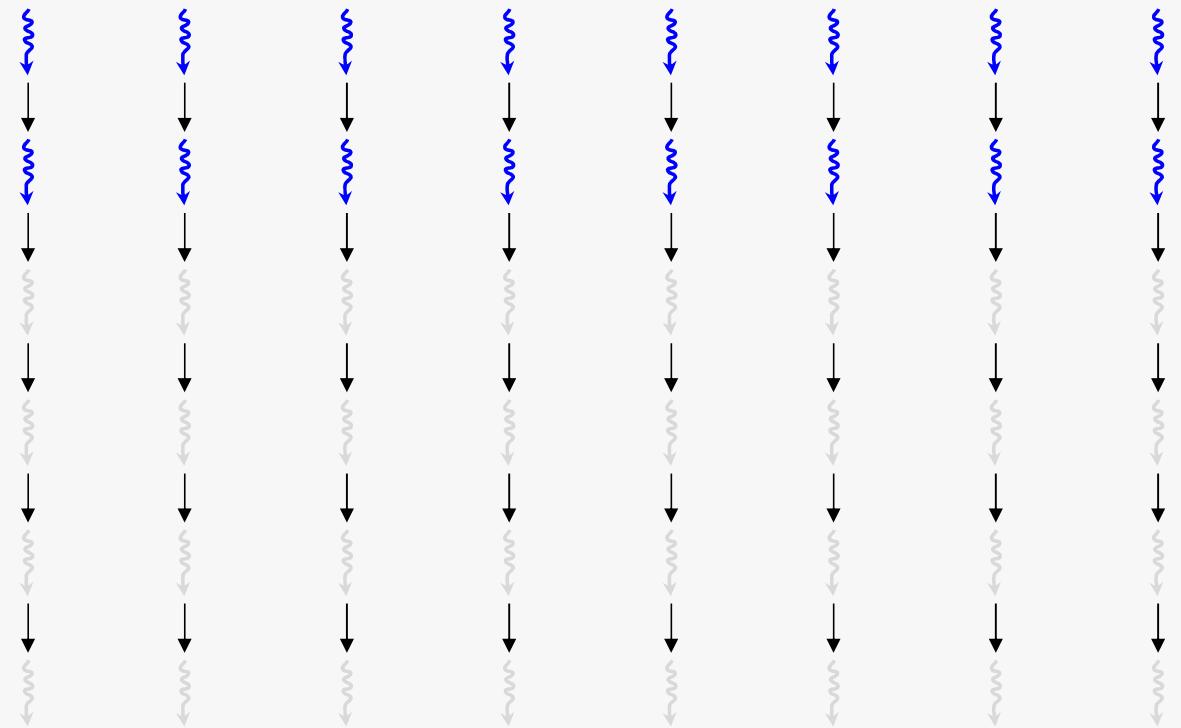
Avoid divergent control flow

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

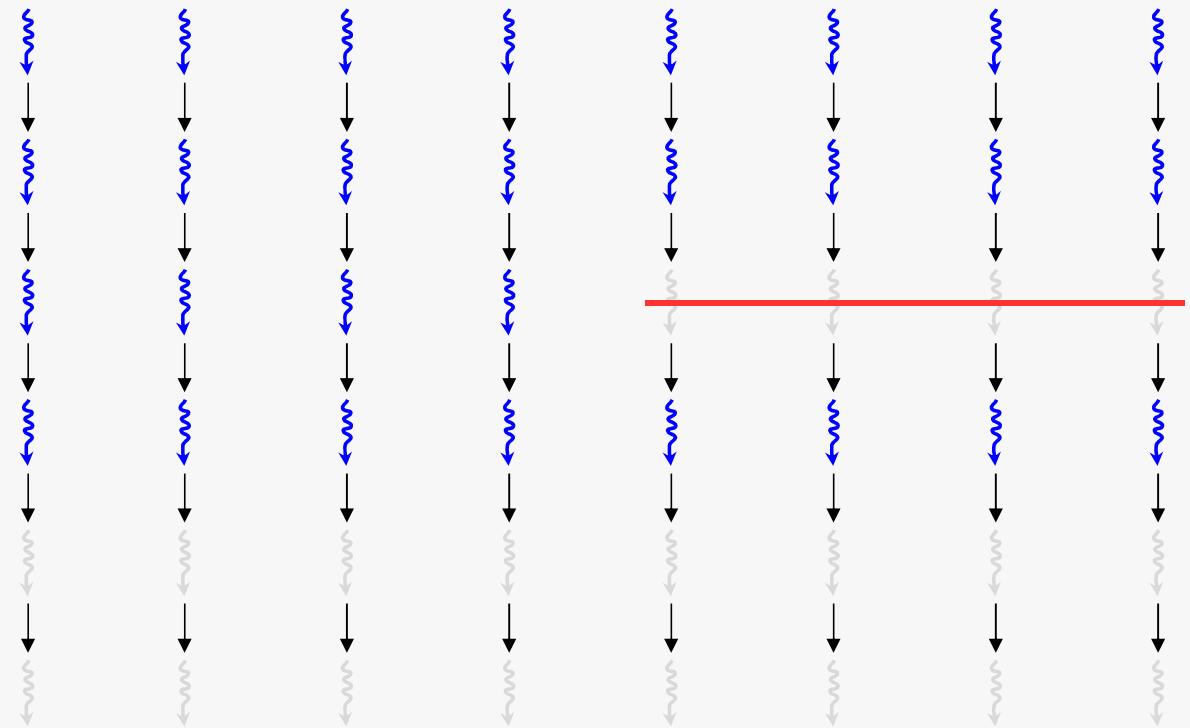
```
a[globalId] = 0;  
  
if (globalId < 4) {  
  
    a[globalId] = x();  
  
} else {  
  
    a[globalId] = y();  
  
}
```



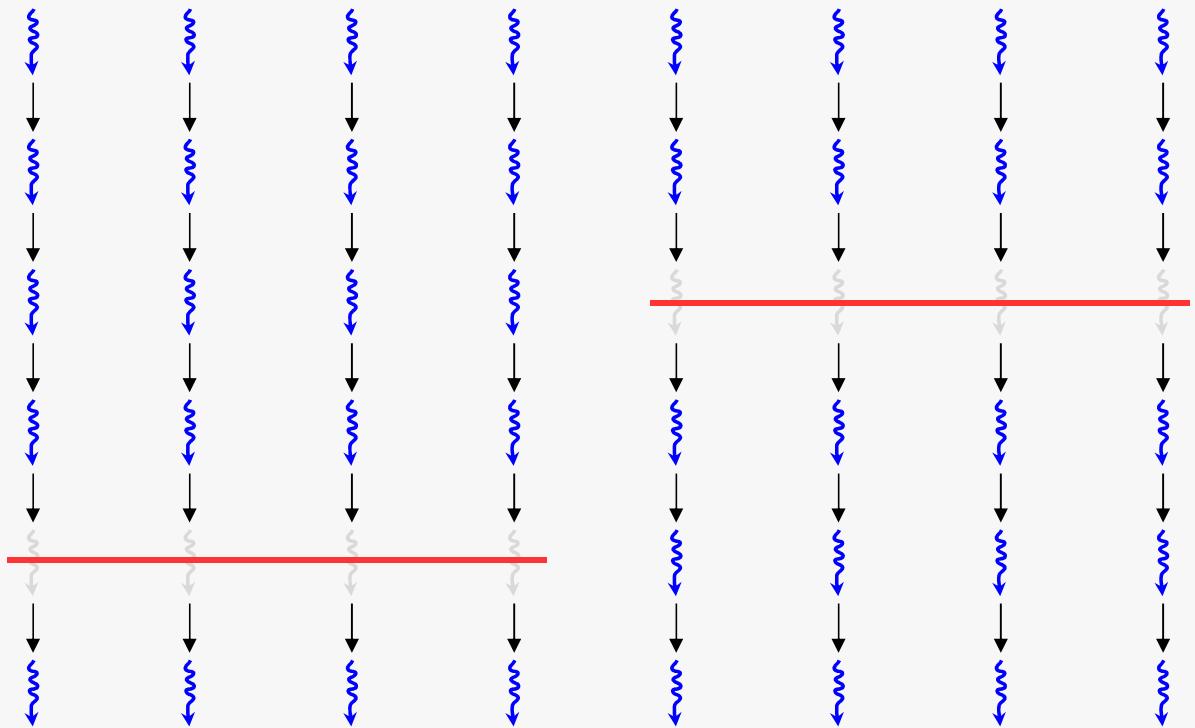
```
a[globalId] = 0;  
  
if (globalId < 4) {  
  
    a[globalId] = x();  
  
} else {  
  
    a[globalId] = y();  
  
}
```



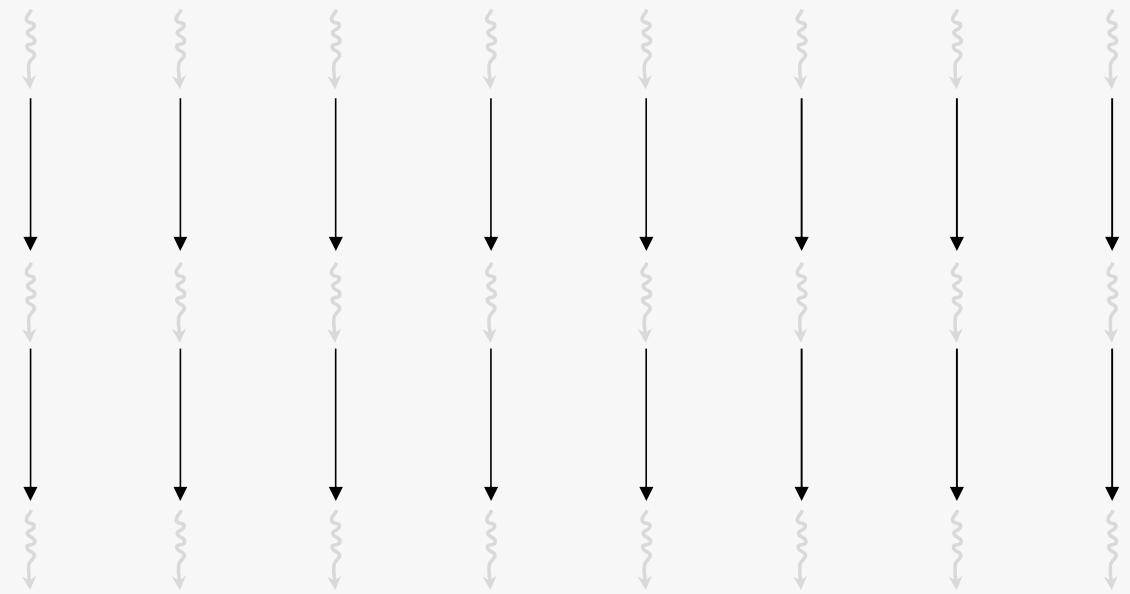
```
a[globalId] = 0;  
  
if (globalId < 4) {  
  
    a[globalId] = x();  
  
} else {  
  
    a[globalId] = y();  
  
}
```



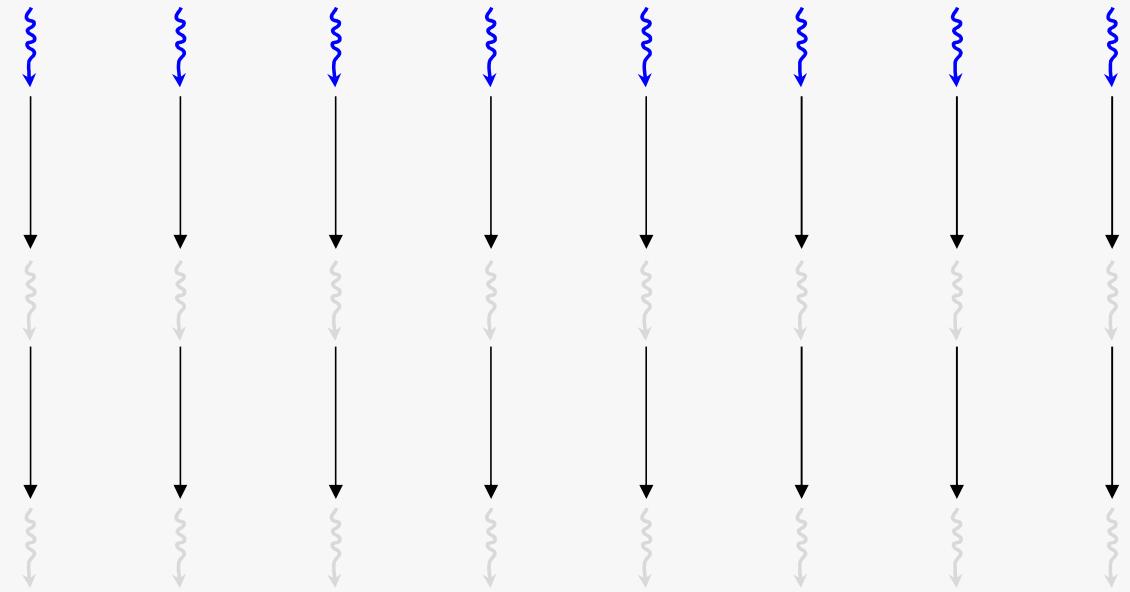
```
a[globalId] = 0;  
  
if (globalId < 4) {  
  
    a[globalId] = x();  
  
} else {  
  
    a[globalId] = y();  
  
}
```



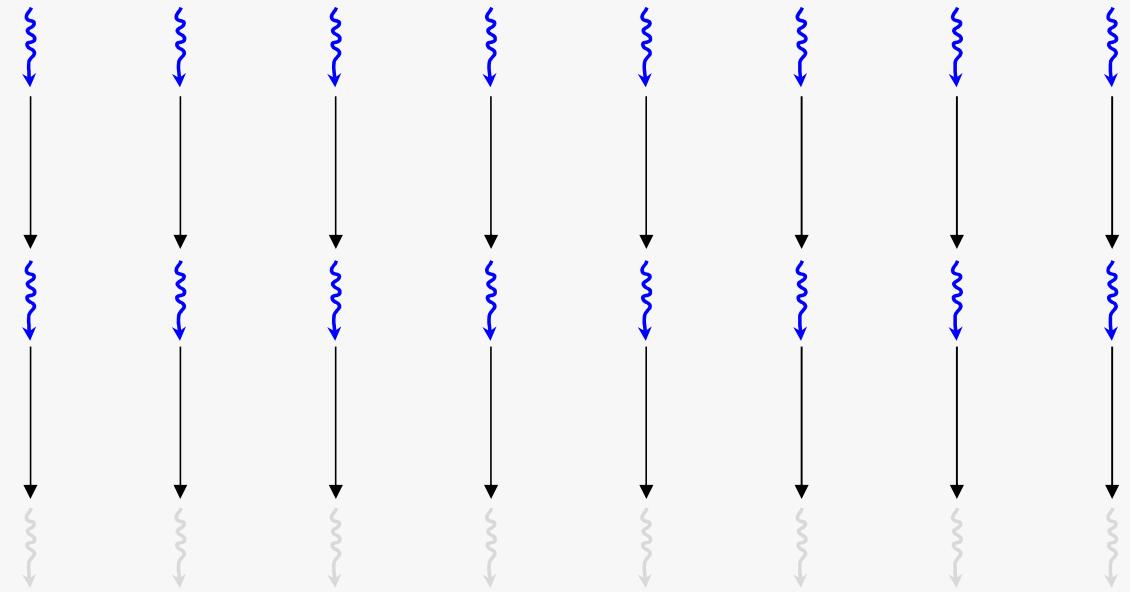
```
...  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
...
```



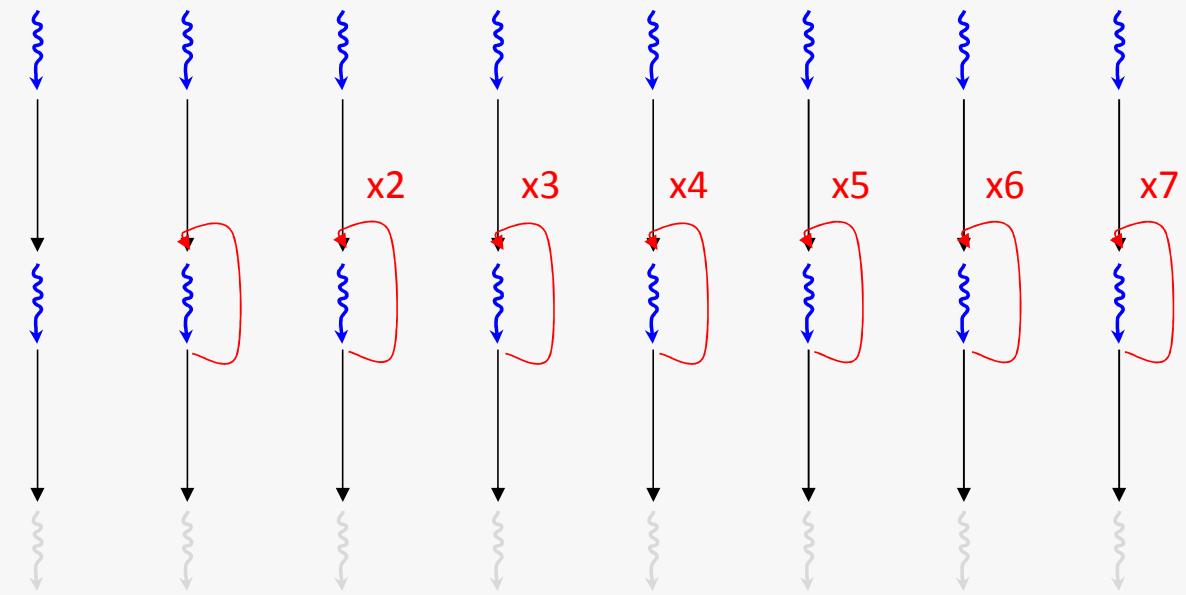
```
...  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
...
```



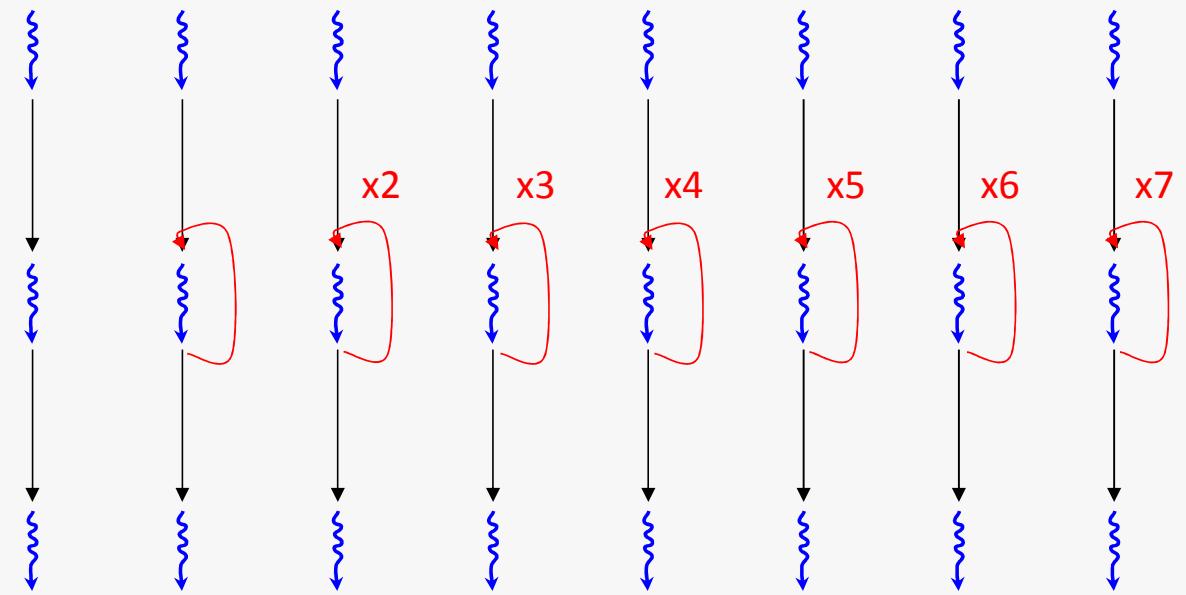
```
...  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
...
```



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



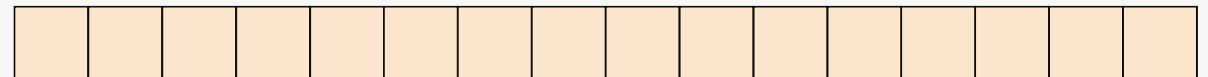
```
...  
for (int i = 0; i <  
    globalId; i++) {  
    do_something();  
}  
...
```



Coalesced global memory access

- Reading and writing from global memory is very expensive
 - *It often means copying across an off-chip bus*
- Reading and writing from global memory is done in strides
 - *This means accessing data that is physically close together in memory is more efficient*

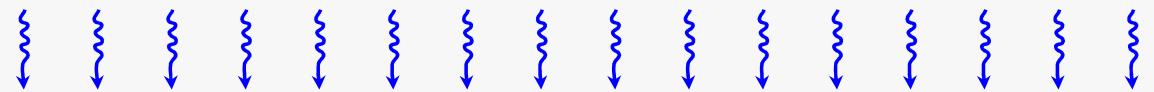
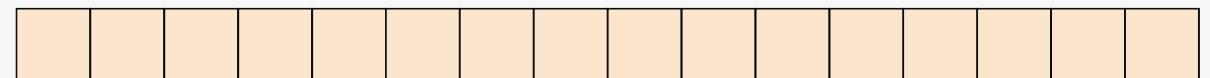
```
float data[size];
```



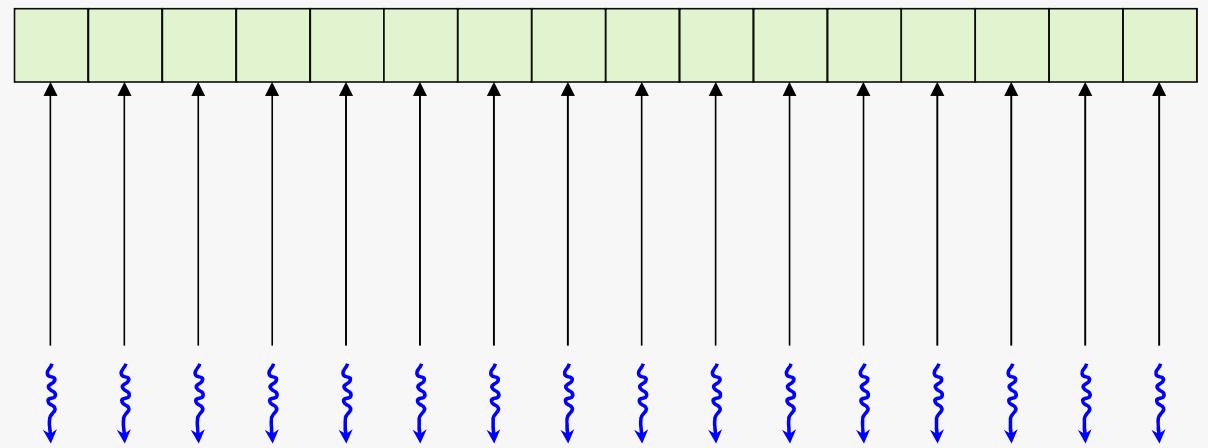
```
float data[size];
```

...

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



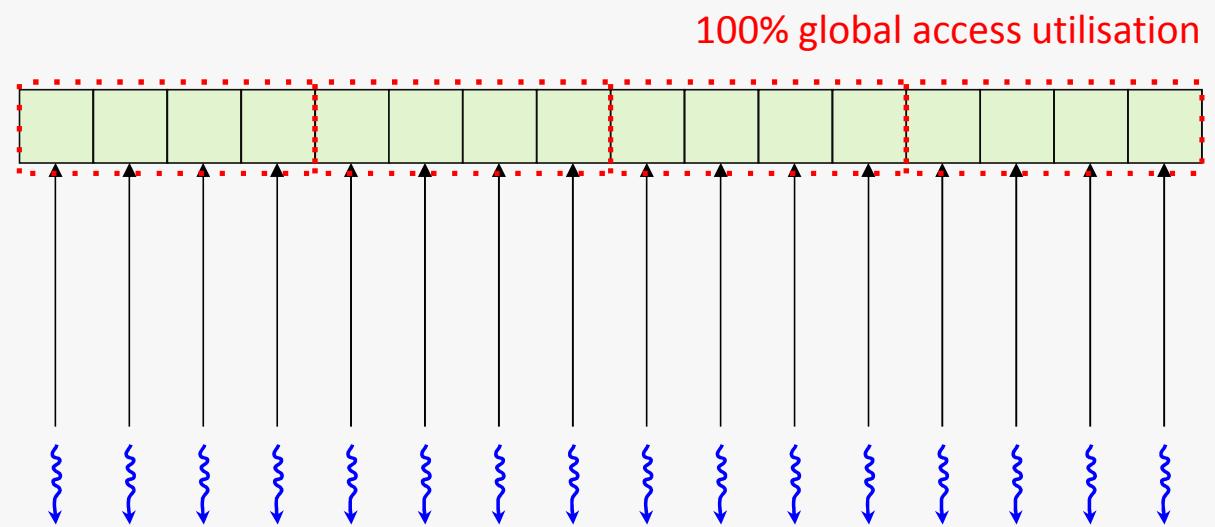
```
float data[size];  
...  
f(a[globalId]);
```



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

```
...
```

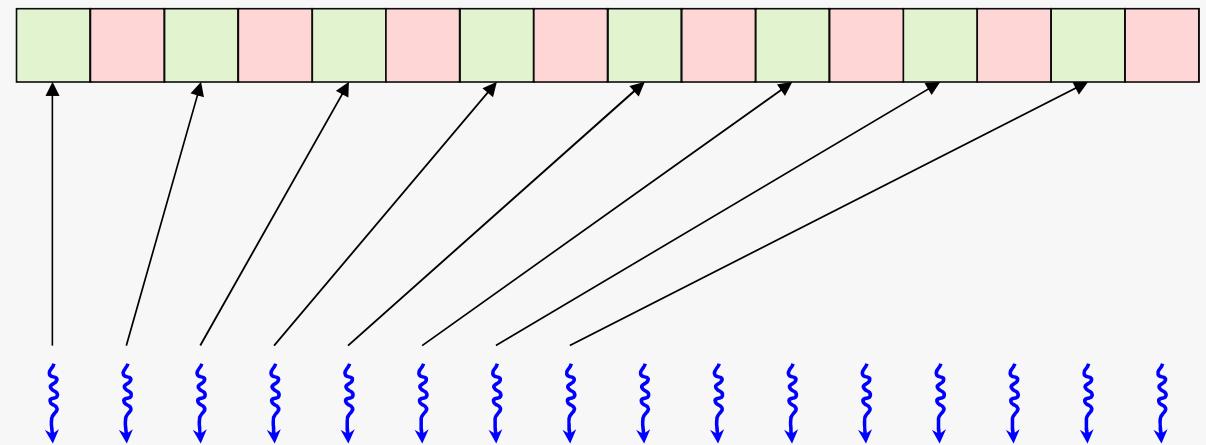
```
f(a[globalId]);
```



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

```
...
```

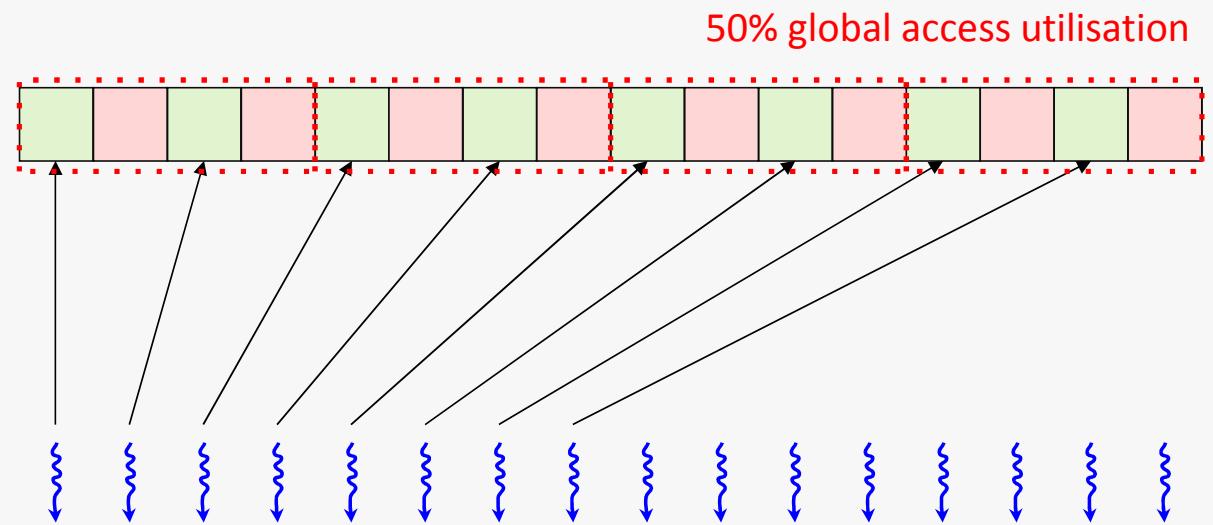
```
f(a[globalId * 2]);
```

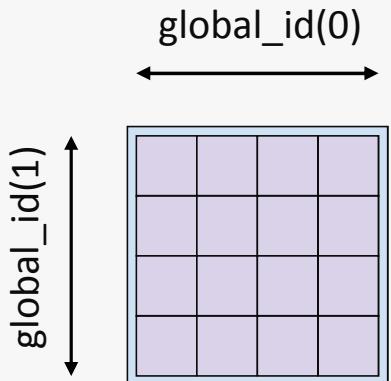


```
float data[size];
```

```
...
```

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





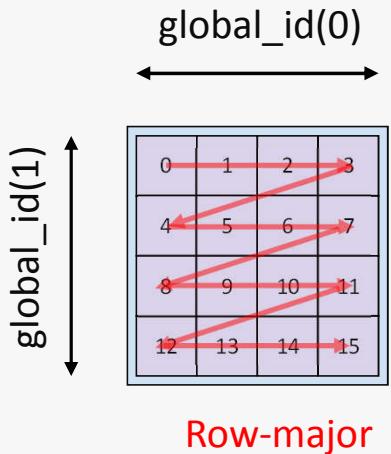
Row-major

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

This becomes very important when dealing with multiple dimensions

It's important to ensure that the order work-items are executed in aligns with the order that data elements that are accessed

This maintains coalesced global memory access



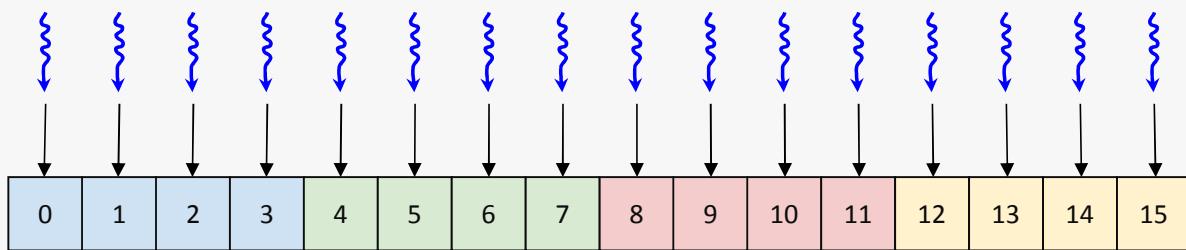
Row-major

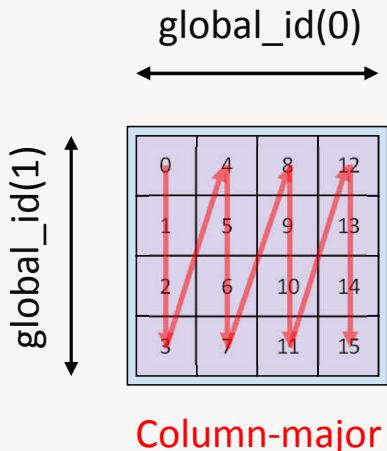
```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id1 * 4) + id0;
a[linearId] = f();
```

Row-major

Here data elements are accessed in row-major and work-items are executed in row-major

Global memory access is coalesced



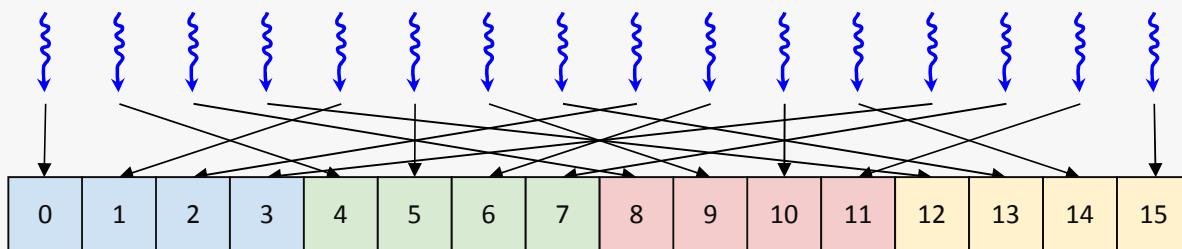


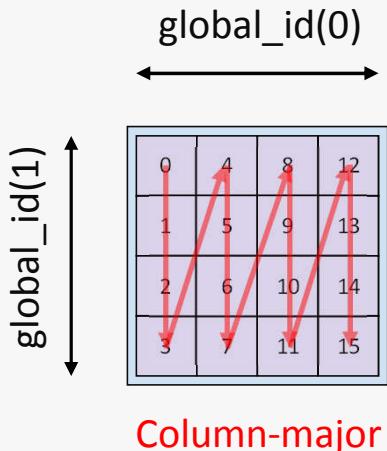
Row-major

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

If the work-items were executed in column-major

Global memory access is no longer coalesced





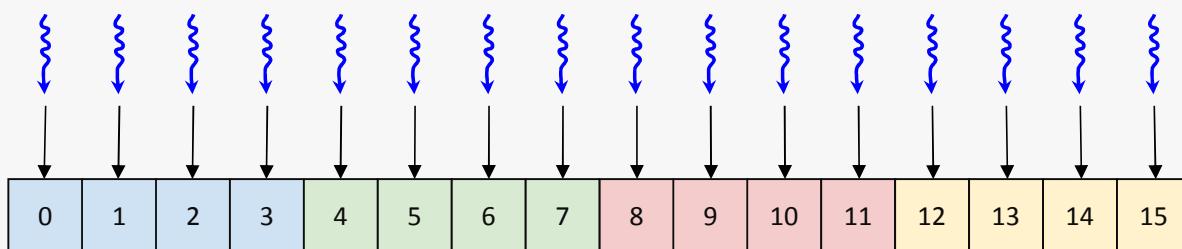
Column-major

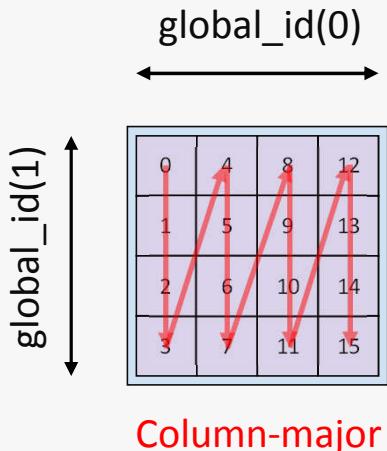
```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id0 * 4) + id1;
a[linearId] = f();
```

Column-major

However if you were to switch
the data access pattern to
column-major

Global memory access is
coalesced again





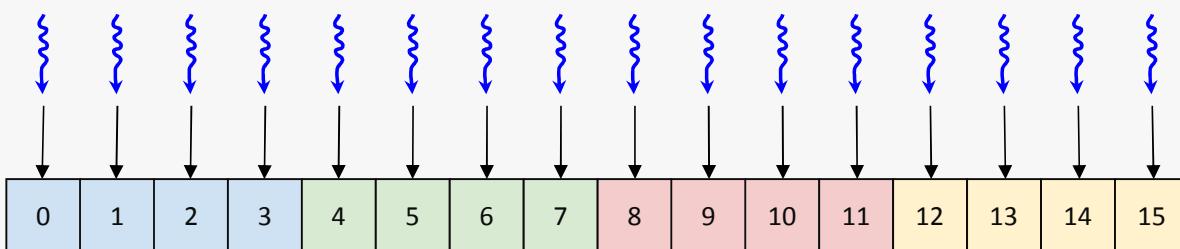
Column-major

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

Column-major

In this case the nd-range is square (width and height are equal)

If the nd-range were to be rectangular you would also have to switch the stride



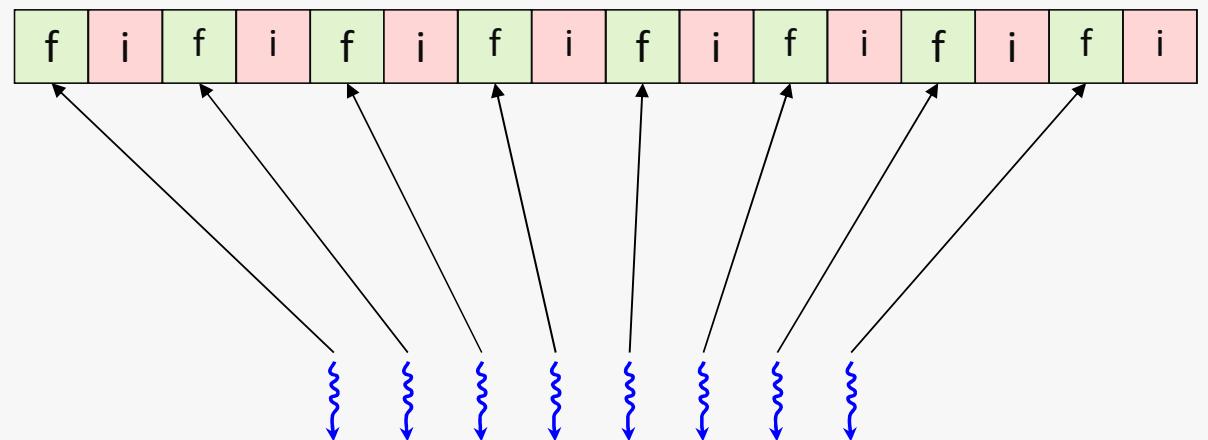
Representing your data as AoS or SoA

- Aos and SoA are two ways to represent your data
 - *AoS: Array of structures*
 - *SoA: Structure of arrays*
- How your algorithm accesses data members can impact which representation is more efficient
 - *Choosing one or the other can impact global memory access coalescing*

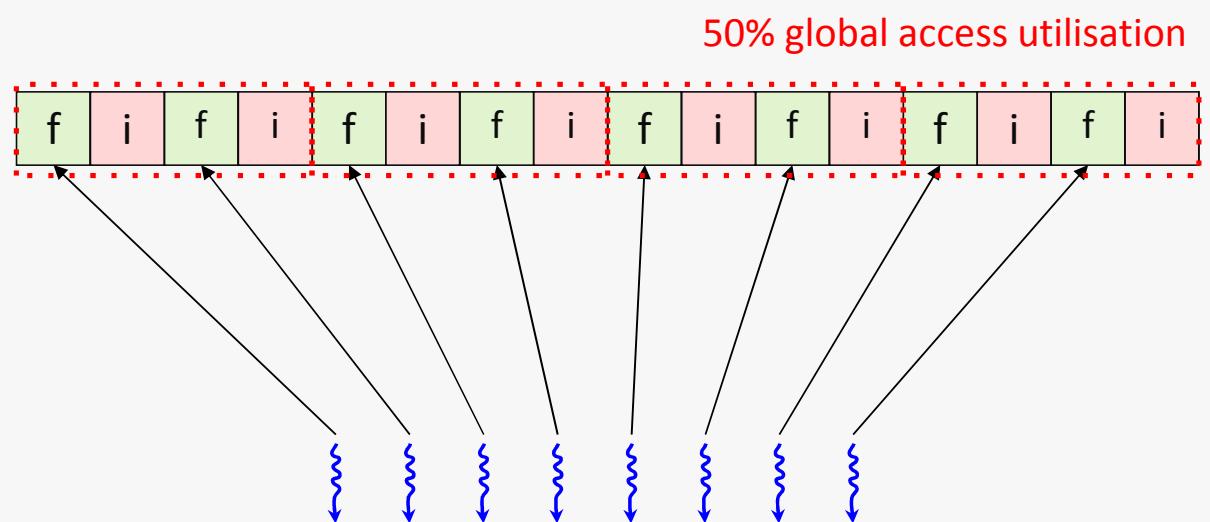
```
struct str {  
    float f;  
    int i;  
};  
  
str data[size];
```

f	i	f	i	f	i	f	i	f	i	f	i	f	i	f	i
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

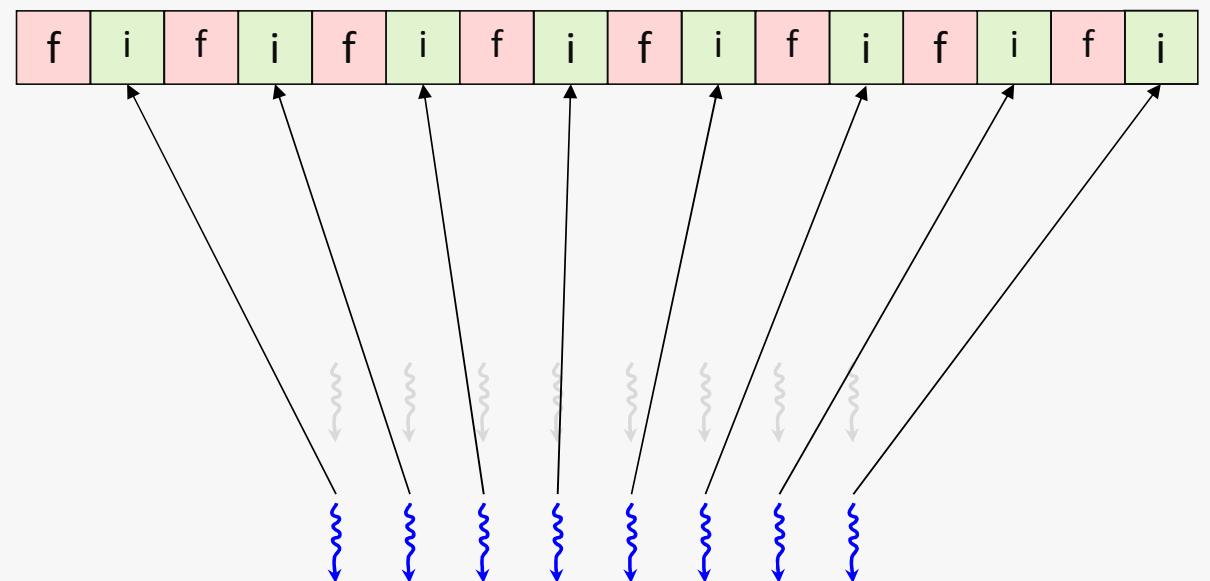
```
struct str {  
    float f;  
    int i;  
};  
  
str data[size];  
  
...  
  
f(a[globalId].f);
```



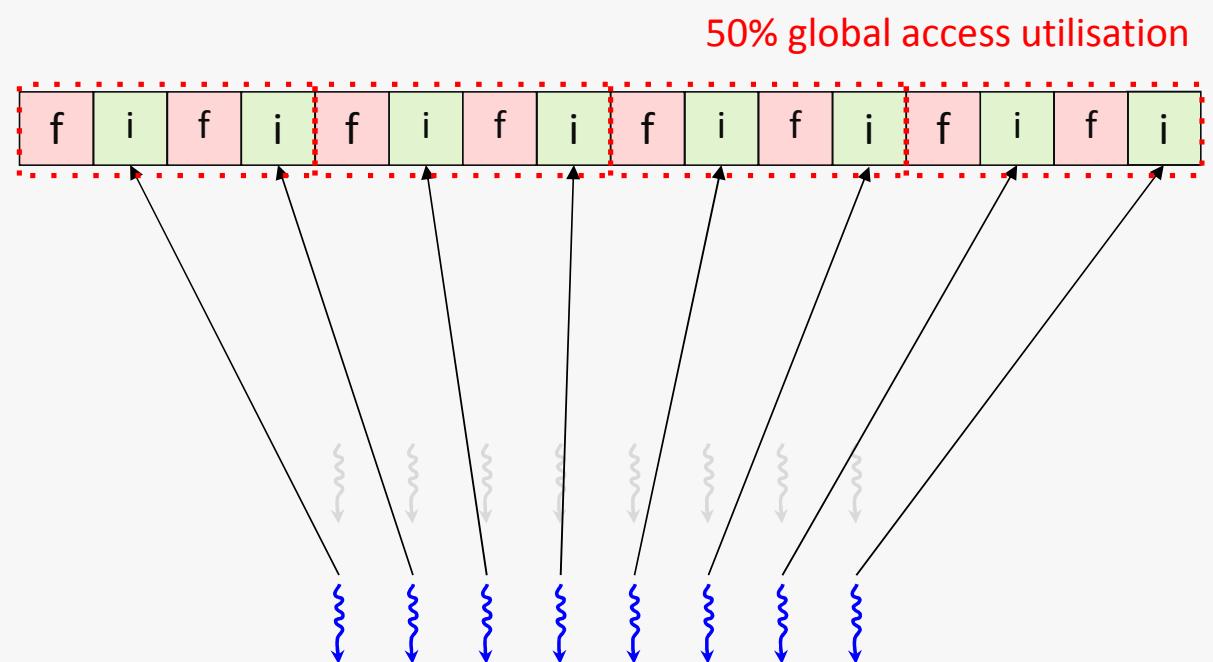
```
struct str {  
    float f;  
    int i;  
};  
  
str data[size];  
  
...  
  
f(a[globalId].f);
```



```
struct str {  
    float f;  
    int i;  
};  
  
str data[size];  
  
...  
  
f(a[globalId].f);  
  
f(a[globalId].i);
```



```
struct str {  
    float f;  
    int i;  
};  
  
str data[size];  
  
...  
  
f(a[globalId].f);  
  
f(a[globalId].i);
```



```
struct str {  
    float fs[size];  
    int is[size];  
};
```

```
str data;
```

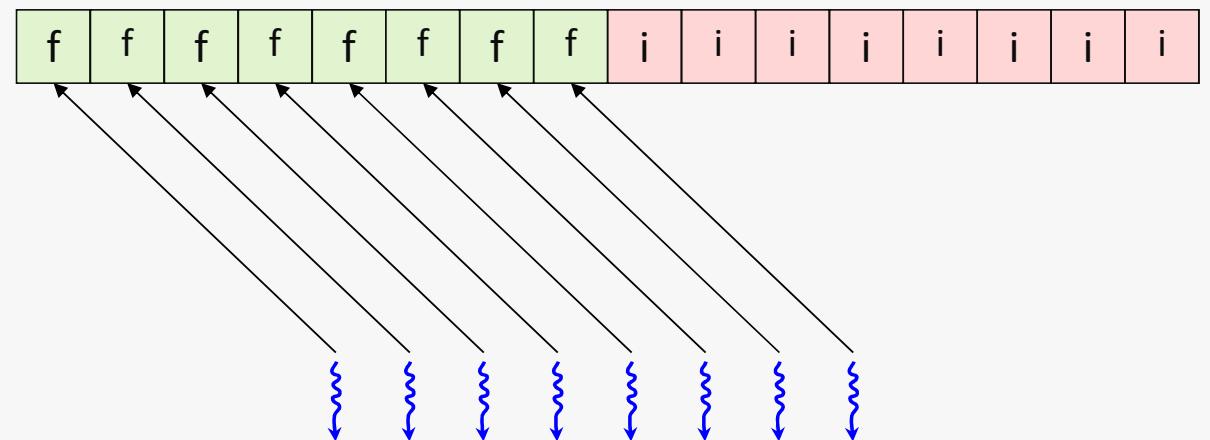


```
struct str {  
    float fs[size];  
    int is[size];  
};
```

```
str data;
```

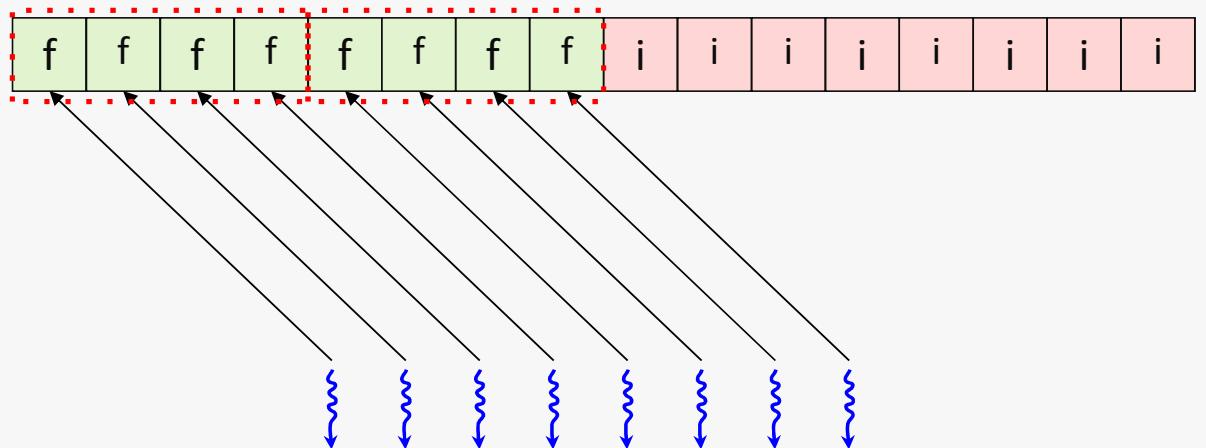
```
...
```

```
f(a[0].fs[globalId]);
```



```
struct str {  
    float fs[size];  
    int is[size];  
};  
  
str data;  
  
...  
  
f(a[0].fs[globalId]);
```

100% global access utilisation

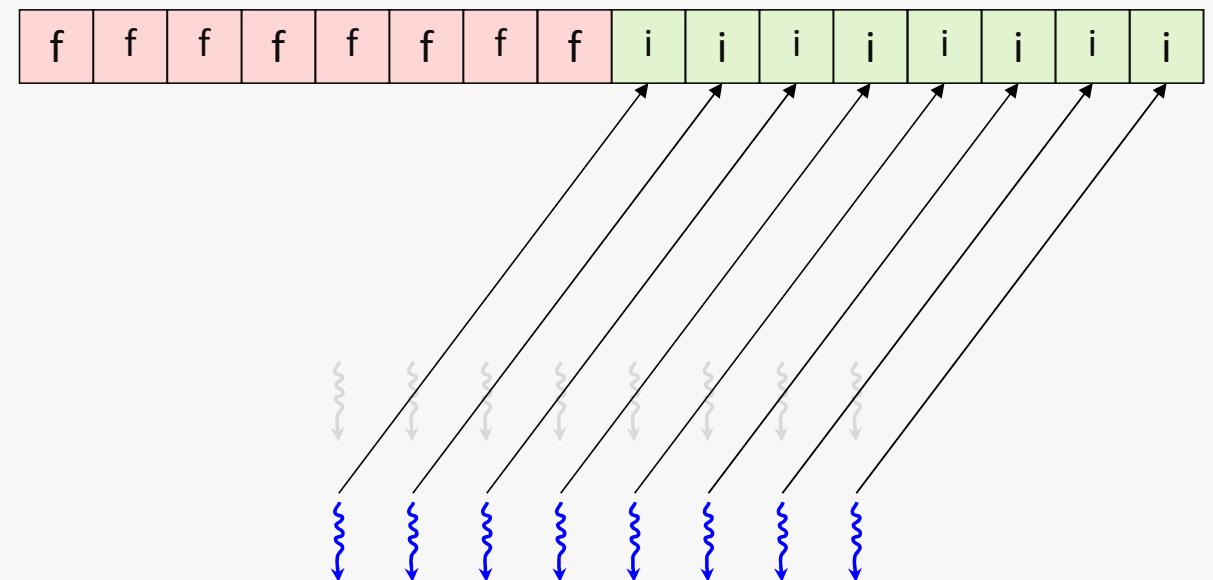


```
struct str {  
    float fs[size];  
    int is[size];  
};
```

```
str data;
```

```
...
```

```
f(a[0].fs[globalId]);  
  
f(a[0].is[globalId]);
```



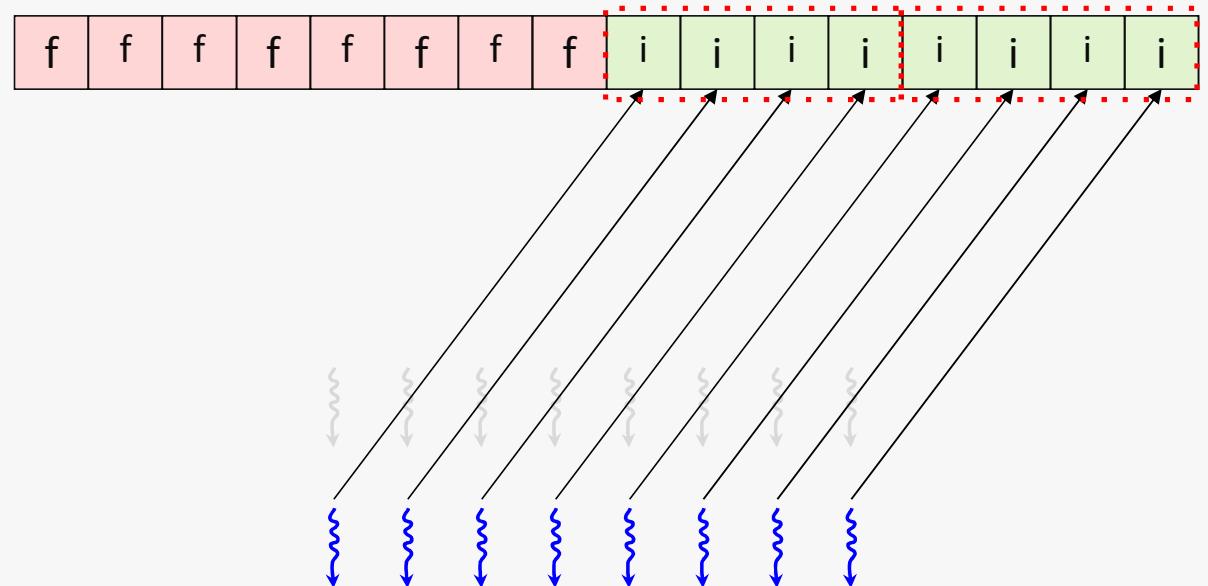
```
struct str {  
    float fs[size];  
    int is[size];  
};
```

```
str data;
```

```
...
```

```
f(a[0].fs[globalId]);  
  
f(a[0].is[globalId]);
```

100% global access utilisation



Make use of vector operations

- GPUs are vector processors
 - *Each processing element is capable of wide instructions which can operate on multiple elements of data at once*
- Many compilers can auto-vectorise
 - *This can affect the amount of performance gain you may see in vectorising your kernels*

```
float rS, gS, bS, aS;  
float r1, g1, b1, a1;  
float r2, g2, b2, a2;
```

...

```
rS = r1 + r2;  
gS = g1 + g2;  
bS = b1 + b2;  
aS = a1 + a2;
```



```
cl::sycl::float4 vS;  
cl::sycl::float4 v1;  
cl::sycl::float4 v2;  
  
...  
  
rS = v1 + v2;
```

128bit FP vector add

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue{gpu_selector{}, async_handler{}};

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

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

            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){ out[i] = inA[i] + inB[i]; });
        });

        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

If we go back to the vector add example from earlier

```

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

int main(int argc, char *argv[]) {
    std::vector<float> dA{ ... }, dB{ ... }, dO{ ... };

    try {
        queue gpuQueue{gpu_selector{}, async_handler{}};

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

        auto vecBufA = bufA.reinterpret<float4>(range<1>(dA.size() / 4));
        auto vecBufB = bufB.reinterpret<float4>(range<1>(dA.size() / 4));
        auto vecBufO = bufO.reinterpret<float4>(range<1>(dA.size() / 4));

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

            auto inA = vecBufA.get_access<access::mode::read>(cgh);
            auto inB = vecBufB.get_access<access::mode::read>(cgh);
            auto out = vecBufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size() / 4),
                [=] (id<1> i){ out[i] = inA[i] + inB[i]; });
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}

```

The representation of a buffer can be changed by calling the **reinterpret** member function

You must provide a new type and a new range that together represent the same total size in bytes

A re-interpreted buffer behaves as if it were a copy of the original buffer, just with a different type when you create an accessor to it

The SYCL vector classes provide arithmetic operators

Key takeaways

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

Ensuring global memory access is coalesced can give a significant performance gain

Vectorisation can give further performance gain



Questions?

Exercise 4:

Image grayscale

- How to create a simple image processing kernel that performs a grayscaling
- How to linearize the work-item id in row-major and column major ordering and how this affects global memory access coalescing
- How to vectorize kernel code using SYCL vector types



Chapter 13: GPU Optimization (part 2)

Gordon Brown

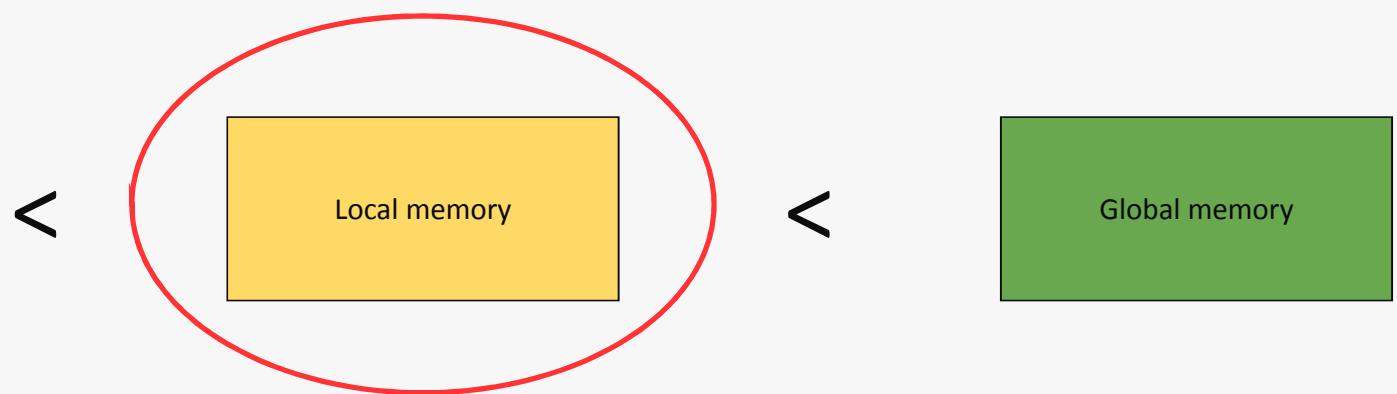
CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about when to use local memory
 - Learn how to declare and use local memory
 - Learn about synchronization of work-groups
 - Learn about occupancy and choosing an appropriate work-group size

Make use of local memory

- Local memory is much lower latency to access than global memory
 - *Cache commonly accessed data and temporary results in local memory rather than reading and writing to global memory*
- Using local memory is not necessarily always more efficient
 - *If data is not accessed frequently enough to warrant the copy to local memory you may not see a performance gain*

Private memory



1	7	5	8	2	3	8	3	4	6	2	2	4	5	8	3
1	3	4	3	2	4	3	4	5	6	1	6	5	7	8	5
9	2	1	8	1	4	6	9	5	1	4	5	1	9	4	7
3	6	2	0	2	2	9	8	2	7	9	4	2	6	1	5
1	7	2	2	8	4	6	8	4	7	6	8	3	2	4	1
4	9	9	5	1	3	7	3	8	1	7	4	1	5	9	4
4	0	6	3	6	9	9	6	8	5	9	9	0	2	1	5
3	8	1	2	4	7	1	7	6	7	7	2	6	3	6	7
6	7	5	4	3	1	4	4	2	6	3	0	5	0	7	0
1	3	4	2	2	8	1	6	4	9	5	3	7	1	2	4
7	5	4	3	7	0	4	0	3	0	4	4	2	8	9	0
0	9	9	8	0	2	9	8	2	1	6	0	6	3	4	1
6	4	0	1	9	1	7	4	8	3	0	5	0	2	0	6
1	5	7	6	3	0	6	5	4	6	0	4	1	8	7	0
3	3	0	5	9	8	2	4	7	1	5	2	0	4	9	7
1	9	0	4	0	3	0	6	1	2	8	7	0	1	2	9

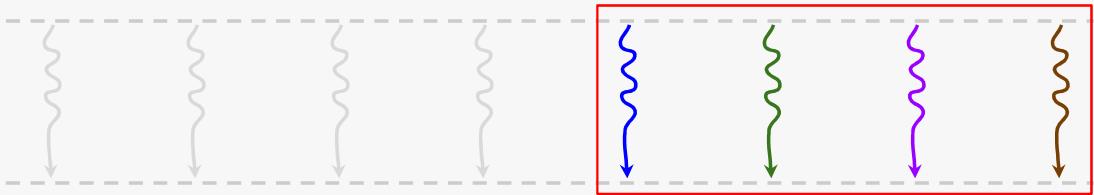
4	6	2	2	4	5	8	3
5	6	1	6	5	7	8	5
5	1	4	5	1	9	4	7
2	7	9	4	2	6	1	5
4	7	6	8	3	2	4	1
8	1	7	4	1	5	9	4
8	5	9	9	0	2	1	5
6	7	7	2	6	3	6	7

A common technique for using local memory is to break up your input into tiles

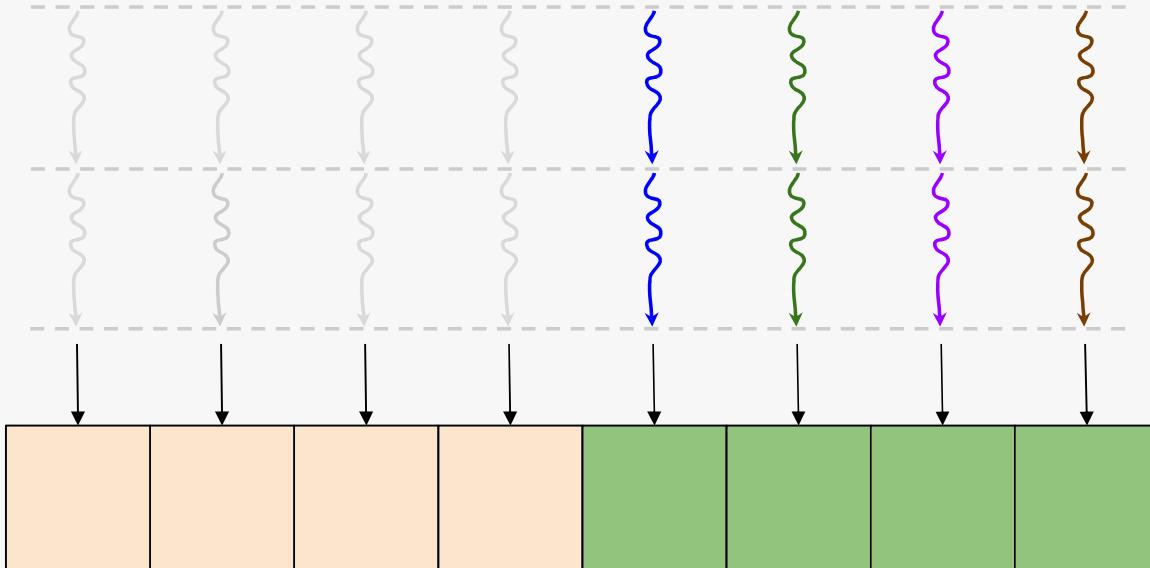
Then each tile can be moved to local memory while the work-group is working on it

Synchronise work-groups when necessary

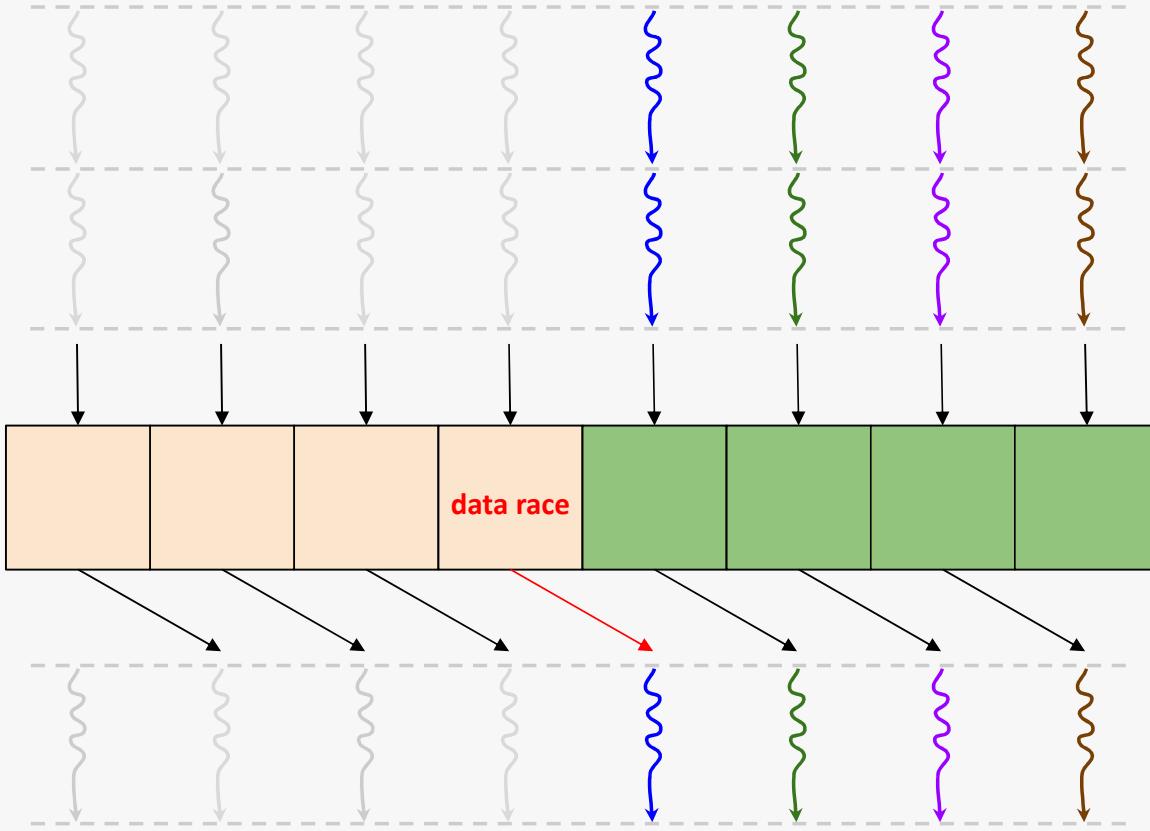
- Synchronising with a work-group barrier waits for all work-items to reach the same point
 - *Use a work-group barrier if you are copying data to local memory that neighbouring work-items will need to access*
 - *Use a work-group barrier if you have temporary results that will be shared with other work-items*



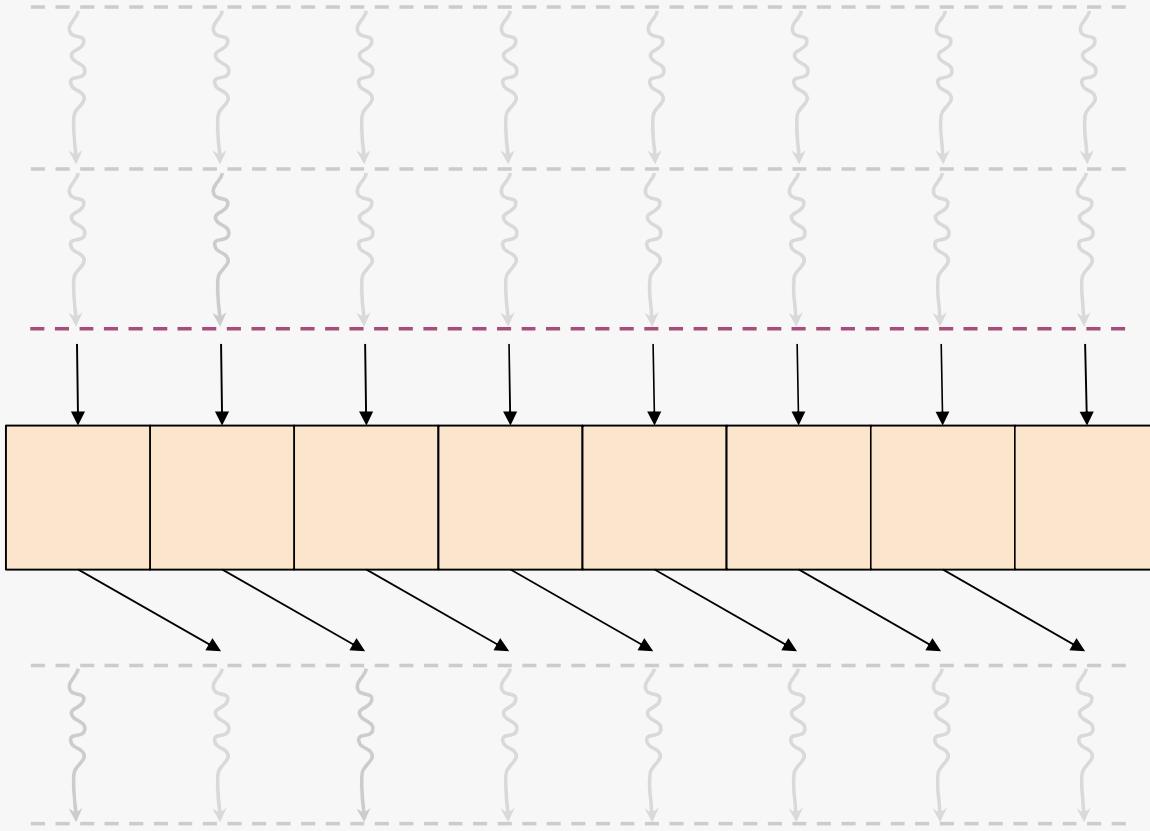
Remember that work-items are not
all guaranteed to execute
concurrently



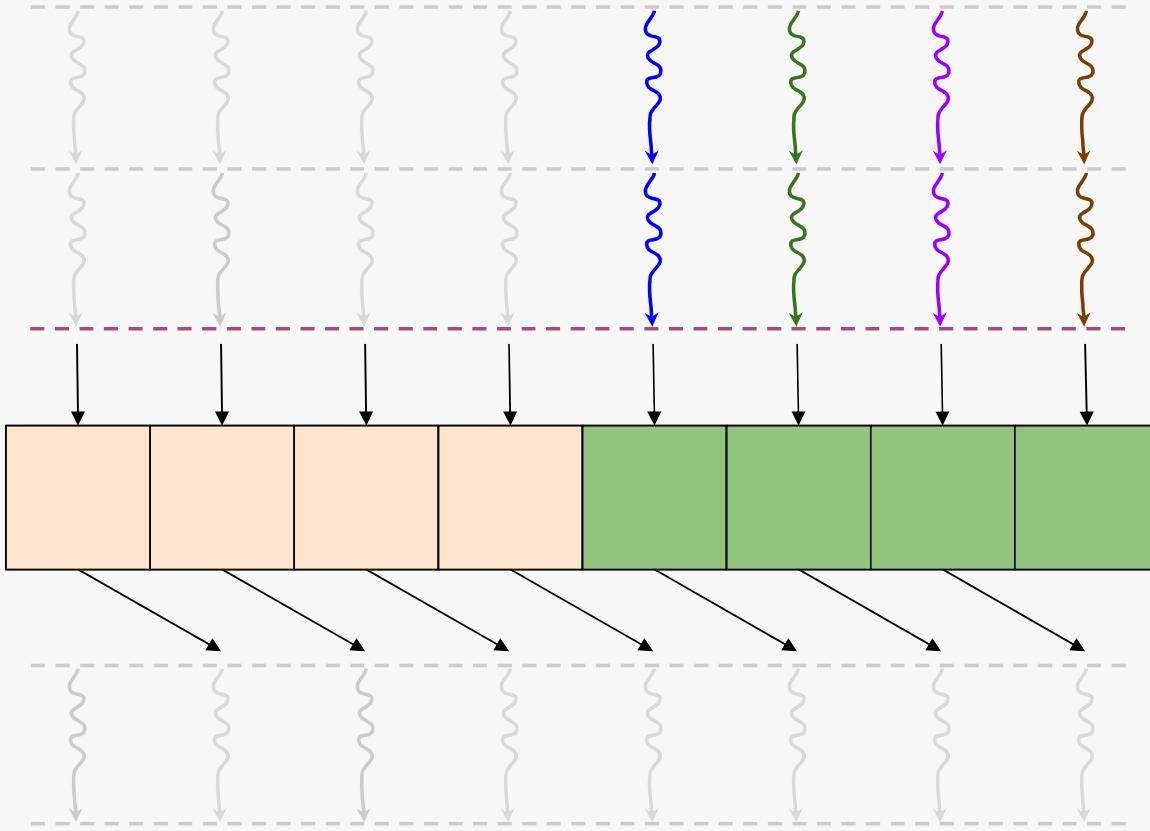
A work-item can share results with other work-items via local and global memory



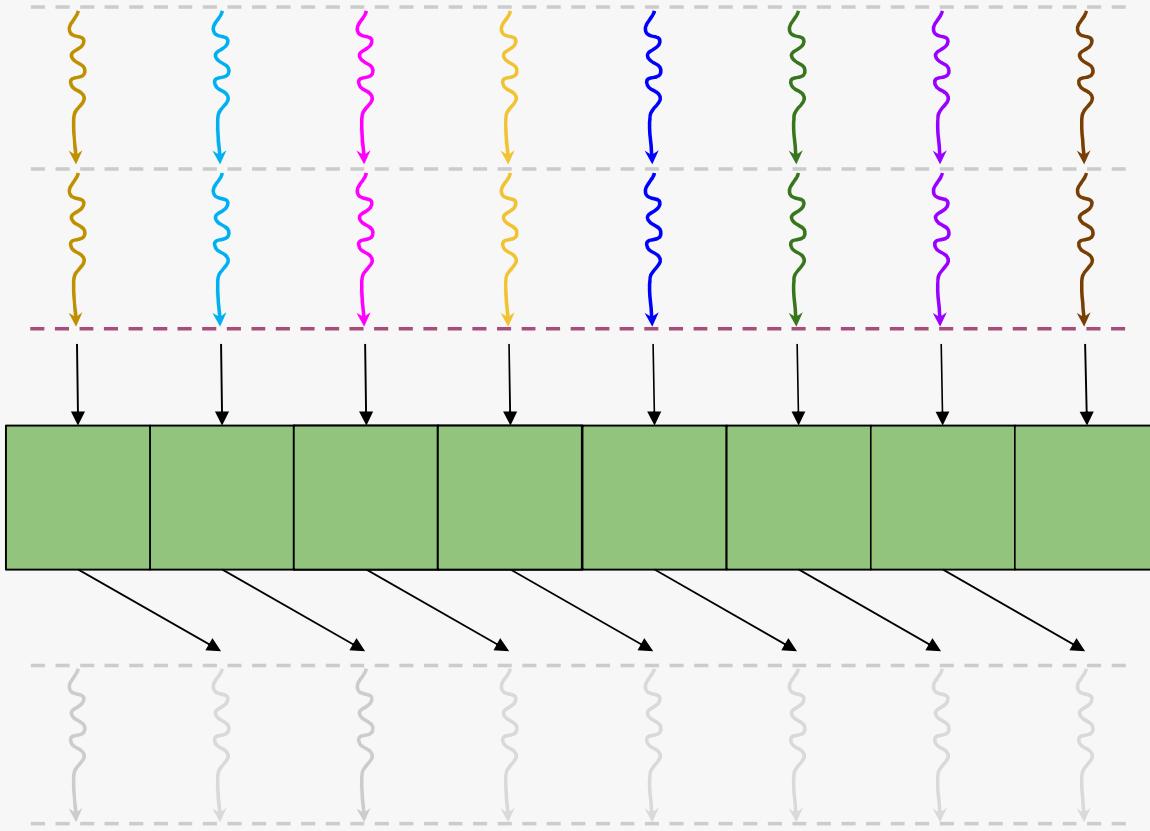
This means that it's possible for a work-item to read a result that hasn't yet been written to yet, you have a data race



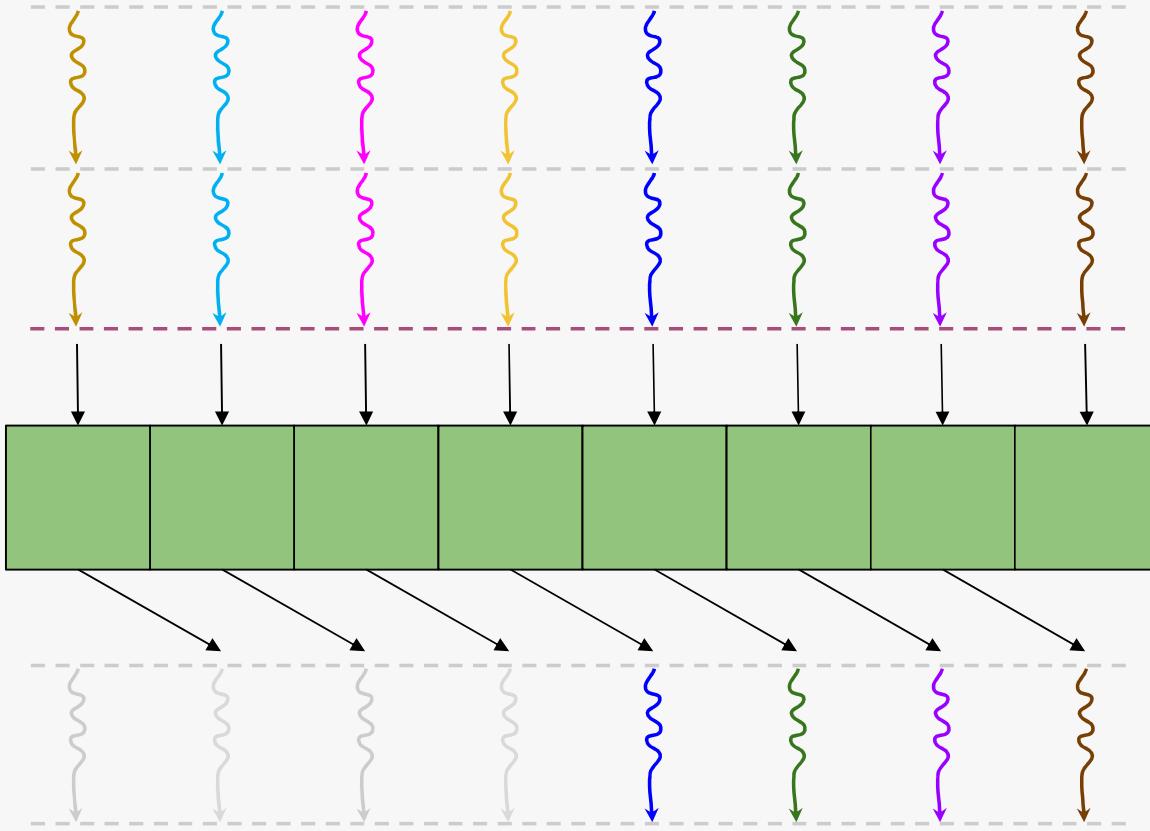
This problem can be solved by a synchronisation primitive called a work-group barrier



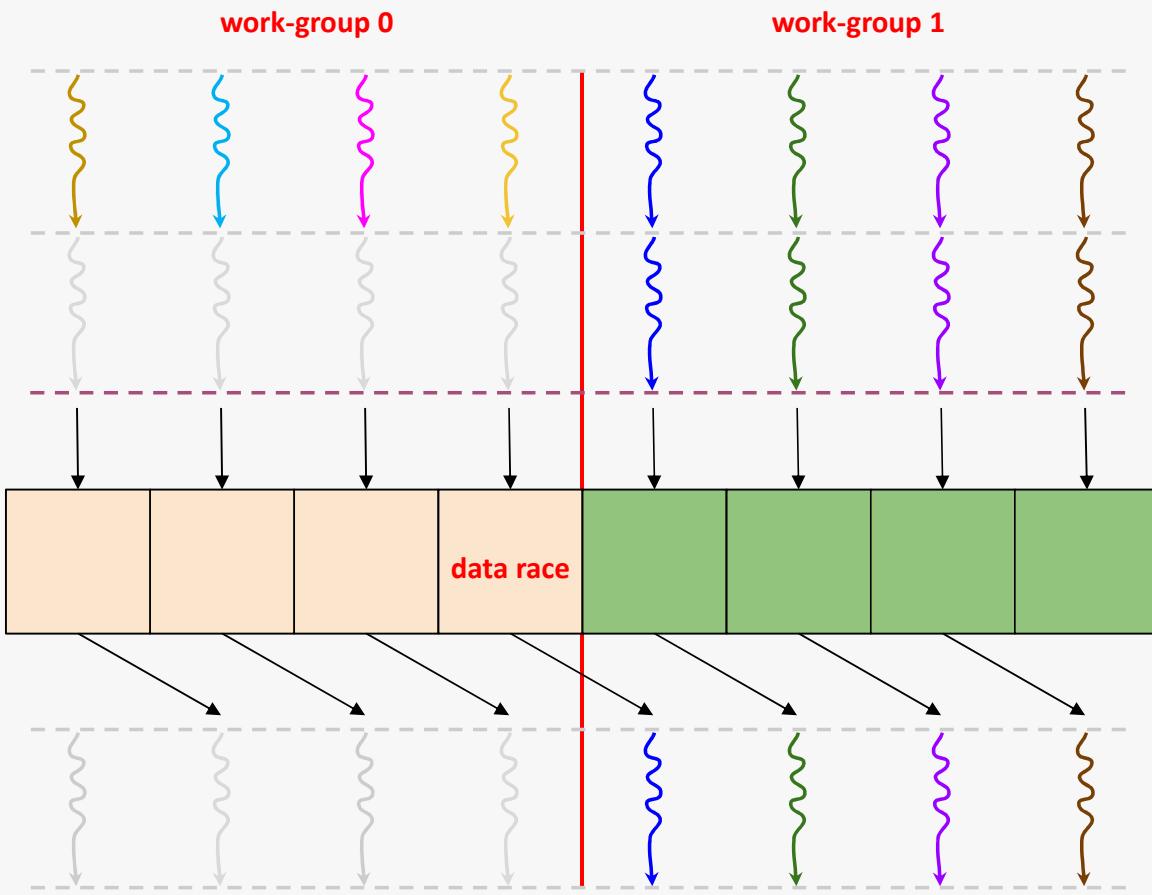
Work-items will block until all work-items in the work-group have reached that point



Work-items will block until all work-items in the work-group have reached that point



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



However this does not apply across work-group boundaries, and you have a data race again

- + Using local memory
- + Using work-group barriers

Choosing a good work-group size

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

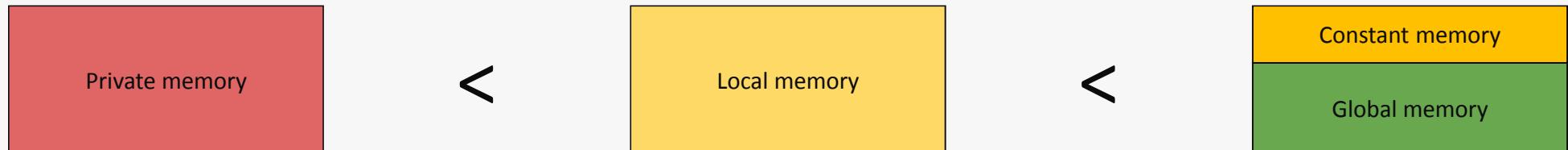
Querying GPU properties that can impact occupancy

- Total number of compute units
 - `info::device::max_compute_units`
- Maximum work-items per compute unit
 - `info::device::max_work_group_size`
- Total local memory available
 - `info::device::local_mem_size`

Ideas for further optimisations

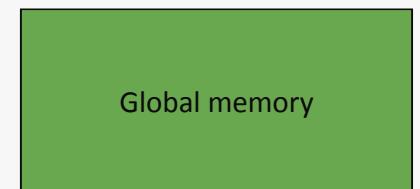
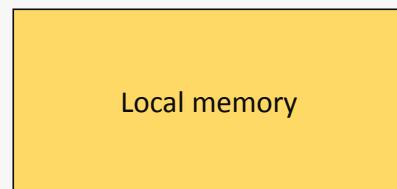
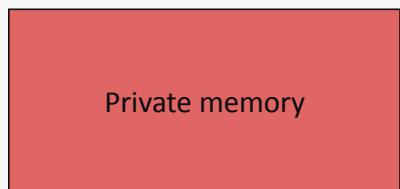
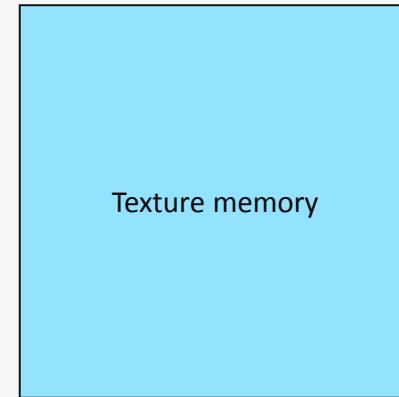
Use constant memory

- Some GPUs provide a region of global memory that is read-only
 - *This can be faster to access as it doesn't require caching*



Use texture memory

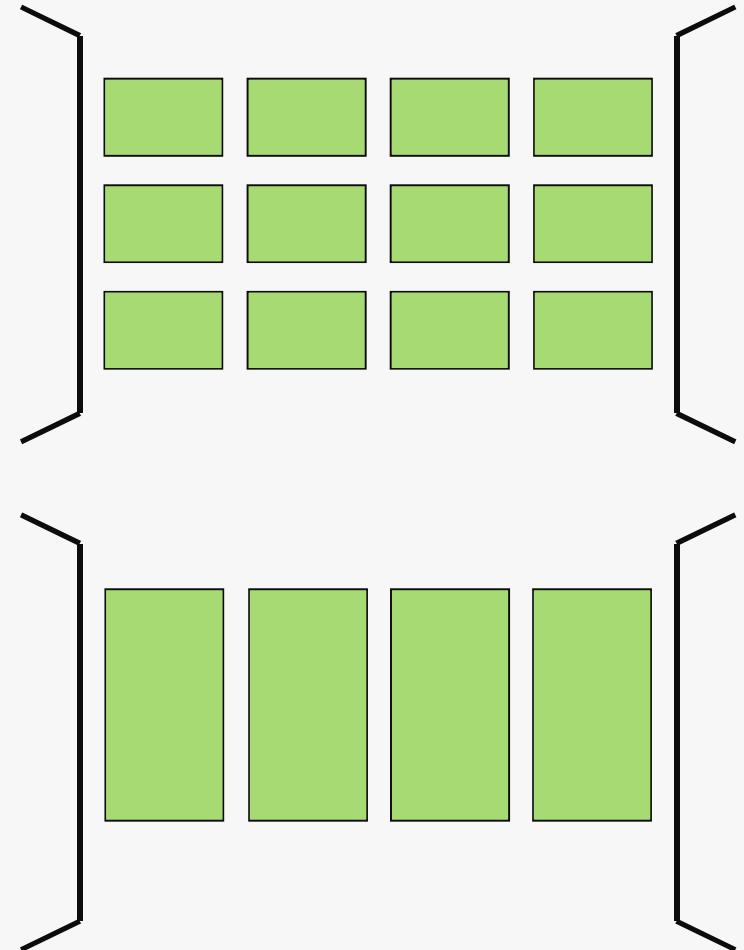
- Most GPUs have texture memory
 - *This can be faster to access for data that is represented as pixels*
 - *This also provides sampling operations*



Batch work together

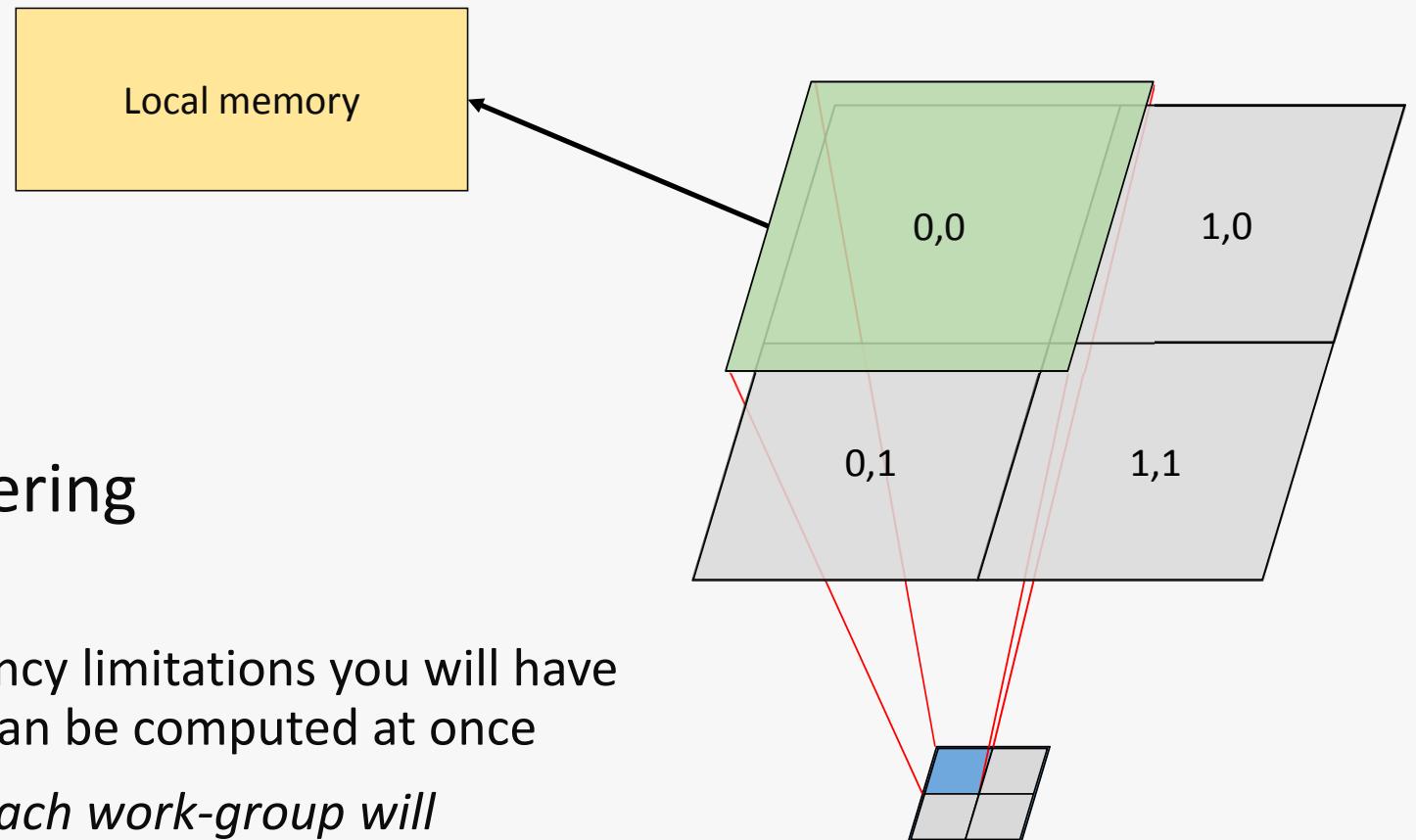
- Hitting occupancy limitations of a GPU can lead to drops in performance gain
 - *This is because single work-items are having to do more chunks of work*

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



Use double buffering

- If you hit occupancy limitations you will have more tiles than can be computed at once
 - *This means each work-group will compute more than one tile*



Copy

Copy
 $\{0, 0\}$

Copy
 $\{1, 0\}$

Copy
 $\{0, 1\}$

Copy
 $\{1, 1\}$

Compute

Compute
 $\{0, 0\}$

Compute
 $\{1, 0\}$

Compute
 $\{0, 1\}$

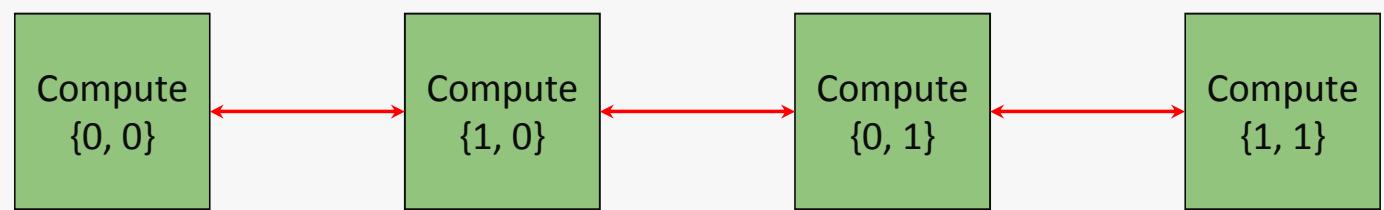
Compute
 $\{1, 1\}$



Copy



Compute



Copy

Copy
 $\{0, 0\}$

Copy
 $\{1, 0\}$

Copy
 $\{0, 1\}$

Copy
 $\{1, 1\}$

Compute

Compute
 $\{0, 0\}$

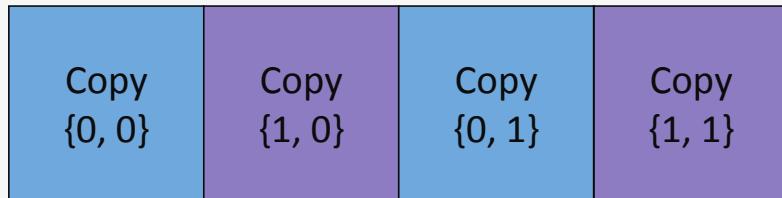
Compute
 $\{1, 0\}$

Compute
 $\{0, 1\}$

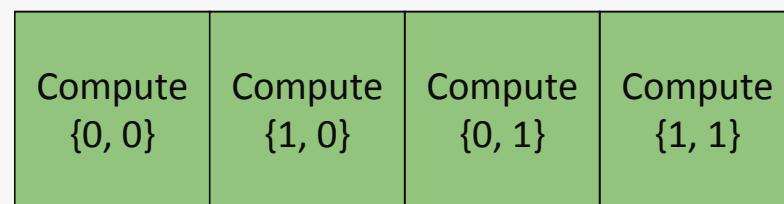
Compute
 $\{1, 1\}$



Copy



Compute



Overlapping copy and compute within kernels allows for better utilisation of GPU processing elements and therefore better throughput

Loop unrolling

```
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange),  
 [=](cl::sycl::nd_item<2> item) {  
  
     int rowOffset = item.get_global_id(1) * WIDTH;  
     int my = item.get_global_id(0) + rowOffset;  
  
     int fIndex = 0;  
     cl::sycl::float4 sum = cl::sycl::float4{0.0f};  
  
     sum += inputAcc[(my - 1 * WIDTH) - 1] * filterAcc[0];  
     sum += inputAcc[(my - 1 * WIDTH)] * filterAcc[1];  
     sum += inputAcc[(my - 1 * WIDTH) + 1] * filterAcc[2];  
     sum += inputAcc[(my * WIDTH) - 1] * filterAcc[3];  
     sum += inputAcc[(my * WIDTH)] * filterAcc[4];  
     sum += inputAcc[(my * WIDTH) + 1] * filterAcc[5];  
     sum += inputAcc[(my + 1 * WIDTH) - 1] * filterAcc[6];  
     sum += inputAcc[(my + 1 * WIDTH)] * filterAcc[7];  
     sum += inputAcc[(my + 1 * WIDTH) + 1] * filterAcc[8];  
  
     outputAcc[my] = sum;  
});
```

- Here we unroll the loop over the filter
 - *This allows the compiler more freedom in how it vectorises and allocates registers*
- However this does make the code more obfuscated and less flexible

Further tips

- Use profiling tools to gather more accurate information about your programs
 - *SYCL provides kernel profiling*
 - *Most OpenCL implementations provide proprietary profiler tools*
- Follow vendor optimisation guides
 - *Most OpenCL vendors provide optimisation guides that detail recommendations on how to optimise programs for their respective GPU*

Key takeaways

Use local memory when you have commonly accessed or shared data

Synchronise work-groups when accessing local memory that another work-item copied

Be aware of occupancy limitations and choose an appropriate work-group size



Questions?

Exercise 5:

Matrix transpose

- How to create a simple matrix transpose kernel
- How to allocate and use local memory
- How to synchronize a work-group using a barrier
- How different work-group sizes effect performance



Chapter 14: Parallel Algorithms

Gordon Brown

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about the standard algorithms
 - Learn about the C++17 parallel algorithms
 - Learn about execution policies
 - Learn about how to adapt algorithms for parallelism

Patterns

Many of the challenges of parallel computing and their solutions are best represented by common computational patterns

This class will focus on the patterns present in the C++ 11/14 standard algorithms, how those patterns must be adapted to provide opportunities for parallelism, and how to apply those patterns to CPU and GPU hardware

Standard algorithms

The C++ standard defines algorithms as:

- *“Describes components that C++ programs may use to perform algorithmic operations on containers and other sequences”*

We're going to look at three of these

transform

accumulate

partial_sum

transform

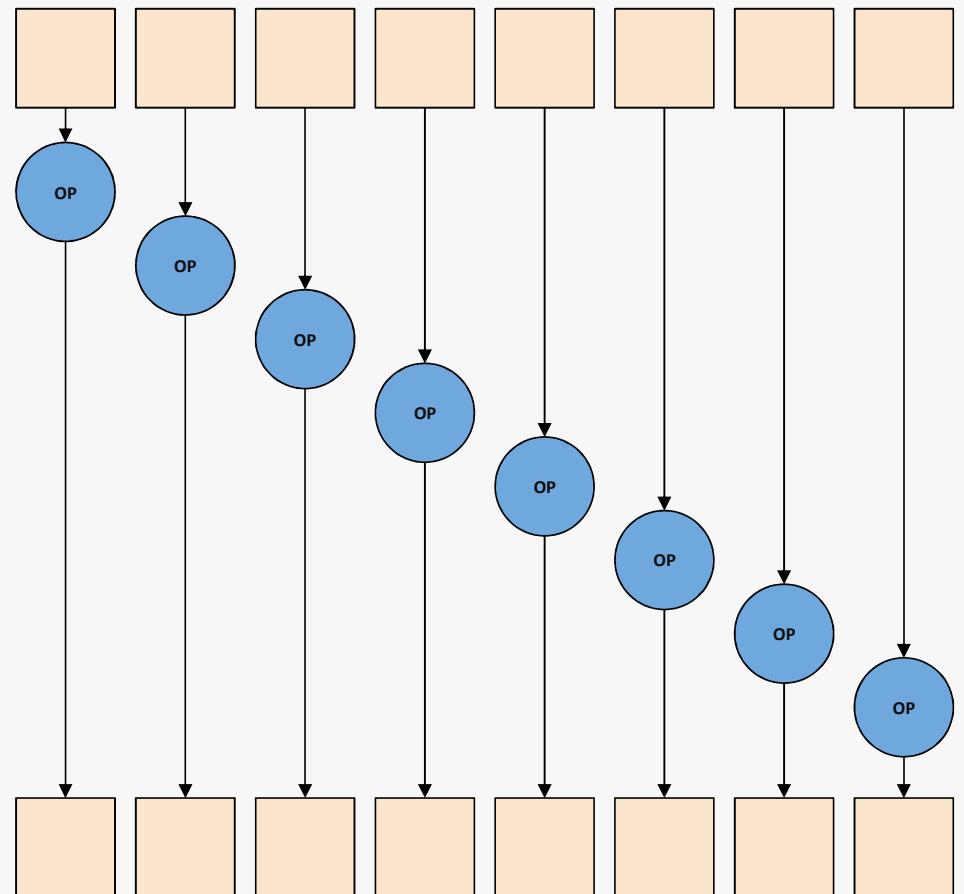
```
d_first transform(first, last,  
                  d_first,  
                  unary_op)
```

```
for each pair of it in [first, last) and d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```

transform

```
d_first transform(first, last,  
                  d_first,  
                  unary_op)
```

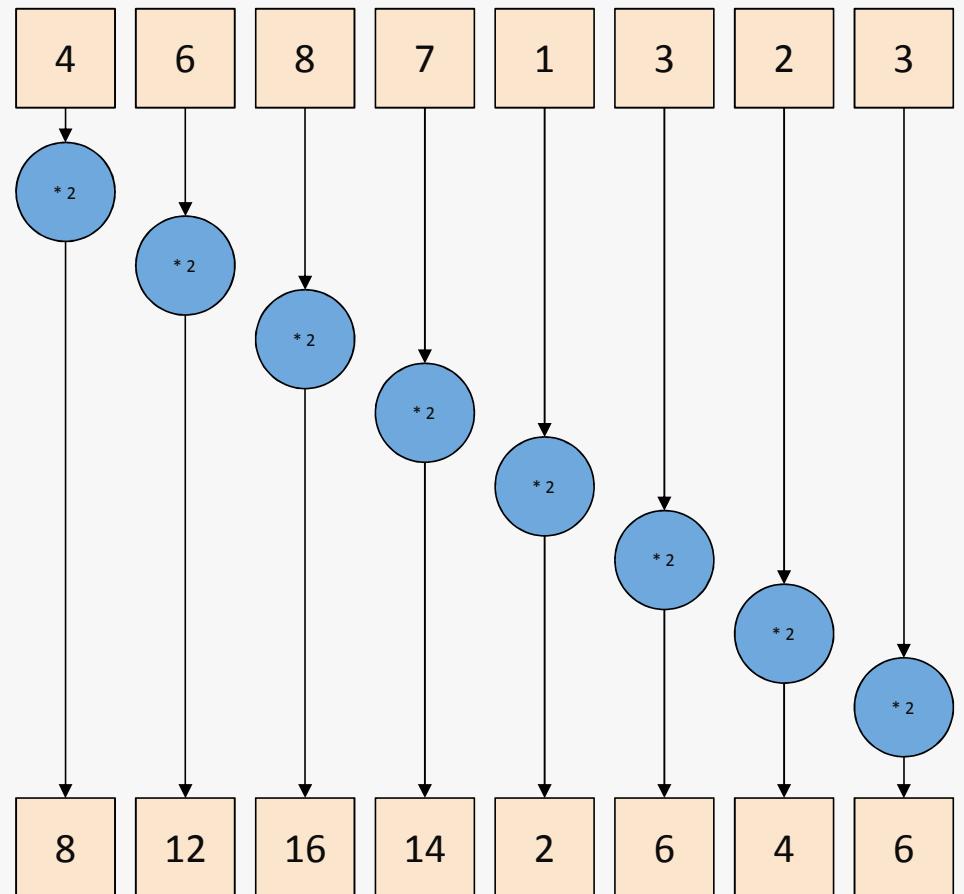
```
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



transform

```
d_first transform(first, last,  
                  d_first,  
                  unary_op)
```

```
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



accumulate

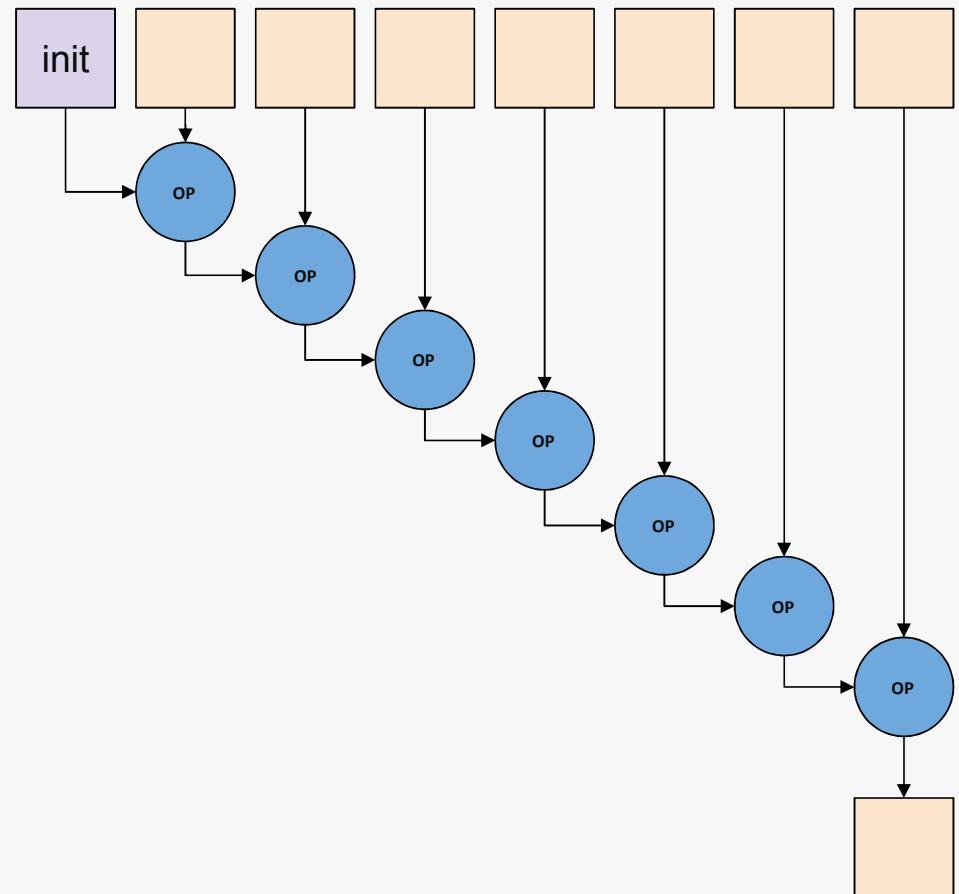
```
result accumulate(first, last,  
                  init,  
                  [binary_op])
```

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

accumulate

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

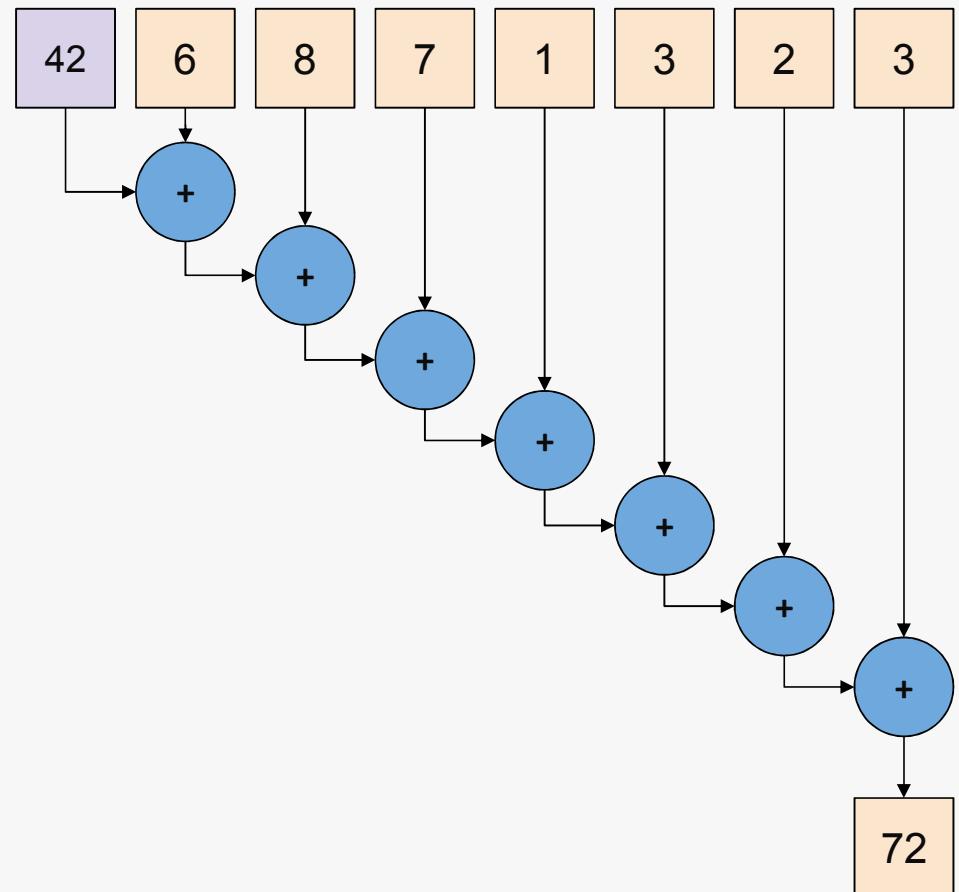
```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```



accumulate

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

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

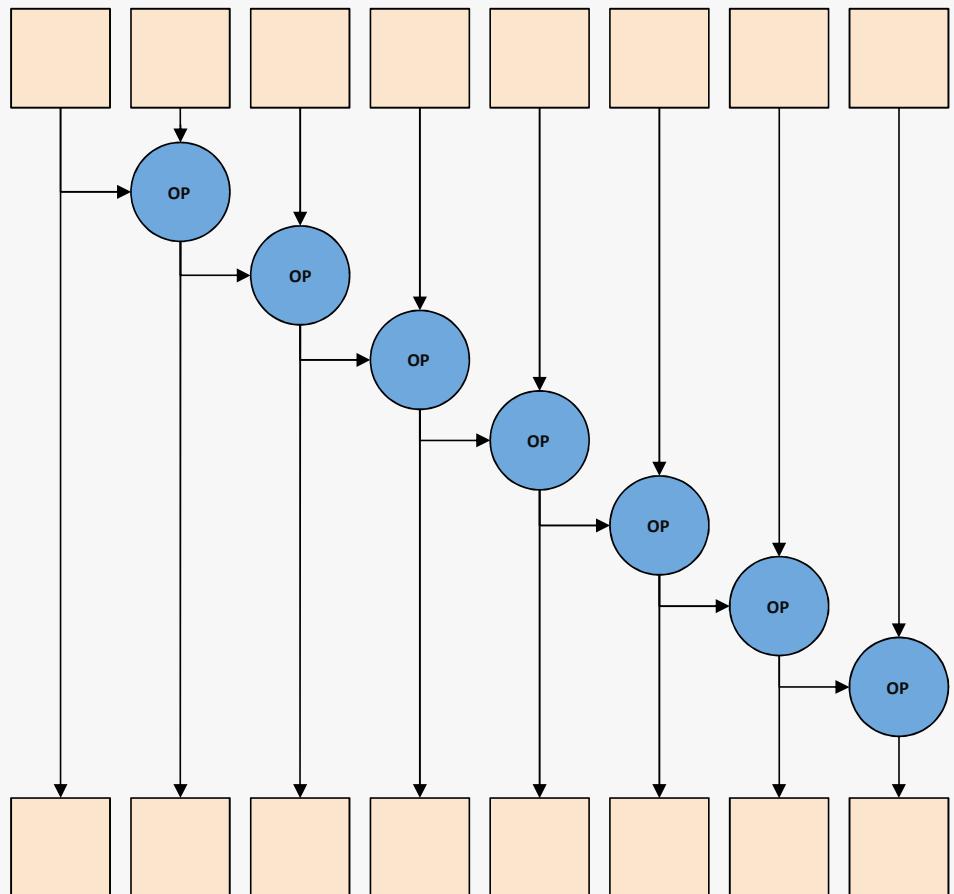


partial_sum

```
d_first partial_sum(first, last,
                     d_first,
                     [binary_op])
first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```

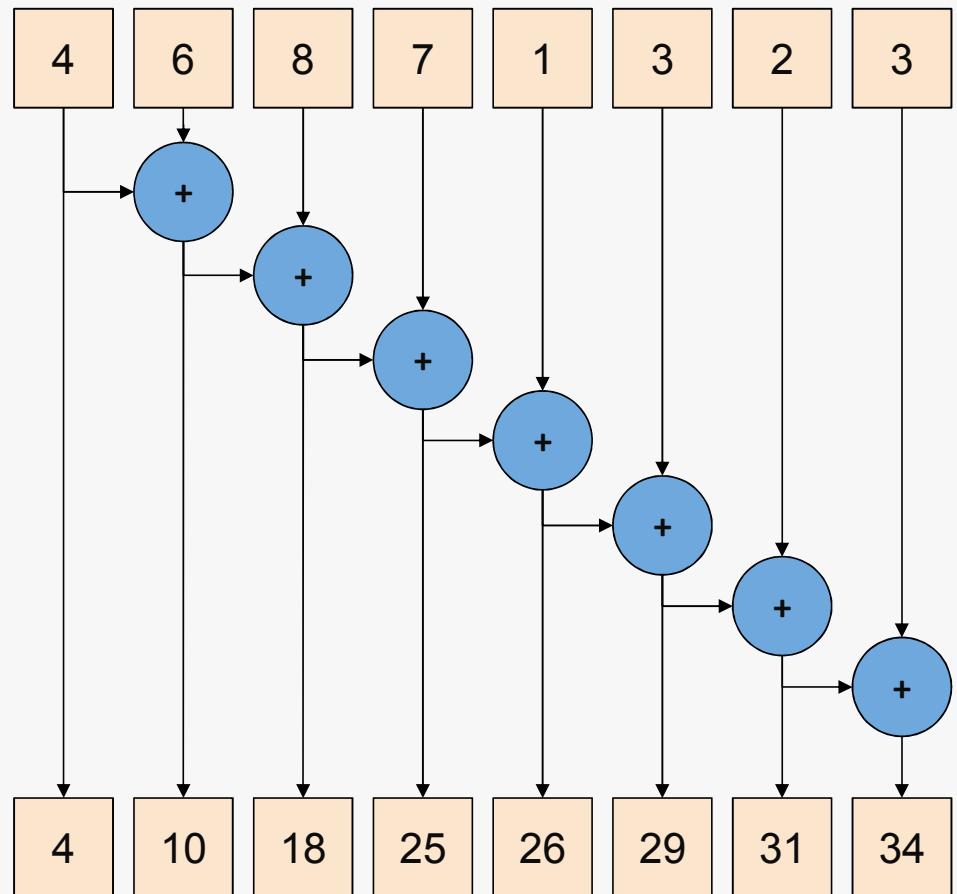
partial_sum

```
d_first partial_sum(first, last,
                     d_first,
                     [binary_op])
first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and
d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```



partial_sum

```
d_first partial_sum(first, last,
                     d_first,
                     [binary_op])
first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and
d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```



Moving from sequential to parallel algorithm

- **Guideline:**
 - Thread Programming \Leftrightarrow assembly language of Parallel programming
 - Algorithm & Pattern \Leftrightarrow High level Parallel Programming
- **The fastest way to get good speedup in C++17**
 - Use parallel algorithms

C++17 parallel algorithms

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

There are three kinds of new algorithms introduced:

- Overloads for most algorithms taking an execution policy
- New unordered versions of existing ordered algorithms
- New fused algorithms

Execution policies

An execution policy describes how a generic algorithm can be parallelized

They allow programmers to request parallelism with a particular set of constraints

Standard execution policies

- `sequenced_execution_policy` (`seq`)
 - Operations are indeterminately sequenced in the calling thread
 - Do not parallelize
- `parallel_execution_policy` (`par`)
 - Operations are indeterminately sequenced with respect to each other within the same thread
 - Parallelize but don't vectorise
- `parallel_unsequenced_execution_policy` (`par_unseq`)
 - Operations are unsequenced with respect to each other and possibly interleaved
 - Parallelize and may vectorise

Parallel versions of existing algorithms

adjacent_difference	for_each[_n]	merge	remove[_copy _copy_if _if]	stable_partition
adjacent_find	generate[_n]	min_element	replace[_copy _copy_if _if]	stable_sort
all_of	includes	minmax_element	reverse[_copy]	swap_ranges
any_of	implace_merge	mismatch	rotate[_copy]	transform
copy[_if _n]		move	search[_n]	
count[_if]	is_heap[_until]	none_of	set_difference	
equal	is_partitioned	nth_element	set_intersection	
	is_sorted[_until]	partial_sort[_copy]	set_symmetric_difference	uninitialized_copy[_n]
fill[_n]	lexicographical_compare	partition[_copy]	set_union	uninitialized_fill[_n]
find[_end _first_of _if _if_not]	max_element		sort	unique[_copy]

Parallel versions of existing algorithms

adjacent_difference	<code>for_each[_n]</code>	merge	<code>remove[_copy _copy_if _if]</code>	stable_partition
adjacent_find	<code>generate[_n]</code>	<code>min_element</code>	<code>replace[_copy _copy_if _if]</code>	stable_sort
all_of	<code>includes</code>	<code>minmax_element</code>	<code>reverse[_copy]</code>	swap_ranges
any_of	<code>implace_merge</code>	<code>mismatch</code>	<code>rotate[_copy]</code>	transform
<code>copy[_if _n]</code>		<code>move</code>	<code>search[_n]</code>	
count[_if]	<code>is_heap[_until]</code>	<code>none_of</code>	<code>set_difference</code>	
equal	<code>is_partitioned</code>	<code>nth_element</code>	<code>set_intersection</code>	
	<code>is_sorted[_until]</code>	<code>partial_sort[_copy]</code>	<code>set_symmetric_difference</code>	<code>uninitialized_copy[_n]</code>
<code>fill[_n]</code>	<code>lexicographical_compare</code>	<code>partition[_copy]</code>	<code>set_union</code>	<code>uninitialized_fill[_n]</code>
<code>find[_end _first_of _if _if_not]</code>	<code>max_element</code>		<code>sort</code>	<code>unique[_copy]</code>

New unordered algorithms

adjacent_difference	<code>for_each[_n]</code>	merge	<code>remove[_copy _copy_if _if]</code>	stable_partition
adjacent_find	<code>generate[_n]</code>	<code>min_element</code>	<code>replace[_copy _copy_if _if]</code>	stable_sort
all_of	<code>includes</code>	<code>minmax_element</code>	<code>reverse[_copy]</code>	swap_ranges
any_of	<code>implace_merge</code>	<code>mismatch</code>	<code>rotate[_copy]</code>	transform
<code>copy[_if _n]</code>	<code>inclusive_scan</code>	<code>move</code>	<code>search[_n]</code>	
<code>count[_if]</code>	<code>is_heap[_until]</code>	<code>none_of</code>	<code>set_difference</code>	
<code>equal</code>	<code>is_partitioned</code>	<code>nth_element</code>	<code>set_intersection</code>	
<code>exclusive_scan</code>	<code>is_sorted[_until]</code>	<code>partial_sort[_copy]</code>	<code>set_symmetric_difference</code>	<code>uninitialized_copy[_n]</code>
<code>fill[_n]</code>	<code>lexicographical_compare</code>	<code>partition[_copy]</code>	<code>set_union</code>	<code>uninitialized_fill[_n]</code>
<code>find[_end _first_of _if _if_not]</code>	<code>max_element</code>	<code>reduce</code>	<code>sort</code>	<code>unique[_copy]</code>

New fused algorithms

adjacent_difference	<code>for_each[_n]</code>	merge	<code>remove[_copy _copy_if _if]</code>	<code>stable_partition</code>
adjacent_find	<code>generate[_n]</code>	<code>min_element</code>	<code>replace[_copy _copy_if _if]</code>	<code>stable_sort</code>
all_of	<code>includes</code>	<code>minmax_element</code>	<code>reverse[_copy]</code>	<code>swap_ranges</code>
any_of	<code>implace_merge</code>	<code>mismatch</code>	<code>rotate[_copy]</code>	<code>transform</code>
<code>copy[_if _n]</code>	<code>inclusive_scan</code>	<code>move</code>	<code>search[_n]</code>	<code>transform_exclusive_scan</code>
<code>count[_if]</code>	<code>is_heap[_until]</code>	<code>none_of</code>	<code>set_difference</code>	<code>transform_inclusive_scan</code>
<code>equal</code>	<code>is_partitioned</code>	<code>nth_element</code>	<code>set_intersection</code>	<code>transform_reduce</code>
<code>exclusive_scan</code>	<code>is_sorted[_until]</code>	<code>partial_sort[_copy]</code>	<code>set_symmetric_difference</code>	<code>uninitialized_copy[_n]</code>
<code>fill[_n]</code>	<code>lexicographical_compare</code>	<code>partition[_copy]</code>	<code>set_union</code>	<code>uninitialized_fill[_n]</code>
<code>find[_end _first_of _if _if_not]</code>	<code>max_element</code>	<code>reduce</code>	<code>sort</code>	<code>unique[_copy]</code>

In this class we will focus on...

`transform` (non-serial) -> new overload of `transform` (serial)

`reduce` -> `unordered accumulate`

`inclusive_scan` -> `unordered partial_sum (inclusive)`

`transform_reduce` -> fused `transform (unordered) & reduce`

Why do we need unordered algorithms?

Having algorithms be unordered means the some algorithms can be implemented to perform the operations in parallel where they couldn't before

Why do we need unordered algorithms?

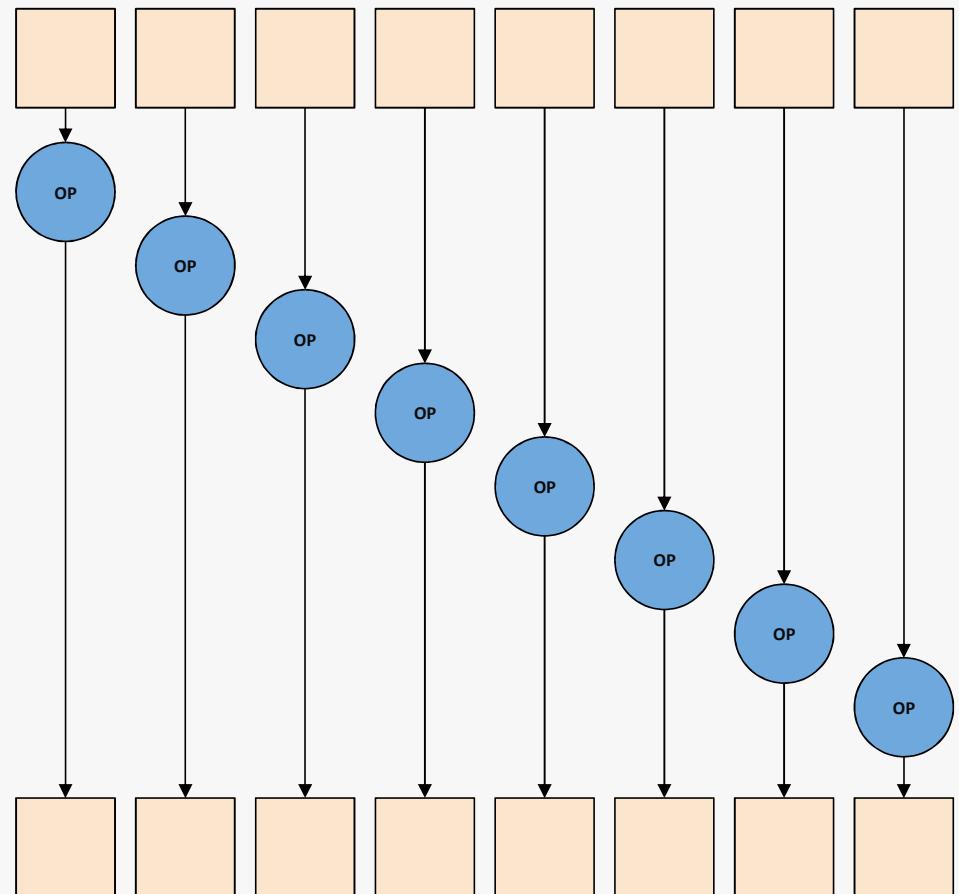
Having algorithms be unordered means the some algorithms can be implemented to perform the operations in parallel where they couldn't before

However allowing algorithms to be unordered comes with some caveats

transform (serial)

```
d_first transform(first, last,  
                 d_first,  
                 unary_op)
```

```
for each pair of it in [first, last) and  
d_it in [d_first, last) in order  
    *d_it = unary_op(*it)  
return d_first
```



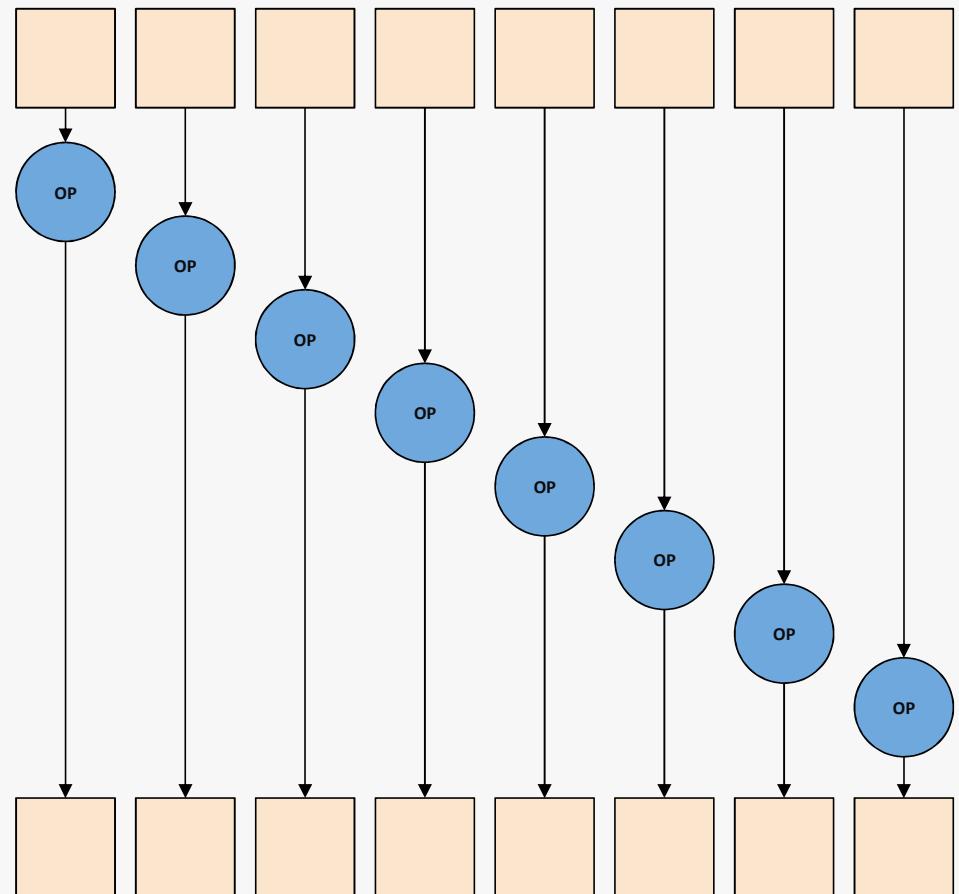
transform (non-serial)

```
d_first transform([execution_policy,]
                  first, last,
                  d_first,
                  unary_op)
```

```
for each pair of it in [first, last) and d_it in [d_first, last)
    *d_it = unary_op(*it)
return d_first
```

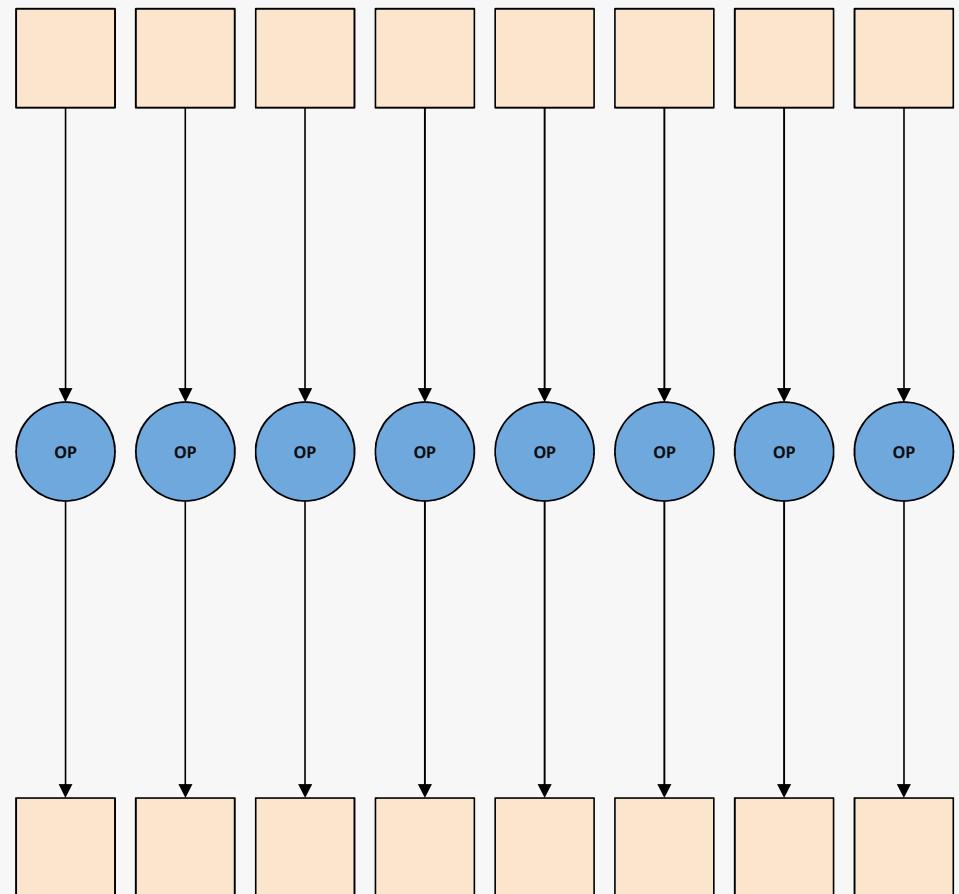
transform (non-serial)

```
d_first  
transform([execution_policy,  
          first, last,  
          d_first,  
          unary_op)  
  
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



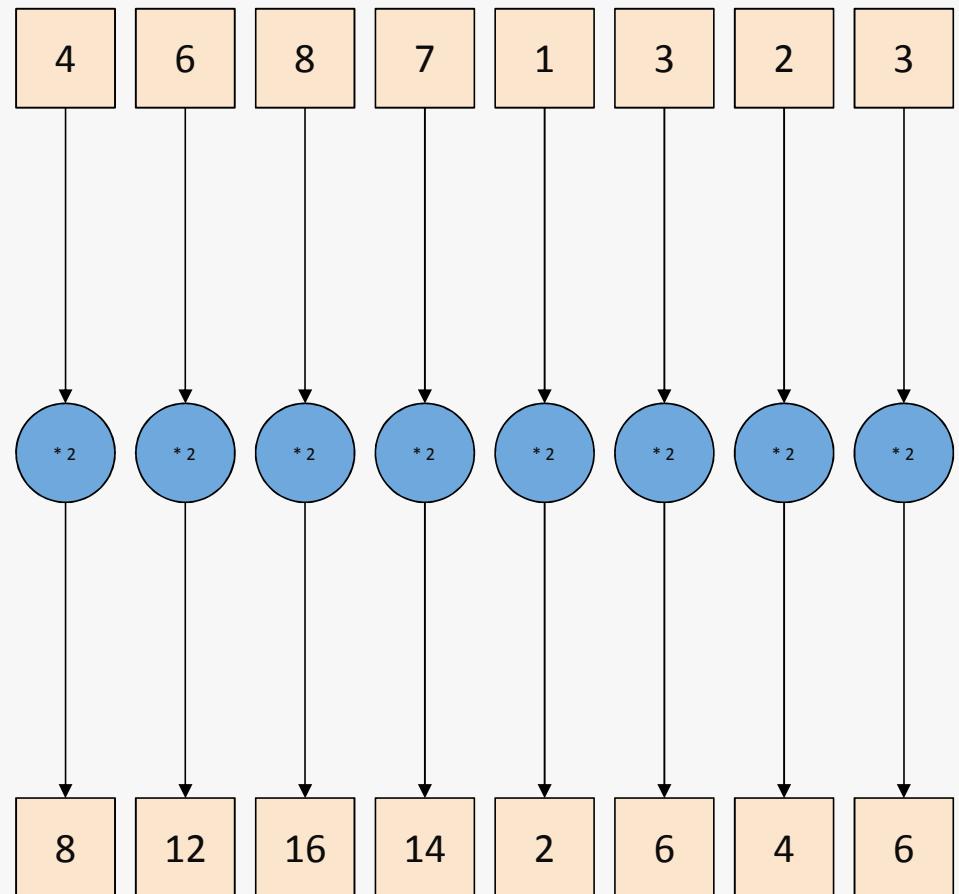
transform (non-serial)

```
d_first  
transform([execution_policy,  
          first, last,  
          d_first,  
          unary_op)  
  
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```

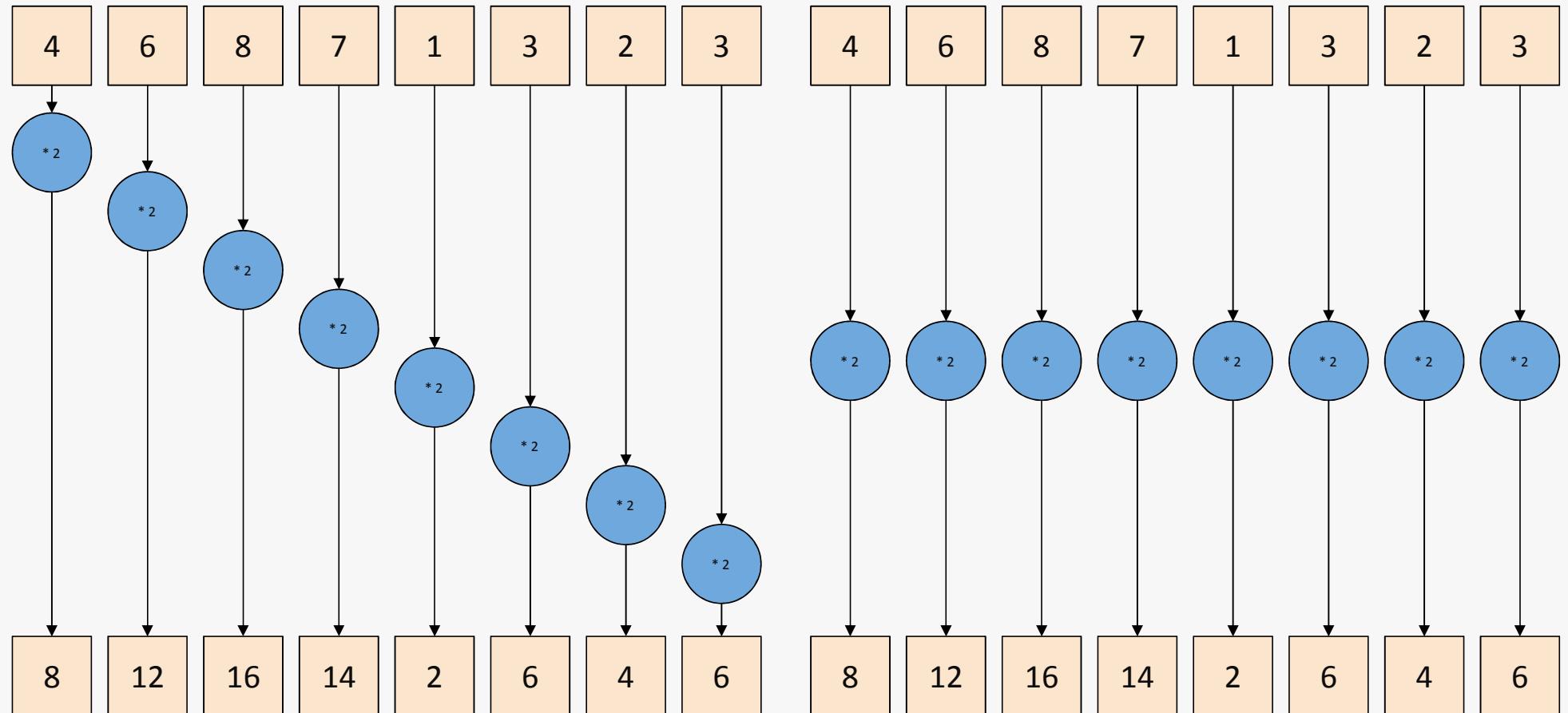


transform (non-serial)

```
d_first  
transform([execution_policy,]  
         first, last,  
         d_first,  
         unary_op)  
  
for each pair of it in [first, last) and  
d_it in [d_first, last)  
    *d_it = unary_op(*it)  
return d_first
```



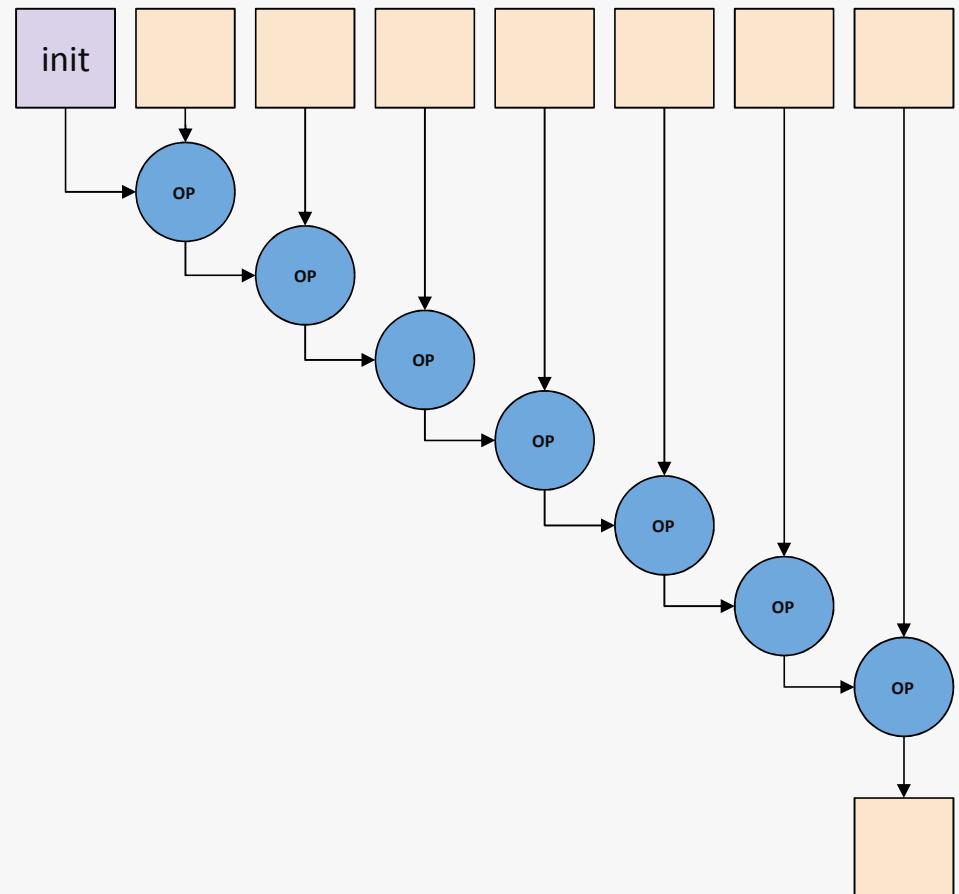
transform (non-serial)



accumulate

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

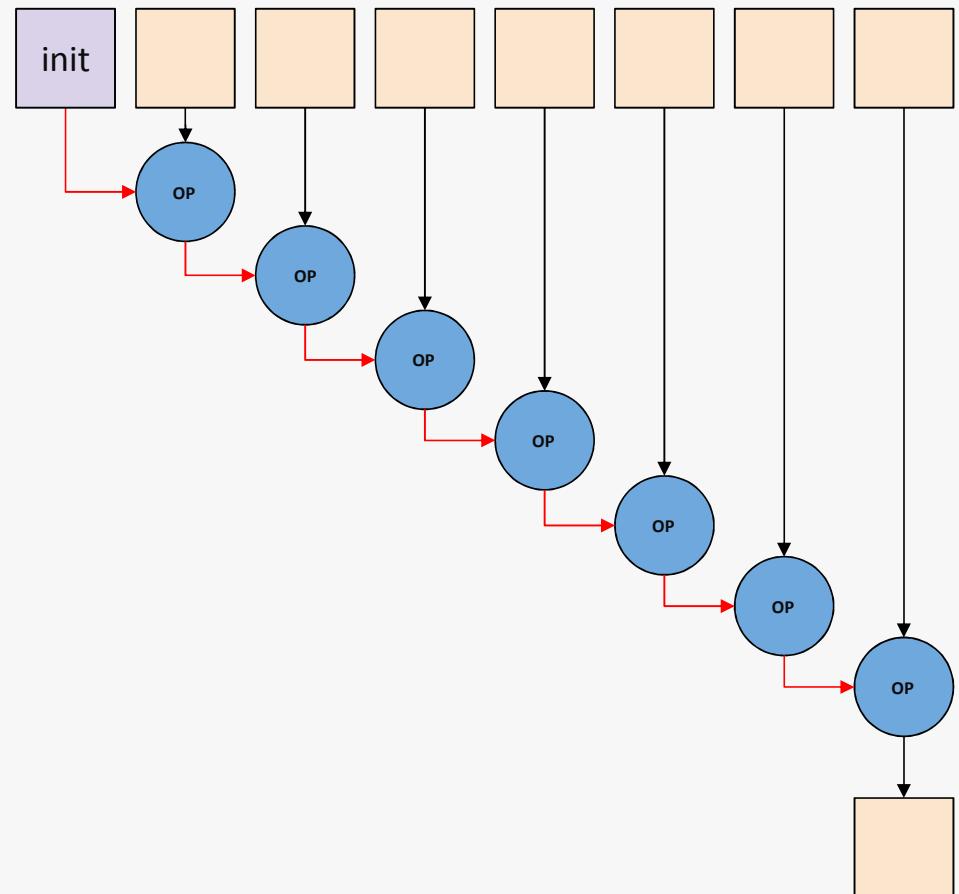
```
first acc = init  
then for each it in [first, last) in order  
    acc = binary_op(acc, *it)  
return acc
```



accumulate

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

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



reduce

```
result reduce([execution_policy,]
              first, last,
              init,
              [binary_op])  
  
acc = GSUM(binary_op, init, *first, ..., *(last-1))
```

What is this GSUM thing?

$\text{GSUM}(\text{op}, \ a^1, \ \dots, \ a^N) == \text{GNSUM}(\text{op}, \ b^1, \ \dots, \ b^N)$
where $b^1, \ \dots, \ b^N$ may be any permutation of $a^1, \ \dots, \ a^N$

$\text{GNSUM}(\text{op}, \ a^1, \ \dots, \ a^N) == a^1$, if $N == 1$

$\text{GNSUM}(\text{op}, \ a^1, \ \dots, \ a^N) == \text{op}(\text{GNSUM}(\text{op}, \ a^1, \ \dots, \ a^K),$
 $\text{GNSUM}(\text{op}, \ a^{K+1}, \ \dots, \ a^N))$, otherwise
where a^K is an arbitrary point between a^1 and a^N

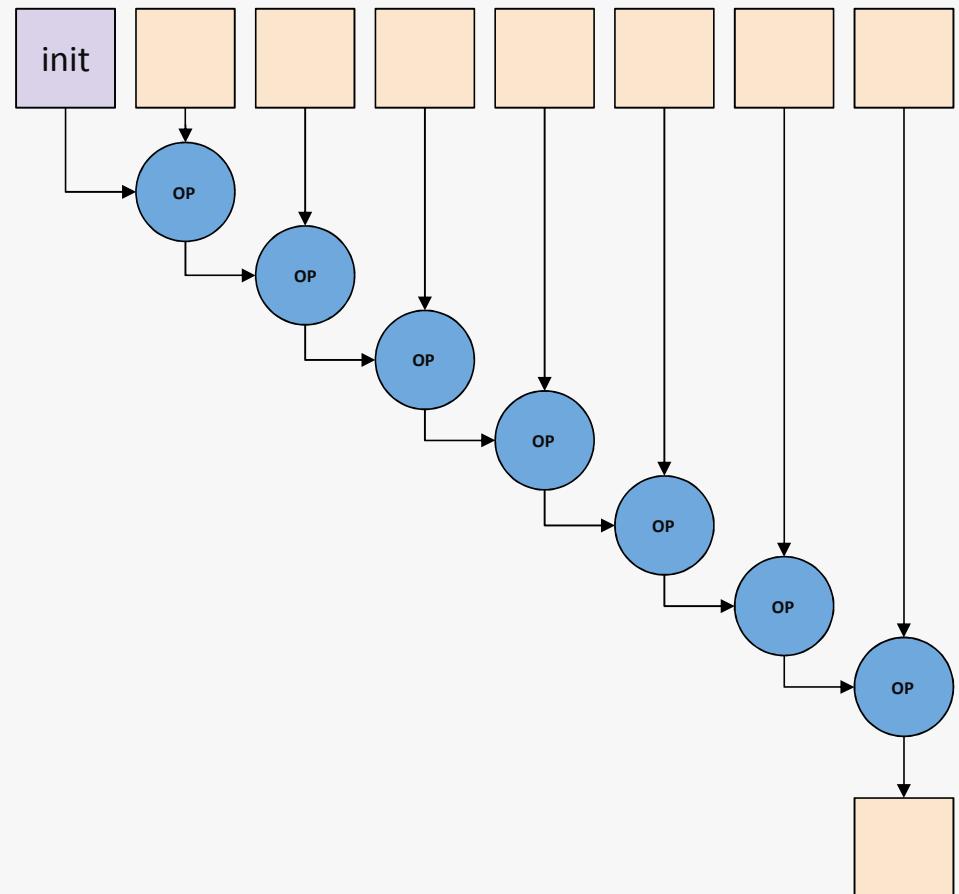
reduce

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

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

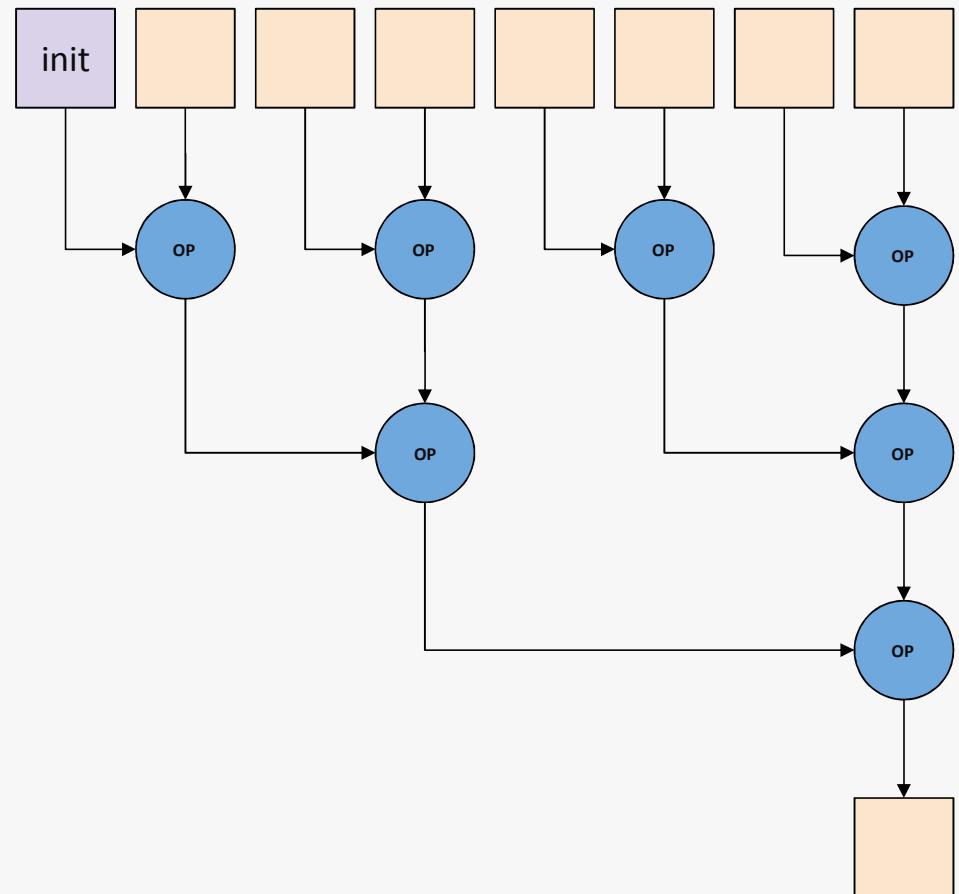
reduce

```
result reduce([execution_policy],  
             first, last,  
             init,  
             [binary_op])  
  
acc = GSUM(binary_op, init, *first,  
           ..., *(last-1))  
  
return acc
```



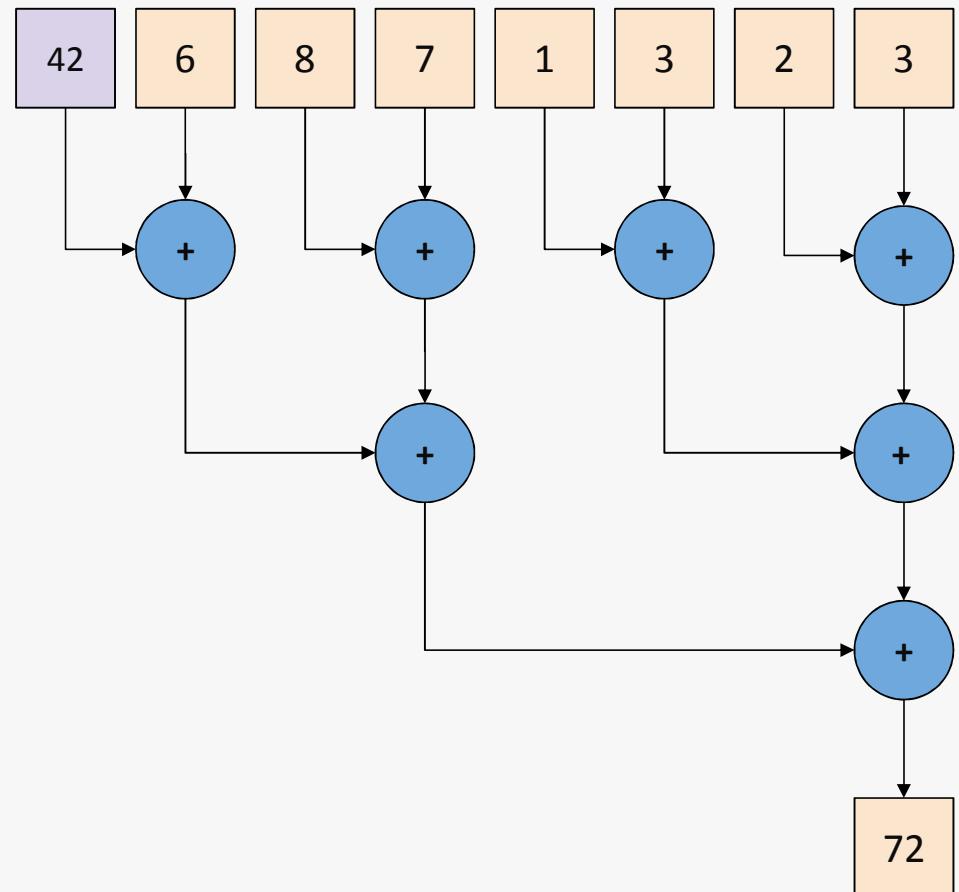
reduce

```
result reduce([execution_policy],  
             first, last,  
             init,  
             [binary_op])  
  
acc = GSUM(binary_op, init, *first,  
           ..., *(last-1))  
return acc
```

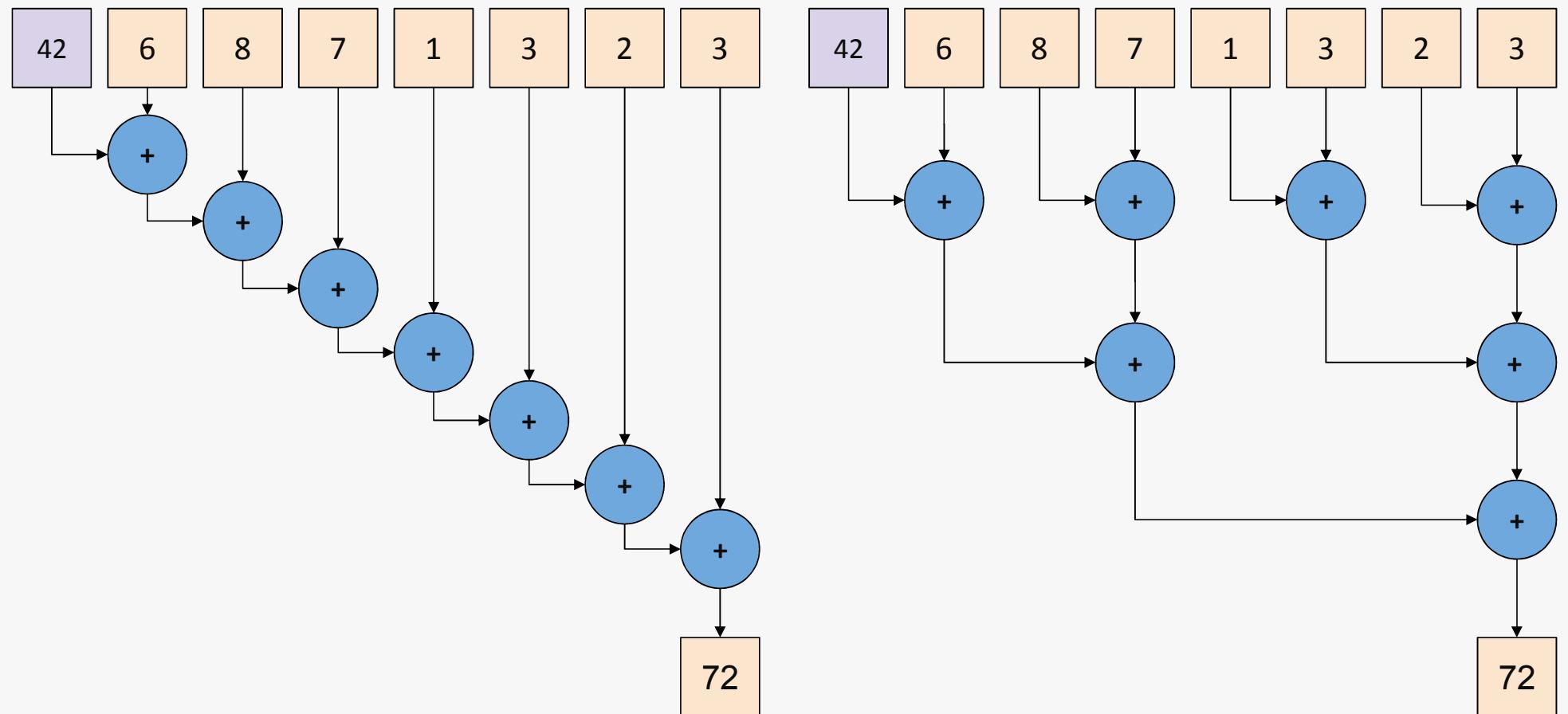


reduce

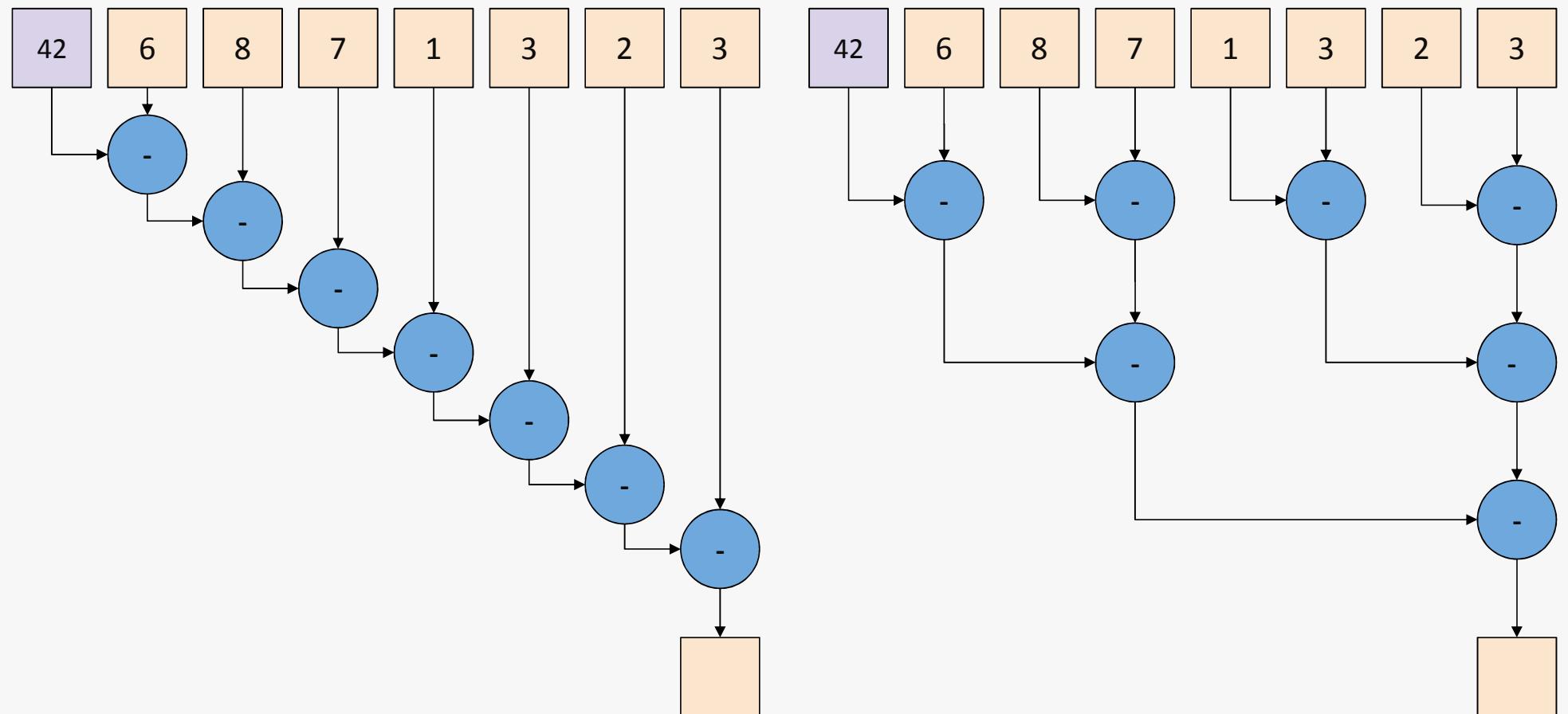
```
result reduce([execution_policy],  
             first, last,  
             init,  
             [binary_op])  
  
acc = GSUM(binary_op, init, *first,  
           ..., *(last-1))  
return acc
```



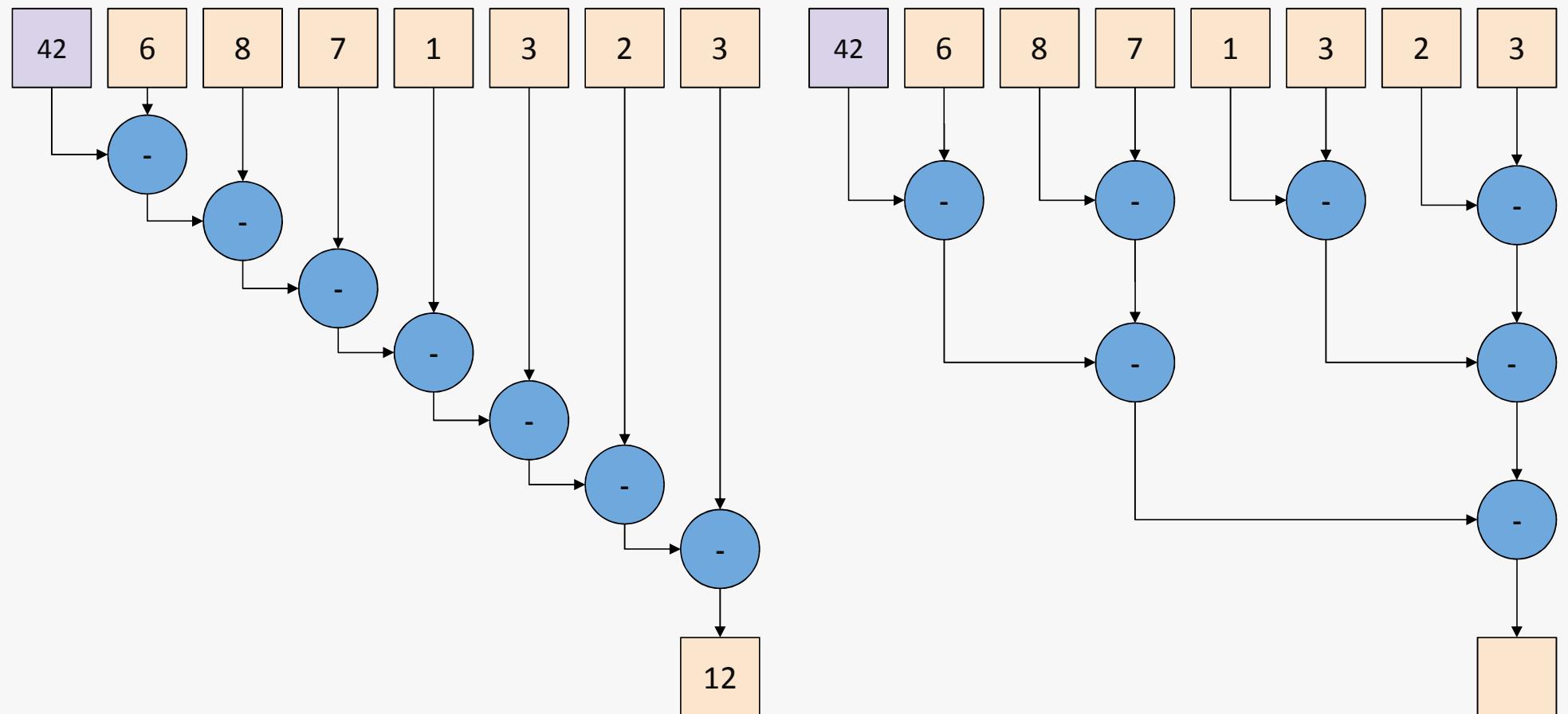
reduce



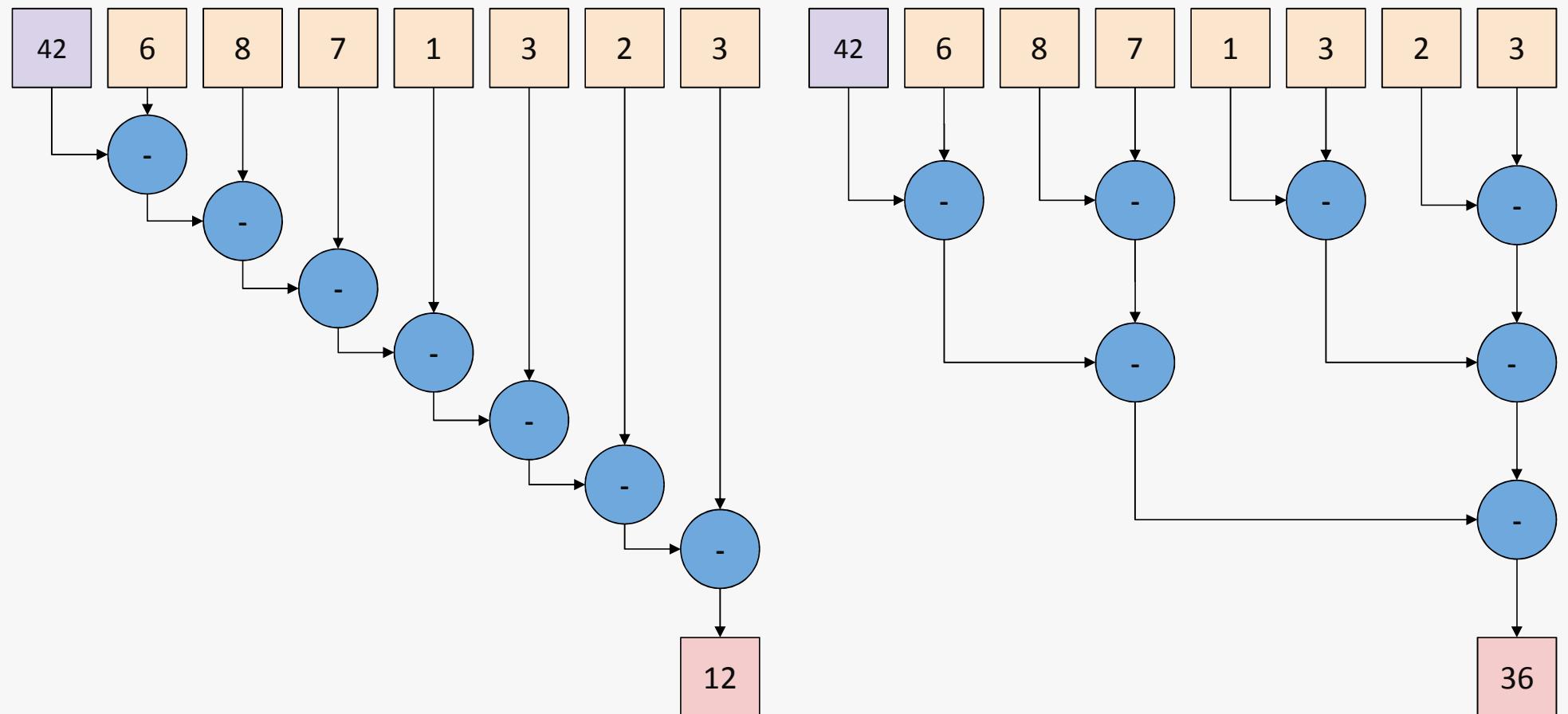
reduce



reduce



reduce



Quick maths revision

Due to the requirements of GSUM and GNSUM, reduce is allowed to be unordered

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

Commutativity

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

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

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

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

$$(x - y) - z == x - (y - z)$$

$$(x / y) / z == x / (y / z)$$

transform_reduce

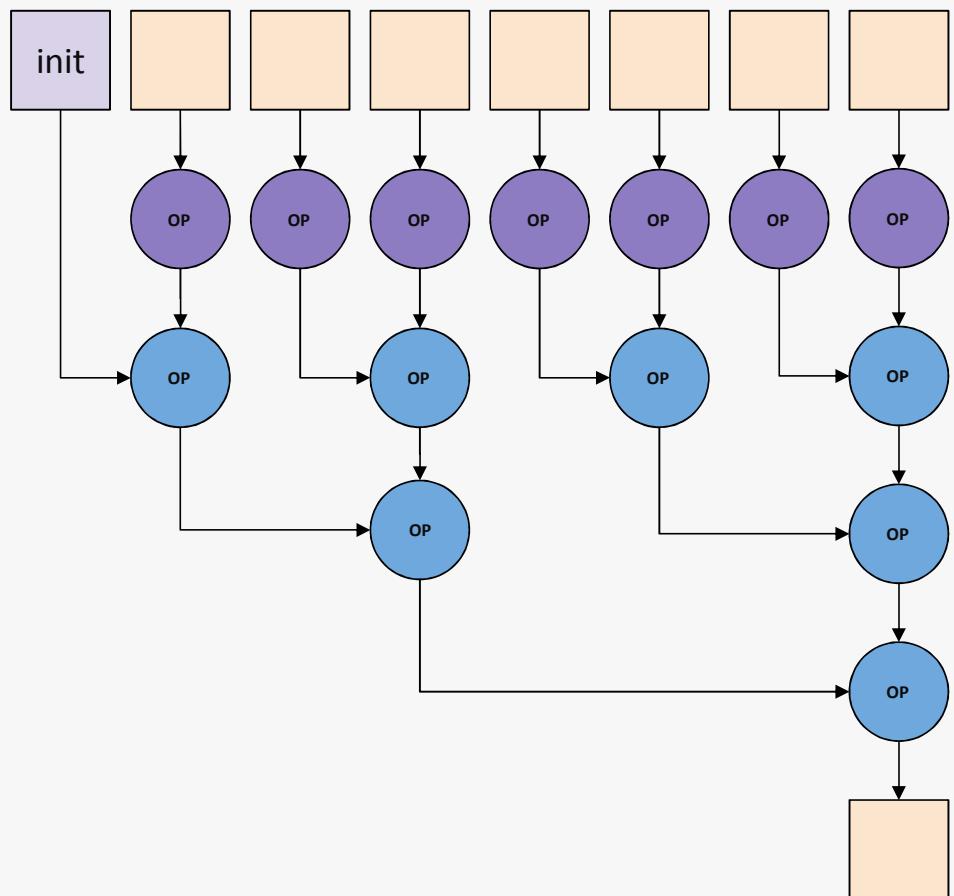
```
result reduce([execution_policy,]
             first, last,
             init,
             [binary_op,]
             [unary_op])
```

```
acc = GSUM(binary_op, init, unary_op(*first),
            ..., unary_op(*(last-1)))
return acc
```

transform_reduce

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

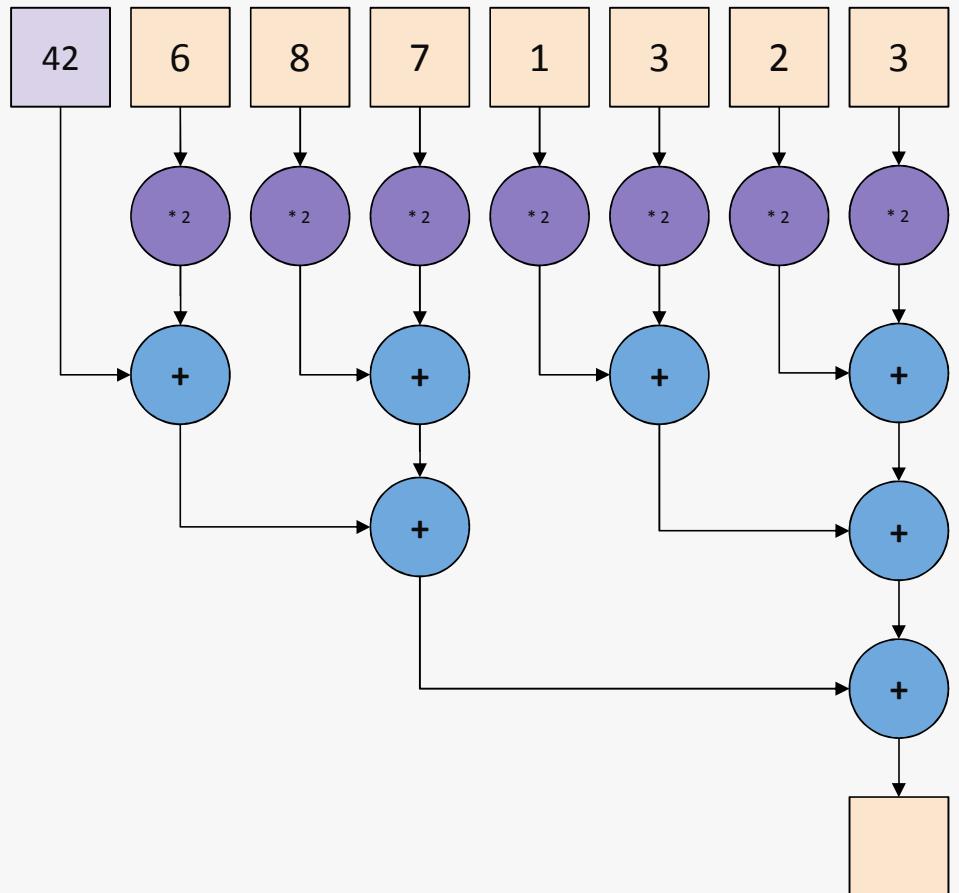
```
acc = GSUM(binary_op, init,
            unary_op(*first),
            unary_op(*last-1))  
return acc
```



transform_reduce

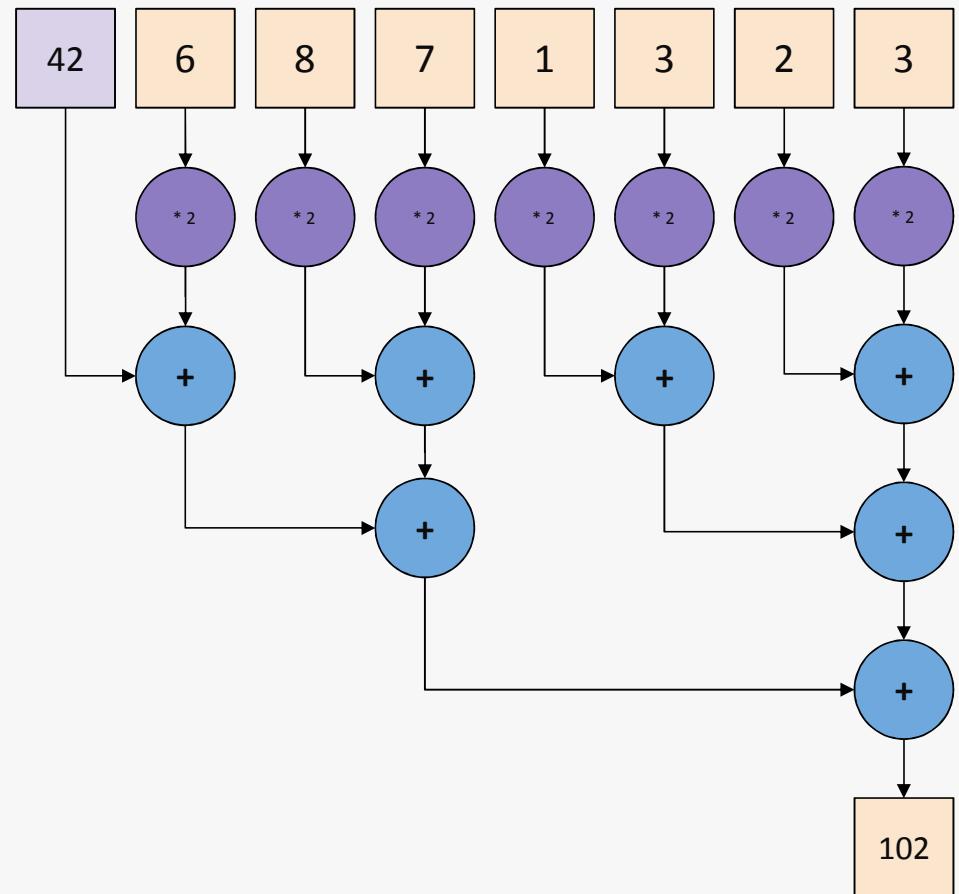
```
result reduce([execution_policy,]
             first, last,
             init,
             [binary_op,]
             [unary_op])
```

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

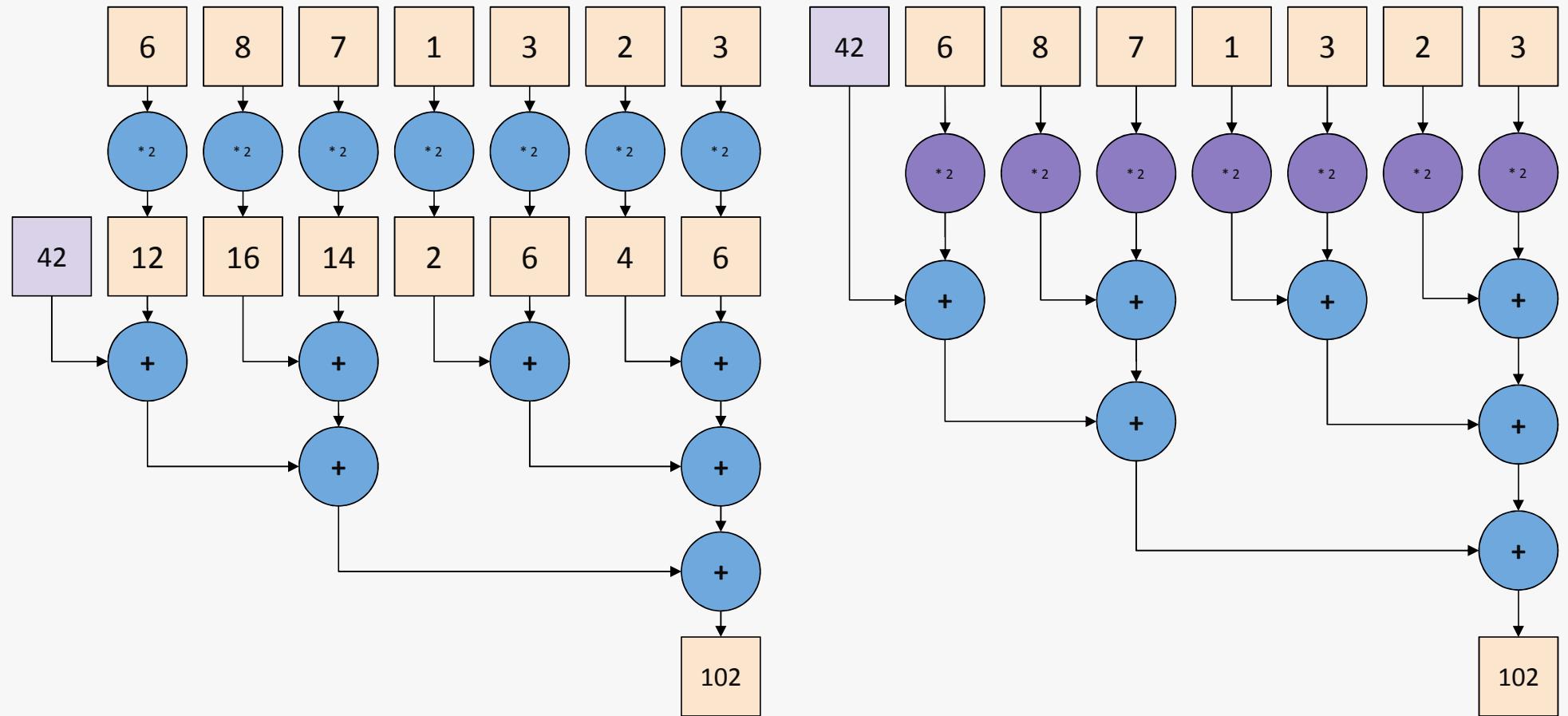


transform_reduce

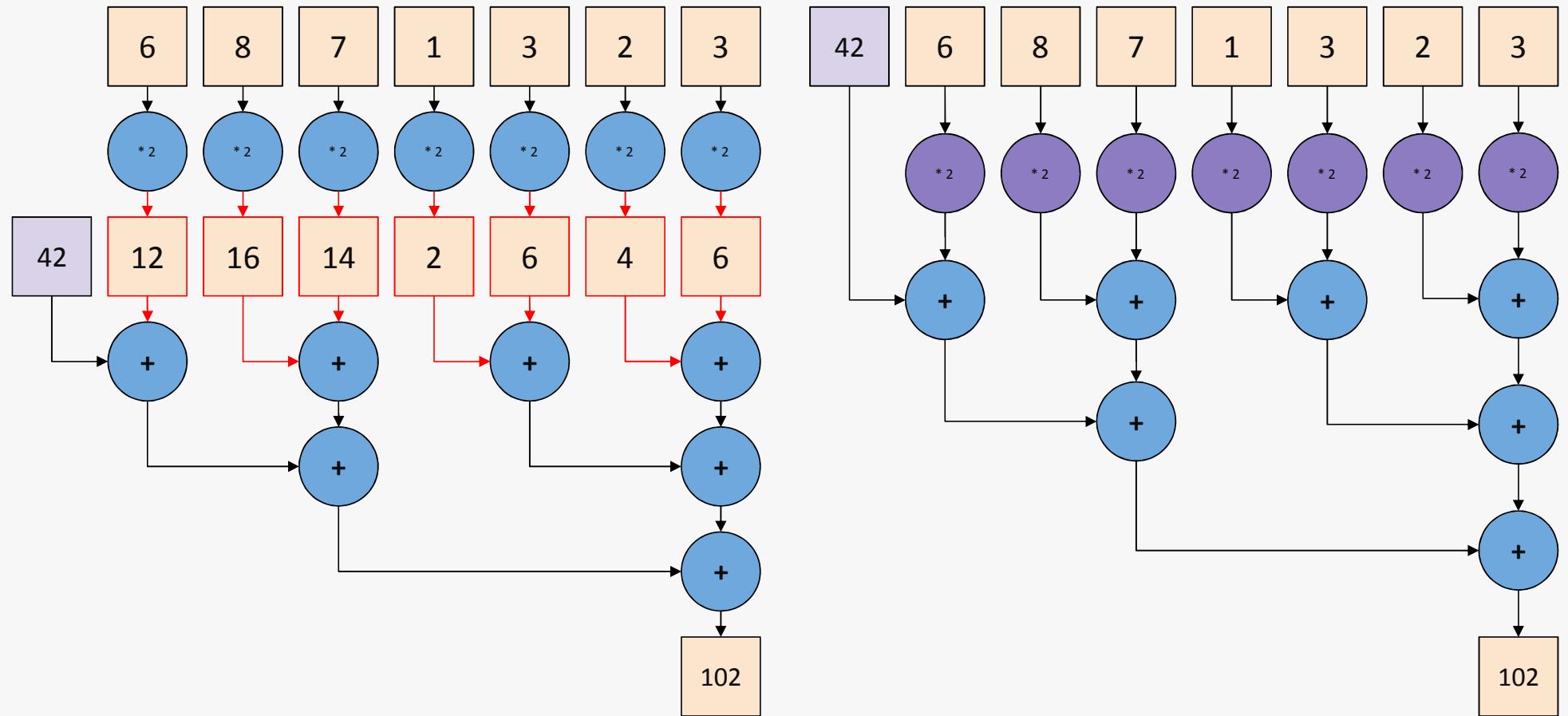
```
result reduce([execution_policy],  
             first, last,  
             init,  
             [binary_op],  
             [unary_op])  
  
acc = GSUM(binary_op, init,  
           unary_op(*first),  
           unary_op(*last-1))  
return acc
```



Benefit of fused transform & reduce

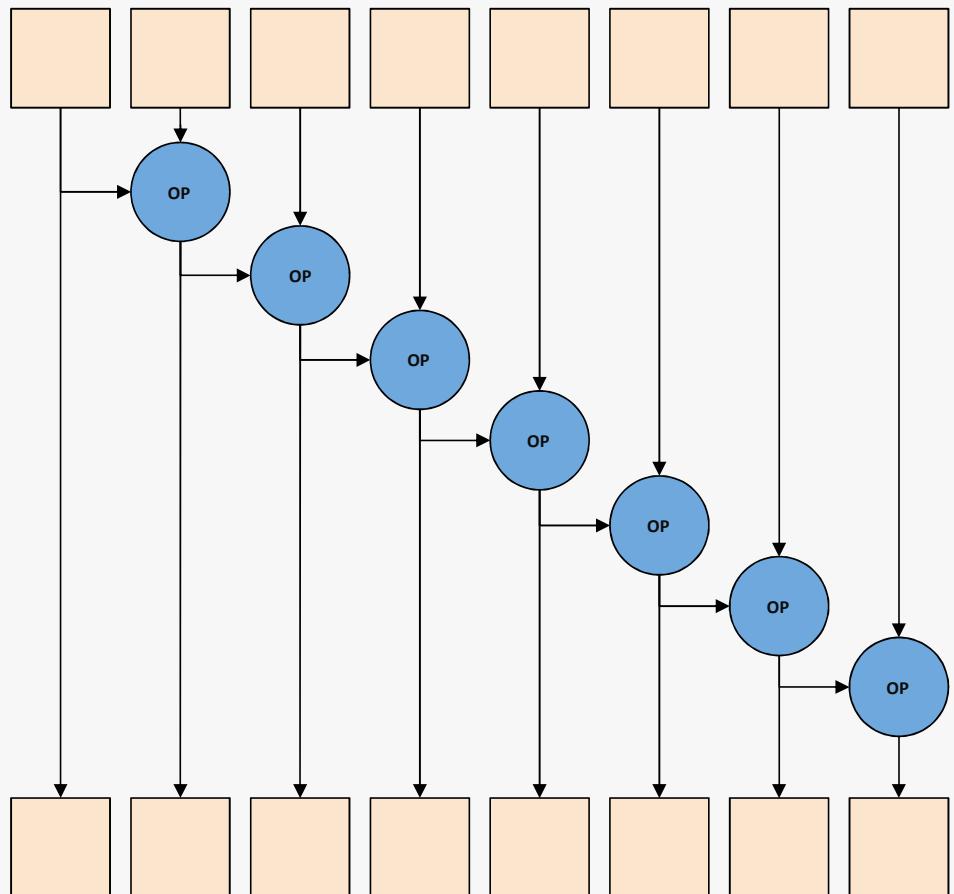


Benefit of fused transform & reduce



partial_sum

```
d_first partial_sum(first, last,
                     d_first,
                     [binary_op])
first sum = *first++
*d_first++ = sum
for each pair of it in [first, last) and
d_it in [d_first, last) in order
    sum = binary_op(sum, *it)
    *d_it = sum
return d_first
```



inclusive_scan

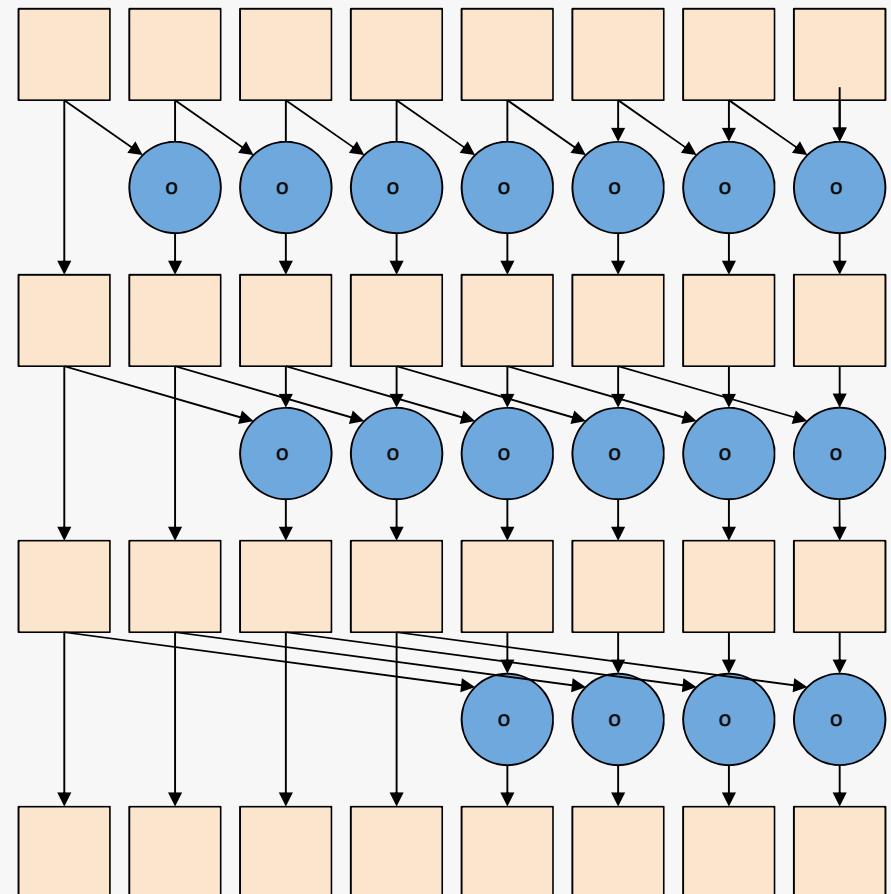
```
d_first partial_sum([execution_policy,]
                     first, last,
                     d_first,
                     [binary_op],
                     [init])
```

```
/* very complicated :(
return d_first
```

inclusive_scan

```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init])
```

```
/* very complicated :( */  
return d_first
```

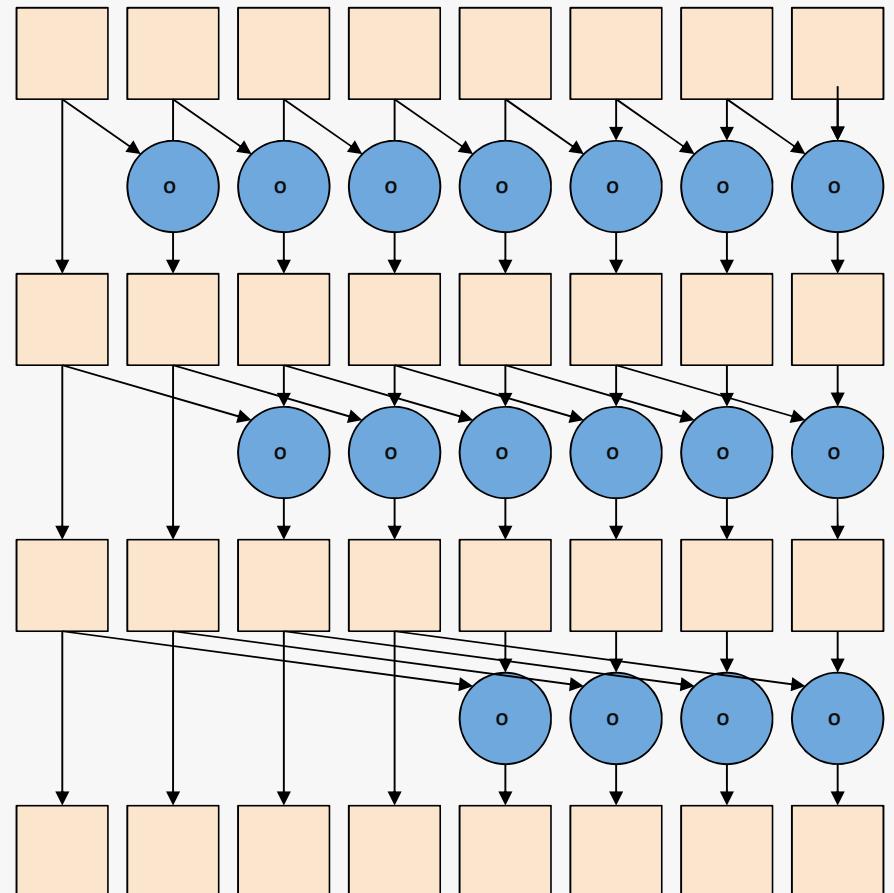


inclusive_scan

```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init]))
```

```
/* very complicated :( */  
return d_first
```

Hillis & Steele Scan

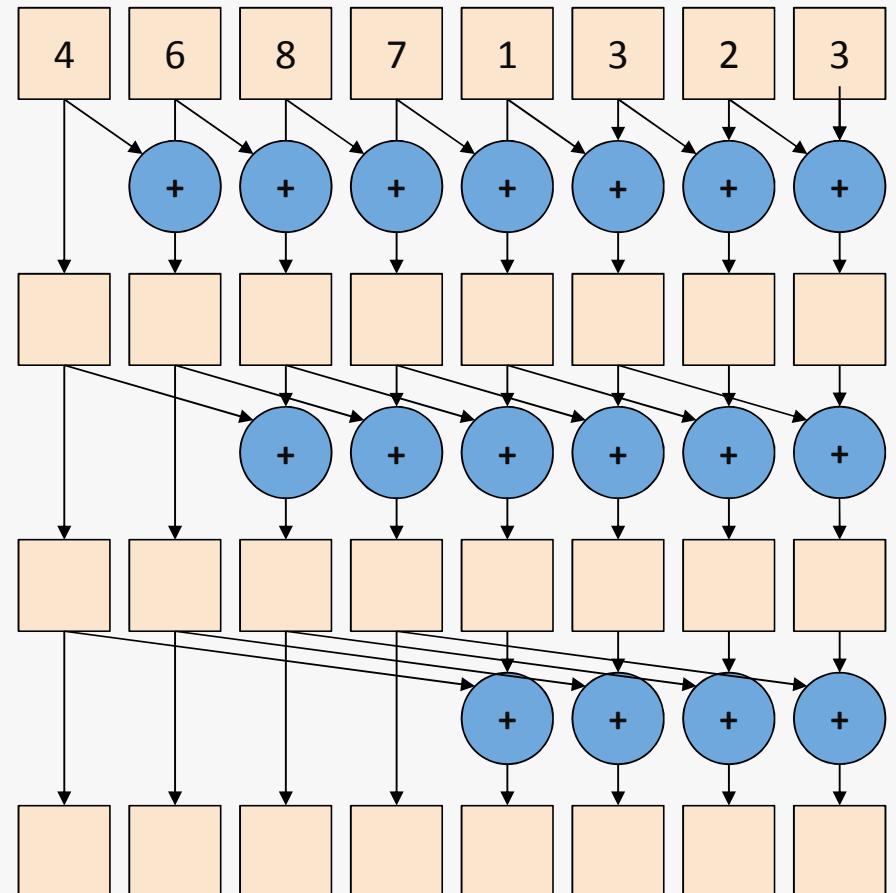


inclusive_scan

```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init])
```

```
/* very complicated :( */  
return d_first
```

Hillis & Steele Scan

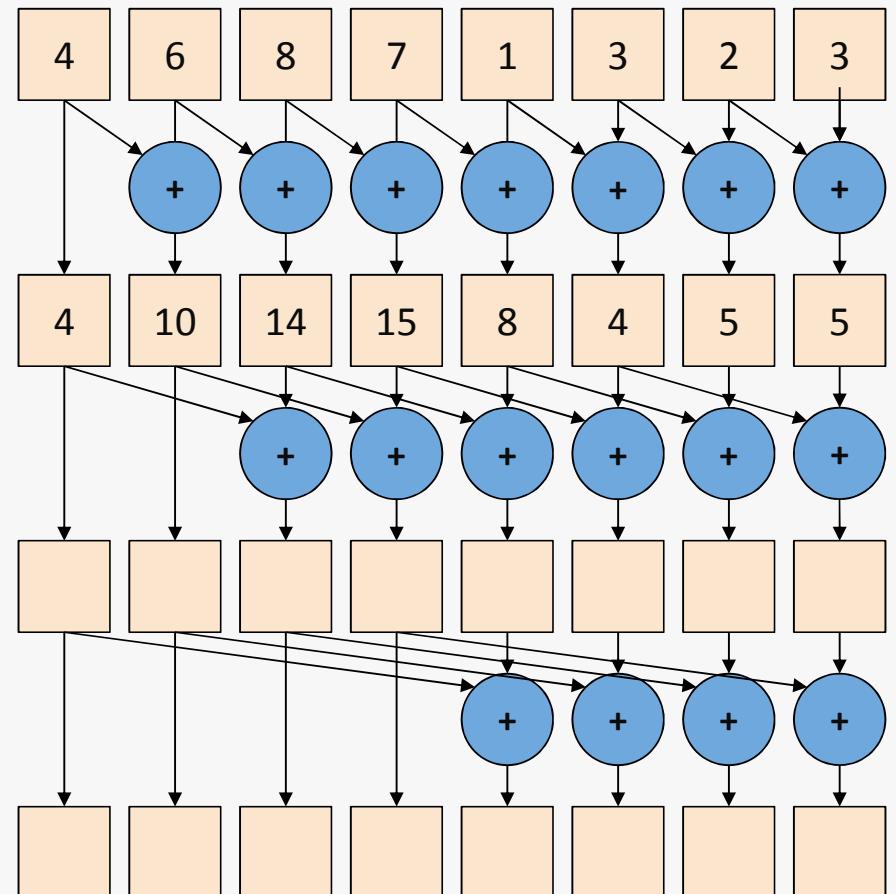


inclusive_scan

```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init])
```

```
/* very complicated :( */  
return d_first
```

Hillis & Steele Scan

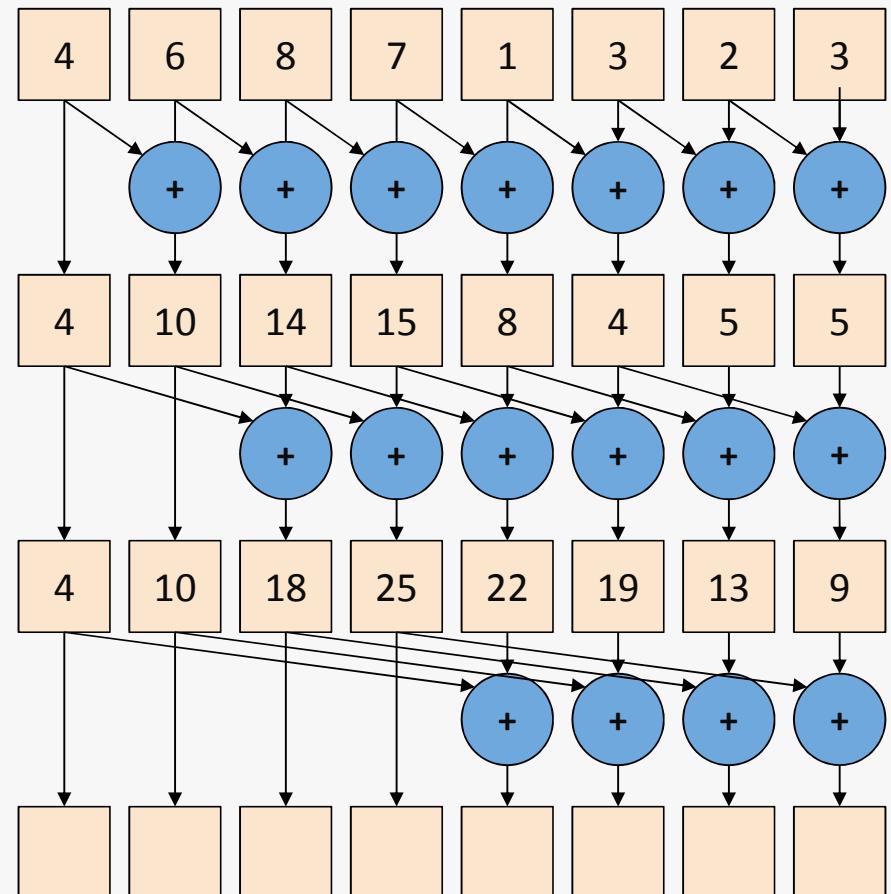


inclusive_scan

```
d_first  
partial_sum([execution_policy,]  
           first, last,  
           d_first,  
           [binary_op],  
           [init])
```

```
/* very complicated :( */  
return d_first
```

Hillis & Steele Scan

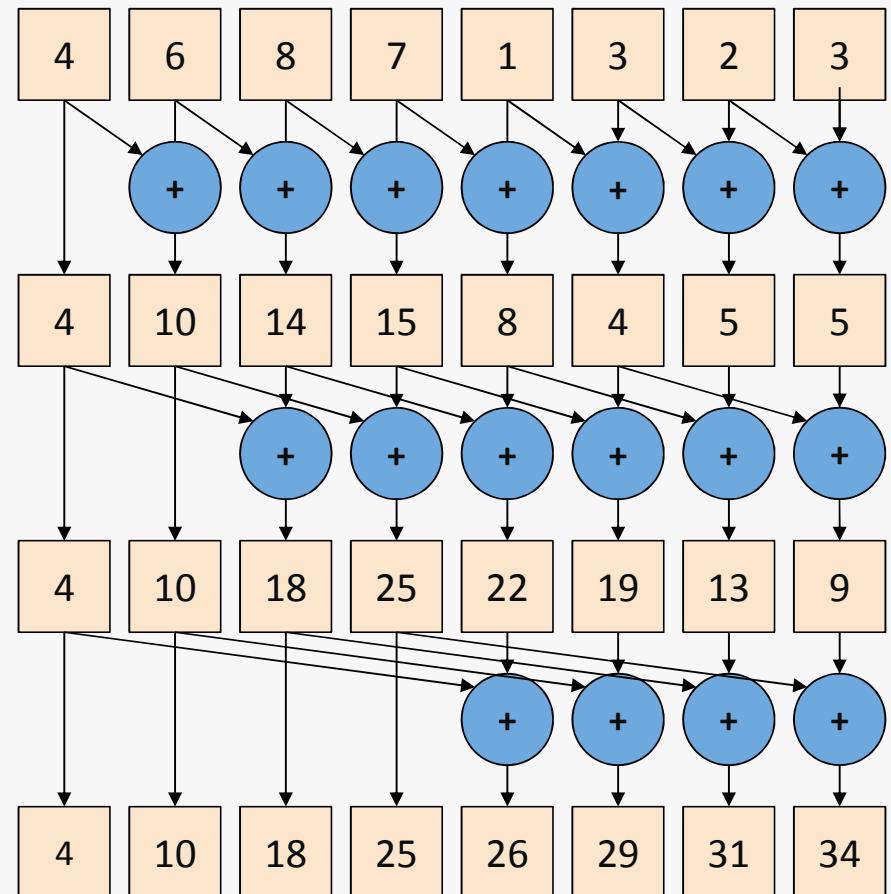


inclusive_scan

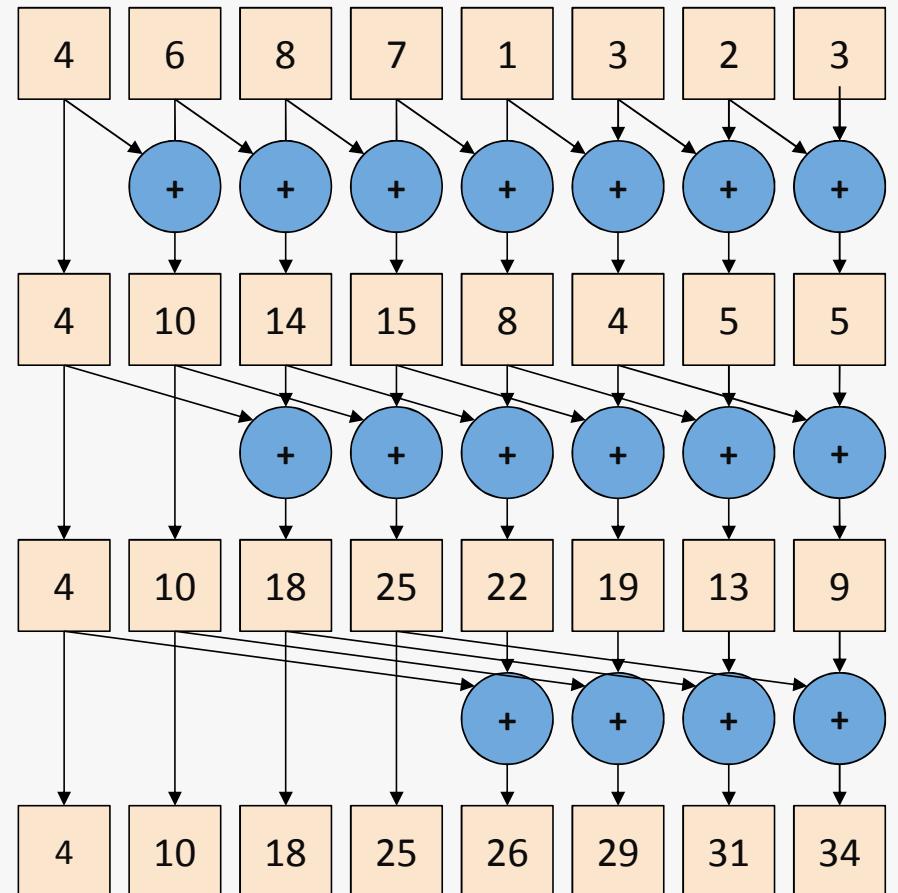
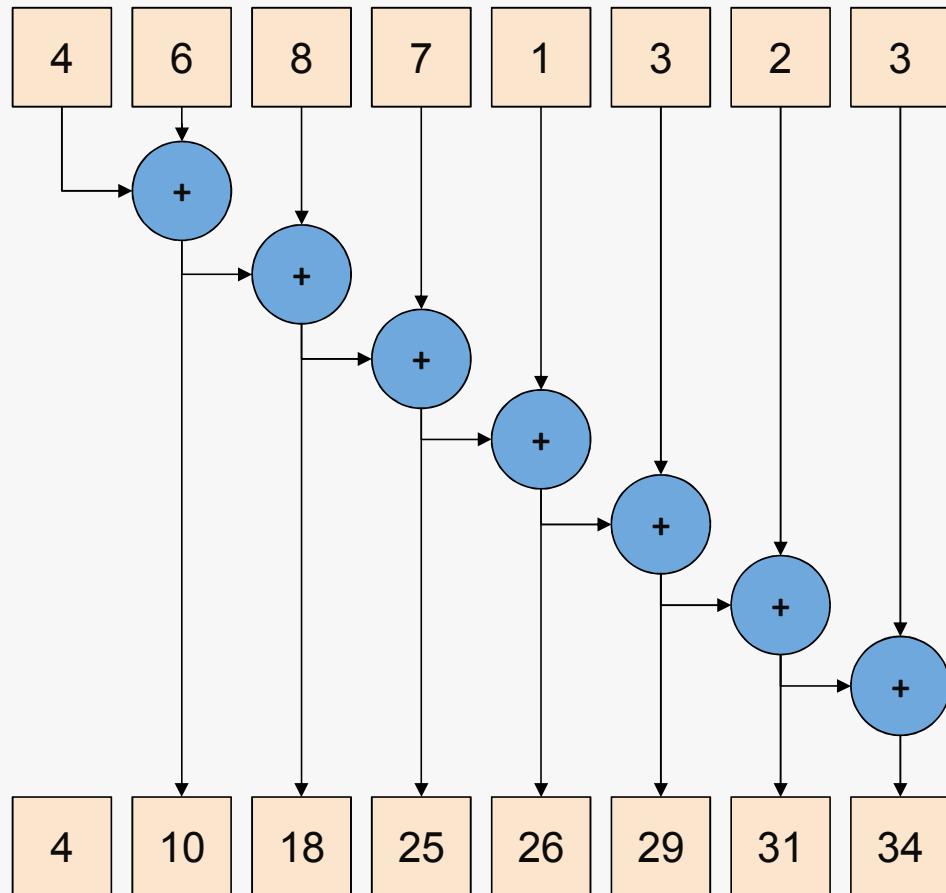
```
d_first  
partial_sum([execution_policy,  
            first, last,  
            d_first,  
            [binary_op],  
            [init])
```

```
/* very complicated :( */  
return d_first
```

Hillis & Steele Scan



inclusive_scan



inclusive_scan

The `binary_op` for `inclusive_scan` is also required to be associative

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

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

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

$$(x - y) - z == x - (y - z)$$

$$(x / y) / z == x / (y / z)$$

A note about iterators

`transform(serial)`, `accumulate`, `partial_sum` use **Input/Output** iterators:

- Increment (without multiple passes)

A note about iterators

`transform` (serial), `accumulate`, `partial_sum` use **Input/Output** iterators:

- Increment (without multiple passes)

`transform` (unordered), `reduce`, `transform_reduce`, `partial_sum` use **Forward** iterators:

- Increment (with multiple passes)

A note about iterators

`transform` (serial), `accumulate`, `partial_sum` use **Input/Output** iterators:

- Increment (without multiple passes)

`transform` (unordered), `reduce`, `transform_reduce`, `partial_sum` use **Forward** iterators:

- Increment (with multiple passes)

When moving data (important for GPUs) use **Contiguous** iterators:

- All elements in the range are laid out contiguously in memory

Measuring algorithm performance

When parallelizing a piece of code it's important to measure any performance benefit

Measuring algorithm performance

When parallelizing a piece of code it's important to measure any performance benefit

You want to isolate the part of the code that you are interested in measuring to ensure that unrelated parts of the code do not influence the measurement

Measuring algorithm performance

When parallelizing a piece of code it's important to measure any performance benefit

You want to isolate the part of the code that you are interested in measuring to ensure that unrelated parts of the code do not influence the measurement

You also want to run the code multiple times and calculate the average to ensure you get a more accurate representation of the execution time

Measuring algorithm performance

Ensure that there are no background processes

- If the OS switches out the hardware threads that are running your benchmark then it can affect the measurement
- You can often ensure this using command line options to bind your process to particular threads

Measuring algorithm performance

Ensure that there are no background processes

- If the OS switches out the hardware threads that are running your benchmark then it can affect the measurement
- You can often ensure this using command line options to bind your process to particular threads

Ensure that you run the benchmark with a range of different input sizes and operation complexities

- There is always overhead involved in parallelism, so sometimes the overhead can affect the performance

Measuring algorithm performance

Be aware of the hardware that you are running on

- The performance can vary depending on the hardware
- Different hardware will have varying overhead in launching work and copying data to where the work is being done

Measuring algorithm performance

Be aware of the hardware that you are running on

- The performance can vary depending on the hardware
- Different hardware will have varying overhead in launching work and copying data to where the work is being done

Always run in release mode

- Debug mode will not give accurate performance evaluation
- This is due to additional code that is generated for debug builds

Measuring algorithms

```
template <typename Func>
auto benchmark(Func &&func, int iterations, std::string
caption)
    std::chrono::duration<double, std::milli>
        totalTime{0};
    for (int i = 0; i < iterations; i++) {
        std::chrono::steady_clock::time_point start =
            std::chrono::steady_clock::now();
        func();
        std::chrono::steady_clock::time_point end =
            std::chrono::steady_clock::now();
        totalTime += (end - start);
    }
    return averageTime;
}
```

This is a function which uses std::chrono to time a number of iterations of a function and calculate the average

Key takeaways

C++17 introduces new unordered and fused algorithms and execution policies to provide the opportunity for parallelism

Unordered algorithms introduce new considerations to how they are implemented and used, and furthermore how they can be parallelised

There are many important things to consider when benchmarking the performance of an algorithm



Questions?