



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

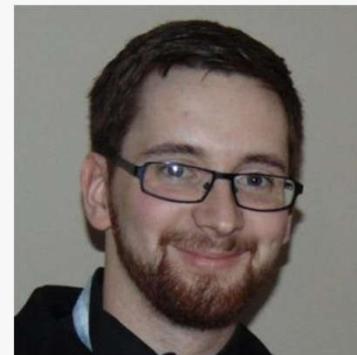
Customers



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: Transpose Chapter 14: Parallel Algorithms



Questions?



Chapter 1: Importance of Parallelism

Gordon Brown

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about the current landscape of computer architectures
 - Learn about the performance benefits of parallelism

Imagine you need to dig a hole...



But you want to get it done faster...

Let's say you have three options:



But you want to get it done faster...

Let's say you have three options:

- Simply dig faster with the one shovel you have



But you want to get it done faster...

Let's say you have three options:

- Simply dig faster with the one shovel you have
- Buy a better shovel that moves more dirt



But you want to get it done faster...

Let's say you have three options:

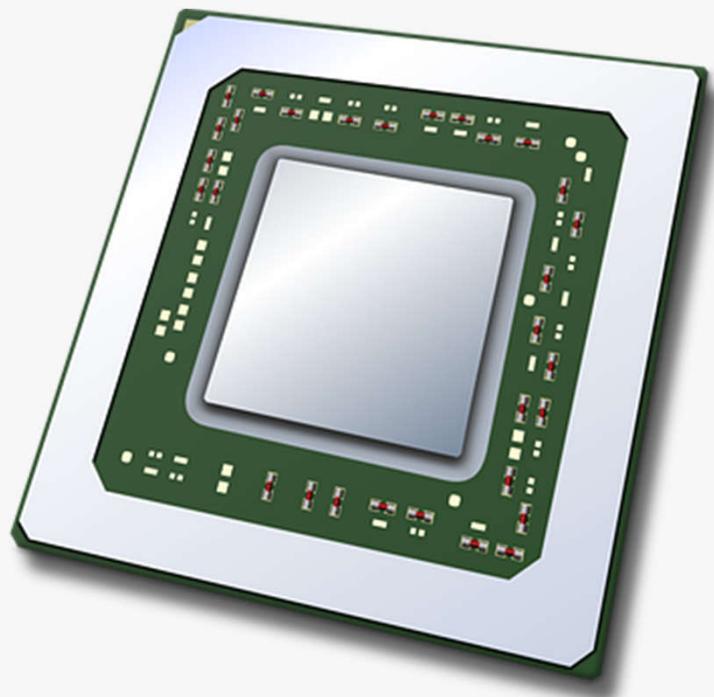
- Simply dig faster with the one shovel you have
- Buy a better shovel that moves more dirt
- Hire additional people to help you dig



This is analogous to processor design



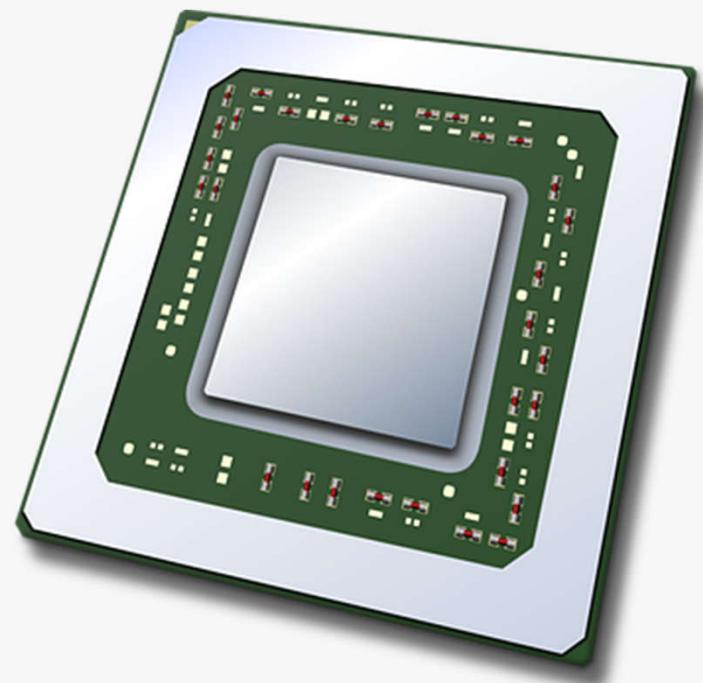
=



When you need a processor to run faster...

You have three options:

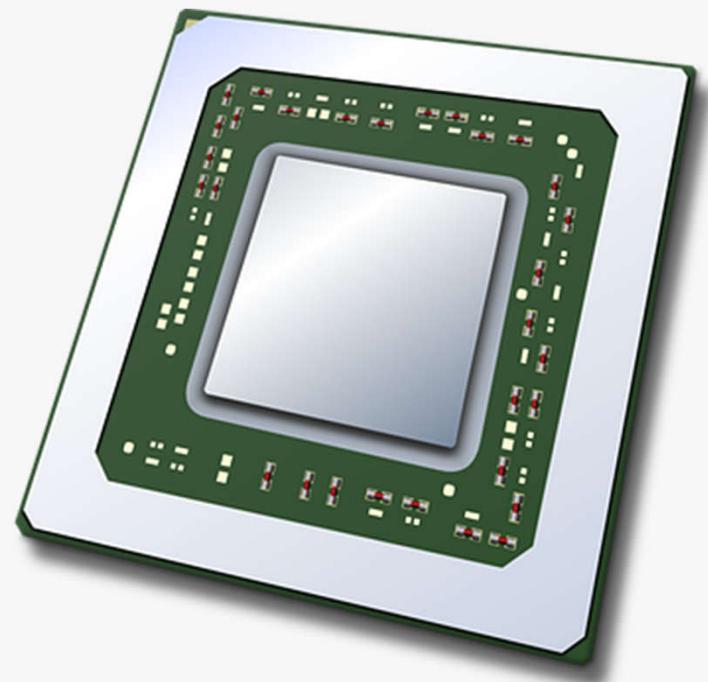
- Run at a higher clock speed



When you need a processor to run faster...

You have three options:

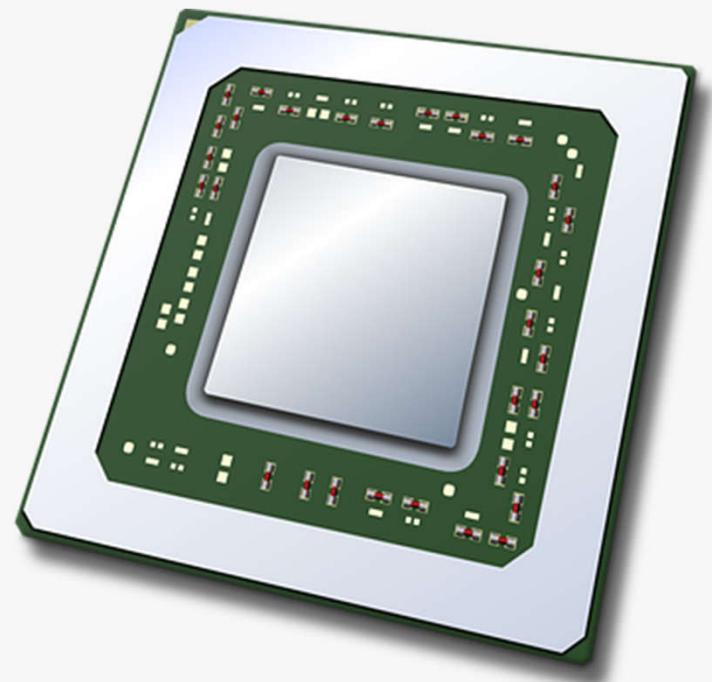
- Run at a higher clock speed
- Do more work on each clock cycle



When you need a processor to run faster...

You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

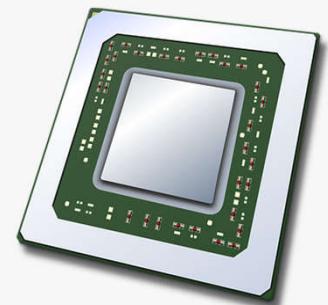


When you need a processor to run faster...

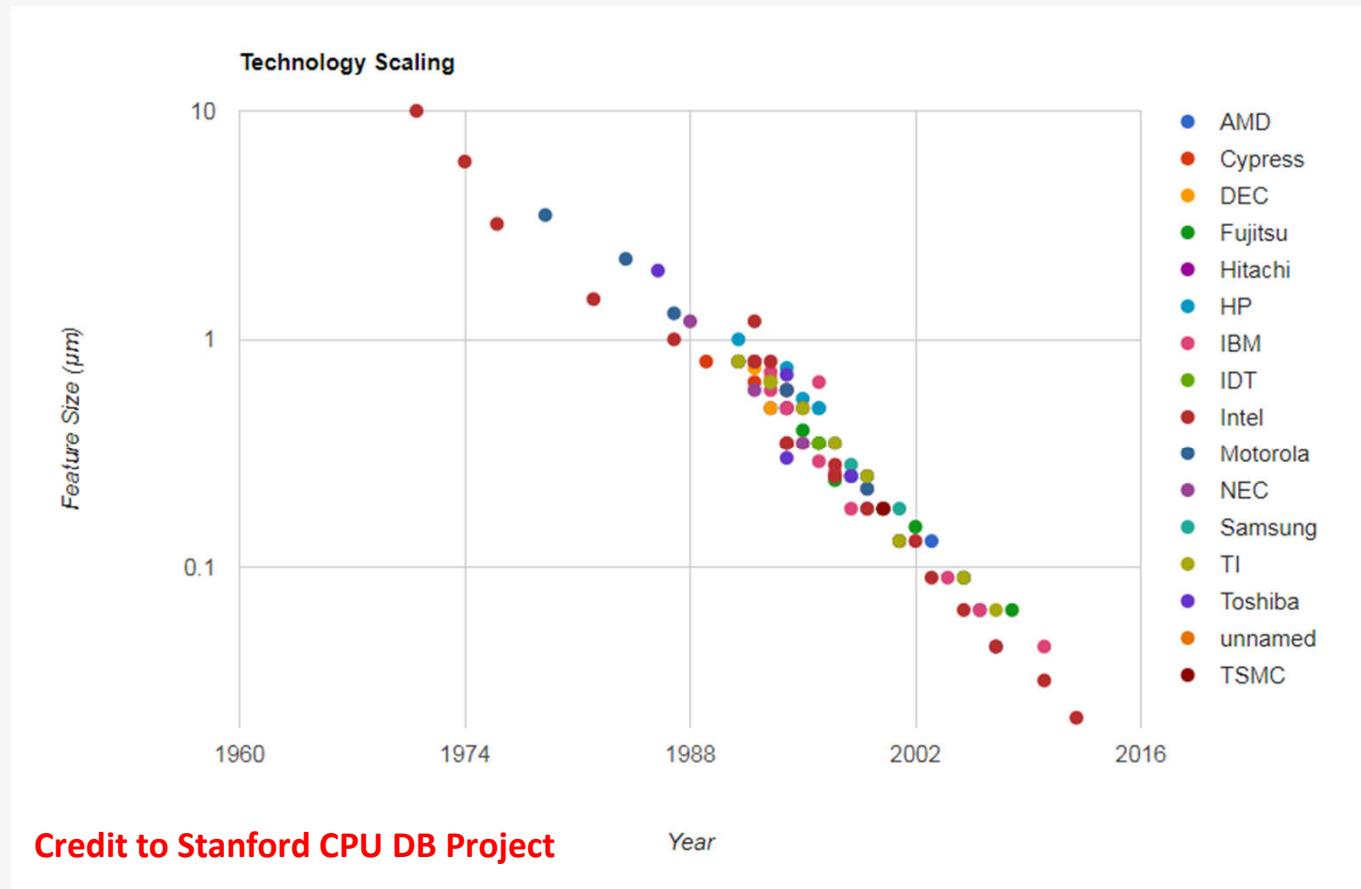
You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

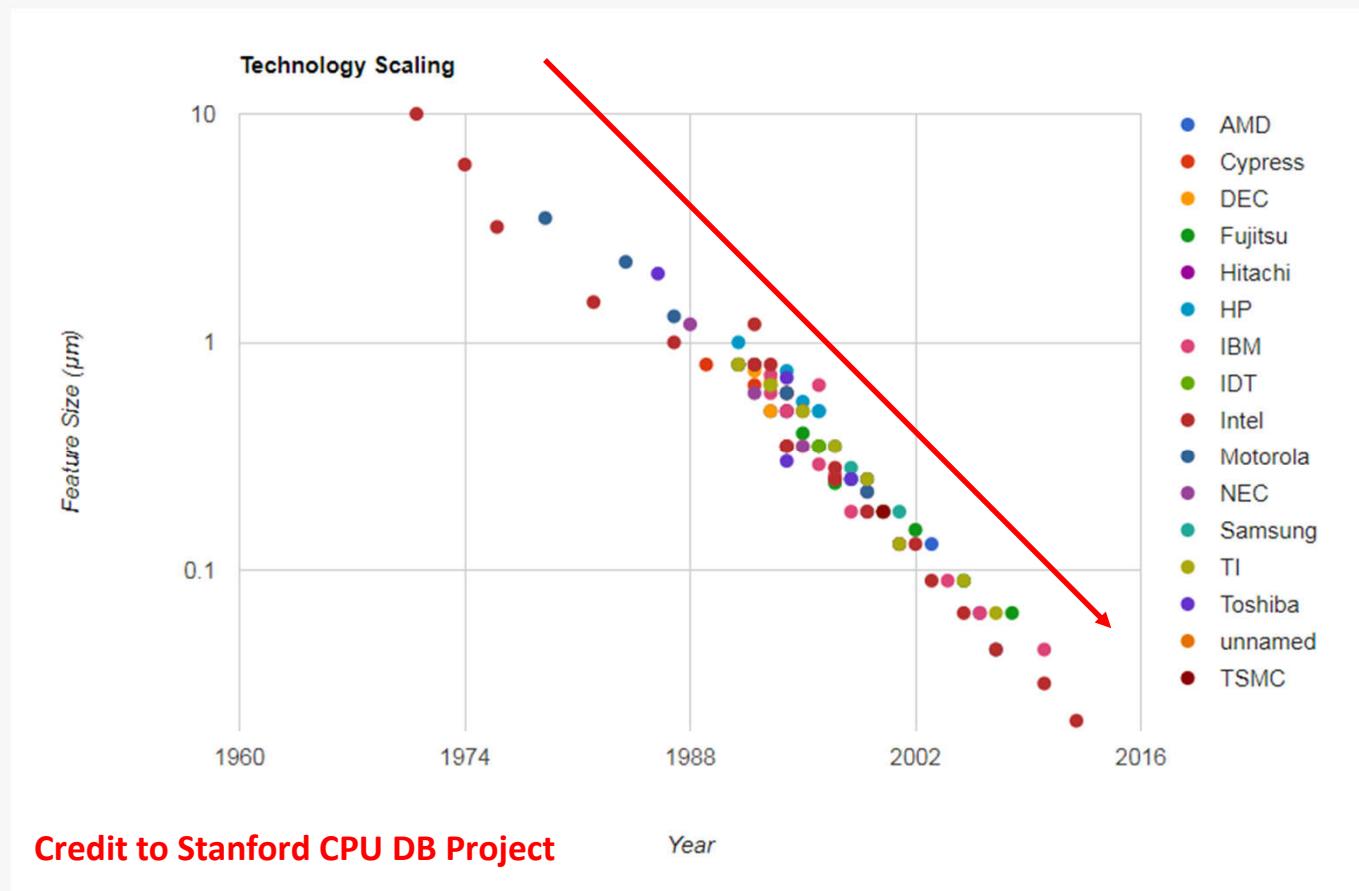
Increasing the clock speed of modern processors increases power consumption



The problem...

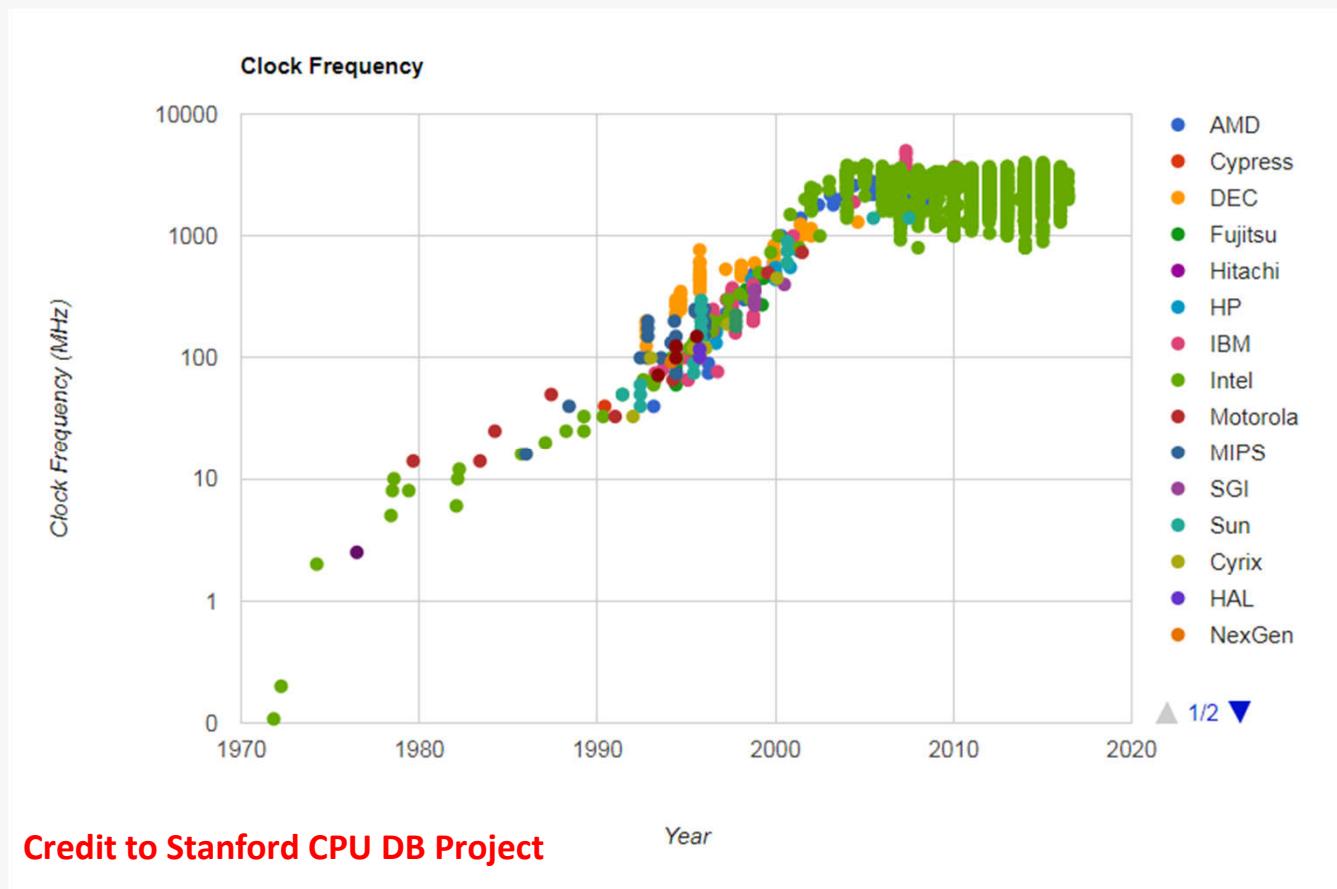


The problem...

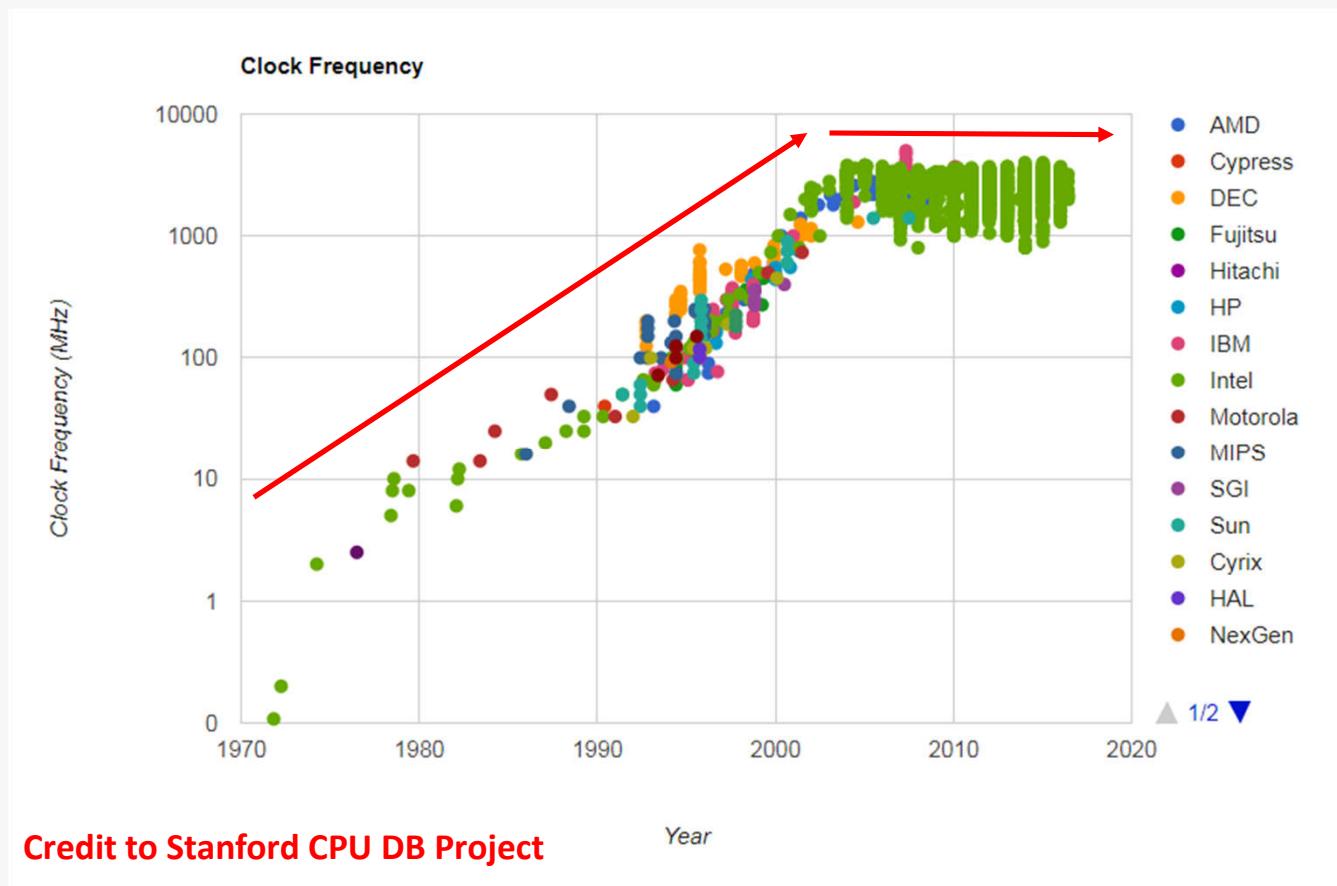


The size of transistors
are continuing to
decrease

The problem...

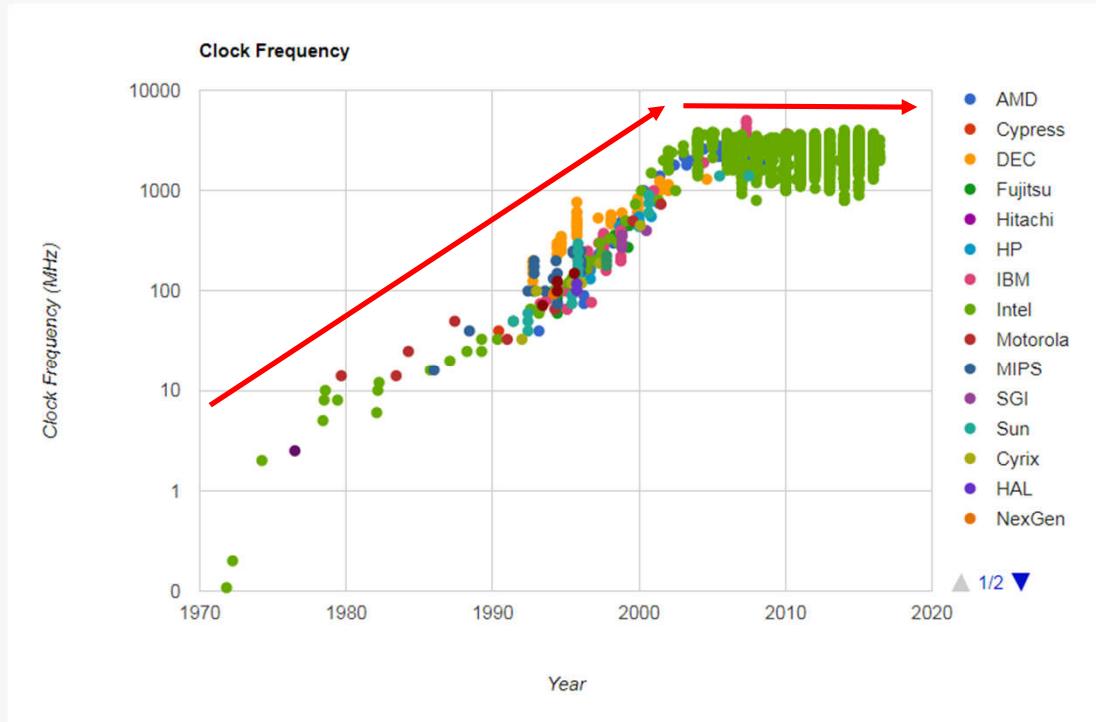


The problem...



However in recent years CPU clock speeds have stopped increasing

CPU clock speeds have stopped increasing despite transistor sizes continuing to decrease, why is this?



The problem is power

- Fitting more transistors on a single processor
 - Requires more power
 - Which generates more heat
- Cooling these devices becomes the limiting factor in processor design
 - Applies to everything from HPC to mobile devices



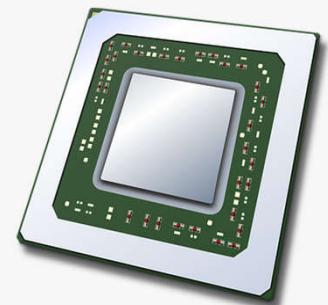
When you need a processor to run faster...

You have three options:

- Run at a higher clock speed
- **Do more work on each clock cycle**
- Have multiple processors work in parallel

Making use of more powerful instructions can provide a performance gain

However there's a limit to how much instruction level parallelism can be achieved

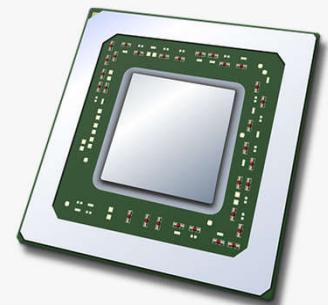


When you need a processor to run faster...

You have three options:

- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel

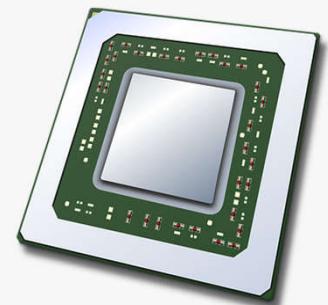
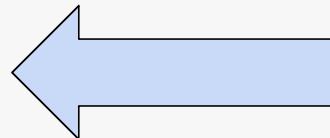
Having a large number of smaller processors execute in parallel can provide additional performance gain



When you need a processor to run faster...

You have three options:

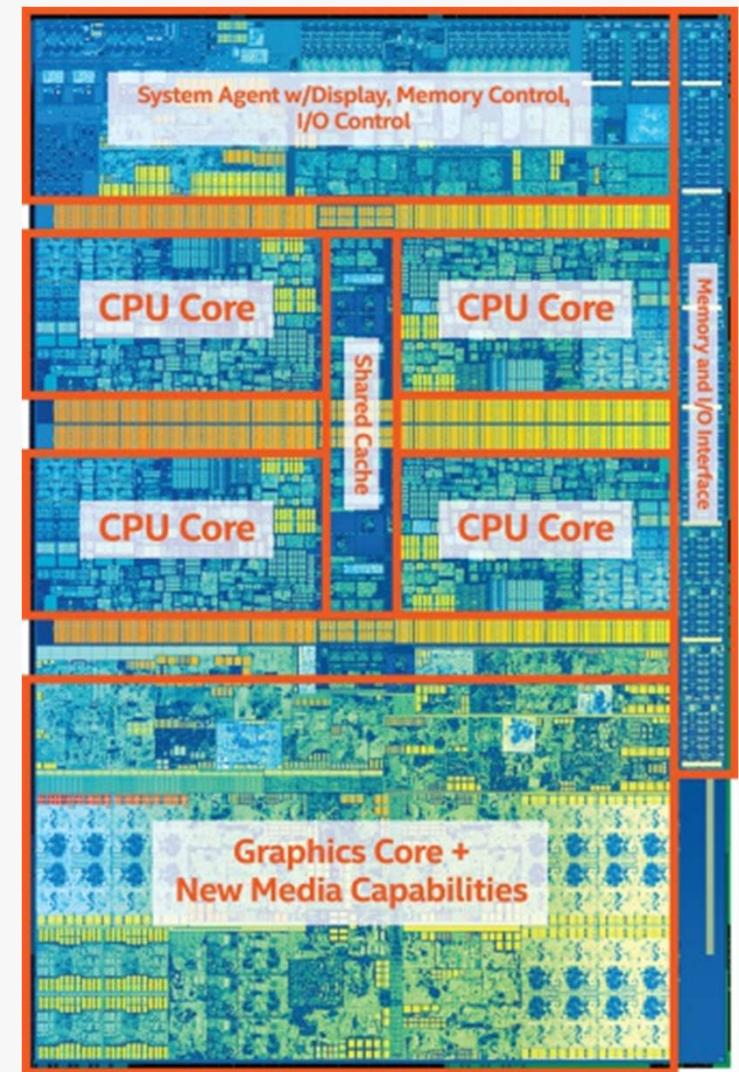
- Run at a higher clock speed
- Do more work on each clock cycle
- Have multiple processors work in parallel



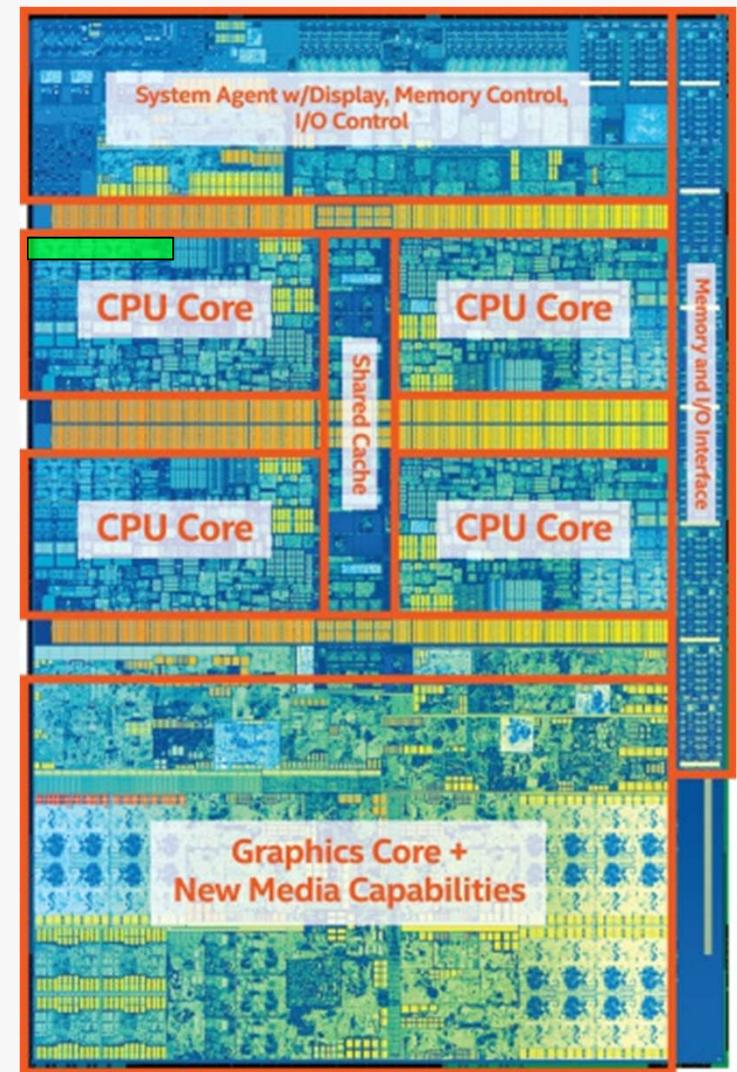
Take a typical Intel chip...

Intel Core i7 7th Gen

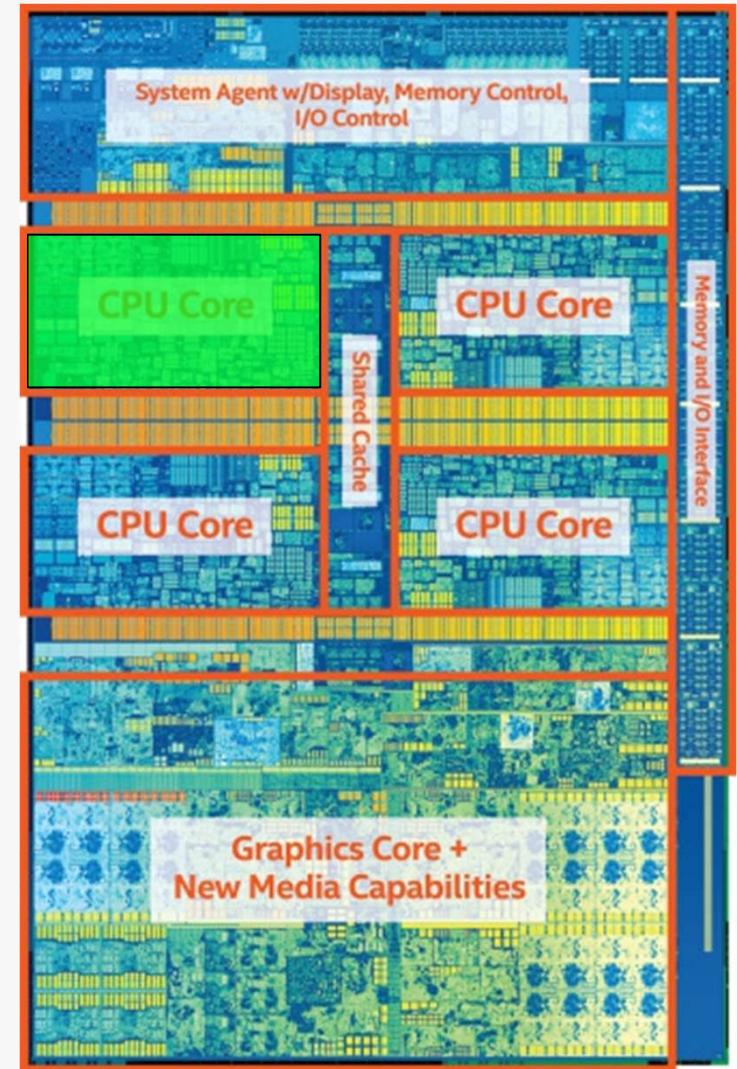
- 4x CPU cores
 - Each with hyperthreading
 - Each with support for 256bit AVX2 instructions
- Intel Gen 9.5 GPU
 - With 1280 processing elements



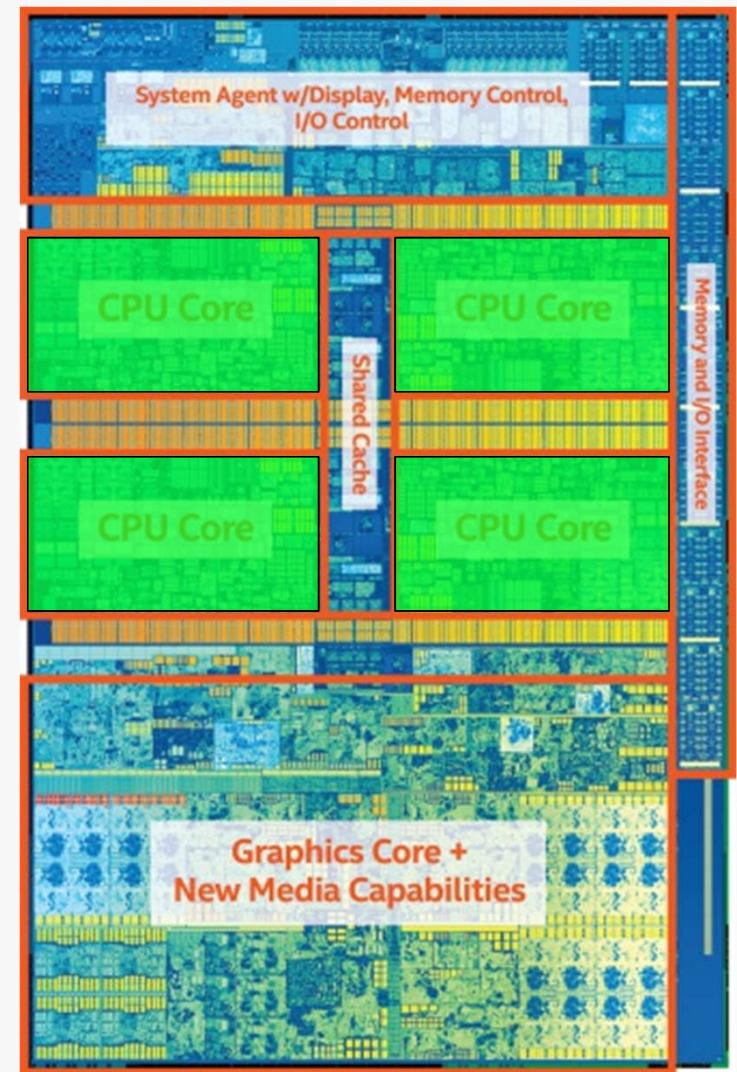
Regular sequential C++ code (non-vectorised) running on a single thread only takes advantage of a very small amount of the available resources of the chip



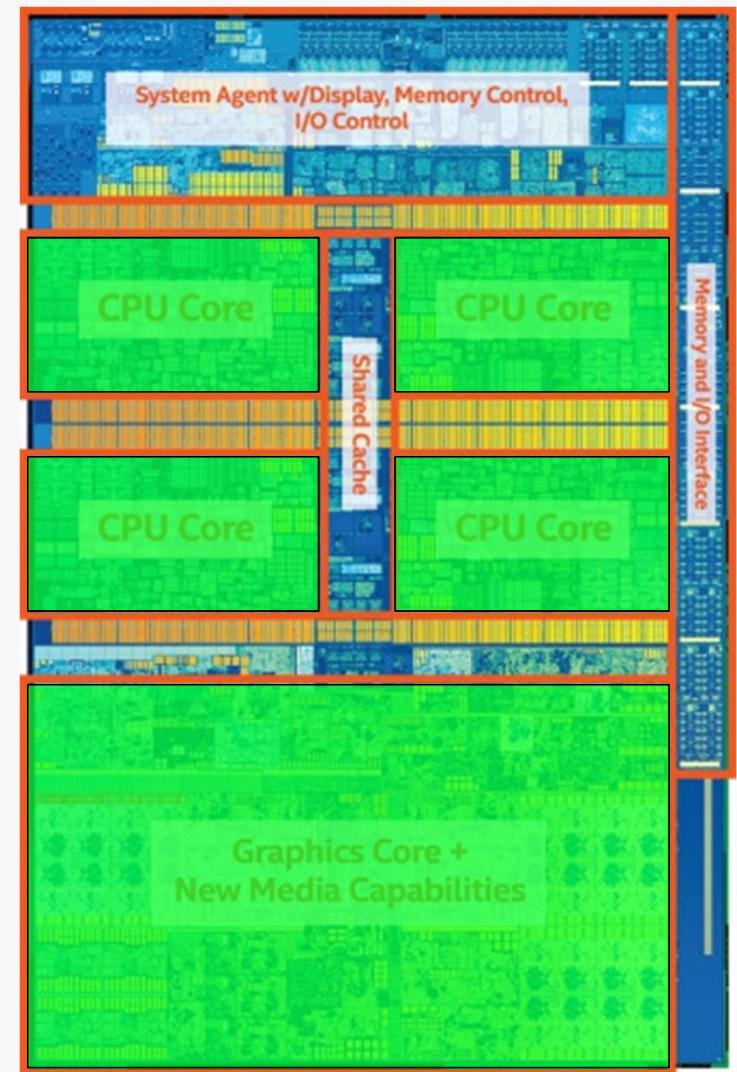
Vectorisation allows you to fully utilise a single CPU core



Multi-threading allows you to fully utilise all CPU cores



Heterogeneous dispatch allows you to fully utilise the entire chip



So now you have multiple diggers...

How do you manage them?



So now you have multiple diggers...

How do you manage them?

- Make sure they all have work to do



So now you have multiple diggers...

How do you manage them?

- Make sure they all have work to do
- Make sure they are all working efficiently



So now you have multiple diggers...

How do you manage them?

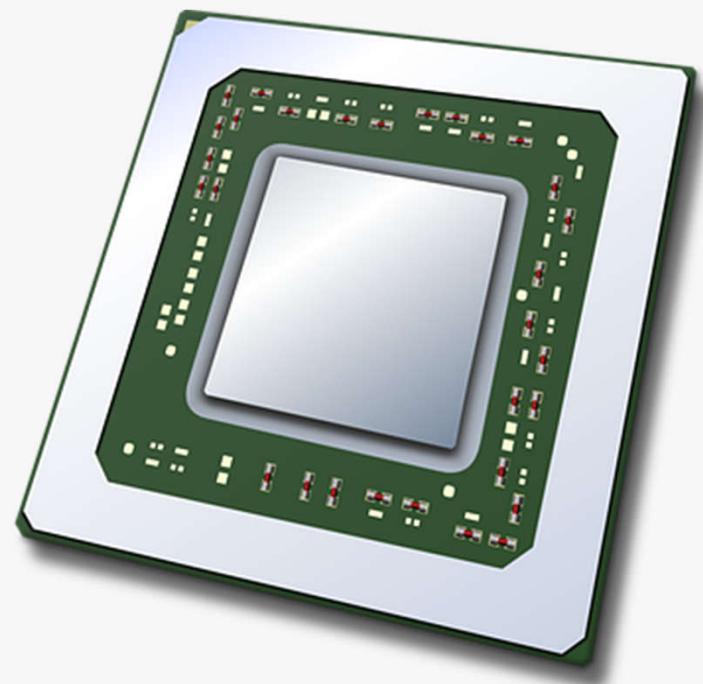
- Make sure they all have work to do
- Make sure they are all working efficiently
- Make sure they don't get in each other's way



This also applies to parallel processors

How do you manage them?

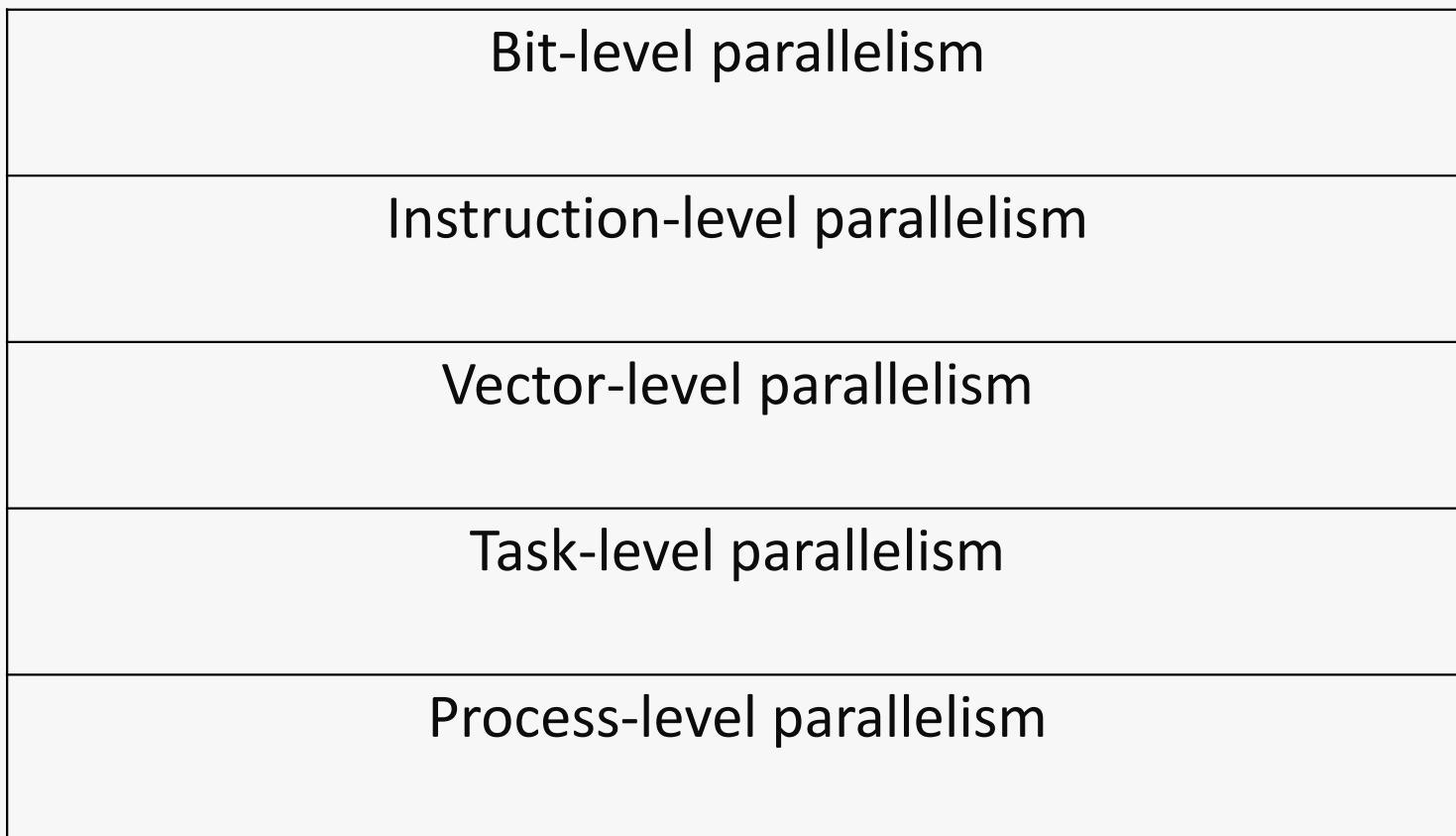
- Make sure they all have work to do
- Make sure they are all working efficiently
- Make sure they don't get in each other's way



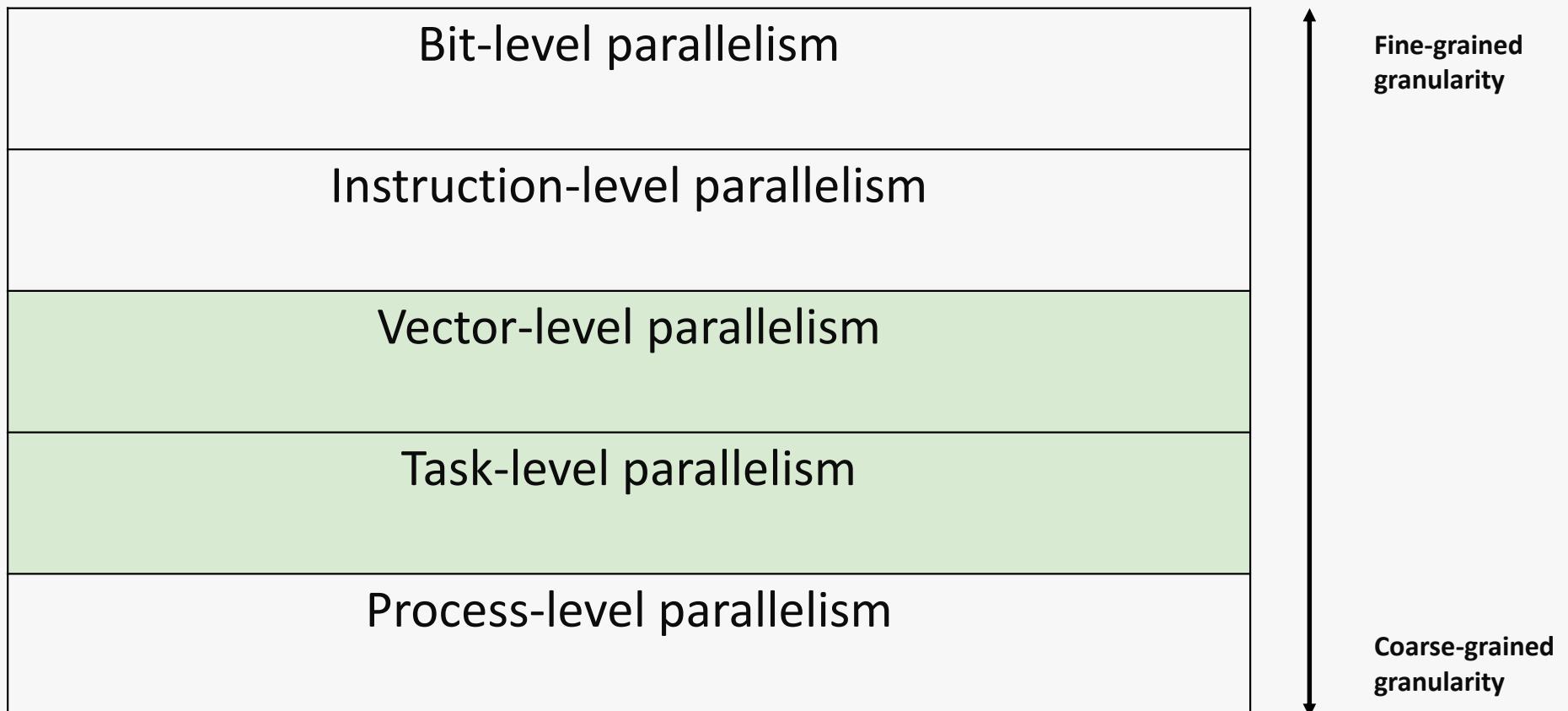
Parallelism vs concurrency



The different kinds of parallelism



The different kinds of parallelism



The challenge of parallel computing

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

The challenge of parallel computing

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

The challenge comes in constructing those tasks in such a way that they:

- Make efficient use of the available hardware
- Adhere to the benefits and limitations of the hardware
- Coordinate effectively to complete the work
- Maintaining the correctness of your application

The challenge of parallel computing

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

The challenge comes in constructing those tasks in such a way that they:

- Make efficient use of the available hardware
- Adhere to the benefits and limitations of the hardware
- Coordinate effectively to complete the work
- Maintaining the correctness of your application

This will be the focus of the class...

Key takeaways

The clock speeds of CPUs are not getting any faster, so gaining further performance must come from parallelism

Sequential code alone uses a very small fraction of the available resources of a typical chip

The challenge of parallel computing is efficiently coordinating tasks across multiple processes in order to solve a problem



Questions?



Chapter 2: Performance Portability

Michael Wong

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about the Iron triangle of parallel computing
 - Learn about the goals of parallel computing
 - Learn about how to iterate through parallel programming tasks

Step Back

So what are the goals of multithreaded programming

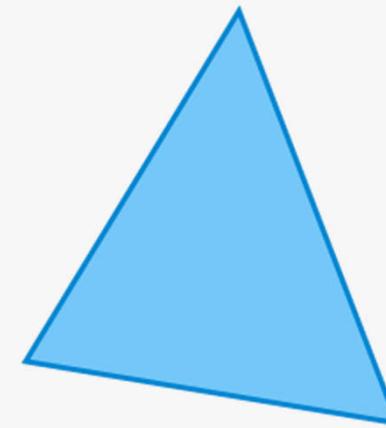
And how do you do it in general?

So What are the Goals?

- Goals of Parallel Programming over and above sequential programming
 1. Performance
 2. Productivity
 3. Generality-Portability

Performance

Generality-Portability



Productivity

The Iron Triangle of Parallel Programming language nirvana

- Oh, really??? What about correctness, maintainability, robustness, and so on?

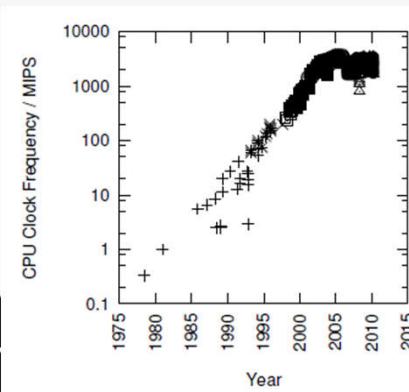
- And if correctness, maintainability, and robustness don't make the list, why do productivity and generality?

- Given that parallel programs are much harder to prove correct than are sequential programs, again, shouldn't correctness really be on the list?

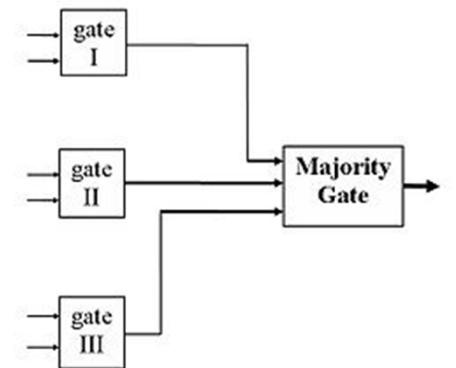
- What about just having fun?

Performance

- Broadly includes scalability and efficiency
- If not for performance why not just write sequential program?
- parallel programming is primarily a performance optimization, and, as such, it is one potential optimization of many.

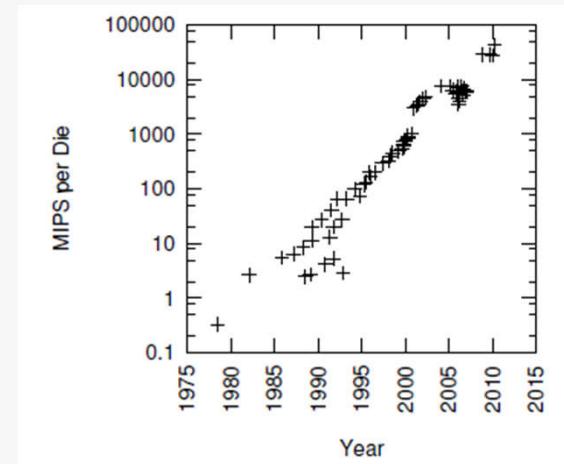
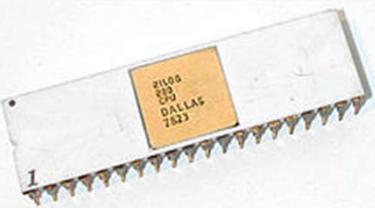


- Are there no cases where parallel programming is about something other than performance?



Productivity

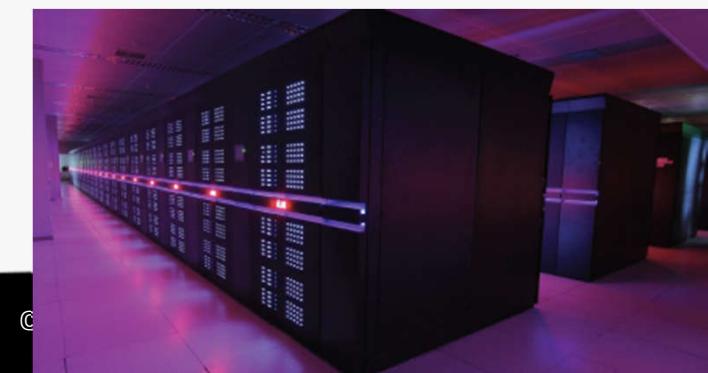
- Perhaps at one time, the sole purpose of parallel software was performance. Now, however, productivity is gaining the spotlight.



- Why all this prattling on about non-technical issues??? And not just any non-technical issue, but productivity of all things? Who cares?

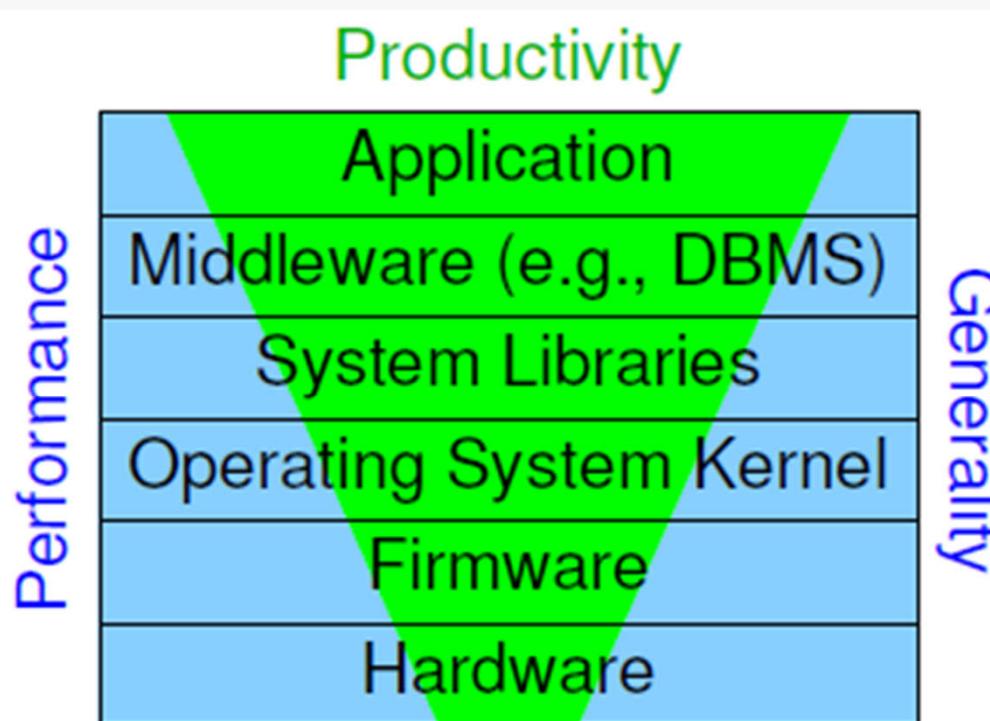


- Given how cheap parallel systems have become, how can anyone afford to pay people to program them?



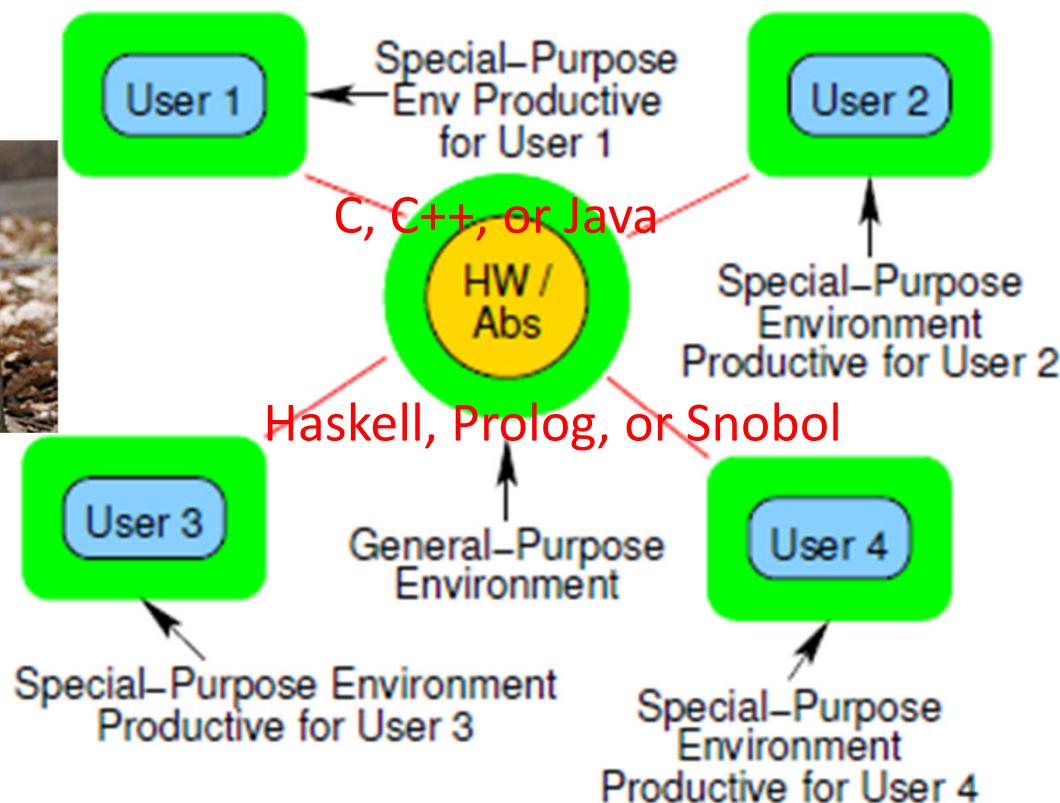
Generality (Portability)

- C/C++ locking+threads
- Java
- MPI
- OpenMP
- SQL



Until such a nirvana appears, it will be necessary to make engineering tradeoffs

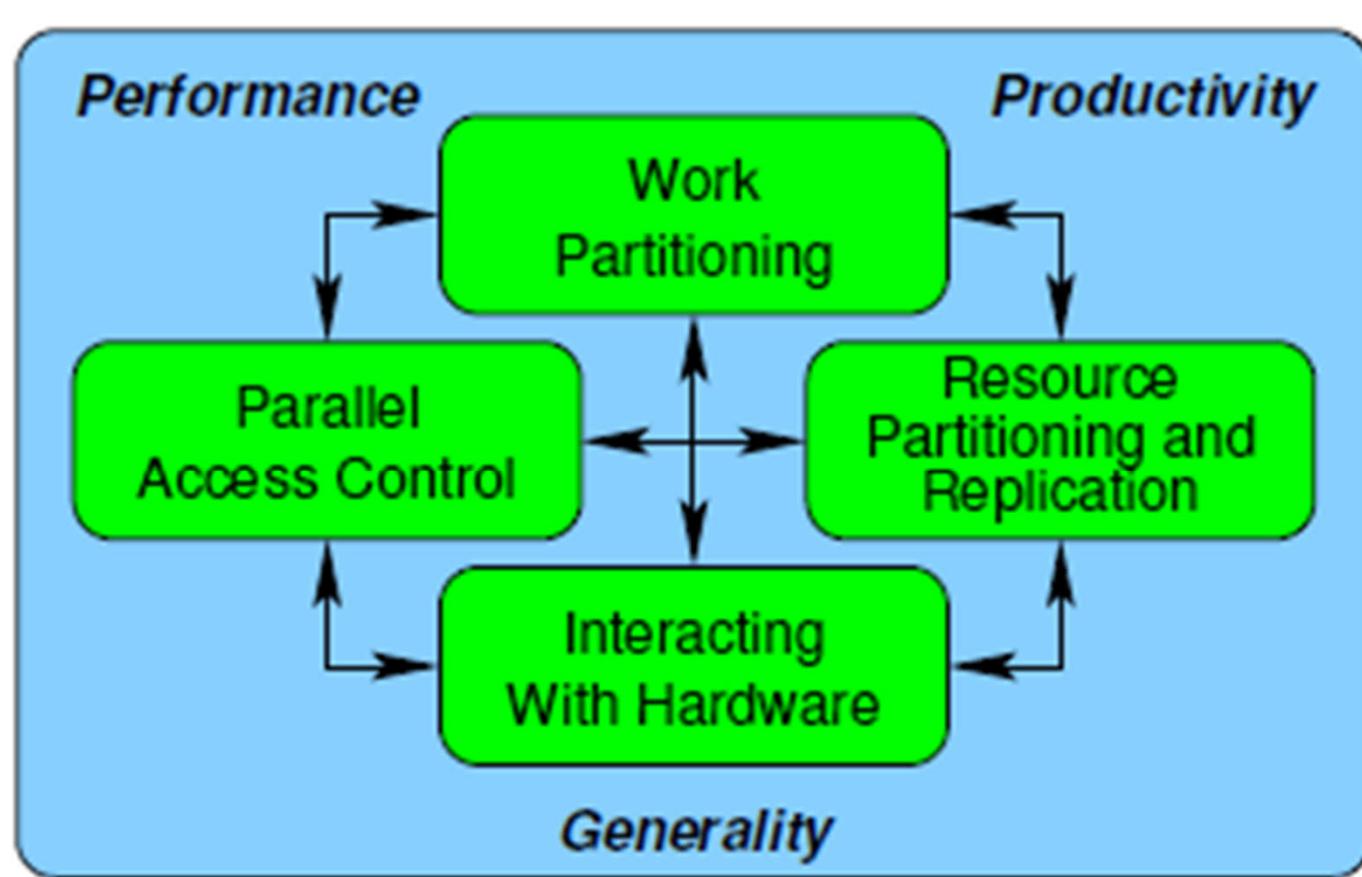
Tradeoff Generality and Performance





- This is a ridiculously unachievable ideal! Why not focus on something that is achievable in practice?

Parallel programming tasks



Work Partitioning

- Greatly increase performance, scalability but can greatly increase complexity
- permitting threads to execute concurrently greatly increases the program's state space, which can make the program difficult to understand and debug
 - Can decrease productivity

- Other than CPU cache capacity, what might require limiting the number of concurrent threads?

Parallel Access Control

- Does access depend on resource location?
- How does thread coordinate access to the resource?

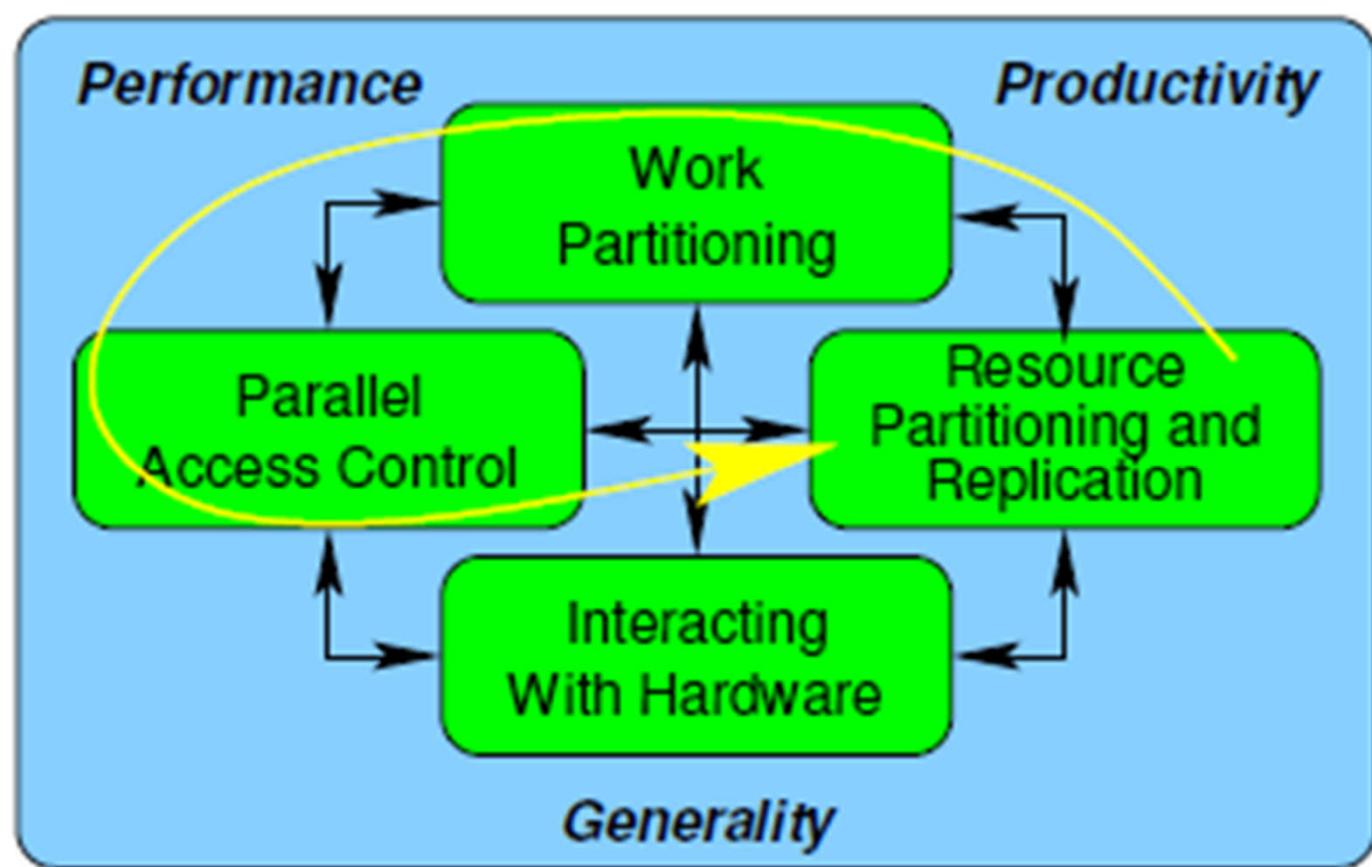
Resource partitioning and replication

- Data may be partitioned over computers, disks, NUMA nodes, CPU cores, pages, cache lines, instances of synchronization primitives, or critical section of code
- Resource partitioning is frequently application dependent

Interacting with Hardware

- developers working with novel hardware features and components will often need to work directly with such hardware
- direct access to the hardware can be required when squeezing the last drop of performance out of a given system

Composite Capabilities



Key takeaways

Iron Triangle of Parallel programming language

Remember how to iterate through parallel programming tasks



Questions?



Chapter 3: C++ Multi-threading

Michael Wong

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn how to create and wait on threads
 - Learn how to pass parameters to threads and share data between threads
 - Learn about invariants, lifetime issues and deadlock
 - Learn about mutexes and condition variables

Parallelism and Concurrency in C11/C++11/C++14/C++17/C++20

- C99/C++98: does not have parallelism or concurrency support
- C++11 have multithreading support
 - Memory model, atomics API
 - Language support: TLS, static init, termination, lambda function
 - Library support: thread start, join, terminate, mutex, condition variable
 - Advanced abstractions: basic futures, thread pools
- C11 will have similar memory model, atomics API. TLS, static init/termination
 - Some minor differences like `__Atomic` qualifier
- C++14 enhanced the memory model with better definition of lock-free vs obstruction-free
 - Clarified `atomic_signal_fence`
 - Improved on out-of-thin-air results
 - Realized consume ordering as described was problematic
- C++17 we added
 - Define strength ordering of memory models
 - `ParallelSTL`, progress guarantees
- C++20 we added
 - Co-routines
 - `Atomic<shared_ptr<T>>`
 - Latches, barriers
 - `SIMD<T>`
 - Cooperative cancellation and jthread

Parallel/concurrency after C++11

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

Parallel/concurrency after C++14

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI, background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	C++11: thread,lambda function, TLS, async C++14: generic lambda	C++11: Async, packaged tasks, promises, futures, atomics,	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: <code>shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence,</code>	C++11: lambda C++14: none

Parallel/concurrency after C++17

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
today's abstractions	C++11: thread, lambda function, TLS, async C++14: generic lambda	C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control false sharing	C++11: locks, memory model, mutex, condition variable, atomics, static init/term, C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence, C++ 17: scoped_lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies	C++11: lambda C++14: none C++17: progress guarantees, TOE, execution policies

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 false sharing 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,

So Why do we need to standardize concurrency?

- Reflects the real world
 - Multi-core processors
 - Solutions for very large problems
 - Internet programming
- Standardize existing practice
 - C++ threads=OS threads
 - shared memory
 - Loosely based on POSIX, Boost thread
 - Does not replace other specifications
 - MPI, OpenMP, UPC, autoparallelization, many others
- Can help existing advanced abstractions
 - TBB, PPL, Cilk,

Concurrency Language and Library

- Core Language: what does it mean to share memory and how it affects variables
 - TLS
 - Static duration variable initialization/destruction
 - Memory model
 - Atomic types and operations
 - Lock-free programming
 - Fences
- Library
 - How to create/synchronize/terminate threads,
 - Thread , mutex , locks
 - RAII for locking, type safe
 - propagate exceptions
 - few advanced abstraction
 - Async() , promises and futures
 - parallel STL

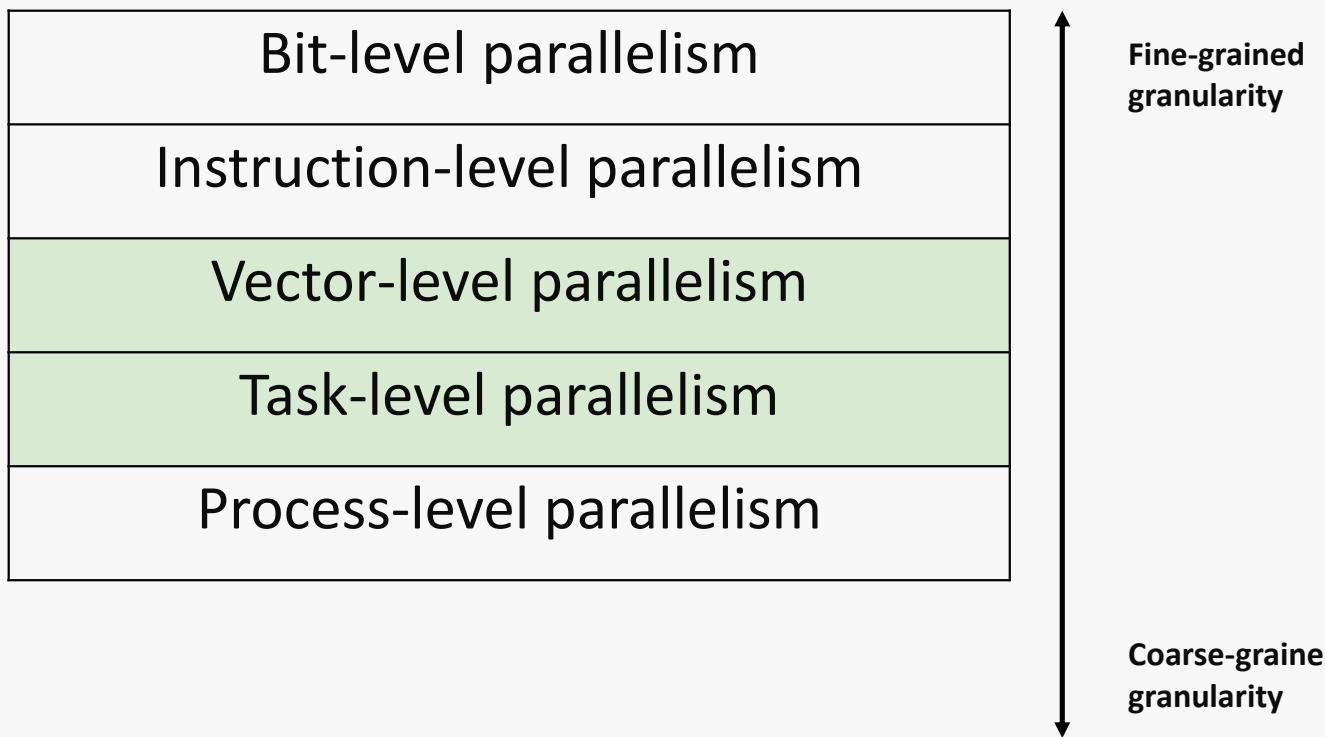
What we got in C++

- Low level support to enable higher abstractions
 - Elementary Thread pools in asynch
- Ease of programming
 - Writing correct concurrent code is hard
 - Lots of concurrency in modern HW
 - Portability with the same natural syntax
 - Not achievable before
 - Uncompromising Performance
 - Stable memory model
 - System level interoperability
- C++ shares threads with other languages

What we are still trying to get

- higher parallel abstractions
 - Transactional memory (TM), executors, queues and counters
 - Task Blocks, networking
- Distributed and Heterogeneous programming
- Complete Compatibility between C and C++
- Total isolation from programmer mistakes

What is Parallelization?

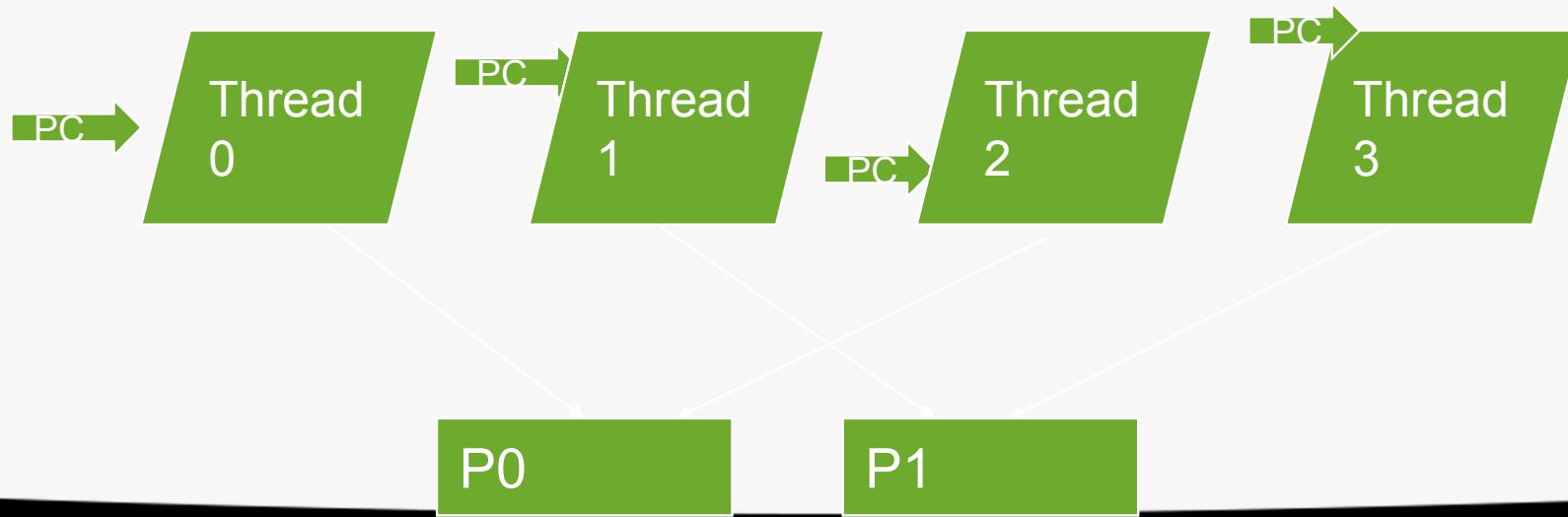


Something is parallel if there is a certain level of independence in the order of operations

- If there is notion of ordering, I must do this, then do that, then it is not parallel

What is a thread?

- **This** is where the parallelism is.
- What is a thread?
 - A series of instructions with its own program counter and state
- What is a process?
 - An instance of running program, has at least one thread



Parallel Overhead

- Total CPU > serial CPU
 - Newly introduced parallel portions in your program now needs to be executed, not a big deal
 - Threads need to communicate data to each other and synchronize
 - this is usually the problem, and why parallel code may not scale
 - Most memory models will give you some elegant way to strip out that overhead
 - But it will get worse when increasing number of threads
 - Efficient parallelization is about minimizing the overhead

Fork-Join Pattern

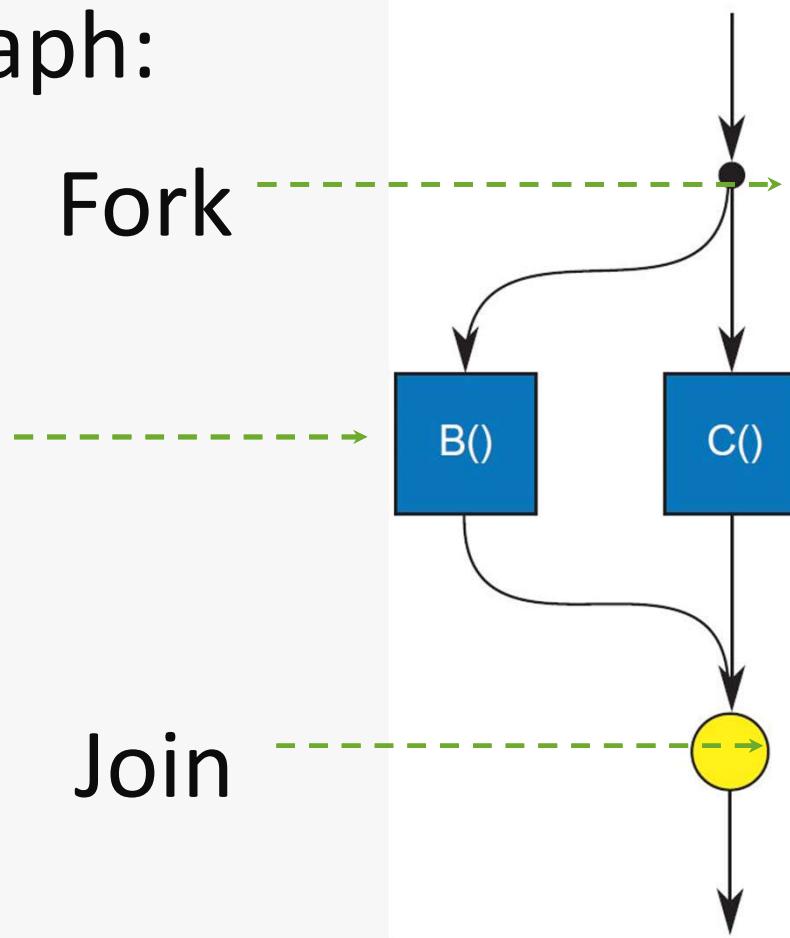
- Control flow **divides** (forks) into multiple flows, then **combines** (joins) later
- During a fork, one flow of control becomes two
- Separate flows are “independent”
 - Does “independent” mean “not dependent” ?
 - No, it just means that the 2 flows of control “are not constrained to do similar computation”
- During a join, two flows become one, and only this one flow continues

Fork-Join Pattern

- Fork-Join directed graph:

Independent work

Is it possible for B() and C() to have dependencies between them?



Fork-Join Pattern

- Typical **divide-and-conquer** algorithm implemented with fork-join:

```
void DivideAndConquer( Problem P ) {  
    if( P is base case ) {  
        Solve P;  
    } else {  
        Divide P into K subproblems;  
        Fork to conquer each subproblem in parallel;  
        Join;  
        Combine subsolutions into final solution;  
    }  
}
```

98

Thread launching

- Basic thread class
 - a. Fork a function execution, join operation, fork-join pattern
 - b. std::thread takes any “callable object” and runs it asynchronously:
- Three different ways of launching a thread
 1. Ordinary function f in t1
 2. Class w with operator()() in t2
 3. Lambda expression in t3

```
void f();  
class W {  
    void operator()()const;  
    void normalize(long double, int,  
std::vector<float>);  
};  
void bar()  
{  
    std::thread t1(f); //f() executes in separate thread  
    W w;  
    t1.join(); //wait for thread t1 to end  
    std::thread t2(w); // run function object  
    w.operator()() asynchronously  
    std::thread t3([]{std::cout << "lambda\n"; });  
}
```

std::thread - cppreference.com X std::system - cppreference.com X Coliru X +

https://en.cppreference.com/w/cpp/thread/thread

cppreference.com Create account Search

Page Discussion View Edit History C++ Thread support library std::thread

std::thread

Defined in header `<thread>`

`class thread;` (since C++11)

The class `thread` represents a single thread of execution. Threads allow multiple functions to execute concurrently. Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a constructor argument. The return value of the top-level function is ignored and if it terminates by throwing an exception, `std::terminate` is called. The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic`). `std::thread` objects may also be in the state that does not represent any thread (after default construction, move from, `detach`, or `join`), and a thread of execution may be not associated with any thread objects (after `detach`). No two `std::thread` objects may represent the same thread of execution; `std::thread` is not `CopyConstructible` or `CopyAssignable`, although it is `MoveConstructible` and `MoveAssignable`.

Member types

Member type	Definition
<code>native_handle_type</code>	<i>implementation-defined</i>

Member classes

`id` represents the *id* of a thread
(public member class)

Member functions

<code>(constructor)</code>	constructs new thread object (public member function)
<code>(destructor)</code>	destructs the thread object, underlying thread must be joined or detached (public member function)
<code>operator=</code>	moves the thread object (public member function)

Observers

<code>joinable</code>	checks whether the thread is joinable, i.e. potentially running in parallel context (public member function)
<code>get_id</code>	returns the <i>id</i> of the thread (public member function)

slack
All the tools your team needs in one place. Slack: Where work happens.
ADS VIA CARBON

16:21

std::thread::thread - cppreference.com

https://en.cppreference.com/w/cpp/thread/thread/thread

cppreference.com

Create account Search

Page Discussion

C++ Thread support library std::thread

std::thread::thread

thread() noexcept; (1) (since C++11)
thread(thread& other) noexcept; (2) (since C++11)
template< class Function, class... Args >
explicit thread(Function&& f, Args&... args); (3) (since C++11)
thread(const thread&) = delete; (4) (since C++11)

Constructs new thread object.

1) Creates new thread object which does not represent a thread.
2) Move constructor. Constructs the thread object to represent the thread of execution that was represented by other. After this call other no longer represents a thread of execution.
3) Creates new std::thread object and associates it with a thread of execution. The new thread of execution starts executing

```
std::invoke(decay_copy(std::forward<Function>(f), decay_copy(std::forward<Args>(args))...);
```

where decay_copy is defined as

```
template <class T>
std::decay_t<T> decay_copy(T& v) { return std::forward<T>(v); }
```

Except that the calls to decay_copy are evaluated in the context of the caller, so that any exceptions thrown during evaluation and copying/moving of the arguments are thrown in the current thread, without starting the new thread.

The completion of the invocation of the constructor *synchronizes-with* (as defined in std::memory_order) the beginning of the invocation of the copy of f on the new thread of execution.

This constructor does not participate in overload resolution if std::decay_t<Function> is the same type as std::thread. (since C++14)

4) The copy constructor is deleted; threads are not copyable. No two std::thread objects may represent the same thread of execution.

Parameters

Thread joining

- An initialized thread object represents an active thread of execution, and can be joined, or detached
- Once called, the thread object is non-joinable
 - Calling either join or detach on non-joinable object will result in runtime exception
 - If you have not called detach or join for joinable threads, then upon calling destructor, it will call std::terminate
 - This program is not well defined
- **Guideline: must always call join or detach**

To join or not to join, that is the question

thread_joinable.cpp

Joining

- Sets a synchronization point between child thread and the parent thread
- Blocks execution of the thread that calls join function, until child thread's execution finish
- After call, the child thread' object can safety be destroyed

Detach

- Separates the child thread from the parent thread
- Allows execution to continue independently
- Allocated resources will be freed once the child thread exits
- No synchronization with main thread
 - Child thread can outlive parent thread
 - Parent thread cannot hold any references to child thread

Joining thread

`thread_join`

Detach thread

`thread_detach`

Need a Better joining mechanism

- For join, introduces a synchronization point, and blocks
- We would like to do some other tasks in between waiting for the child thread
- In real world, you will do many tasks between spinning off child threads

What happens if I have an exception?

thread_exception

A more reliable join: first try

thread_exception

RAII

- Resource acquisition is initialization
- When we execution a program, objects will be constructed from top to bottom (in execution order), then destructed in reverse order
 - First created object will be destroyed last

Thread Guard

Thread_guard (make sure other operation is caught)

Passing parameters by values

thread_param_by_val

Pass parameters by reference

thread_param_by_ref

Guideline

- passing by references with detach is dangerous

Data Life time in multithreaded mode

- Data can change during async calls

```
int x, y, z;  
Widget *pw;
```

...

call f(x, y) asynchronously (i.e., on a new thread);

- *As f execute*
 - *x,y,z, pw could be out of scope*
 - *pw could be deleted*
 - *Values might change*
- *Exact details will depend on parameters (unlike single threaded mode)*

116

Life time and parameters

void f(int xParam, int yParam); // pass by value:

// f unaffected by

// changes to x, y

void f(int& xParam, int& yParam); // pass by ref:

void f(const int& xParam, const int& yParam); // f affected by

// changes to x, y

int x, y, z;

Widget *pw;

...

call f(x, y) asynchronously;

- No declaration insulates f from changes to z, pw, and *pw.¹¹⁷

Data Lifetime

- Data lifetime issues critical in multi-threading (MT) design.
 - A special aspect of synchronization/race issues.
 - Even shared immutable data subject to lifetime issues.

When thread2 outlives thread1

thread_lifetime

Guideline for std::thread arguments

- By-reference/by-pointer parameters in async calls always risky.
 - Prefer pass-by-value.
 - Including via lambdas!
- 2 strategies to avoid lifetime issues
 - Copy data for use by the asynchronous call.
 - Ensure referenced objects live long enough.

```
void f(int xParam); // function to call asynchronously
{
    int x;
    ...
    std::thread t1([&]{ f(x); }); // risky! closure holds a ref to x
    std::thread t2([=]{ f(x); }); // okay, closure holds a copy of x
    ...
} // x destroyed
```

120

Copying arguments for Async Calls

- std::thread's variadic constructor (conceptually) copies everything:

```
void f(int xVal, const Widget& wVal);
```

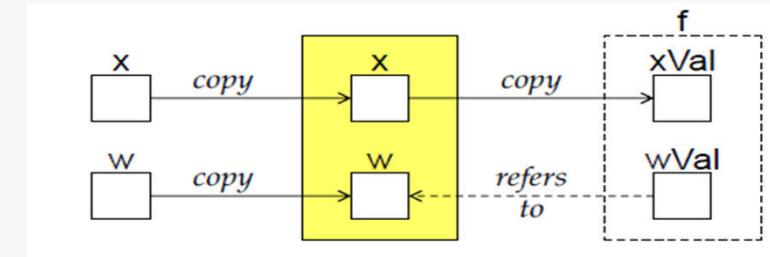
```
int x;
```

```
Widget w;
```

```
...
```

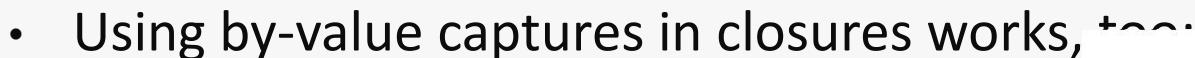
```
std::thread t(f, x, w); // invoke copy off on copies of x, w
```

- Copies of f, x, w, guaranteed to exist until asynch call returns.
- Inside f, wVal refers to a *copy of w, not w itself.*
- Copying optimized to moving whenever possible.



121

Copying Arguments

- Using by-value captures in closures works, 

```
void f(int xVal, const Widget& wVal);
```

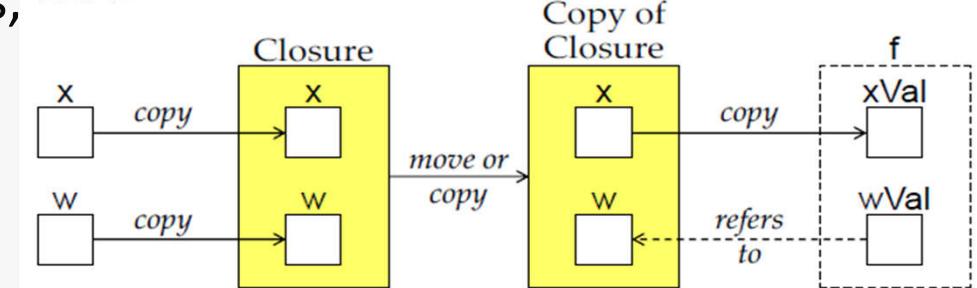
```
int x;
```

```
Widget w;
```

```
...
```

```
std::thread t([=]{ f(x, w); }); // invoke copy off on copies of x, w
```

- Closure contains copies of x and w.
- Closure copied by thread ctor; copy exists until f returns.
 - Copying optimized to moving whenever possible.
- Inside f, wVal refers to a *copy of w, not w itself.*



122

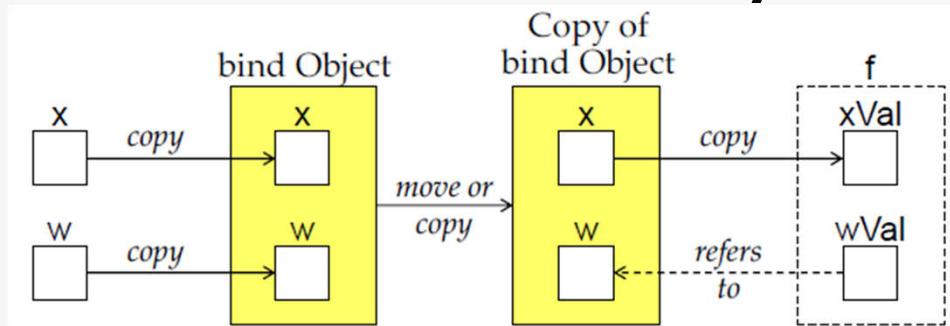
Copying Arguments with Bind: another way

- Another approach is based on std::bind:

```
void f(int xVal, const Widget& wVal);
int x;
Widget w;
...
```

```
std::thread t(std::bind(f, x, w)); // invoke f with copies of x, w
```

- Object returned by bind contains copies of x and w.
- That object copied by thread ctor; copy exists until f returns.
- Inside f, wVal refers to a *copy of w, not w itself.*
- Lambdas are usually a better choice than bind.
- Easier for readers to understand.
- More efficient.



123

Guideline for Copying Arguments

- Options for creating argument copies with sufficient lifetimes:
 - Use variadic thread constructor.
 - Use lambda with by-value capture.
 - Use bind.
- **Prefer** to:
 - Use lambda, more elegant, and copying is explicit

124

Ensure References live long enough

- One way is to delay locals' destruction until asynch call is complete:

```
void f(int xVal, const Widget& wVal); // as before
{
    int x;
    Widget w;
    ...
    std::thread t([&]{ f(x, w); });
    // wVal really refers to w
    ...
    t.join(); // destroy w only after t finishes
}
```

125

What if you really want to pass by ref or need to mix ? Method 1

- Use lambdas:
- Given

```
void f(int xVal, int yVal, int zVal, Widget& wVal);
```

- **Lambdas: use by-reference capture:**

```
{  
    Widget w;  
    int x, y, z;  
    ...  
    std::thread t([=, &w]{ f(x, y, z, w); }); // pass copies of x, y, z;  
    ... // pass w by reference  
}
```

126

- You're responsible for avoiding data races on w.

What if you really want to pass by ref or need to mix ? Method 2 and 3

- Use **Variadic thread constructor or bind: Use C++11's std::ref:**
 - Creates objects that act like references.
 - Copies of a std::ref-generated object refer to the same object.

```
void f(int xVal, int yVal, int zVal, Widget& wVal); // as before
{
    static Widget w;
    int x, y, z;
    ...
    std::thread t1(f, x, y, z, std::ref(w)); // pass copies of
    std::thread t2(std::bind(f, x, y, z, std::ref(w))); // x, y, z; pass w
    ... // by reference
}
```

127

To move or not to move

thread_move

Guideline

- Don't move one thread into another without calling thread constructors or thread assignment operator especially if the destination thread already has a thread

What is your id?

thread_id

Use these tools to do parallel reduction

- Divide and conquer: divide our data set into several blocks, and use multiple threads to accumulate the values in each block
- Reduction: take the sum from each thread and accumulate that, leading to a reduction
- Main problems of finding how many threads to run
 - Oversubscription: when we run too many threads, then the processor have, each thread will have a switch context which harms performance
 - Overload: if we run small amount of data in each thread and we spawn millions of threads leading to excessive overhead (work is smaller then the cost of new threads), then that will kill performance

How many threads should we run

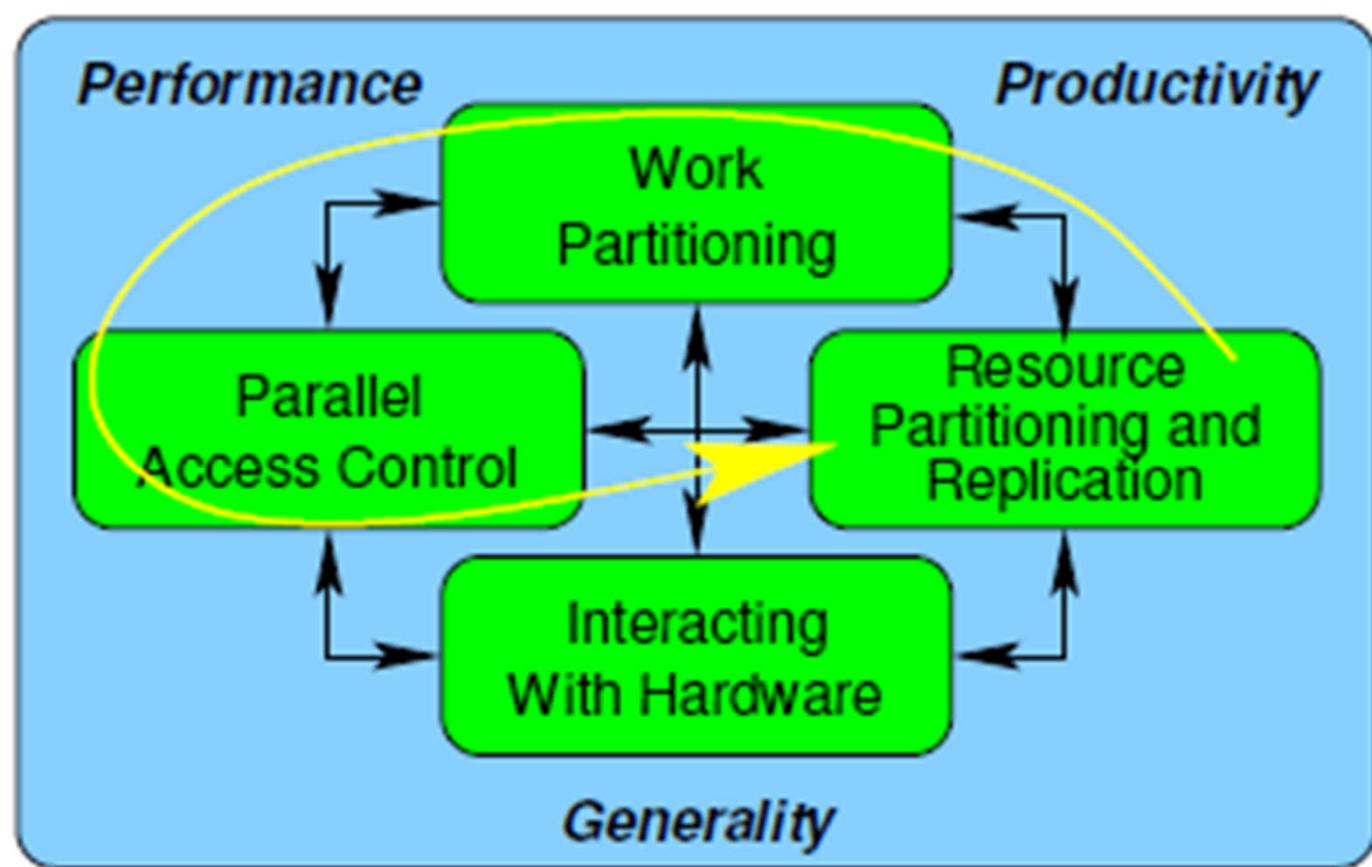
- We need to set minimum number of elements to process on a particular thread
- Find maximum number of threads to avoid overload
- Next find maximum number of threads that processors can run parallel using hardware_concurrency
- Now select minimum of these 2 values as # of running threads

Example

Want to process 6000 elements

1. Set minimum number elements to run per thread as 600, so we need 10 threads to avoid overload
2. Ask `hardware_concurrency` and it returns 12, this is the number of threads you can run in parallel
3. This means we should run $\min(10,12)$ and run 10 threads

Composite Capabilities



Key takeaways

When launching thread, you **must always call join or detach**

Passing argument by references with detach is dangerous

Don't move one thread into another without calling thread constructors or thread assignment operator especially if the destination thread already has a thread

You can build higher abstractions with threads, but wait you can do better!



Questions?



Chapter 4a: C++ Synchronization, Futures, Promises and Packaged Tasks

Michael Wong

CppCon 2019 – Sep 2019

- Learning objectives:
 - Learn about std::async and std::future
 - Learn about mutexes, locks and lock_guard
 - Learn about lock-based programming
 - Learn about std::promise
 - Learn about condition variables
 - Learn about packaged tasks

Locks and Invariants

Most common problems in multithreaded programming are due to invariants being broken while updating

Invariants: statements that are always true for a data structure
E.g. for list, size variable always contains number of elements

Race condition

Whenever there are two concurrent accesses to data, and one of them is a write, and the two accesses are not ordered by a happens before relationship

Practically: in concurrency, anytime when the outcome depends on timing of order of execution of operations of two or more threads

Most likely it will cause a broken invariant

But most of the time, it will work, especially if there are very few threads

Mutex/Critical region

- Prevent simultaneous update of shared variables
- Can cause race conditions
- Force only one thread at a time through

```
for (i=0; i<n;++i){
```

...

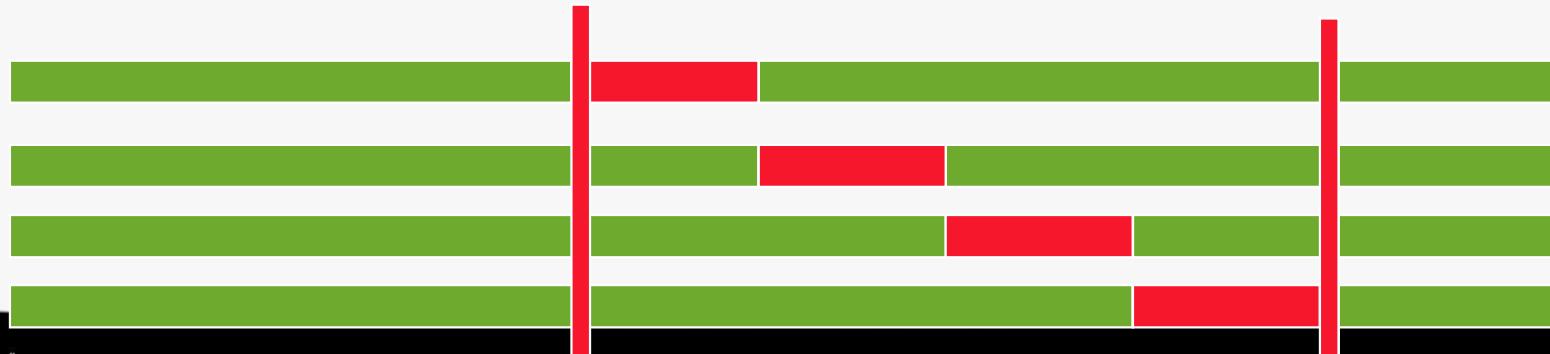
```
    sum+=a[i];
```

...

```
}
```

One at a
time
please

If sum is a shared
variable, this loop can not
run parallel



Mutex

Used to protect shared data

Most of STL is not thread safe

Pushing an element on a list:

1. Create a new node
2. Setting the nodes next to current head node
3. Changing head pointer to point to new node

Mutexes

C++11 has four types:

- `std::mutex`: non-recursive, no timeout support
- `std::timed_mutex`: non-recursive, timeout support
- `std::recursive_mutex`: recursive, no timeout support
- `std::recursive_timed_mutex`: recursive, timeout support

Recursively locking non-recursive mutexes ⇒ undefined behavior.

std::mutex - cppreference.com Coliru https://en.cppreference.com/w/cpp/thread/mutex Create account Search

Page Discussion View Edit History C++ Thread support library std::mutex

std::mutex

Defined in header `<mutex>`

`class mutex;` (since C++11)

The `mutex` class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.

`mutex` offers exclusive, non-recursive ownership semantics:

- A calling thread *owns* a `mutex` from the time that it successfully calls either `lock` or `try_lock` until it calls `unlock`.
- When a thread owns a `mutex`, all other threads will block (for calls to `lock`) or receive a `false` return value (for `try_lock`) if they attempt to claim ownership of the `mutex`.
- A calling thread must not own the `mutex` prior to calling `lock` or `try_lock`.

The behavior of a program is undefined if a `mutex` is destroyed while still owned by any threads, or a thread terminates while owning a `mutex`. The `mutex` class satisfies all requirements of `Mutex` and `StandardLayoutType`.

`std::mutex` is neither copyable nor movable.

Member types

Member type	Definition
<code>native_handle_type</code> (optional)	<i>implementation-defined</i>

Member functions

<code>(constructor)</code>	constructs the <code>mutex</code> (public member function)
<code>(destructor)</code>	destroys the <code>mutex</code> (public member function)
<code>operator=</code> [deleted]	not copy-assignable (public member function)

Locking

<code>lock</code>	locks the <code>mutex</code> , blocks if the <code>mutex</code> is not available (public member function)
<code>try_lock</code>	tries to lock the <code>mutex</code> , returns if the <code>mutex</code> is not available (public member function)
<code>unlock</code>	unlocks the <code>mutex</code> (public member function)

Native handle

<code>native_handle</code>	returns the underlying implementation-defined native handle object (public member function)
----------------------------	--

SendGrid
Transactional Email Delivery. Start sending for Free with a 5-min Integration.
ADS VIA CARBON

Mutual Exclusion

- A mutex is a primitive object used for controlling access in a multi-threaded system.
- A mutex is a shared object (a resource)
- Simplest use:

```
std::mutex m;  
int sh; // shared data  
// ...  
m.lock();  
// manipulate shared data:  
sh+=1;  
m.unlock();
```

145

Mutex – try_lock()

- Don't wait unnecessarily

```
std::mutex m;  
int sh; // shared data  
// ...  
if (m.try_lock()) { // manipulate shared data:  
    sh+=1;  
    m.unlock();  
}  
else {  
    // maybe do something else  
}
```

146

Mutex – try_lock_for()

- Don't wait for too long:

```
std::timed_mutex m;  
int sh; // shared data  
// ...  
if (m.try_lock_for(std::chrono::seconds(10))) { // Note: time  
    // manipulate shared data:  
    sh+=1;  
    m.unlock();  
}  
else {  
    // we didn't get the mutex; do something else  
}
```

147

Mutex – try_lock_until()

- We can wait until a fixed time in the future:

```
std::timed_mutex m;  
int sh; // shared data  
// ...  
if (m.try_lock_until(midnight)) { // manipulate shared data:  
    sh+=1;  
    m.unlock();  
}  
else {  
    // we didn't get the mutex; do something else  
}
```

148

Recursive Mutex

- In some important use cases it is hard to avoid recursion

```
std::recursive_mutex m;  
int sh; // shared data  
// ...  
void f(int i)  
{  
    // ...  
    m.lock();  
    // manipulate shared data:  
    sh+=1;  
    if (--i>0) f(i);  
    m.unlock();  
    // ...  
}
```

149

C++14 additional Mutexes

C++14 adds:

- `std::shared_timed_mutex`:
- Non-recursive reader/writer lock w/timeout support.
- Adds `lock_shared/try_lock_shared` to `std::timed_mutex` API.

Based on Boost's `shared_mutex`, but with fewer capabilities:

- Can't upgrade read lock to exclusive lock.

Locking with mutex

mutex_lock_unlock

Guideline

Always remember to unlock

RAll classes for Mutexes: lock_guard

Mutexes typically managed by RAll classes:

`std::lock_guard`: lock mutex in ctor, unlock it in dtor.

```
std::mutex m; // mutex object
```

```
{
```

```
std::lock_guard<std::mutex> L(m); // lock m
```

```
... // critical section
```

```
} // unlock m
```

No other operations.

No copying/moving, no assignment, no manual unlock, etc.

Locks `std::shared_timed_mutexes` in exclusive (write) mode.

std::lock_guard - cppreference.com Coliru

https://en.cppreference.com/w/cpp/thread/lock_guard

cppreference.com

Create account Search

Page Discussion

C++ Thread support library std::lock_guard

View Edit History

std::lock_guard

Defined in header `<mutex>`

`template< class Mutex >`

`class lock_guard;`

The class `lock_guard` is a mutex wrapper that provides a convenient RAI-style mechanism for owning a mutex for the duration of a scoped block.

When a `lock_guard` object is created, it attempts to take ownership of the mutex it is given. When control leaves the scope in which the `lock_guard` object was created, the `lock_guard` is destructed and the mutex is released.

The `lock_guard` class is non-copyable.

Template parameters

Mutex - the type of the mutex to lock. The type must meet the [BasicLockable](#) requirements

Member types

Member type Definition

`mutex_type` Mutex

Member functions

(constructor)	constructs a <code>lock_guard</code> , optionally locking the given mutex (public member function)
(destructor)	destructs the <code>lock_guard</code> object, unlocks the underlying mutex (public member function)
<code>operator=</code> [deleted]	not copy assignable (public member function)

Example

Run this code

```
#include <thread>
#include <mutex>
#include <iostream>

int g_i = 0;
std::mutex g_i_mutex; // protects g_i

void safe_increment()
```

Now change lock to RAII lock guard

`mutex_lock_guards`

Returning pointer or reference to unprotected data

Mutex Unprotected Data

Passing code to the protected data structure which you don't have control

Mutex unprotected data access

RAII classes for Mutex: unique_lock

std::unique_lock: much more flexible.

May lock mutex after construction, unlock before destruction.

Moveable, but not copyable.

Supports timed mutex operations:

Try locking, timeouts, etc.

Typically the best choice for unshared timed mutexes.

Locks std::shared_timed_mutexes in exclusive (write) mode.

```
using RTM = std::recursive_timed_mutex; // typedef
```

```
RTM m; // mutex object
```

```
{
```

```
std::unique_lock<RTM> L(m); // lock m
```

```
... // critical section
```

```
L.unlock(); // unlock m
```

```
...
```

```
} // nothing happens
```

RAII for mutexes: std::lock

- A lock represents local ownership of a non-local resource(the mutex)

```
std::mutex m;
```

```
int sh; // shared data
```

```
void f()
```

```
{
```

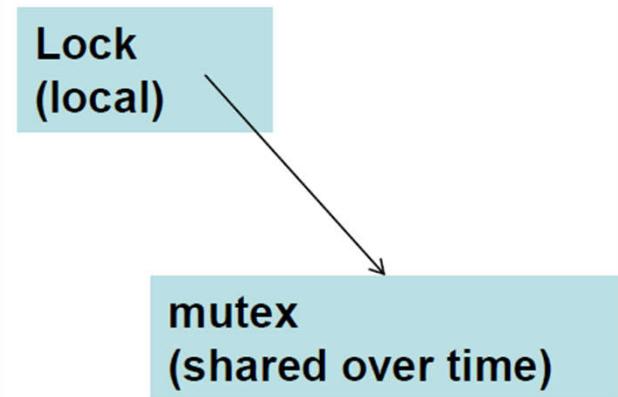
```
// ...
```

```
    std::unique_lock lck(m); // grab (acquire) the mutex
```

```
    // manipulate shared data:
```

```
    sh+=1;
```

```
} // implicitly release the mutex
```



159

Guideline for using mutex locks

Villain:

Returning pointer or reference to unprotected data

Passing code to the protected data structure which you don't have control with

Race condition in the interface

Implicit prohibition of partitioning.

Callback functions requiring locking.

Object-oriented spaghetti code.

Hero:

Standalone application -e.g. Linux kernel

Locking for Parallel library: just another tool

Don't use either callbacks or signals.

Don't acquire locks from within callbacks or signal handlers.

Let the caller control synchronization.

Parameterize the library API to delegate locking to caller.

Explicitly avoid callback deadlocks.

Explicitly avoid signal-handler deadlocks.

std::shared_lock(C++14)

C++14 adds:

```
std::shared_lock: like std::unique_lock, but locks  
std::shared_timed_mutexes for shared (read) access.  
using STM = std::shared_timed_mutex; // typedef  
STM m; // mutex object  
{  
    std::shared_lock<STM> L(m); // lock m in read mode  
    ... // critical section  
    // for reading  
    L.unlock(); // unlock m  
    ...  
} // nothing happens
```

Deadlock

Single mutex with lock_guard

Multiple mutex

No mutex

Multiple Mutex acquisition and Transactional Memory

Acquiring mutexes in different orders leads to deadlock:

```
int weight, value;  
std::mutex wt_mux, val_mux;  
{ // Thread 1  
    std::lock_guard<std::mutex> wt_lock(wt_mux); // wt 1st  
    std::lock_guard<std::mutex> val_lock(val_mux); // val 2nd  
    //work with weight and value // critical section  
}  
{ // Thread 2  
    std::lock_guard<std::mutex> val_lock(val_mux); // val 1st  
    std::lock_guard<std::mutex> wt_lock(wt_mux); // wt 2nd  
    //work with weight and value // critical section  
}
```

Lock ordering deadlocks

Single mutex with lock_guard

Potential Deadlock

- Unstructured use of multiple locks is hazardous:

```
std::mutex m1;  
std::mutex m2;  
int sh1; // shared data  
int sh2;  
// ...  
void f()  
{  
    // ...  
    std::unique_lock lck1(m1);  
    std::unique_lock lck2(m2);  
    // manipulate shared data:  
    sh1+=sh2;  
}
```

166

Multiple Mutex Acquisition: shared_lock

Works with std::shared_lock as well as std::unique_lock.

E.g., for write access to assignment target, read access to source:

```
class Widget {  
public:  
    Widget& operator=(const Widget& rhs)  
    {  
        if (this != &rhs) {  
            std::unique_lock<std::shared_timed_mutex> dest(m, std::defer_lock);  
            std::shared_lock<std::shared_timed_mutex> src(rhs.m, std::defer_lock);  
            std::lock(dest, src); // lock dest in write mode, src in read mode  
            ... // assign data  
        } // unlock mutexes  
        return *this;  
    }  
private:  
    mutable std::shared_timed_mutex m;  
    ...  
};
```

```
Widget w1, w2; // No deadlock!  
...  
// Thread 1           // Thread 2  
w1 = w2;           w2 = w1;
```

RAII for mutexes: std::lock

- We can safely use several locks

```
void f()
{
    // ...
    std::unique_lock lck1(m1,std::defer_lock); // make locks but don't yet
    //try to acquire the mutexes
    std::unique_lock lck2(m2,std::defer_lock);
    std::unique_lock lck3(m3,std::defer_lock);
    // ...
    lock(lck1,lck2,lck3);
    // manipulate shared data
} // implicitly release the mutexes
```

168

How to solve Multiple mutex acquisition

std::lock solves this problem:

```
{ // Thread 1  
    std::unique_lock<std::mutex> wt_lock(wt_mux, std::defer_lock);  
    std::unique_lock<std::mutex> val_lock(val_mux, std::defer_lock);  
    std::lock(wt_lock, val_lock); // get mutexes w/o  
    // deadlock  
    work with weight and value // critical section  
}  
  
{ // Thread 2  
    std::unique_lock<std::mutex> val_lock(val_mux, std::defer_lock);  
    std::unique_lock<std::mutex> wt_lock(wt_mux, std::defer_lock);  
    std::lock(val_lock, wt_lock); // get mutexes w/o  
    // deadlock  
    work with weight and value // critical section  
}
```

Use unique lock to solve deadlock

Mutex deadlock unique lock

Another Multiple mutex solution

Locks are Impractical for Generic Programming=callback

Thread 1:
m1.lock();
m2.lock();
...

Thread 2:
m2.lock();
m1.lock();
...

=
deadlock

Easy. Order Locks.

Now let's get slightly more real:

What about Thread 1 +

A thread running f():
template <class T>
void f(T &x, T y) {
 unique_lock<mutex> _(m2);
 x = y;
}

?

What locks does $x = y$ acquire?

What locks does $x = y$ acquire?

IBM

- Depends on the type T of x and y.
 - The author of f() shouldn't need to know.
 - That would violate modularity.
 - But lets say it's `shared_ptr<TT>`.
 - Depends on locks acquired by TT's destructor.
 - Which probably depends on its member destructors.
 - Which I definitely shouldn't need to know.
 - But which might include a `shared_ptr<TTT>`.
 - Which acquires locks depending on TTT's destructor.
 - Whose internals I definitely have no business knowing.
 - ...
 - And this was for an unrealistically simple f()!
 - We have no realistic rules for avoiding deadlock!
 - In practice: Test & fix?

```
template <class T>  
void f(T &x, T y) {  
    unique_lock<mutex> _(m2);  
    x = y;  
}
```

Transactions Naturally Fit Generic Programming Model

- Composable, no ordering constraints

f() implementation:
`template <class T>
void f(T &x, T y) {
 transaction {
 x = y;
 }
}`

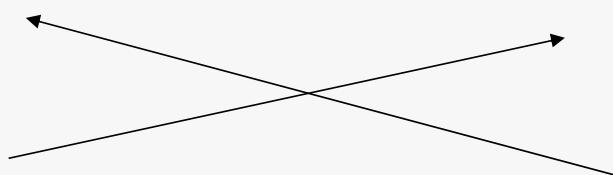
Class implementation:
`class ImpT
{
 ImpT& operator=(ImpT T&
rhs)
 {
 transaction {
 // handle assignment
 }
 }
};`

Impossible to deadlock

Deadlock with No mutex

Thread 1:

`t2.join();`



Thread 2:

`t1.join();`

What can you do with locks and mutexes?

Can implement thread safe stack

LIFO data structure

Implement using singly linked list

Push and pop from the head

Problems:

Race condition in the interface

Condition Variables

Allow threads to communicate about changes to shared data.

- Consumers [wait](#) until producers [notify](#) about changed state.

Rules:

- Call `wait` while holding locked mutex.
- `wait` unlocks mutex, blocks thread, enqueues it for notification.
- At notification, thread is unblocked and moved to mutex queue.
- “Notified threads awake and run with the mutex locked.”

Condition variable types:

- [condition_variable](#): wait on `std::unique_lock<std::mutex>`.
- Most efficient, appropriate in most cases.
- [condition_variable_any](#): wait on any lock type.
- Possibly less efficient, more flexible.
- E.g., works with
 - `std::shared_lock<std::shared_timed_mutex>`

Condition Variables: Wait

[wait parameters](#):

- Mutex for shared data (required).
- Timeout (optional).
- Predicate that must be true for thread to continue (optional).
 - Allows library to handle spurious wakeups.
 - Often specified via lambda.

[Notification options](#):

- [notify_one](#) waiting thread.
 - When all waiting threads will do and only one needed.
 - No guarantee that only one will be awakened.
- [notify_all](#) waiting threads.

Wait Examples

```
std::atomic<bool> readyFlag(false);
std::mutex m;
std::condition_variable cv;
{
    std::unique_lock<std::mutex> lock(m);
    while (!readyFlag)                                // loop for spurious wakeups
        cv.wait(lock);                               // wait for notification
    cv.wait(lock, []{ return readyFlag == true; });   // ditto, but library loops
    if (cv.wait_for(lock, std::chrono::seconds(1)),    // if (notification rcv'd
        []{ return readyFlag == true; }) {             // or timeout) and
        ...                                         // predicate's true...
    }                                               // critical section
}
else {
    ...                                         // timed out w/o getting
}                                               // into critical section
}
```

Notification Example

```
std::atomic<bool> readyFlag(false); // as before
std::condition_variable cv;
{
    ...
    readyFlag = true;
    cv.notify_one(); // wake ~1 thread
}
{
    ...
    readyFlag = true;
    cv.notify_all(); // wake all threads
}
// 1 will then block on m)
```

`notify_all` moves all blocked threads from the condition variable queue to the corresponding mutex queue.

What can you do with condition variable and wait

Build a thread safe queue

Push

FIFO data structure

Pop

Implemented using singly
linked list

Front
Empty

Push from tail, and pop
from head

Size
(same as stack interface)

Synchronous vs asynchronous

Synchronous operations

Blocks the caller process
until its operation
completes

Asynchronous operations

Non-blocking

Initiates the operation, but the
caller continues, and can discover
completion in future

Good for when computing long
running task in a separate thread

How it works

Caller thread

1. Asynchronous task caller get a future associated with asynchronous task and creates shared state
2. Dispatch the async task
3. When caller need the result, it calls get method on the future
4. If get method blocks, then async task has not finished yet

Async thread

1. Async thread initiated
2. When ready, modifies shared state that is linked to std::future
3. Completes execution

std::async - cppreference.com × Coliru × +

https://en.cppreference.com/w/cpp/thread/async

67%

C++ Thread support library

std::async

Defined in header `<future>`

```
template< class Function, class... Args >
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
    async( Function&& f, Args&&... args );
```

(since C++11) (until C++17) (since C++17) (since C++20)

```
template< class Function, class... Args >
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( Function&& f, Args&&... args );
```

(1) (since C++17) (until C++20)

```
template< class Function, class... Args >
[[nodiscard]] std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( Function&& f, Args&&... args );
```

(since C++20)

```
template< class Function, class... Args >
std::future<std::result_of_t<std::decay_t<Function>(std::decay_t<Args>...)>>
    async( std::launch policy, Function&& f, Args&&... args );
```

(since C++11) (until C++17) (since C++17) (since C++20)

```
template< class Function, class... Args >
std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( std::launch policy, Function&& f, Args&&... args );
```

(2) (since C++17) (until C++20)

```
template< class Function, class... Args >
[[nodiscard]] std::future<std::invoke_result_t<std::decay_t<Function>, std::decay_t<Args>...>>
    async( std::launch policy, Function&& f, Args&&... args );
```

(since C++20)

The template function `async` runs the function `f` asynchronously (potentially in a separate thread which may be part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call.

- Behaves as if (2) is called with `policy` being `std::launch::async | std::launch::deferred`. In other words, `f` may be executed in another thread or it may be run synchronously when the resulting `std::future` is queried for a value.
- Calls a function `f` with arguments `args` according to a specific launch policy `policy`.
 - If the `async` flag is set (i.e. `(policy & std::launch::async) != 0`), then `async` executes the callable object `f` on a new thread of execution (with all thread-locals initialized) as if spawned by `std::thread(std::forward<Function>(f), std::forward<Args>(args)...)`, except that if the function `f` returns a value or throws an exception, it is stored in the shared state accessible through the `std::future` that `async` returns to the caller.
 - If the `deferred` flag is set (i.e. `(policy & std::launch::deferred) != 0`), then `async` converts `f` and `args...` the same way as by `std::thread` constructor, but does not spawn a new thread of execution. Instead, `lazy evaluation` is performed: the first call to a non-timed wait function on the `std::future` that `async` returned to the caller will cause the copy of `f` to be invoked (as an rvalue) with the copies of `args...` (also passed as rvalues) in the current thread (which does not have to be the thread that originally called `std::async`). The result or exception is placed in the shared state associated with the future and only then it is made ready. All further accesses to the same `std::future` will return the result immediately.
 - If both the `std::launch::async` and `std::launch::deferred` flags are set in `policy`, it is up to the implementation whether to perform asynchronous execution or lazy evaluation.
 - If neither `std::launch::async` nor `std::launch::deferred`, nor any implementation-defined policy flag is set in `policy`, the behavior is undefined.

In any case, the call to `std::async synchronizes-with` (as defined in `std::memory_order`) the call to `f`, and the completion of `f` is `sequenced-before` making the shared state ready. If the `async` policy is chosen, the associated thread completion `synchronizes-with` the successful return from the first function that is waiting on the shared state, or with the return of the last function that releases the shared state, whichever comes first.

Parameters

<code>f</code>	- <i>Callable</i> object to call
<code>args...</code>	- parameters to pass to <code>f</code>
<code>policy</code>	- bitmask value, where individual bits control the allowed methods of execution

Bit	Explanation
<code>std::launch::async</code>	enable asynchronous evaluation
<code>std::launch::deferred</code>	enable lazy evaluation

Type requirements

Parallel reduction with `async`

`async`

Basic thread replacement but adds exception and value

Building blocks:

- `std::async`: Request asynchronous execution of a function.
- `Future`: token representing function's result.

Unlike raw use of `std::thread` objects:

- Allows values or exceptions to be returned.
- Just like “normal” function calls.

async

```
double bestValue(int x, int y);           // something callable
std::future<double> f =                  // in concept, run λ
    std::async( []{ return bestValue(10, 20); } ); // asynchronously;
                                                // get future for it
...
                                                // do other work
double val = f.get();                    // get result (or
                                                // exception) from λ
```

As usual, auto reduces verbiage:

```
auto f = std::async( []{ return bestValue(10, 20); } );
...
auto val = f.get();
```

Async Launch Policies

`std::launch::async`: function runs on a new thread.

- Maintains calling thread's responsiveness (e.g., GUI threads).

```
auto f = std::async(std::launch::async, doBackgroundWork);
```

`std::launch::deferred`: function runs on thread invoking `get` or

- wait on `std::async`'s future:

```
auto f = std::async(std::launch::deferred,  
[]{ return bestValue(10, 20); });
```

...

```
auto val = f.get(); // run λ synchronously here
```

- Useful for debugging, performance tuning.

By default, implementation chooses, presumably with goals:

- Take advantage of all hardware concurrency, i.e., scale.
- Avoid oversubscription.

Futures

Two kinds:

- `std::future<T>`: result may be accessed only once.
 - Suitable for most use cases.
 - Moveable, not copyable.
 - Exactly one future has right to access result.
- `std::shared_future<T>`: result may be accessed multiple times.
 - Appropriate when multiple threads access a single result.
 - Both copyable and moveable.
 - Multiple `std::shared_futures` for the same result may exist.
 - Creatable from `std::future`.
 - Such creation transfers ownership.

Futures results

Result retrieval via get:

- Blocks until a return is available, then grabs it.
- For `future<T>`, “grabs” \equiv “moves (if possible) or copies.”
- For `shared_future<T>` or `anyKindOfFuture<T&>`, “grabs” \equiv “gets reference to.”
- “Return” may be an exception (which is then propagated).

Futures Alternatives

An alternative is wait:

Blocks until a return is available.

```
std::future<double> f = std::async([]{ return bestValue(10, 20); });

...
f.wait(); // block until λ is done
```

A timeout may be specified.

Most useful when std::launch::async specified.

```
std::future<double> f =
std::async(std::launch::async, []{ return bestValue(10, 20); });

...
while ( f.wait_for(std::chrono::seconds(0)) != // if result of λ
    std::future_status::ready) { // isn't ready,
    ... // do more work
}
double val = f.get(); // grab result
```

When you don't care about the return from future

Useful when callers want to know only when a callee finishes.

Callable objects returning void.

Callers uninterested in return value.

But possibly interested in exceptions.

```
void initDataStructs(int defValue);  
void initGUI();  
  
std::future<void> f1 = std::async( []{ initDataStructs(-1); } );  
std::future<void> f2 = std::async( []{ initGUI(); } );  
... // init everything else  
  
f1.get(); // wait for asynch. inits. to  
f2.get(); // finish (and get exceptions,  
// if any)  
... // proceed with the program
```

Packaged tasks

Wraps any callable target so it can be invoked asynchronously

Return value or exception is stored in a shared state

Unlike `async`, packaged task is not called automatically after construction, but has to be called explicitly.

To run synchronously, do nothing and it will be executed sequentially

To run asynchronously, have to detach the task

Have to specify template parameter in a special way

```
template< class R, class ...Args >class packaged_task<R(Args...)>;
```

1. `packaged_task<int(int,int)> task(callable objectf)`
1. `packaged_task<int(int,int)> task([](int a, int b) { return std::pow(a, b); })`
1. `packaged_task<int()> task(std::bind(f, 2, 11))`

std::packaged_task - cppreference Coliru X +

https://en.cppreference.com/w/cpp/thread/packaged_task

Create account Search

Page Discussion View Edit History

C++ Thread support library std::packaged_task

std::packaged_task

Defined in header `<future>`

```
template< class > class packaged_task; //not defined (1) (since C++11)
template< class R, class ...Args > (2) (since C++11)
class packaged_task<R(Args...);
```

The class template `std::packaged_task` wraps any *Callable* target (function, lambda expression, bind expression, or another function object) so that it can be invoked asynchronously. Its return value or exception thrown is stored in a shared state which can be accessed through `std::future` objects.

Just like `std::function`, `std::packaged_task` is a polymorphic, allocator-aware container: the stored callable target may be allocated on heap or with a provided allocator. (until C++17)

Member functions

(constructor)	constructs the task object (public member function)
(destructor)	destructs the task object (public member function)
<code>operator=</code>	moves the task object (public member function)
<code>valid</code>	checks if the task object has a valid function (public member function)
<code>swap</code>	swaps two task objects (public member function)

Getting the result

<code>get_future</code>	returns a <code>std::future</code> associated with the promised result (public member function)
-------------------------	--

Execution

<code>operator()</code>	executes the function (public member function)
<code>make_ready_at_thread_exit</code>	executes the function ensuring that the result is ready only once the current thread exits (public member function)
<code>reset</code>	resets the state abandoning any stored results of previous executions (public member function)

Non-member functions

<code>std::swap(std::packaged_task)</code> (C++11)	specializes the <code>std::swap</code> algorithm (function template)
--	---

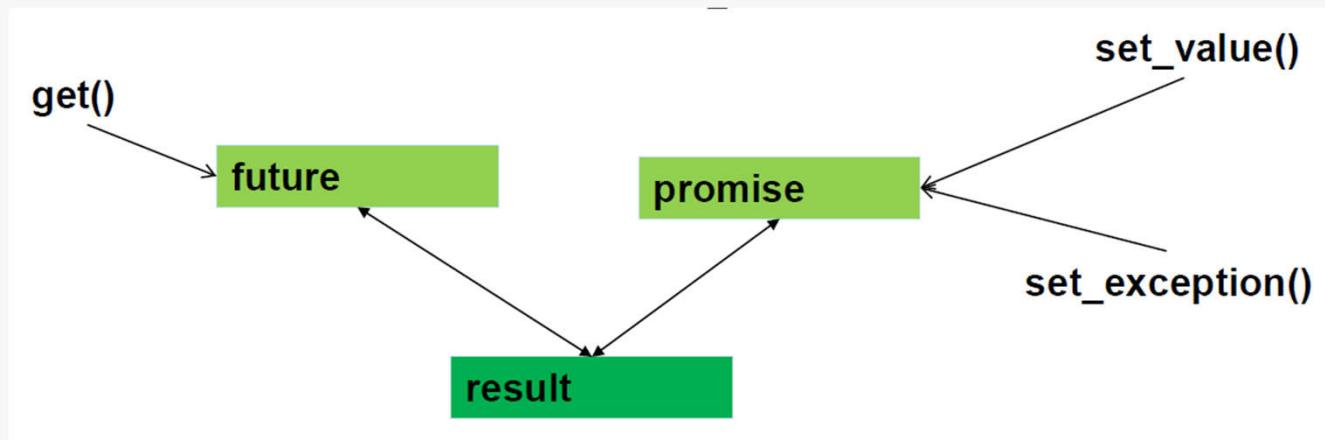
Packaged task

packaged_task

promises

- Promise is paired with a future
- A thread with access to the future object can wait for the result to be set while another thread that has access to the associated promise object can set the value to store it and make the future ready

Future and Promise



- `future+promise` provides a simple way of passing a value from one thread to another
 - No explicit synchronization
 - Exceptions can be transmitted between threads

Future and Promise

- Get from a **future<X>** called f:
X v = f.get(); *// if necessary wait for the value to get*
- Put to a **promise<X>** called p(attached to f):

```
try {  
    X res;  
    // compute a value for res  
    p.set_value(res);  
} catch (...) {  
    // oops: couldn't compute res  
    p.set_exception(std::current_exception());  
}
```

std::promise

Defined in header <future>

```
template< class R > class promise;           (1) (since C++11)
```

```
template< class R > class promise<R&>;      (2) (since C++11)
```

```
template<> class promise<void>; (3) (since C++11)
```

- 1) base template
 - 2) non-void specialization, used to communicate objects between threads
 - 3) void specialization, used to communicate stateless events

The class template `std::promise` provides a facility to store a value or an exception that is later acquired asynchronously via a `std::future` object created by the `std::promise` object. Note that the `std::promise` object is meant to be used only once.

Each promise is associated with a *shared state*, which contains some state information and a *result* which may be not yet evaluated, evaluated to a value (possibly void) or evaluated to an exception. A promise may do three things with the shared state:

- *make ready*: the promise stores the result or the exception in the shared state. Marks the state ready and unblocks any thread waiting on a future associated with the shared state.
 - *release*: the promise gives up its reference to the shared state. If this was the last such reference, the shared state is destroyed. Unless this was a shared state created by `std::async` which is not yet ready, this operation does not block.
 - *abandon*: the promise stores the exception of type `std::future_error` with error code `std::future_errc::broken_promise`, makes the shared state *ready*, and then *releases* it.

The promise is the "push" end of the promise-future communication channel: the operation that stores a value in the shared state *synchronizes-with* (as defined in `std::memory_order`) the successful return from any function that is waiting on the shared state (such as `std::future::get`). Concurrent access to the same shared state may conflict otherwise: for example multiple callers of `std::shared_future::get` must either all be read-only or provide external synchronization.

Member functions

(constructor)	constructs the promise object (public member function)
(destructor)	destructs the promise object (public member function)
operator=	assigns the shared state (public member function)
swap	swaps two promise objects (public member function)

Getting the result

Promises

promises

Promises and exception

```
1. #include <iostream>
2. #include <thread>
3. #include <future>
4. #include <stdexcept>
5. void throw_exception(){}
6.     throw std::invalid_argument("input cannot be negative");
7. }
8. void calculate_square_root(std::promise<int>& prom){
9.     int x=1;
10.    std::cout << "Please, enter an integer value: ";
11.    try {
12.        std::cin >> x;
13.        if (x < 0)
14.        {
15.            throw_exception();
16.        }
17.        prom.set_value(std::sqrt(x));
18.    }
19.    catch (std::exception&)
20.    {
21.        prom.set_exception(std::current_exception());
22.    }
}
```

```
1. void print_result(std::future<int>& fut) {
2.     try {
3.         int x = fut.get();
4.         std::cout << "value: " << x << '\n';
5.     }
6.     catch (std::exception& e) {
7.         std::cout << "[exception caught: " << e.what()
8.             << "]\n";
9.     }
10. }
11. int main(){
12.     std::promise<int> prom;
13.     std::future<int> fut = prom.get_future();
14.
15.     std::thread printing_thread(print_result, std::ref(fut));
16.     std::thread calculation_thread(calculate_square_root,
17.                                     std::ref(prom));
18.
19.     printing_thread.join();
20.     calculation_thread.join();
21.     return 0;
22. }
```

Key takeaways

Supports Lock-based data structures

Always Maintain invariants

No race conditions and deadlocks

Handle exceptions



Questions?



Chapter 4b: C++ Atomics & Memory Model

Michael Wong

CppCon 2019 – Sep 2019

- Learning objectives:

- Learn about what a memory model is and why we need one
- Learn about atomics

Memory Model and Consistency model, a quick tutorial

- Sequential Consistency (SC)

Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:

- “... the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program”
- But chip/compiler designers can be annoyingly helpful:
- It can be more expensive to do exactly what you wrote.
- Often they'd rather do something else, that could run faster

Sequential Consistency: a tutorial

- The semantics of the **single threaded** program is defined by the program order of the statements. This is the strict sequential order. For example:

x = 1;

r1 = z;

y = 1;

r2 = w;

Sequential Consistency for program understanding

- Suppose we have two threads.

Thread 1 is the sequence of statement above. Thread 2 is:

Thread 1:

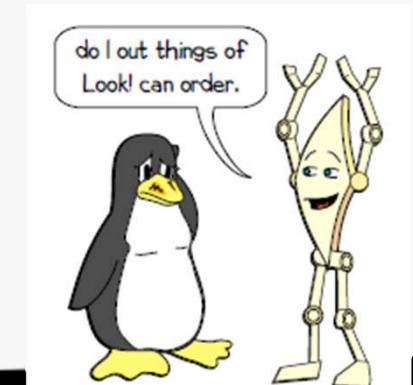
$x = 1;$ $w=1;$
 $r1 = z;$ $r3=y;$
 $y = 1;$ $z=1;$
 $r2 = w;$ $r4=x;$

Thread2:

- **2 of 4! Possible interleavings:**

$x = 1;$ $x=1;$
 $w = 1;$ $w=1;$
 $r1 = z;$ $r3=y;$
 $r3 = y;$ $z=1;$
 $y = 1;$ $r4=x;$
 $z = 1;$ $r1=z;$
 $r2 = w;$ $y=1;$
 $r4 = x;$ $r2=w;$

(All variables are initialized to zero.)



Now add fences to control reordering

Thread 1:

```
x = 1;
```

```
r1 = z;
```

```
fence();
```

```
y = 1;
```

```
r2 = w;
```

Is r3==1 and r4==1 possible?

Is r1==1 and r2==1 possible?

Thread2:

```
w=1;
```

```
r3=y;
```

```
fence();
```

```
z=1;
```

```
r4=x;
```



Memory Model and instruction reordering

- Definitions:
- Instruction reordering: When a program executes instructions, especially memory reads and writes, in an order that is different than the order specified in the program's source code.
- Memory model: Describes how memory reads and writes may appear to be executed relative to their program order.
- Affects the valid optimizations that can be performed by compilers, physical processors, and caches.

What is a memory model?

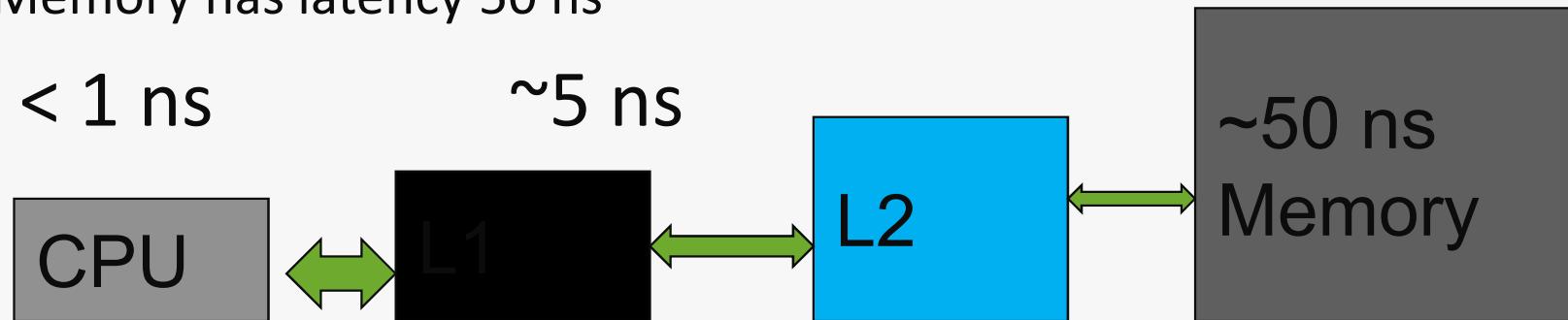
- A set of rules that describe allowable semantics for memory accesses on a computer program
 - Defines the expected value or values returned by any loads in the program
 - Determines when a program produces undefined behavior (e.g load from uninitialized variables)
- Stroustrup: represents a contract between the implementers and the programmers to ensure that most programmers do not have to think about the details of modern computer hardware
- critical component of concurrent programming

Thread switching and you

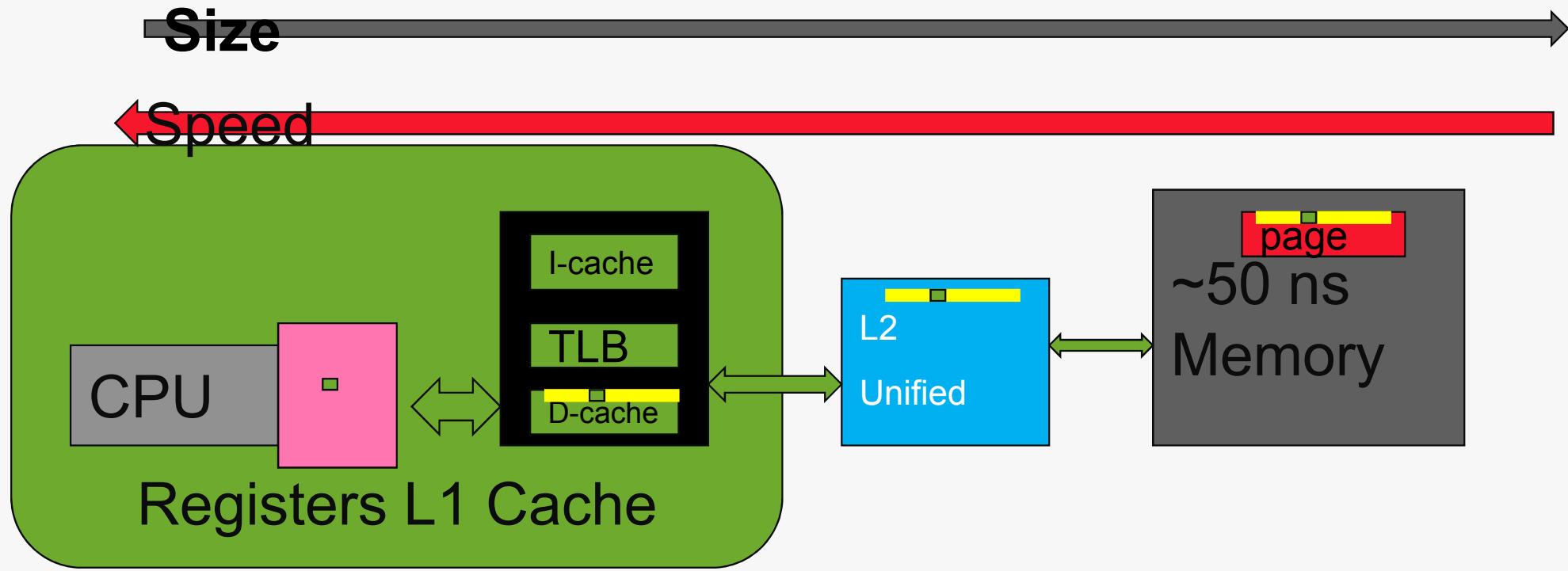
- Execution of a sequential program
- Software, not hardware
- A processor can run a thread
- Put it aside
 - Thread does I/O
 - Thread runs out of time
- You work in an office
- When you leave for lunch, someone else takes over your office.
- If you don't take a break, a security guard shows up and escorts you to the cafeteria.
- When you return, you may get a different office

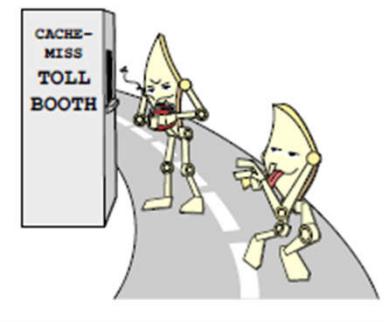
Memory vs CPU

- Memory latency is crucial to getting right performance, and affects parallelization
- Knowing the memory characteristics will help to write faster code, design language
- 1 GHz means one result every nanosecond
- Memory has latency 50 ns

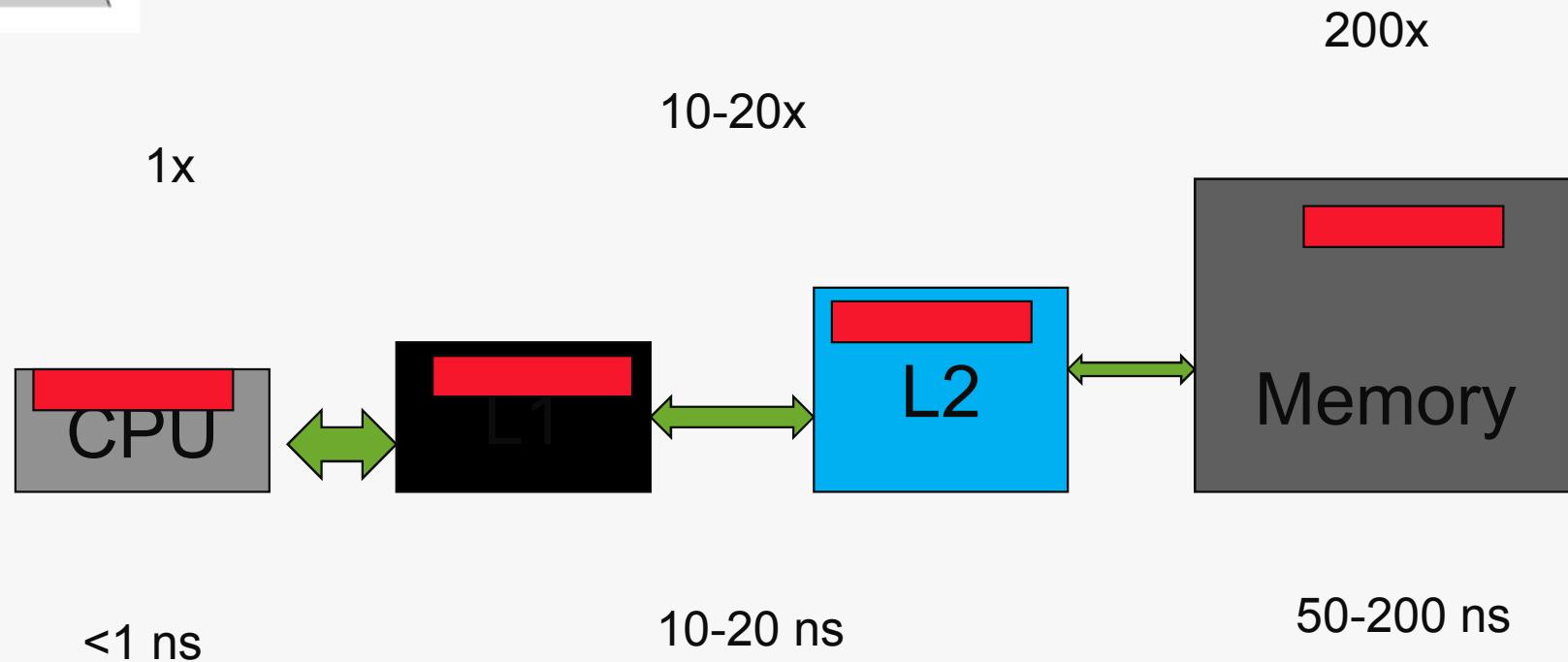


The memory hierarchy

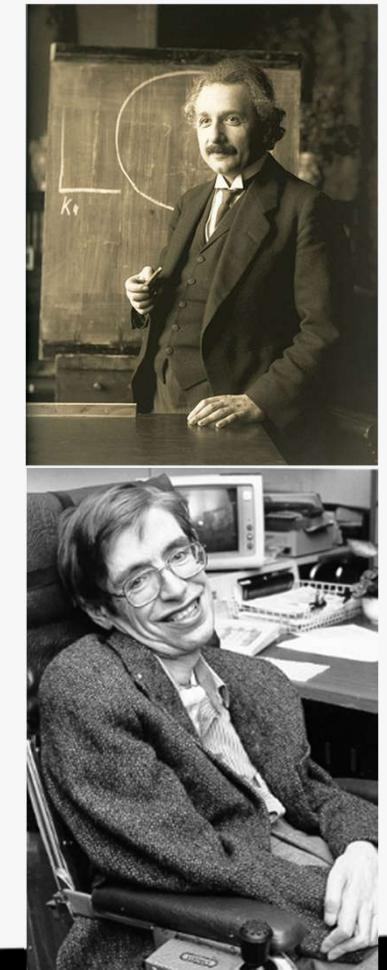
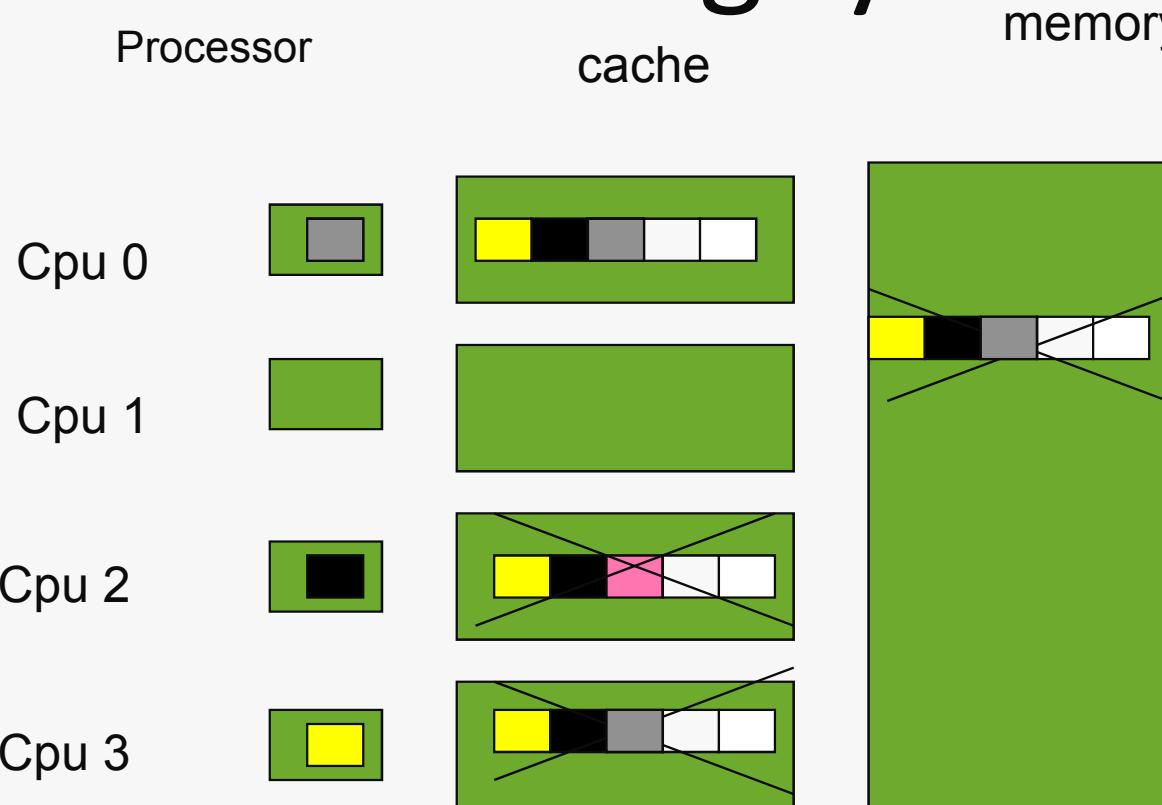




Caches and Memory

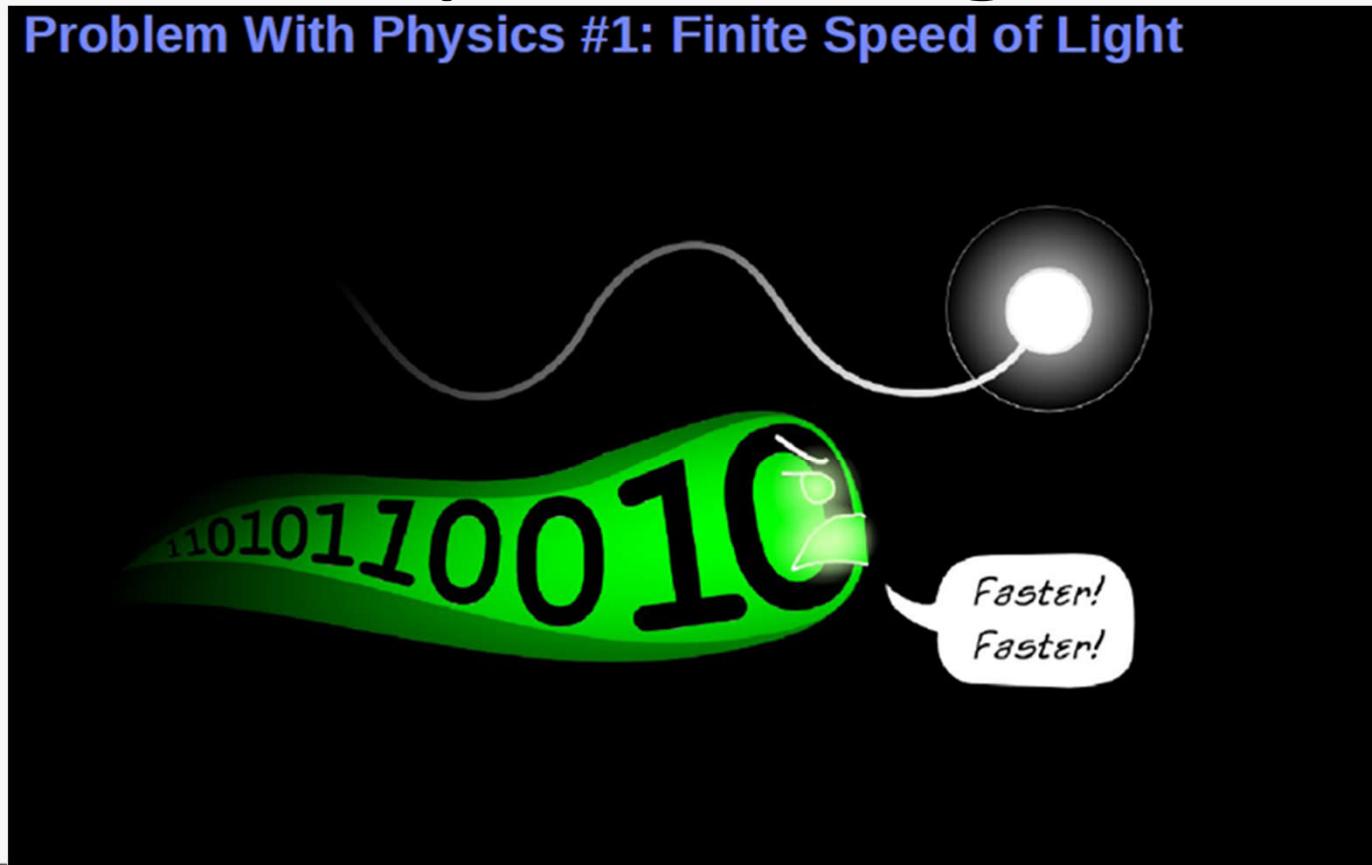


Caches in an MP system and why do these guys hate us?



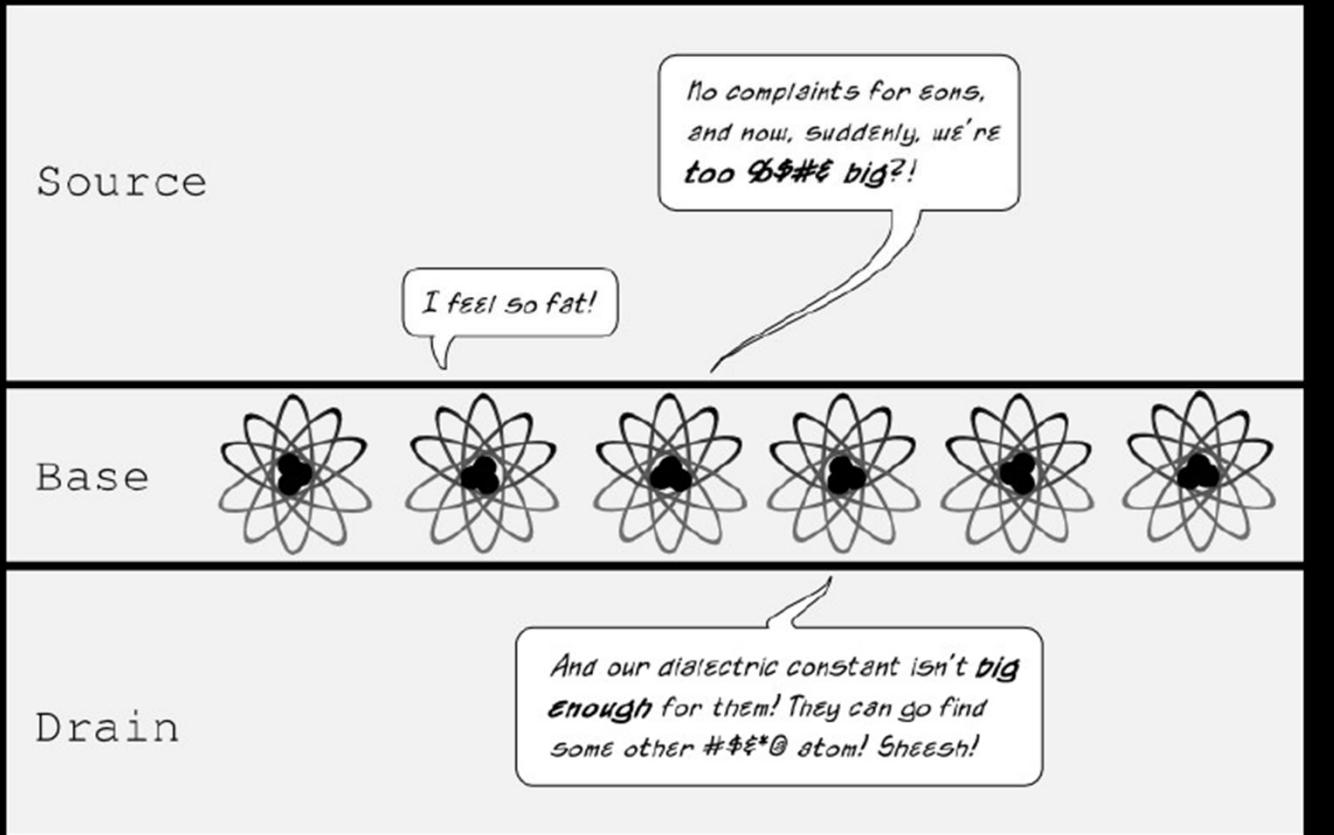
Problem With Physics #1:finite speed of light

Problem With Physics #1: Finite Speed of Light



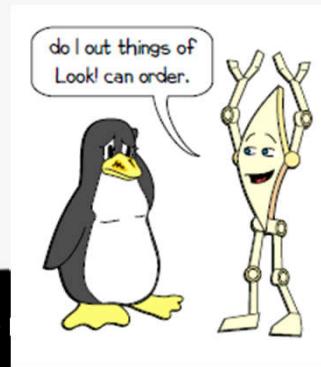
Problem with Physics #2: atomic nature of matter

Problem With Physics #2: Atomic Nature of Matter



Memory Model

- One of the most important aspect of C++0x /C1x is almost invisible to most programmers
 - memory model
 - How threads interact through memory
 - What assumptions the compiler is allowed to make when generating code
 - 2 aspects
 - How things are laid out in memory
 - What happens when two threads access the same memory location and one of them is a modify
 - » Data race
 - » Modification order
 - » **Atomicity/isolation**
 - » **Visibility**
 - » **Ordering**



Memory Model

- Locks and atomic operations communicate non-atomic writes between two threads
- **Volatile is not atomic (this is not Java)**
- Data races cause undefined behavior
- Some optimizations are no longer legal

Message shared memory

- Writes are explicitly communicated
 - Between pairs of threads
 - Through a lock or an atomic variable
- The mechanism is acquire and release
 - One thread releases its memory writes
 - `V=32; atomic_store_explicit(&a,3, memory_order_release);`
 - Another thread acquires those writes
 - `i=atomic_load_explicit(&a, memory_order_acquire);`

What is a memory location

- A non-bitfield primitive data object
- A sequence of adjacent bitfields
 - Not separated by a structure boundary
 - Not interrupted by the null bitfield
 - Avoid expensive atomic read-modify-write operations on bitfields

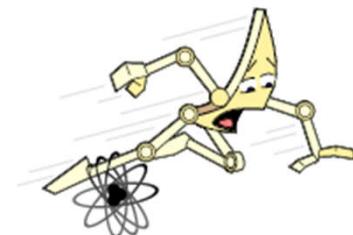
Data race condition

1. A non-atomic write to a memory location in one thread AND
2. A non-atomic read from or write to that same location in another thread AND
3. With no happens-before relations between them
 - Result in **Undefined Behaviour**

Atomics: To Volatile or Not Volatile

- Too much history in volatile to change its meaning
- It is not used to indicate atomicity like Java
- Volatile atomic
 - means something from the environment may also change this in addition to another thread OR
 - Real time requirements the compiler does not know about, so it further restricts compiler optimization

Atomic Design



- Want shared variables
 - **that can be concurrently updated without introducing data race,**
 - **that are atomically updated and read**
 - half updated states are not visible,
- that are implemented without lock overhead whenever the hardware allows,
- that provide access to hardware atomic read-modify write (fetch-and-add, xchg, cmpxchg) instructions whenever possible.

Race Free semantics and Atomic Memory operations

- If a program has a race, it has undefined behavior
 - This is sometimes known as “catch fire” semantics
 - No compiler transformation is allowed to introduce a race
 - More restrictions on invented writes
 - Possibly fewer speculative stores **and (potentially) loads**
- There are atomic memory operations that don’t cause races (or they race but are well-defined)
 - Can be used to implement locks/mutexes
 - Also useful for lock-free algorithms
- Atomic memory operations are expressed as library function calls
 - Reduces need for new language syntax

Atomic Operations and Type

- Data race: if there is no enforced ordering between two accesses to a single memory location from separate threads, one or both of those accesses is not atomic, and one or both is a write, this is a data race, and causes undefined behavior.
- These types avoid undefined behavior and provide an ordering of operations between threads

Lock-free atomics

- Large atomics have no hardware support
 - Implemented with locks
- Locks and signals don't mix
 - Test for lock-free
- Compile-time macros for basic types
 - Always lock-free
 - Never lock-free
 - May be lock-free
- RTTI for each type

Memory Ordering Operations

```
enum memory_order {  
    memory_order_relaxed, // just atomic, no constraint  
    memory_order_release,  
    memory_order_acquire,  
    memory_order_consume,  
    memory_order_acq_rel, // both acquire and release  
    memory_order_seq_cst}; // sequentially consistent
```

- Every atomic operation has a default form, implicitly using seq_cst, and a form with an explicit order argument
- When specified, argument is expected to be just an enum constant

3 Memory Models, but 6 Memory Ordering Constraints

- Sequential Consistency
 - Single total order for all SC ops on all variables
 - default
- Acquire/Release/consume
 - Pairwise ordering rather than total order
 - Independent Reads of Independent Writes don't require synchronization between CPUs
- Relaxed Atomics
 - Read or write data without ordering
 - Still obeys happens-before
- Operations on variable have attributes, which can be explicit

Operations available on atomic types

	atomic_flag	bool/other-type	T*	integral
test_and_set, clear	Y			
is_lock_free		Y	Y	Y
load, store, exchange, compare_exchange_weak +strong		Y	Y	Y
fetch_add (+=), fetch_sub (-=), ++, --			Y	Y
fetch_or (=), fetch_and (&=), fetch_xor (^=)				Y

Examples

```
int x = 0;  
atomic<int> y = 0;  
Thread 1:  
    x = 17;  
    y.store(1,  
            memory_order_release);  
    // or:    y.store(1);
```

```
Thread 2:  
    while  
        (y.load(memory_order_acquire)  
        != 1)  
    // or:    while (y.load() != 1)  
  
    assert(x == 17);
```

```
int x = 0;  
atomic<int> y = 0;  
Thread 1:  
    x = 17;  
    y = 1;  
Thread 2:  
    while (y != 1)  
        continue;  
    assert(x == 17);
```

Key takeaways

Atomics for experts only

Supports Lock free data structures

Use sequential consistency for understanding

Use more relaxed models for performance



Questions?



Chapter 5: Introduction to SYCL

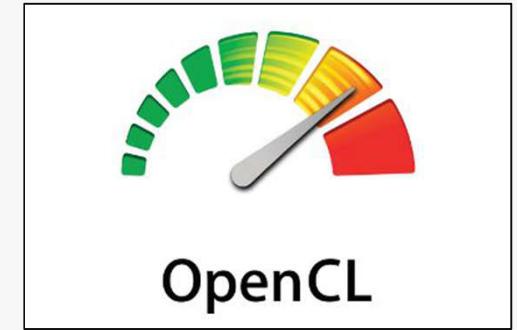
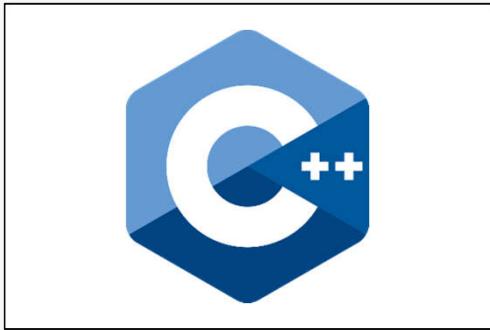
Gordon Brown

CppCon 2019 – Sep 2019

- Learning objectives:

- Learn about the SYCL 1.2.1 specification and its implementations
- Learn about the major features that SYCL provides
- Learn about the components of a SYCL implementation
- Learn about the anatomy of a typical SYCL application
- Learn where to find useful resources for SYCL

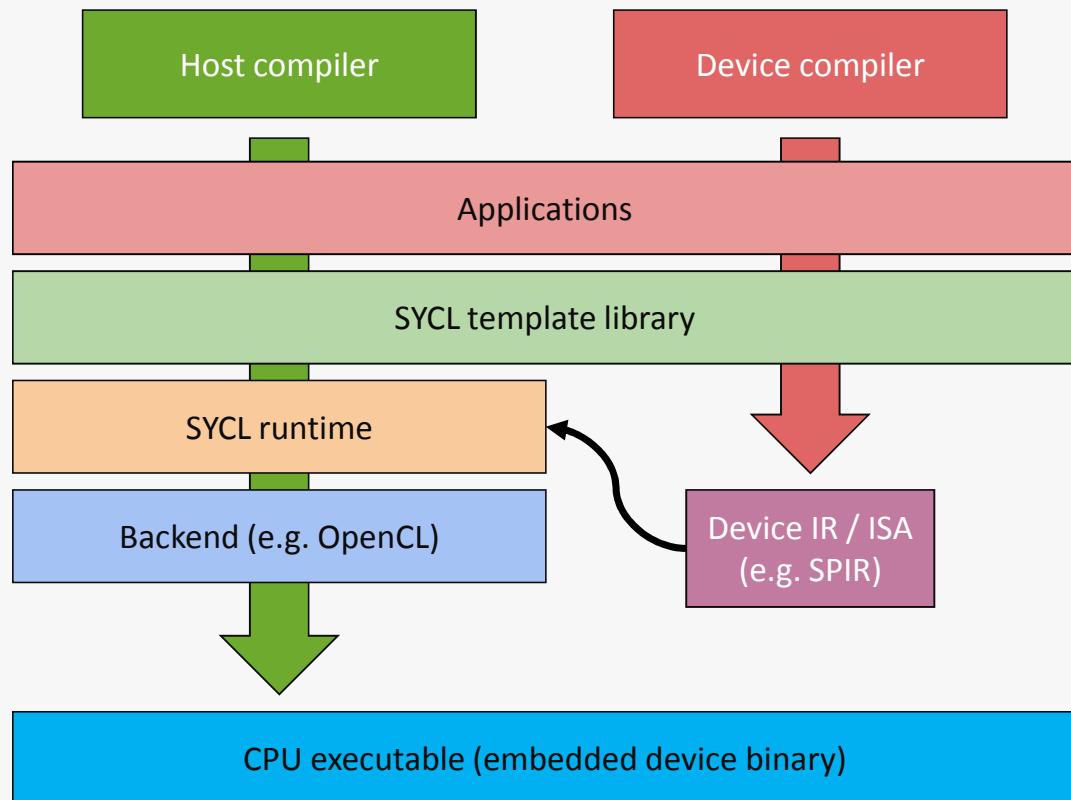
What is SYCL?



SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

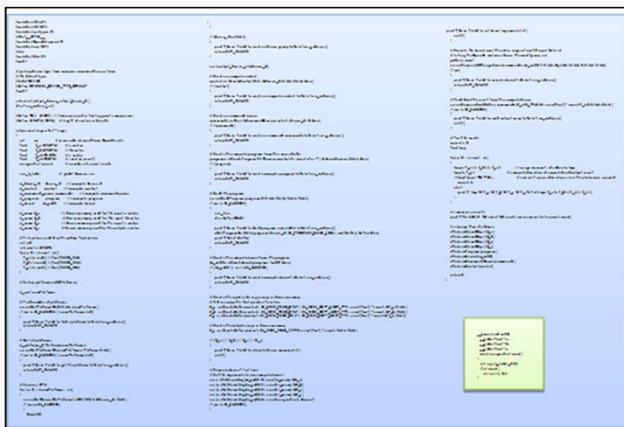
SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

SYCL is a **single-source**, high-level, standard C++ programming model, that can target a range of heterogeneous platforms



- SYCL allows you write both host CPU and device code in the same C++ source file
 - This requires two compilation passes; one for the host code and one for the device code

SYCL is a single-source, **high-level**, standard C++ programming model, that can target a range of heterogeneous platforms



Typical OpenCL hello world application



Typical SYCL hello world application

- SYCL provides high-level abstractions over common boilerplate code
 - Platform/device selection
 - Buffer creation
 - Kernel compilation
 - Dependency management and scheduling

SYCL is a single-source, high-level, **standard C++** programming model, that can target a range of heterogeneous platforms

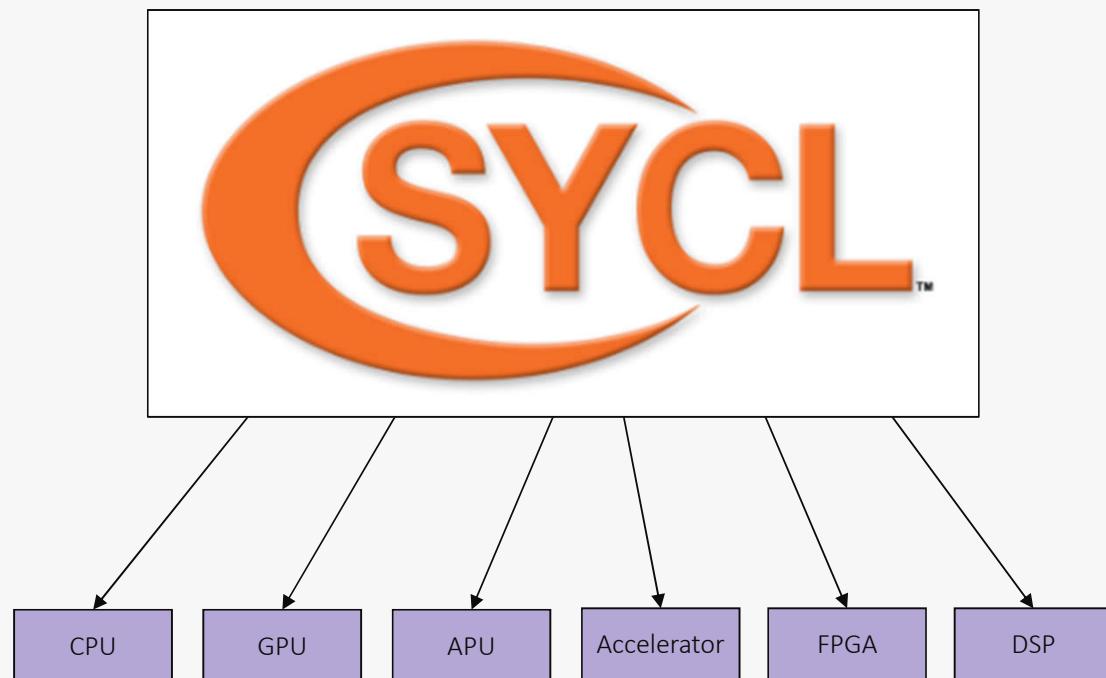
```
array view<float> a, b, c;

std::vector<float> a, b, c;
#pragma parallel_for
for(int i = 0; i < a.size(); i++) {
    c[i] = global vec_add(a[i], b[i]);
}
float *a, *b, *c;
vec_add<<<range>>>(a, b, c);
```

- SYCL allows you to write standard C++
 - No language extensions
 - No pragmas
 - No attributes

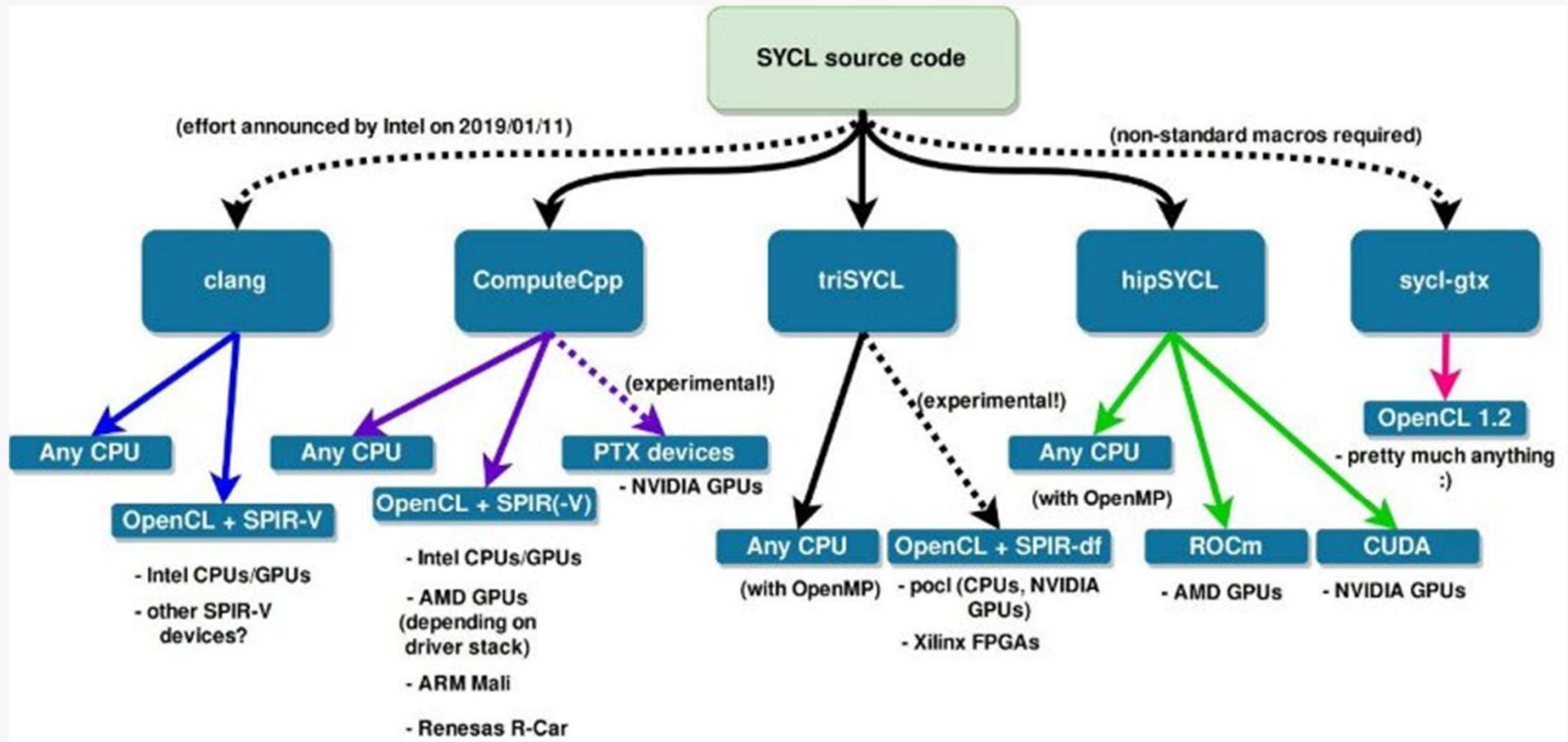
```
cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
    c[idx] = a[idx] + b[idx];
});
```

SYCL is a single-source, high-level, standard C++ programming model, that can **target a range of heterogeneous platforms**

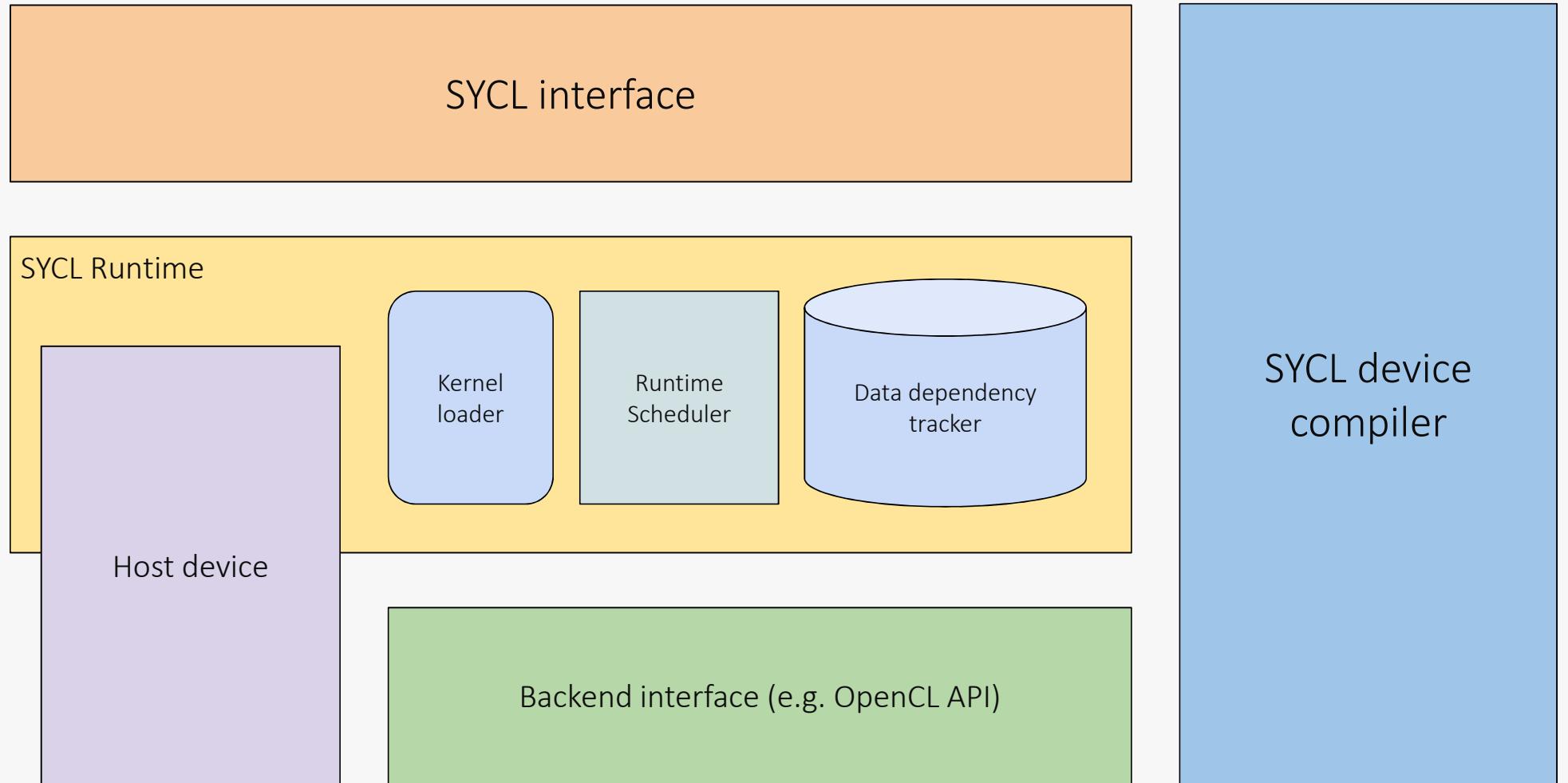


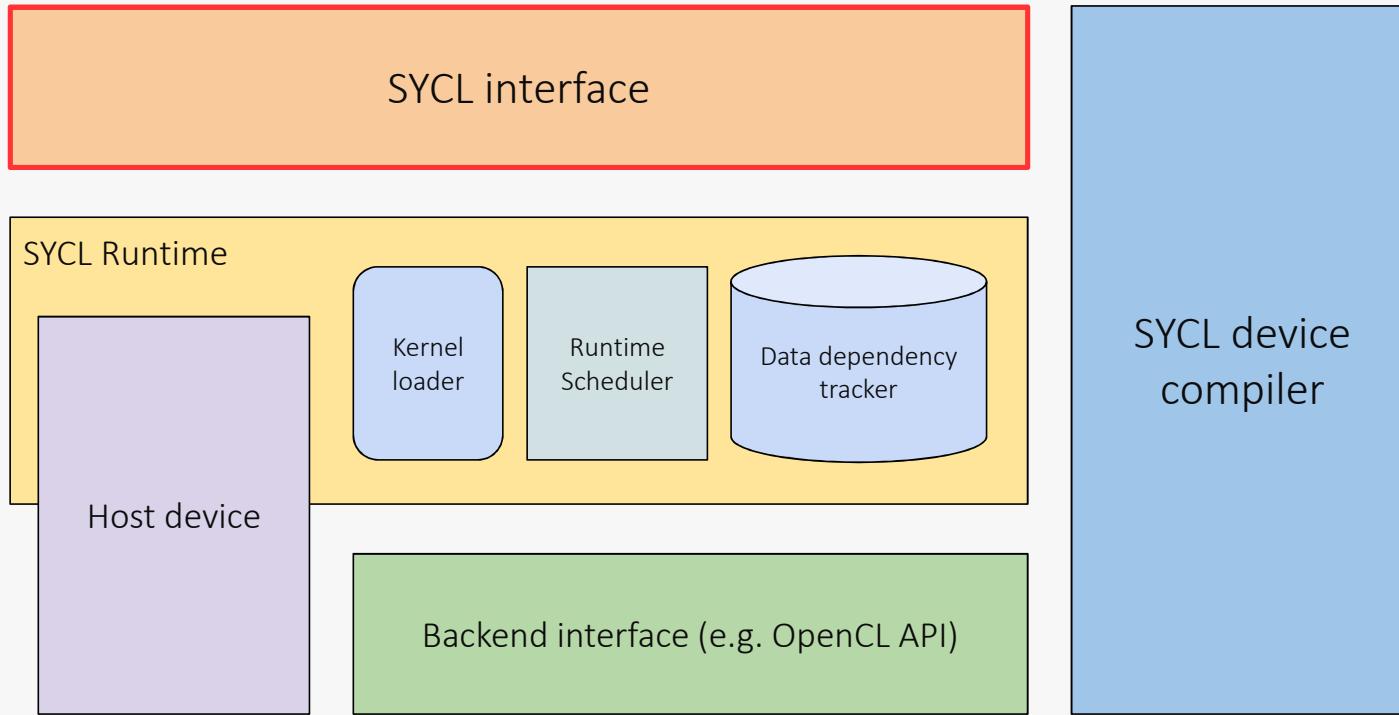
- SYCL can target any device supported by its backend
- SYCL can target a number of different backends
 - Currently the specification is limited to OpenCL
 - Some implementations support other non-standard backends

Who is implementing SYCL?

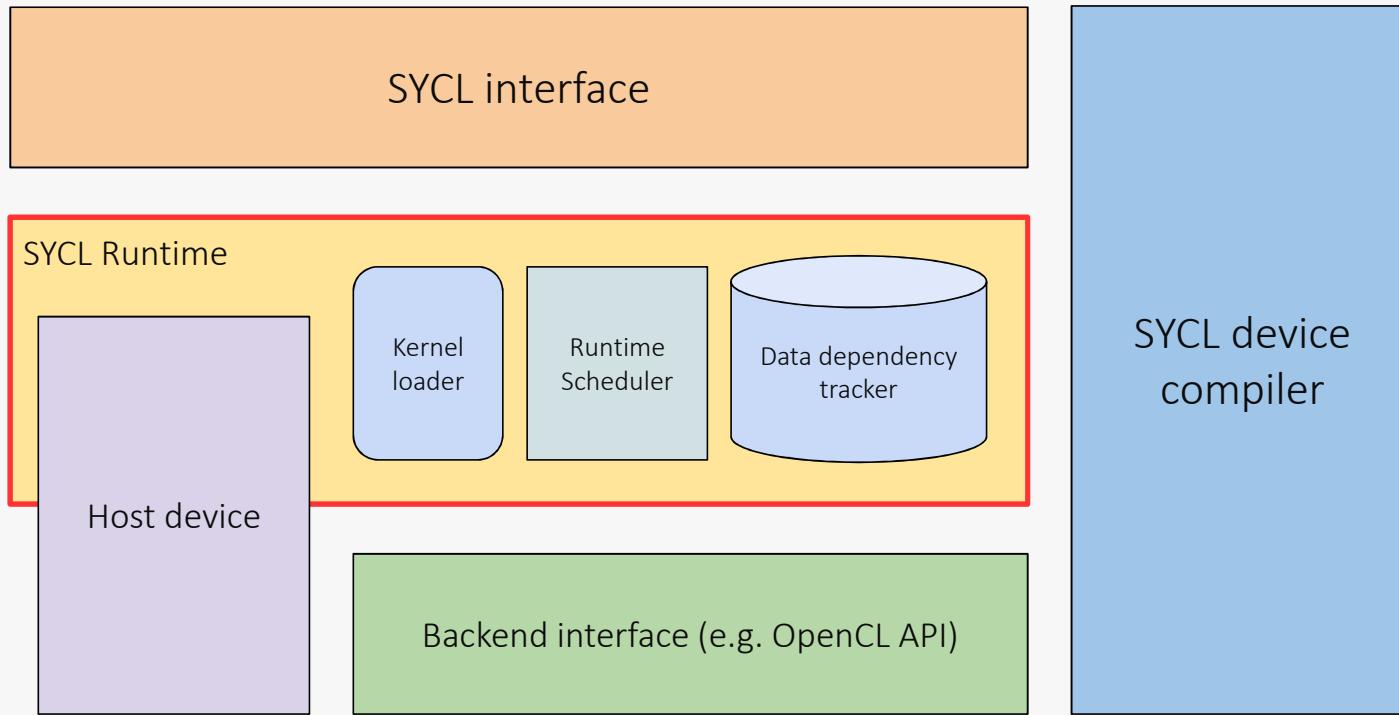


What is in a SYCL implementation?

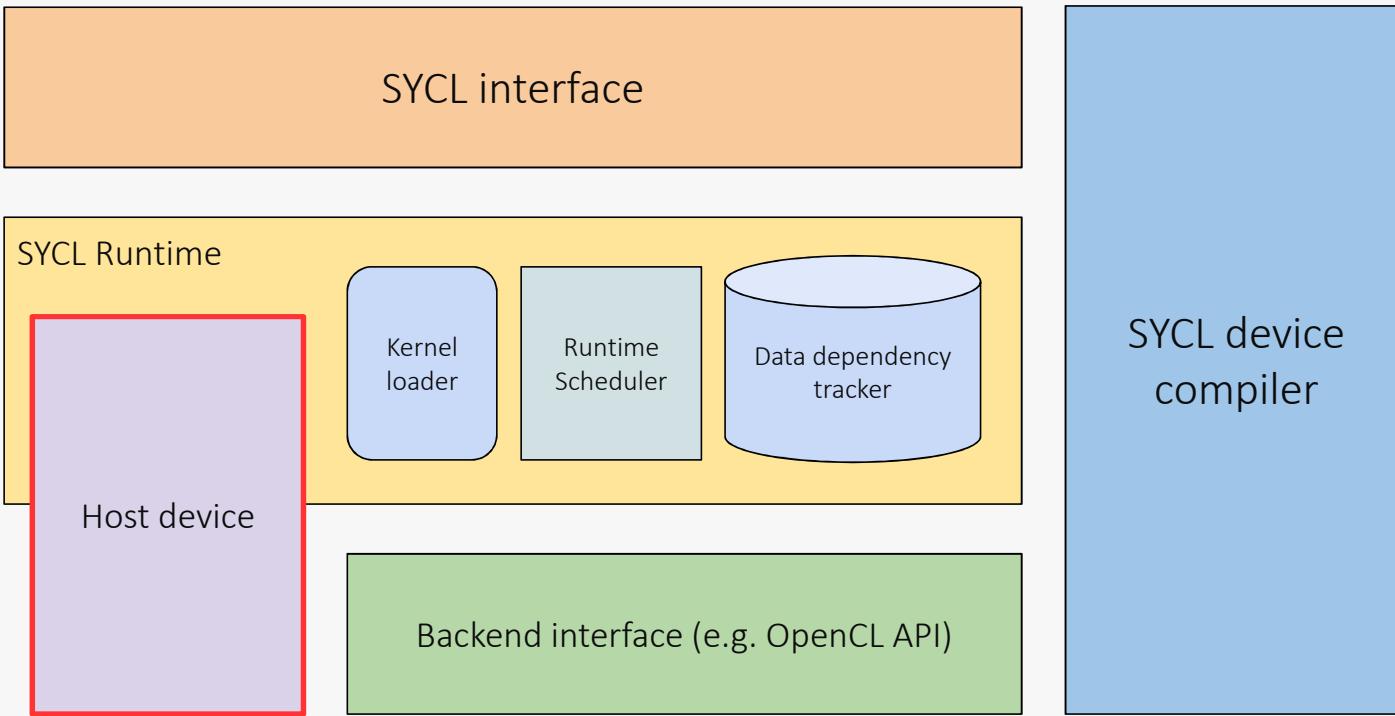




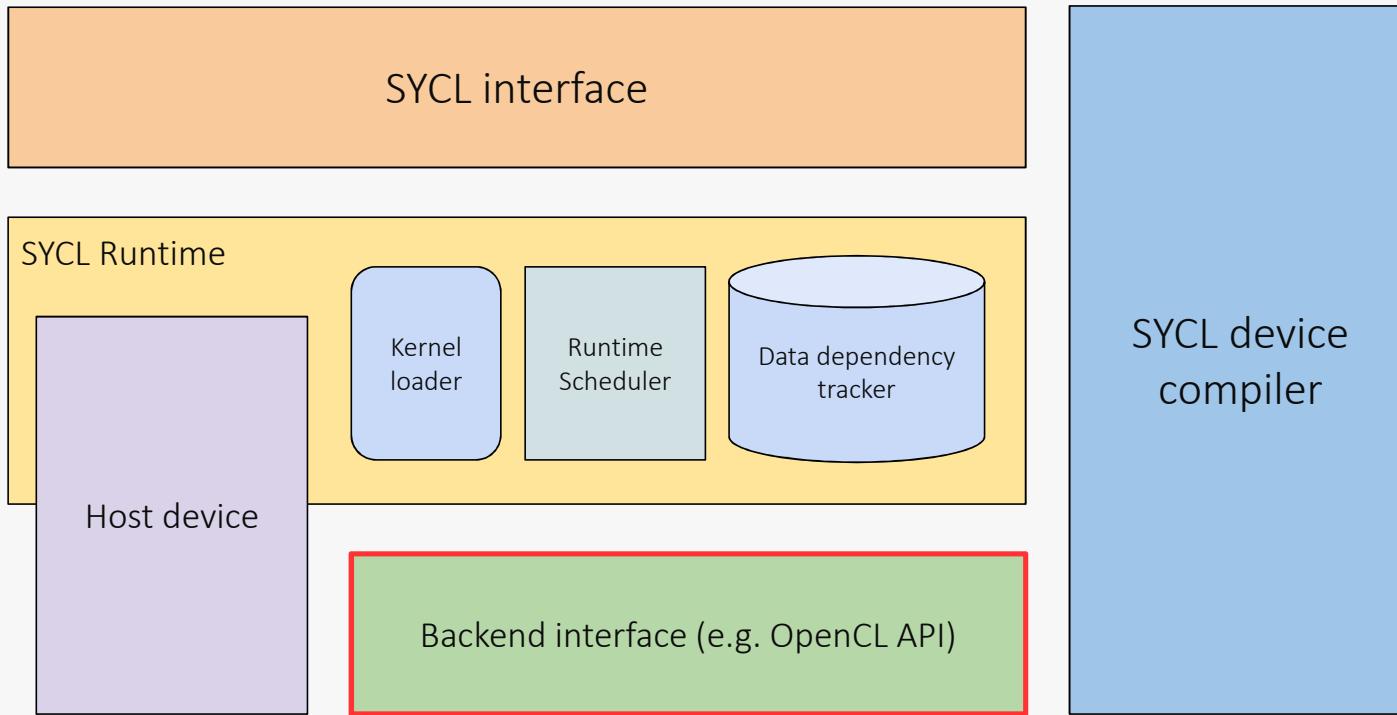
- The SYCL interface is a C++ template library that users and library developers program to
 - The same interface is used for both the host and device code



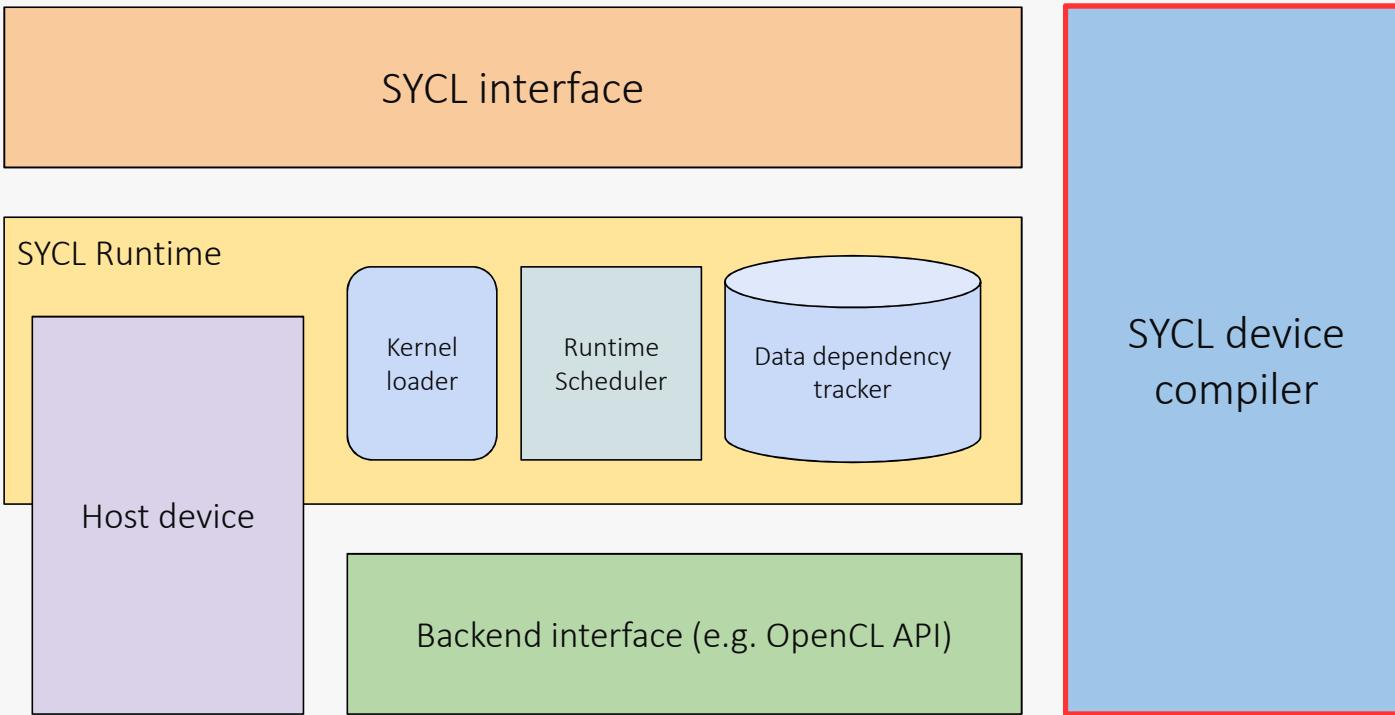
- The SYCL runtime is a library that schedules and executes work
 - It loads kernels, tracks data dependencies and schedules commands



- The host device is an emulated backend that is executed as native C++ code and emulates the SYCL execution and memory model
 - The host device can be used without backend drivers and for debugging purposes



- The backend interface is where the SYCL runtime calls down into a backend in order to execute on a particular device
 - The standard backend is OpenCL but some implementations have supported others



- The SYCL device compiler is a C++ compiler which can identify SYCL kernels and compile them down to an IR or ISA
 - This can be SPIR, SPIR-V, GCN, PTX or any proprietary vendor ISA

VL1

VL1 this backend is not working, better remove it for now
Victor Lomuller, 31/07/2019

What does a SYCL application look like?

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

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

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

}

}
```

First we include the SYCL header which contains the runtime API

We also import the cl::sycl namespace here for the purposes of presenting

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

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

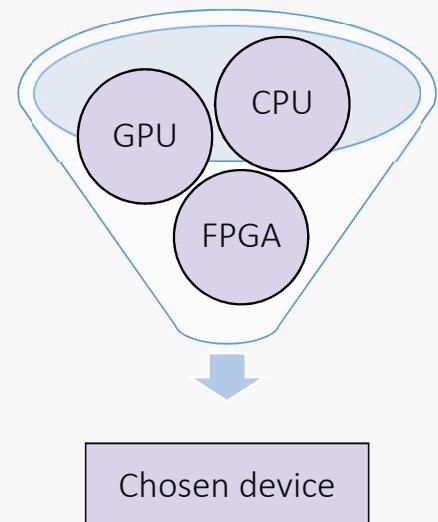
    queue gpuQueue{gpu_selector{}};

}

}
```

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

The queue default constructor uses a the `default_selector`, which allows the runtime to select a device for you



```

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

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

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

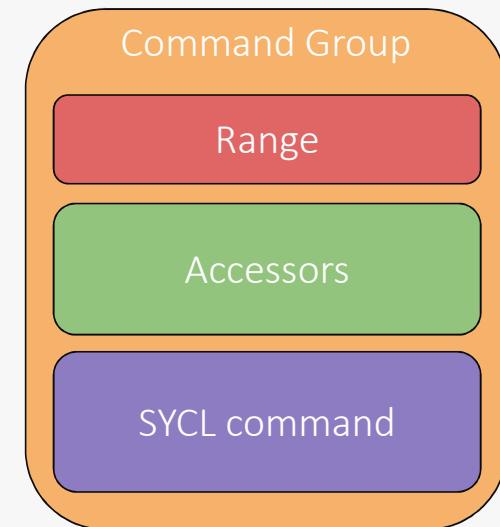
}

```

With a queue we can submit a command group

A command group contains:

- A SYCL command (e.g. a SYCL kernel function)
- Execution range
- Accessors



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

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

    queue gpuQueue{gpu_selector{}};

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

    }) ;

}
```

We initialize three vectors, two inputs and an output

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

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

    queue gpuQueue{gpu_selector{}};

    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) {

    });

}
```

We create a buffer for each vector to manage the data across host and device

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

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

    queue gpuQueue{gpu_selector{}};

    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) {

    });

}
```

Buffers synchronize on destruction via RAII

So adding this scope means that all kernels writing to the buffers will wait and the data will be copied back to the vectors on leaving this scope

```

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

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

    queue gpuQueue{gpu_selector{}};

    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);

    });

}

```

We create an accessor for each of the buffers

Read access for the two input buffers and write access for the output buffer

```

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

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

    queue gpuQueue{gpu_selector{}};

    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]; };
    });
}

}

```

We define a SYCL kernel function for the command group using the parallel_for API

The first argument here is a range, specifying the iteration space

The second argument is a lambda function that represents the entry point for the SYCL kernel

This lambda takes an id parameter that describes the current iteration being executed

```

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

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

    queue gpuQueue{gpu_selector{}};

    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]; });
    });

}

```

The template parameter to `parallel_for` is used to name the lambda

The reason for this is that C++ does not have a standard ABI for lambdas so they are represented differently across the host and device compiler

SYCL kernel functions follow C++ ODR rules, which means that if a SYCL kernel is in a template context, the kernel name needs to reflect that context, so must contain the same template arguments

```

#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 */ }
}

```

In SYCL errors are handled using exception handling, so you should always wrap SYCL code in a try-catch block

Some exceptions are thrown synchronously at the point of using a SYCL API

Other exceptions are asynchronous and are stored by the runtime and passed to an **async handler** when the queue is told to throw

Useful SYCL resources

- The latest SYCL specification is SYCL 1.2.1
 - Available at: <https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf>
- The specification is now available open source
 - Github project: <https://github.com/KhronosGroup/SYCL-Docs>



 A screenshot of a GitHub repository page for "KhronosGroup / SYCL-Docs". The repository has 1 commit, 0 branches, 0 releases, and 1 contributor. The main file listed is "SYCL 1.2.1 revision 5 specification.pdf". Below the file listing is a section titled "SYCL-Docs" which contains instructions for building the specification.

SYCL Open Source Specification

1 commit · 0 branches · 0 releases · 1 contributor · View license

[Create new file](#) · [Upload files](#) · [Find file](#) · [Close or download](#)

[SYCL 1.2.1 revision 5 specification.pdf](#) · Last commit [12 days ago](#)

[Index](#) · [CODE_OF_CONDUCT.md](#) · [COPYRIGHT.txt](#) · [LICENSE.txt](#) · [README.txt](#)

SYCL-Docs

SYCL Open Source Specification

This repository contains the source and tool chain used to generate the formal SYCL specifications found on <https://www.khronos.org/sycl/>

Building

To build the PDF you need the `LaTeX` framework installed.

A good start in Debian/Ubuntu for example is to install the `texlive-full` and `make` packages.

To build the PDF specification

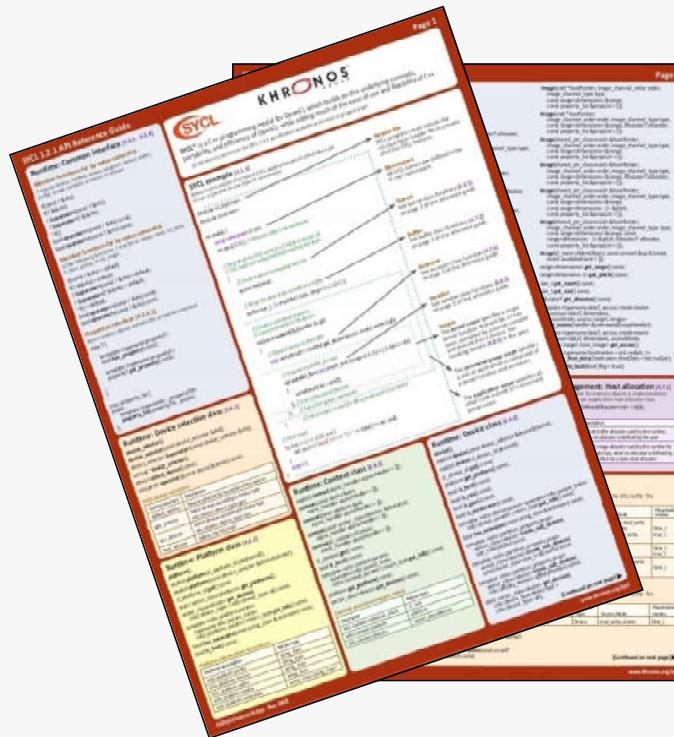
- There is a Khronos backed website for collecting SYCL related news and articles
 - Available at: <http://sycl.tech/>

The screenshot shows the homepage of sycl.tech. At the top, there is a navigation bar with links for News, Videos, Projects, Careers, Specification, and Get SYCL. The main title "SYCL" is prominently displayed. A search bar with the placeholder "search sycl.tech" is located on the right. Below the navigation, there is a section titled "News & Updates" featuring several news items:

- hipSYCL Gets New Compilation Toolchain For Taking SYCL Directly To CUDA & ROCm** (Phoronix.com) - A code snippet is shown on the left, and a thumbnail image of a dragon-like creature is on the right. The article was posted 6 days ago.
- Intel: SYCL compiler - zero-cost abstraction and type safety for heterogeneous computing** (Llvm.org) - A thumbnail image of a dragon-like creature is shown. The article was posted a month ago.
- Intel Continues Working On Their SYCL Compiler For Upstreaming To LLVM** (Phoronix.com) - A thumbnail image of a dragon-like creature is shown. The article was posted a month ago.
- Khronos Continues Working On Better OpenCL + LLVM Integration with SYCL** (Phoronix.com) - A thumbnail image of a dragon-like creature is shown. The article was posted 2 months ago.
- Codeplay Software Releases ComputeCpp Professional Edition** (Khronos.org) - A thumbnail image of a dark interface with the ComputeCpp logo is shown. The article was posted 2 months ago.

At the bottom right of the page, there is a message: "Activate Windows" and "Go to Settings to activate Windows."

- There are Khronos produced SYCL 1.2.1 reference cards
 - Available at: <https://www.khronos.org/files/sycl/sycl-121-reference-card.pdf>



- There is an API reference for ComputeCpp
 - Available at: <https://developer.codeplay.com/products/computecpp/ce/api-reference>

The screenshot shows the API Reference page for ComputeCpp Community Edition. The left sidebar contains a tree view of the API structure, starting with `cl.h` and its sub-directories like `cl::sycl`, `cl::sycl::accessor`, etc. The main content area features three sections: **Recommended** (cl::sycl::event, cl::sycl::buffer, cl::sycl::gpu_selector, cl::sycl::context, cl::sycl::program, cl::sycl::private_memory, cl::sycl::stream, device.h), **Most Visited** (cl::sycl::buffer, cl::sycl::accessor, cl::sycl::handler, cl::sycl, cl::sycl::stream, cl::sycl::h_item), and **Newest** (sycl_interface.h, sycl_math_builtin_symbols.h, sycl_math_builtins_common.h, sycl_math_builtins_floating_point.h, sycl_math_builtins_fp_half_precision.h). A search bar at the top right allows users to search for functions, classes, enums, etc. The bottom of the page includes links for social media sharing, copyright information (© Codeplay Software Ltd.), privacy policy, contact us, and activation instructions for Windows.

Key takeaways

SYCL is an open standard, single source C++ programming model

SYCL allows you to target a wide range of heterogenous devices including GPUs

Checkout the SYCL reference cards to learn more about the programming API



Questions?

Exercise 0

Setting up ComputeCpp

- How to install ComputeCpp and its dependencies
- How to create and compile a SYCL application using ComputeCpp



Chapter 6: CPU & GPU Architecture

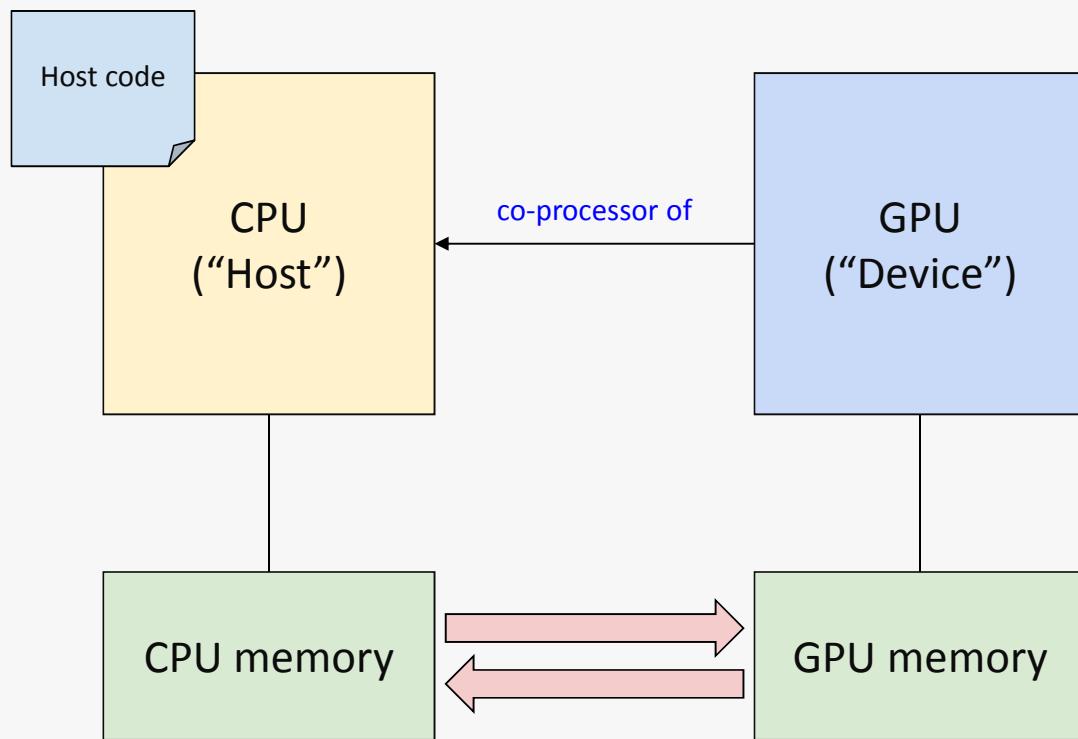
Gordon Brown

CppCon 2019 – Sep 2019

- Learning objectives:

- Learn about typical CPU and GPU architectures
- Learn about the key differences between the CPU and GPU

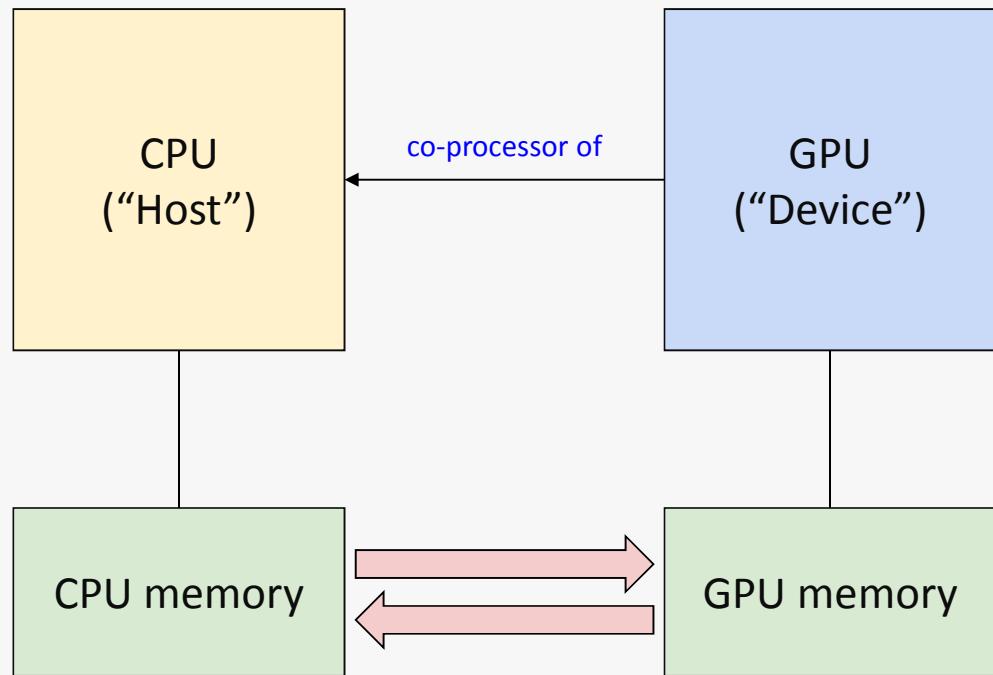
Programming on heterogeneous systems



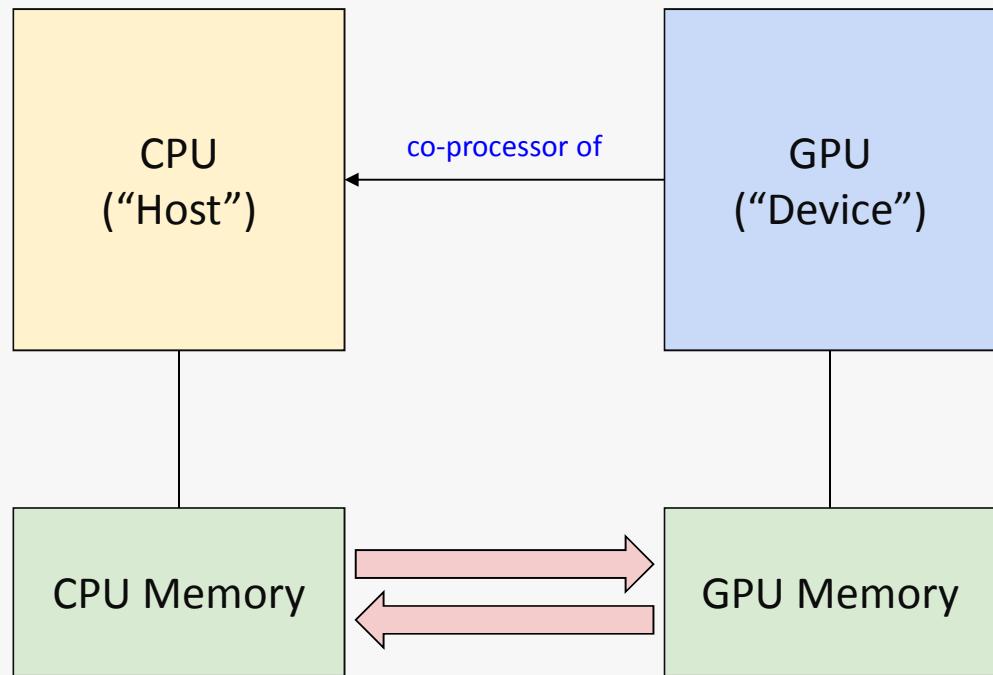
CPU vs GPU

- Small number of large processors
 - More control structures and less processing units
 - Can do more complex logic
 - Requires more power
 - Optimise for latency
 - Minimising the time taken for one particular task
 - Flexible programming model
-
- Large number of small processors
 - Less control structures and more processing units
 - Can do less complex logic
 - Lower power consumption
 - Optimised for throughput
 - Maximising the amount of work done per unit of time
 - Restricted programming model

Programming on heterogeneous systems

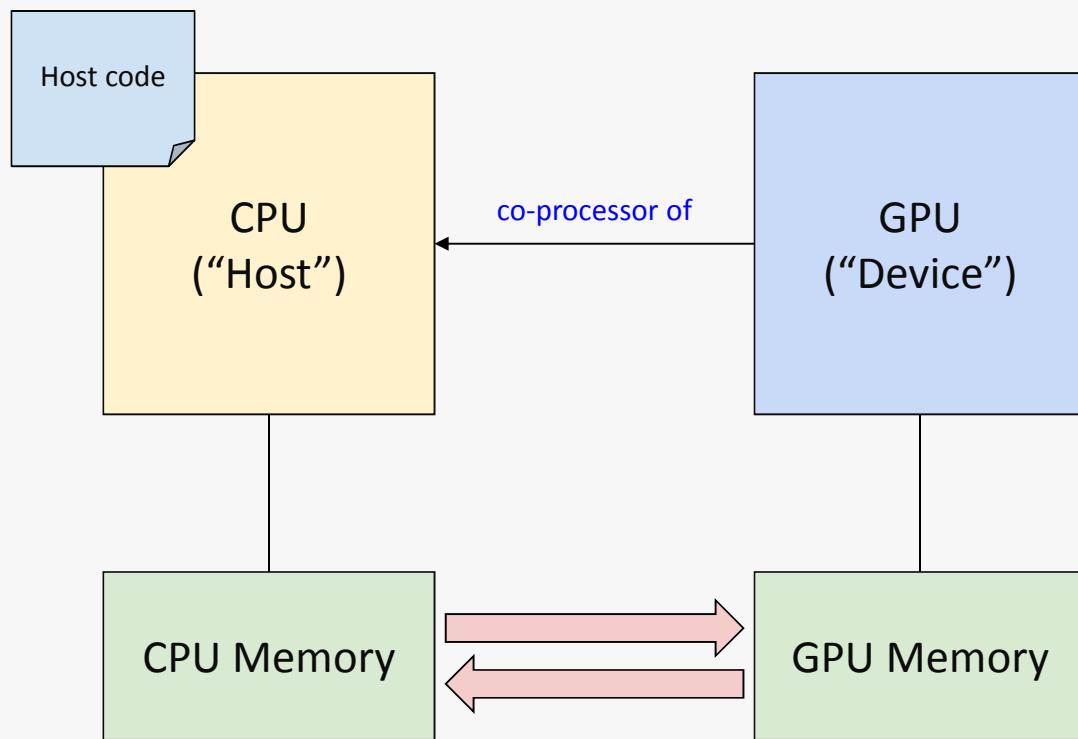


Programming on heterogeneous systems

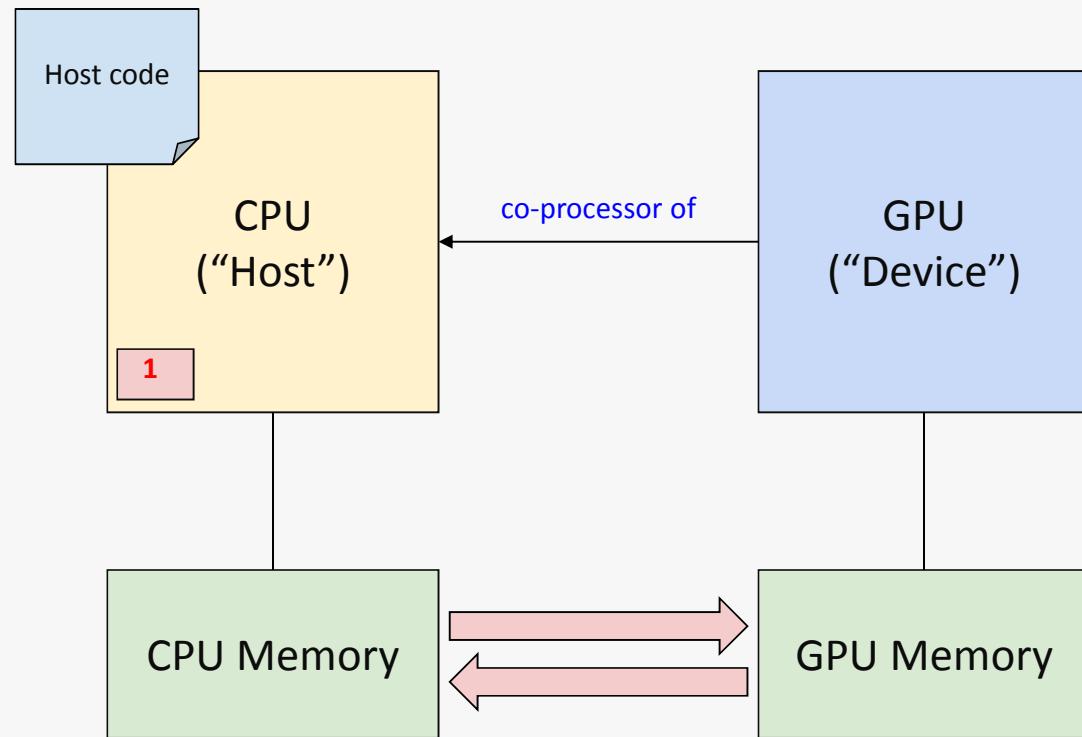


Explicit heterogeneous programming models can allow you to write SPMD code that will run on SIMD CPUs and GPUs

Programming on a SIMD CPU

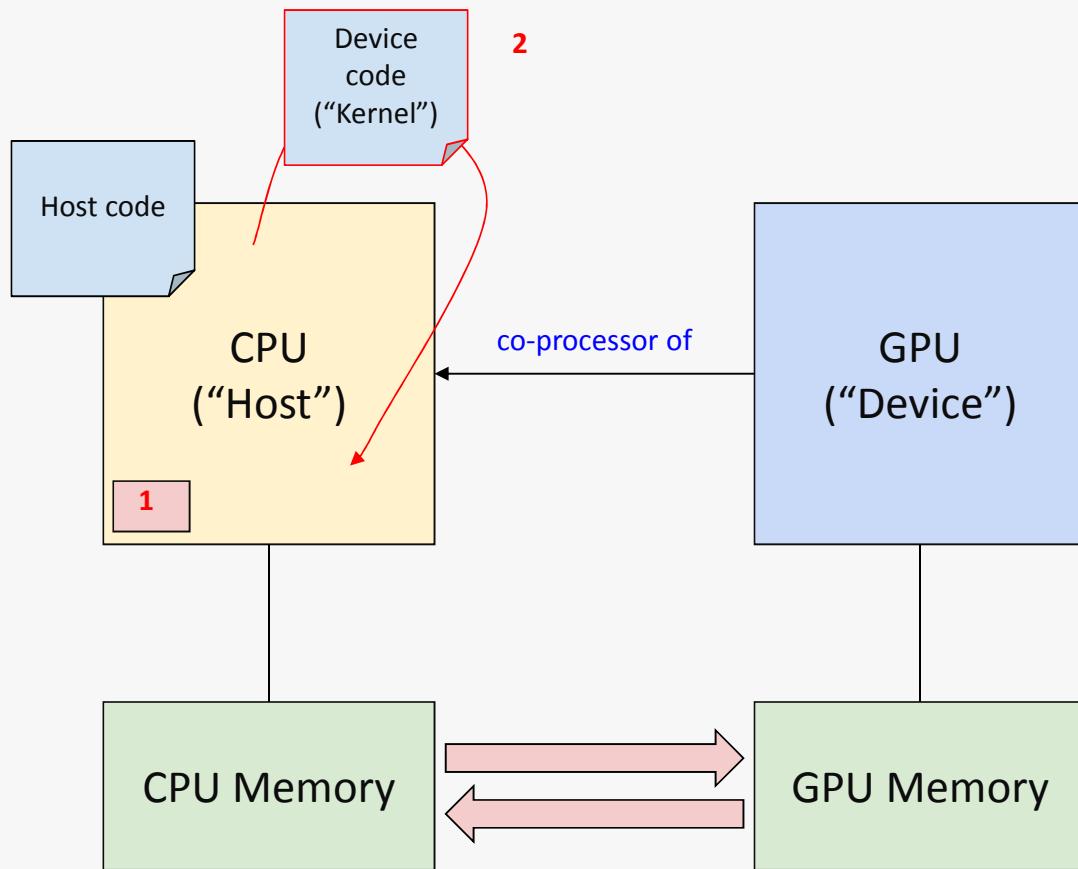


Programming on a SIMD CPU



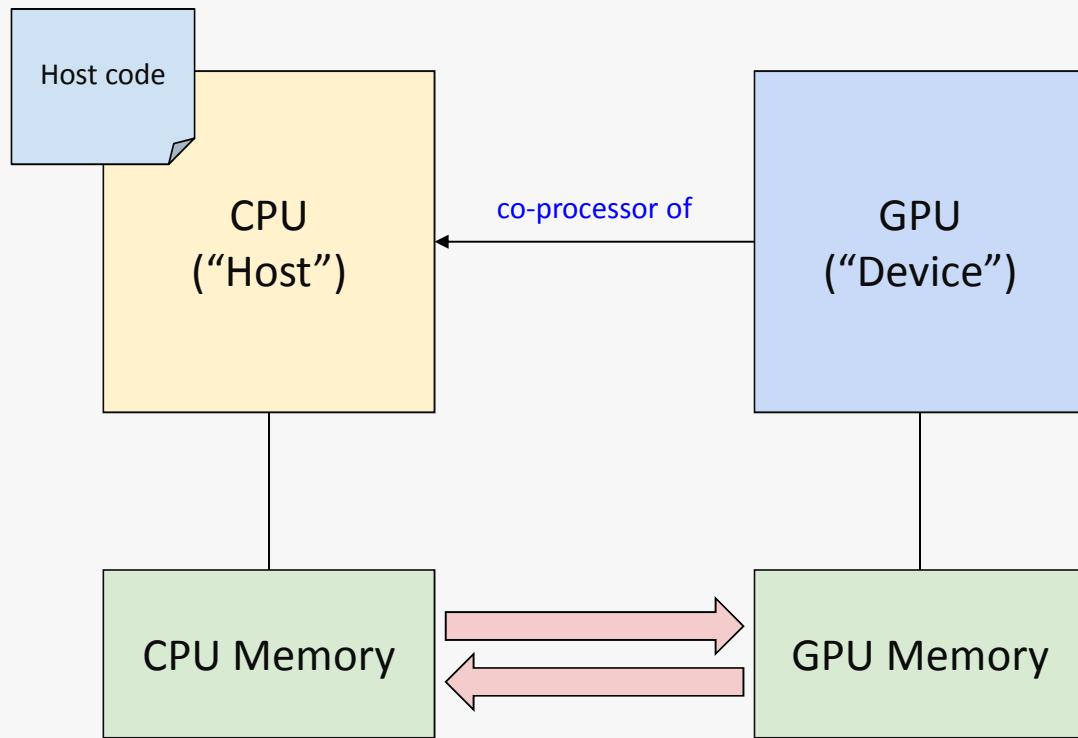
1. The CPU allocates memory on the CPU

Programming on a SIMD CPU

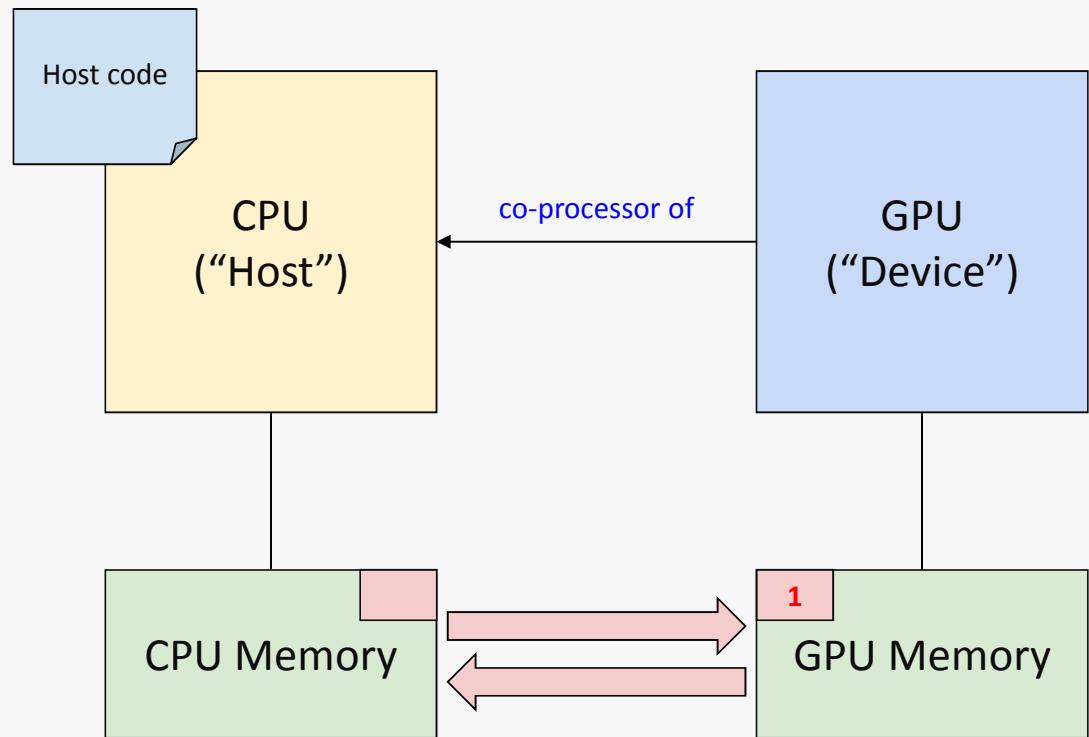


1. The CPU allocates memory on the CPU
2. The CPU executes a kernel using threads and SIMD instructions

Programming on a GPU

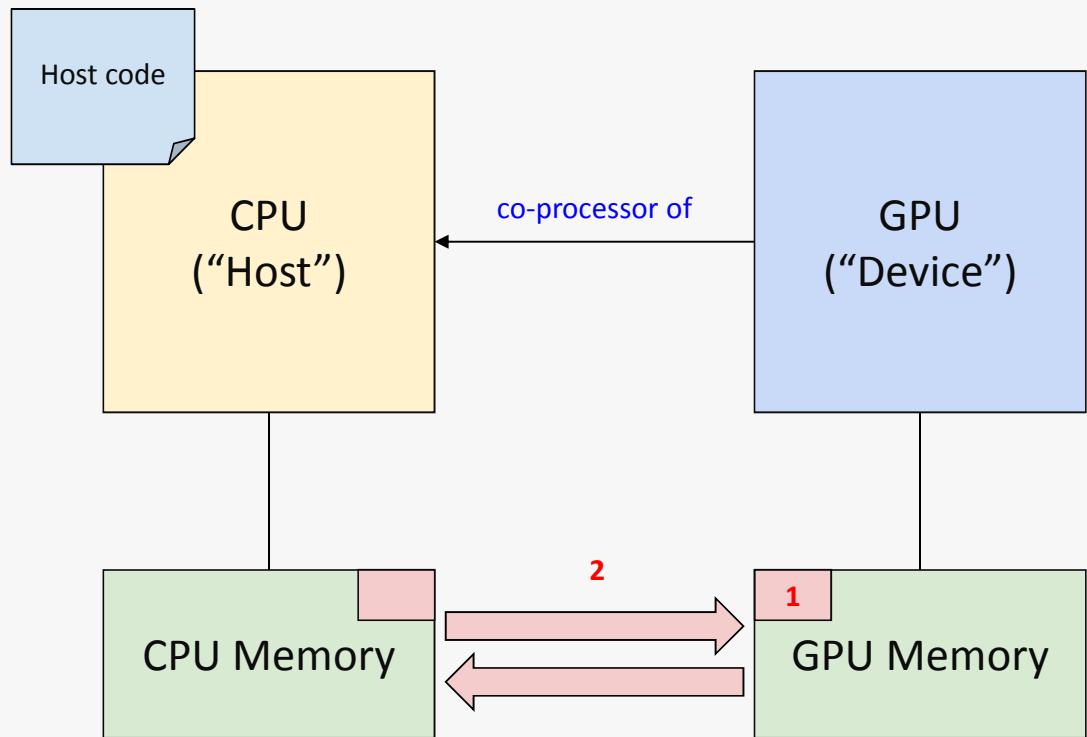


Programming on a GPU



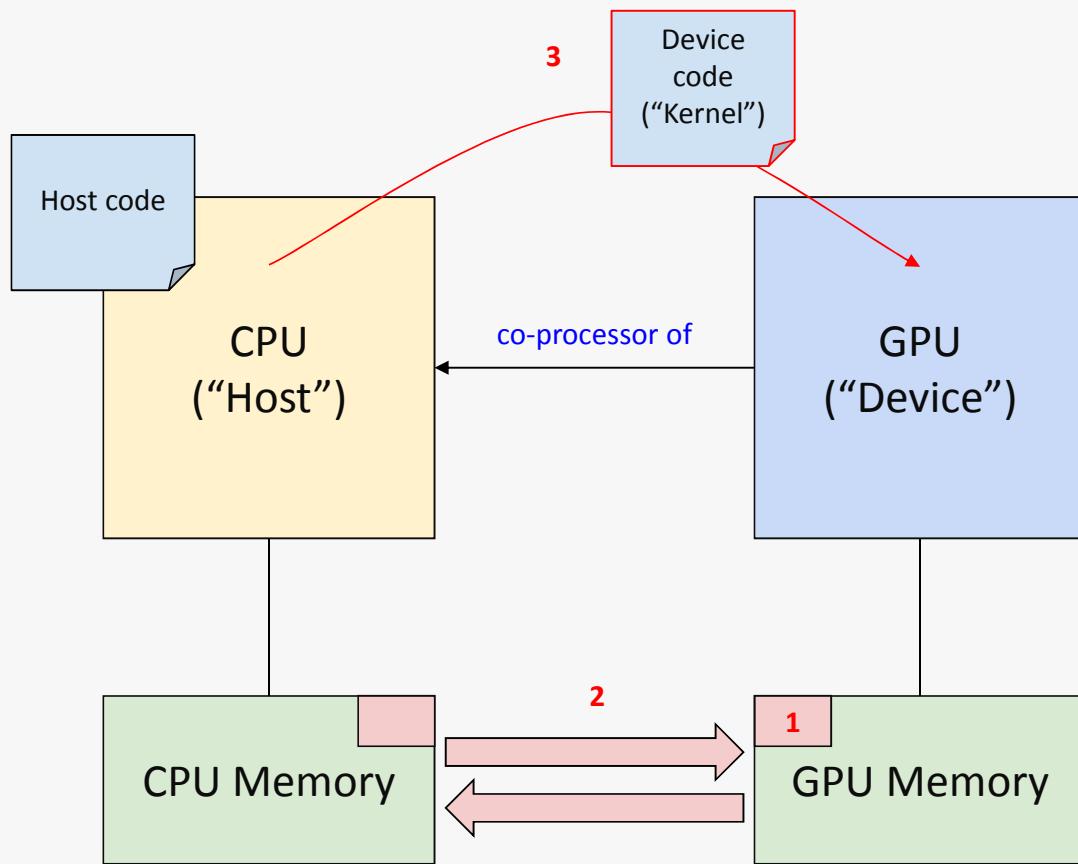
1. The CPU allocates memory on the GPU

Programming on a GPU



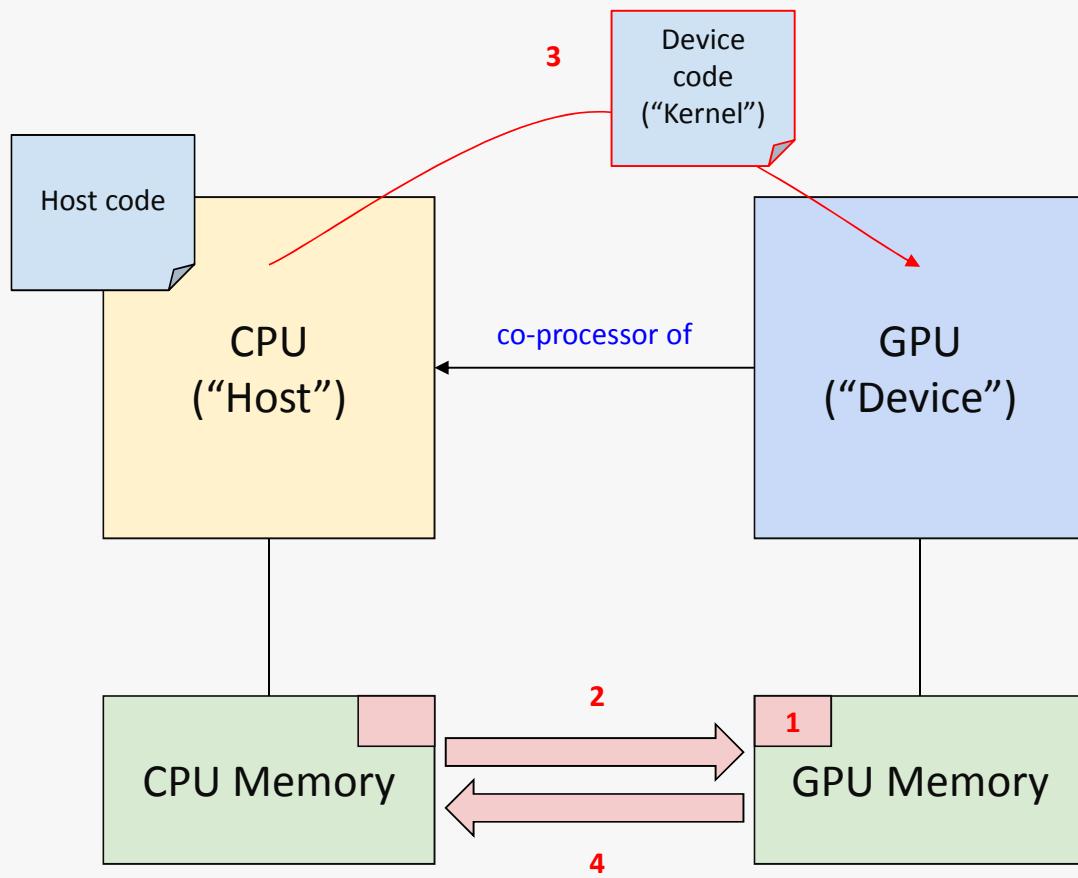
1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU

Programming on a GPU



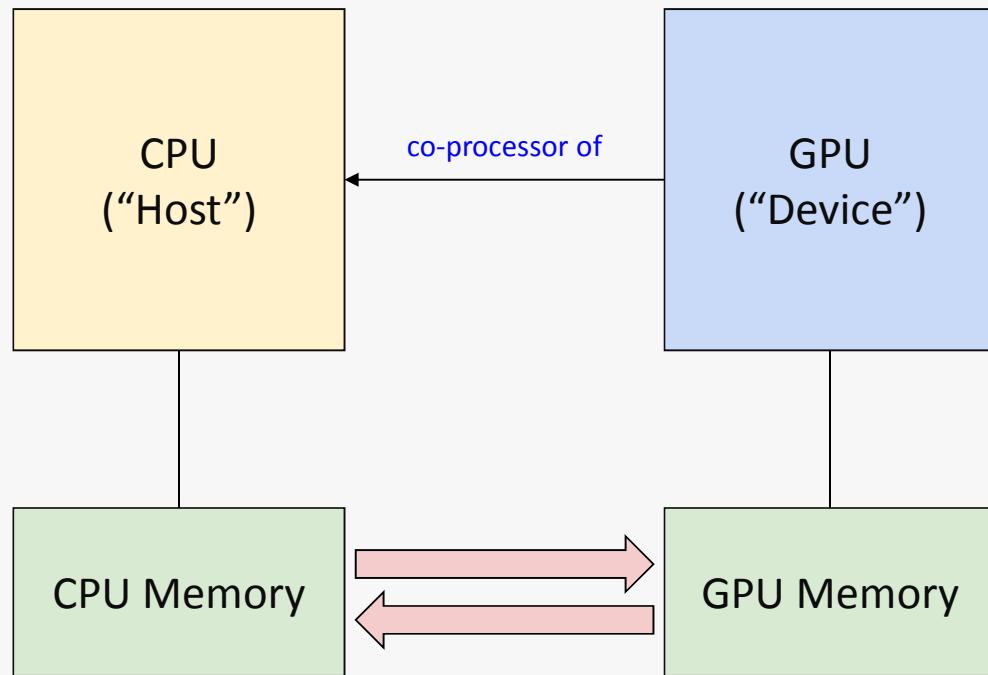
1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU

Programming on a GPU

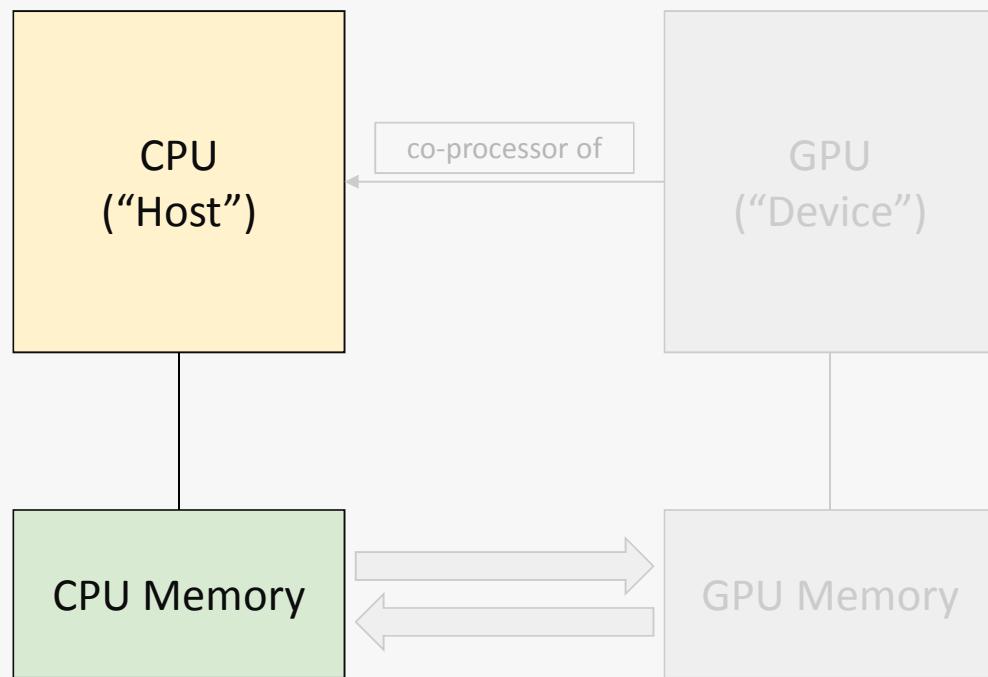


1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU
4. The CPU copies data to CPU from GPU

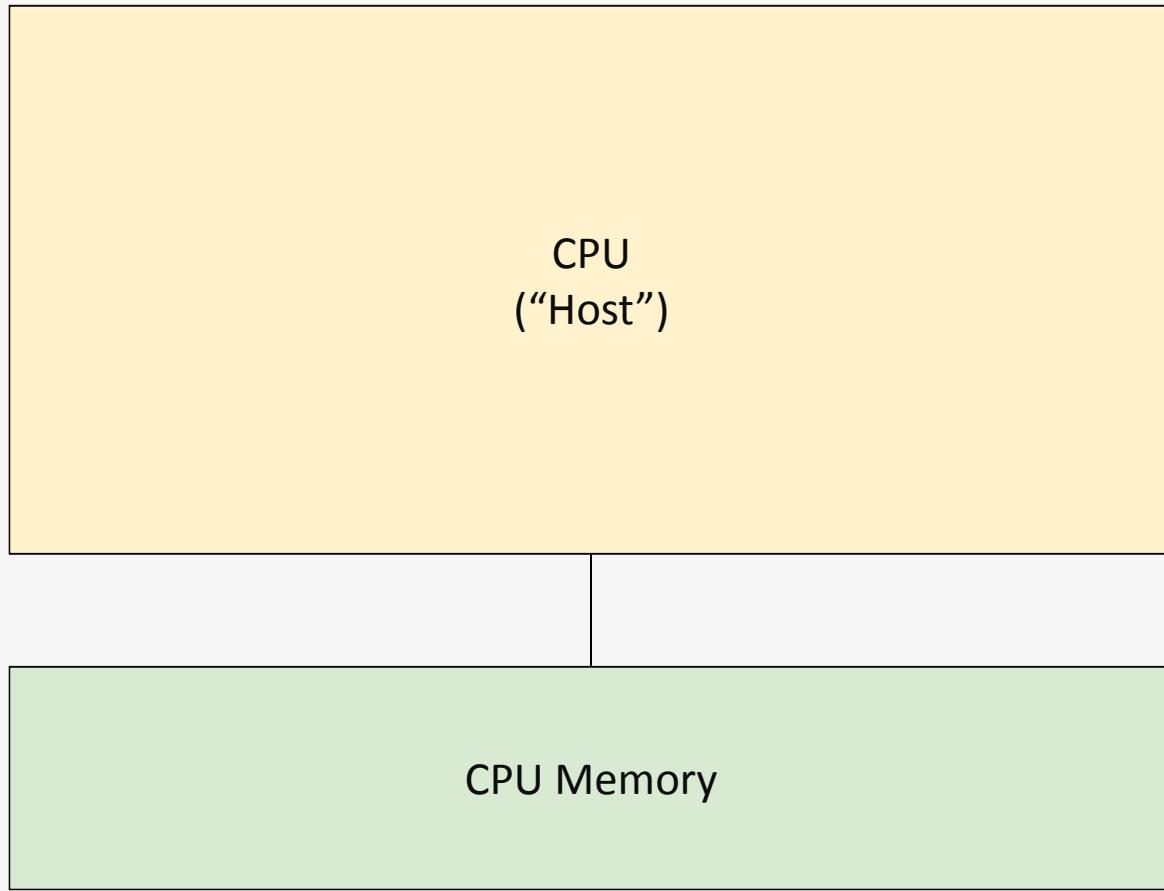
So let's take a look now at each of these



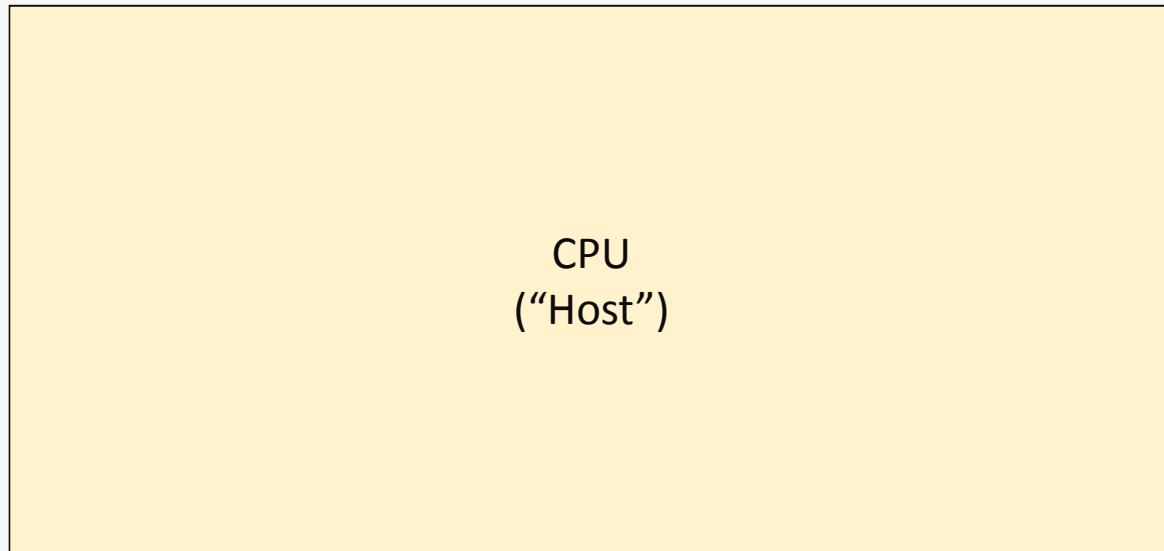
Let's take a look at the CPU...



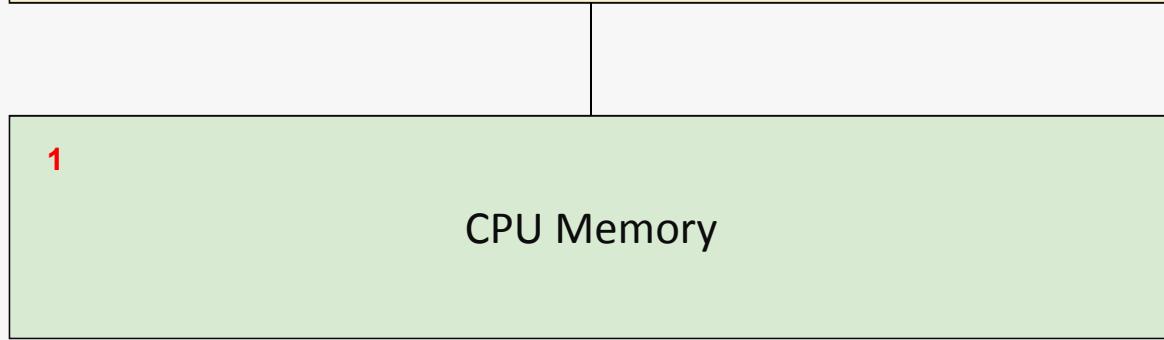
Let's take a look at the CPU...



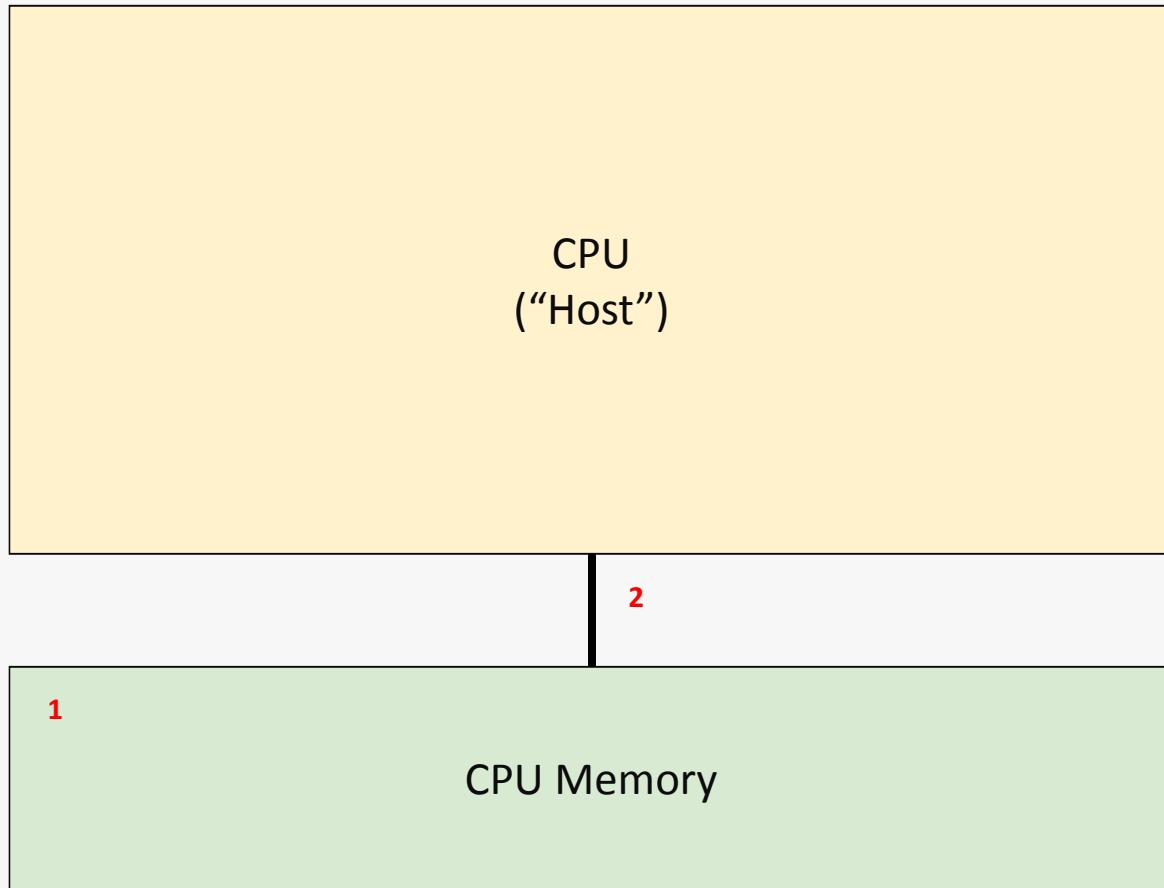
Let's take a look at the CPU...



1. A CPU has a region of dedicated memory

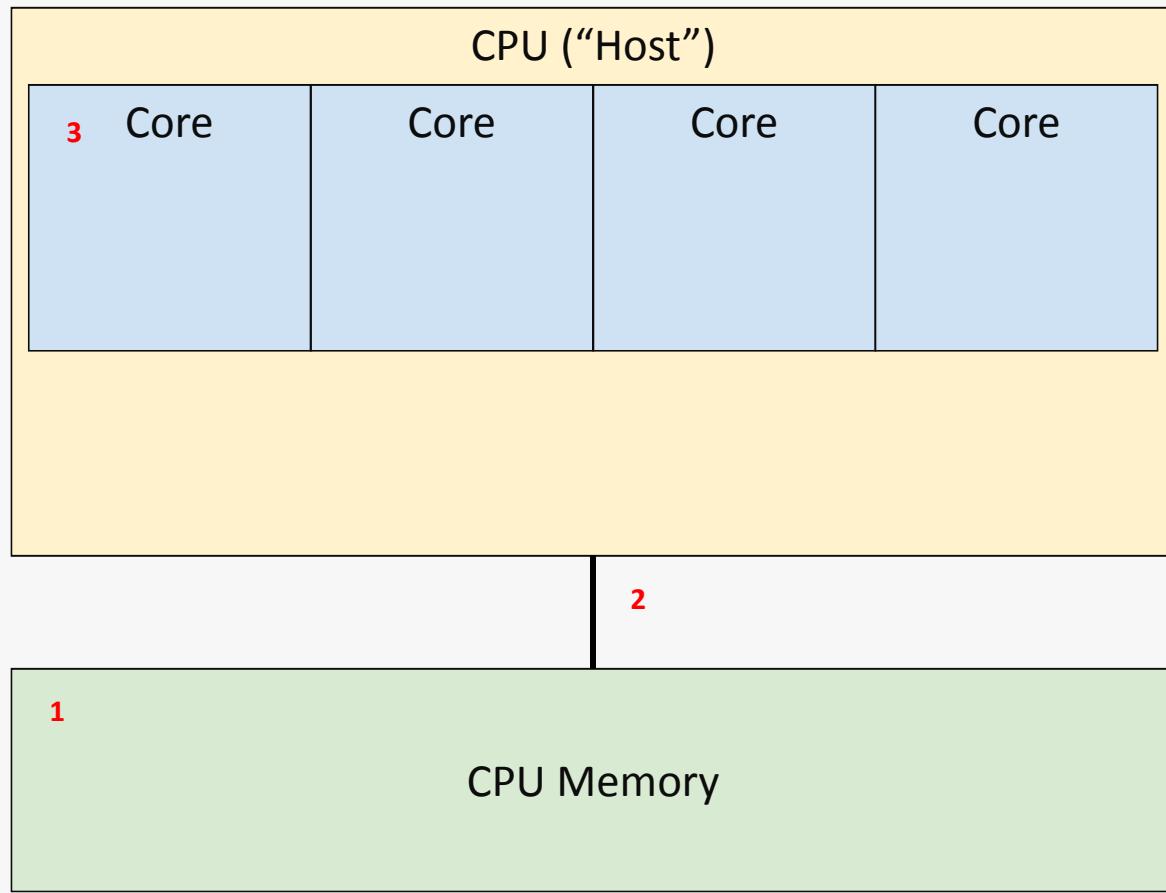


Let's take a look at the CPU...



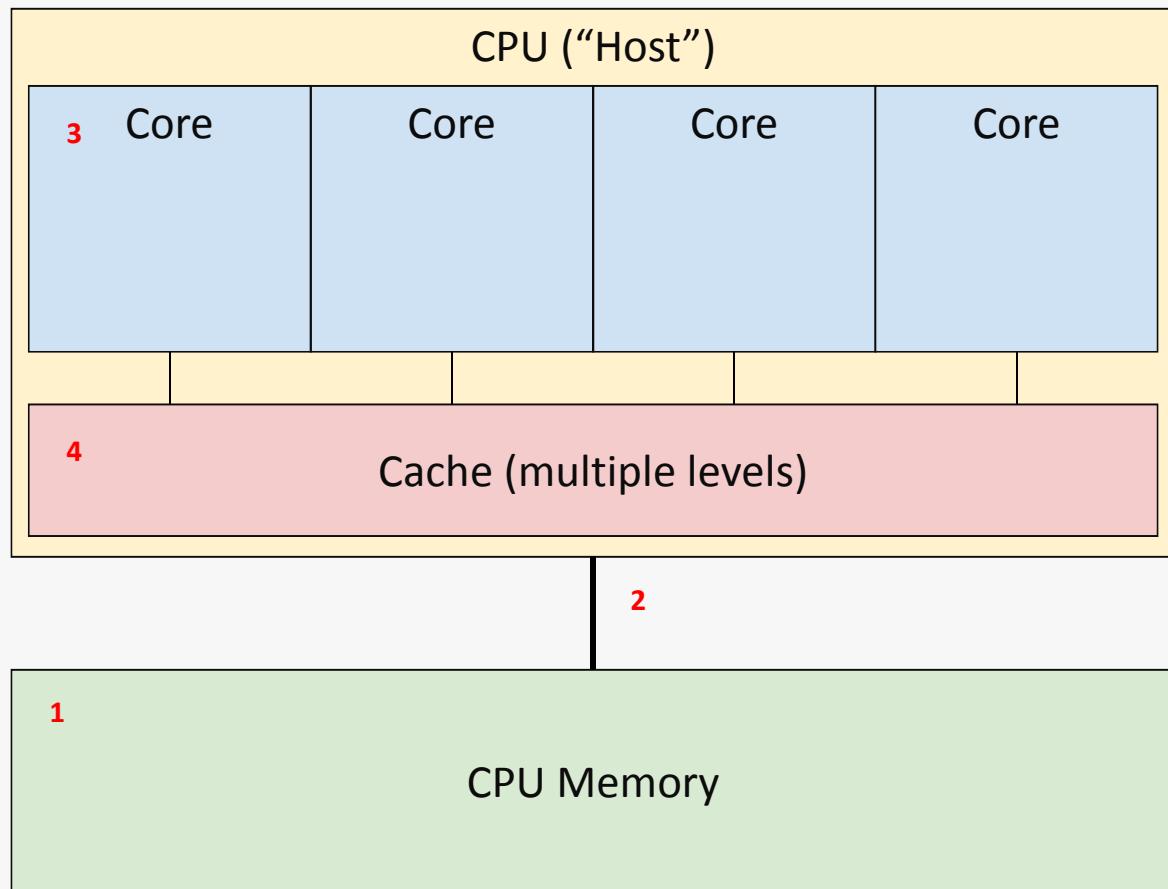
1. A CPU has a region of dedicated memory
2. CPU memory is connected to the CPU via a bus

Let's take a look at the CPU...



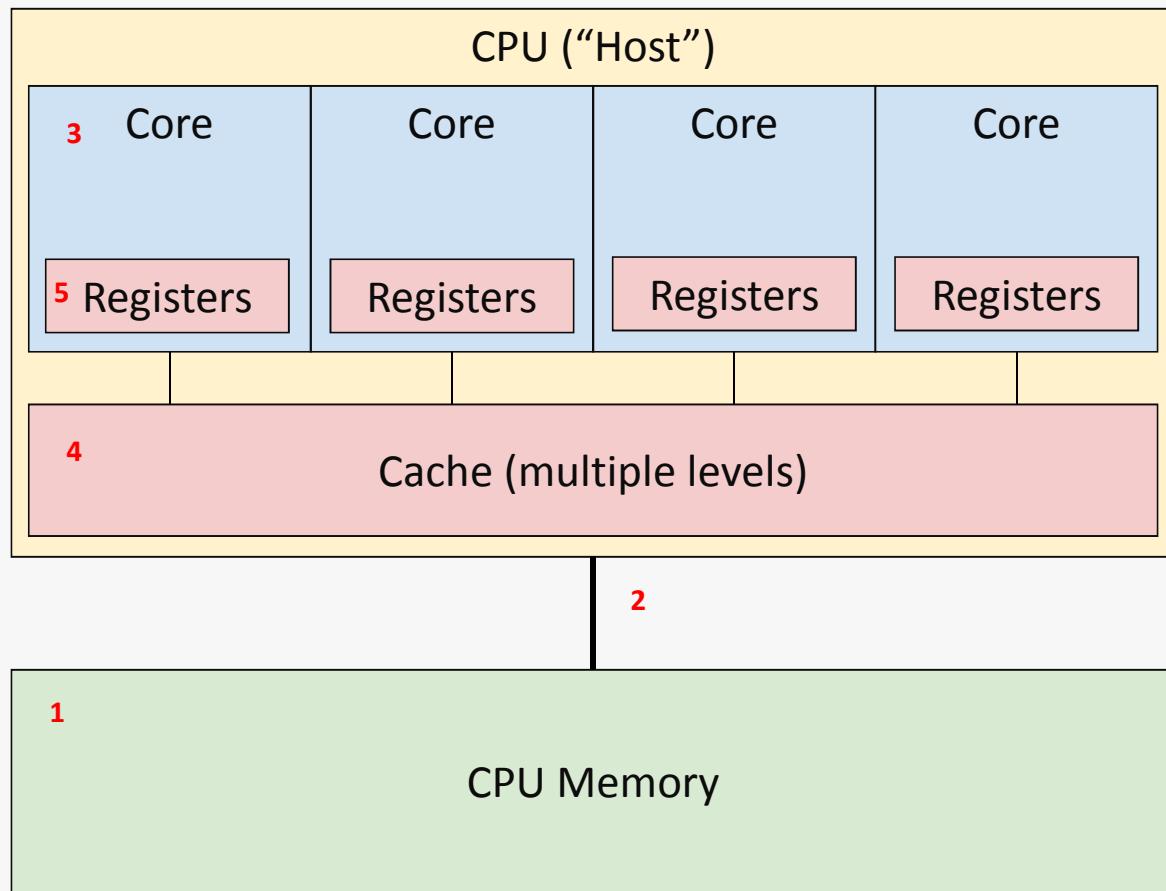
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores

Let's take a look at the CPU...



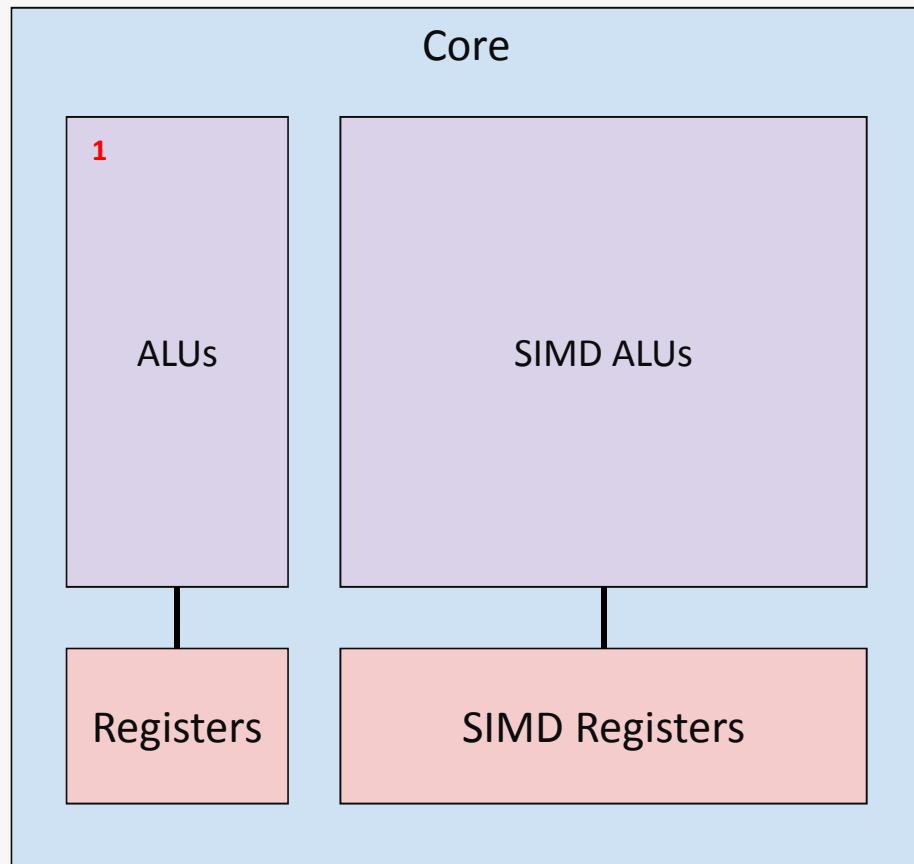
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores
4. A CPU has a number of caches of different levels

Let's take a look at the CPU...



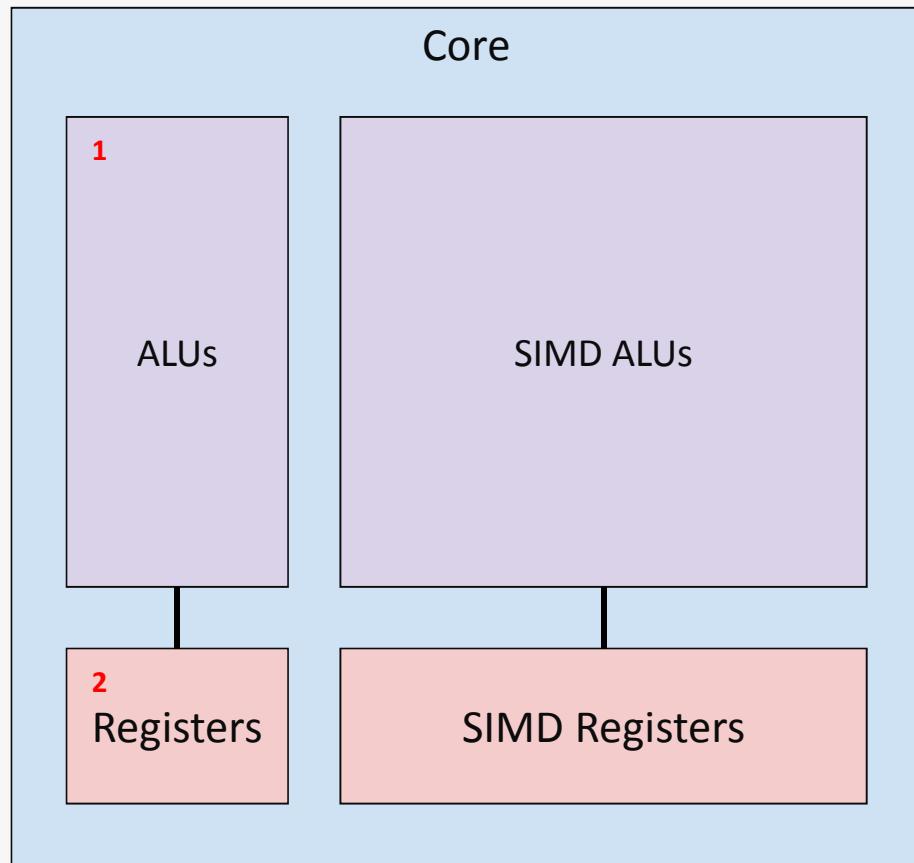
1. A CPU has a region of dedicated memory
2. The CPU memory is connected to the CPU via a bus
3. A CPU has a number of cores
4. A CPU has a number of caches of different levels
5. Each CPU core has dedicated registers

Inside a CPU core



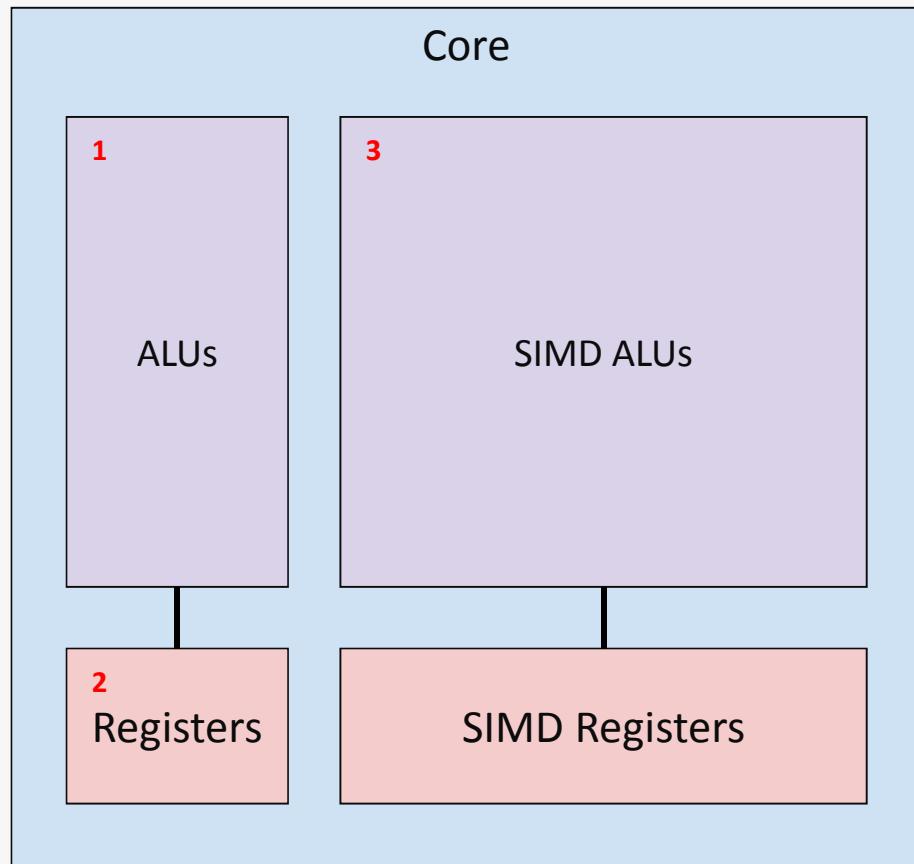
1. A CPU core you have a number of standard ALUs

Inside a CPU core



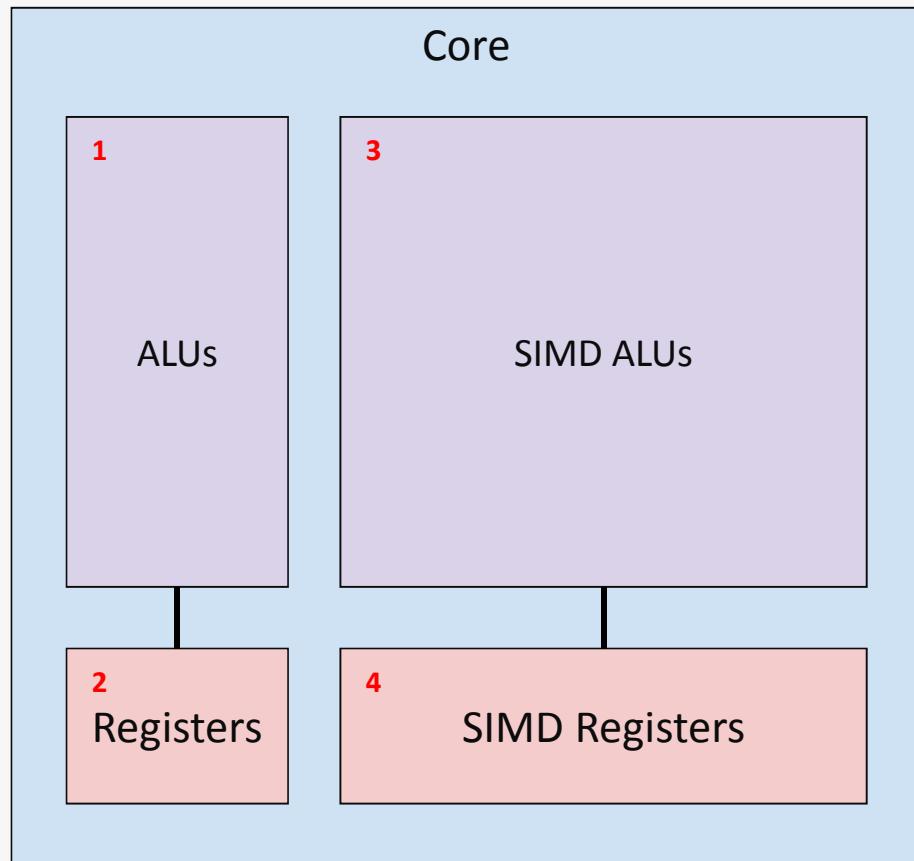
1. A CPU core you have a number of standard ALUs
2. These are connected to standard registers

Inside a CPU core



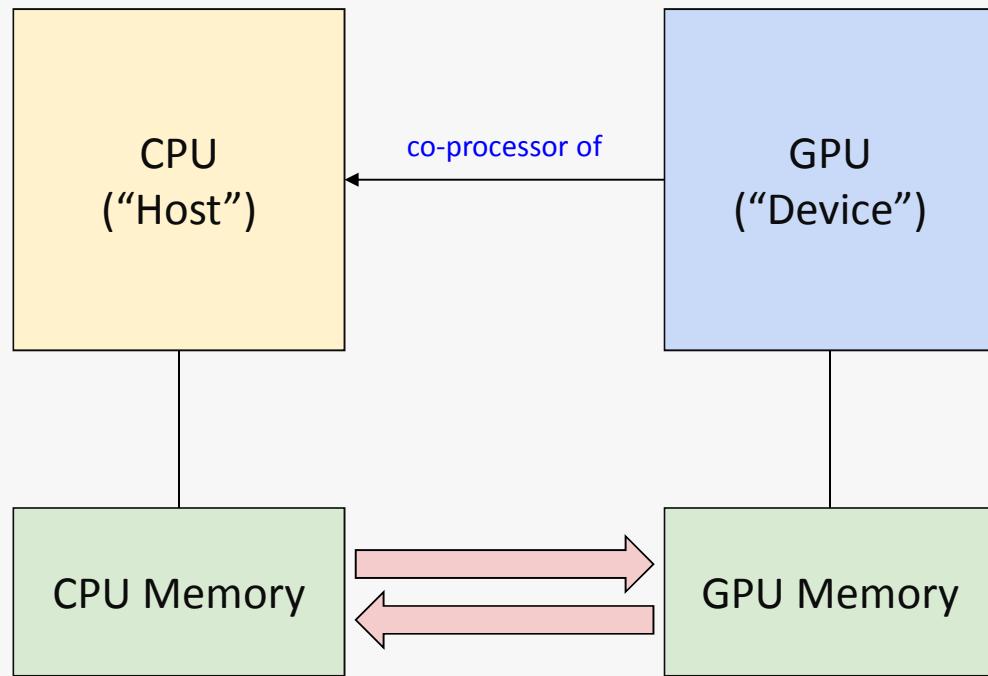
1. A CPU core you have a number of standard ALUs
2. These are connected to standard registers
3. A CPU core also have SIMD ALUs

Inside a CPU core

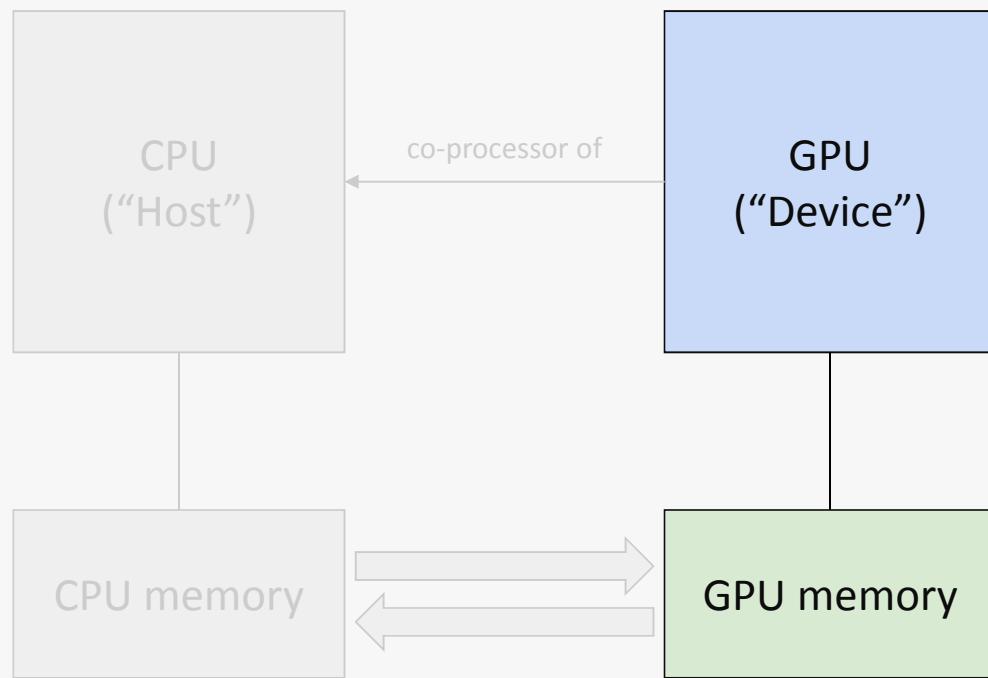


1. A CPU core you have a number of standard ALUs
2. These are connected to standard registers
3. A CPU core also have SIMD ALUs
4. These are connected to SIMD registers

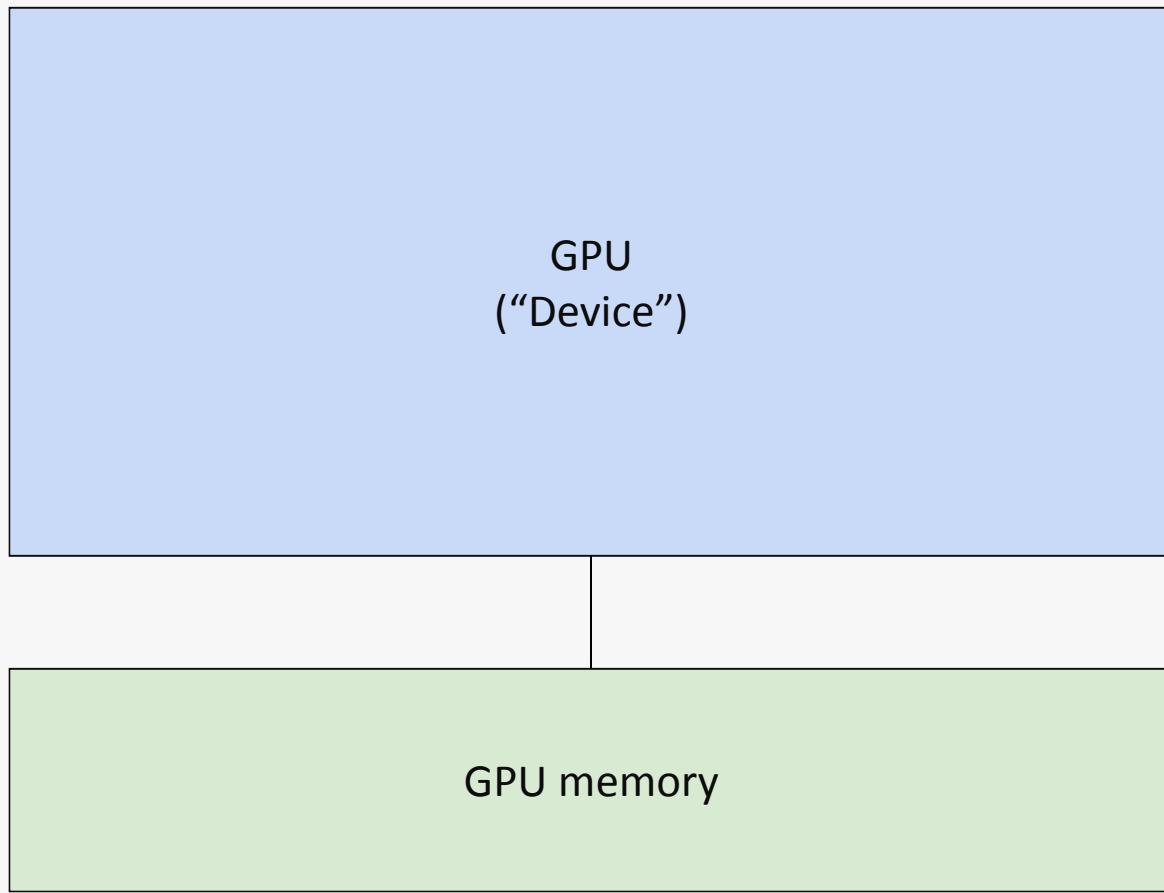
Now let's take a look at the GPU...



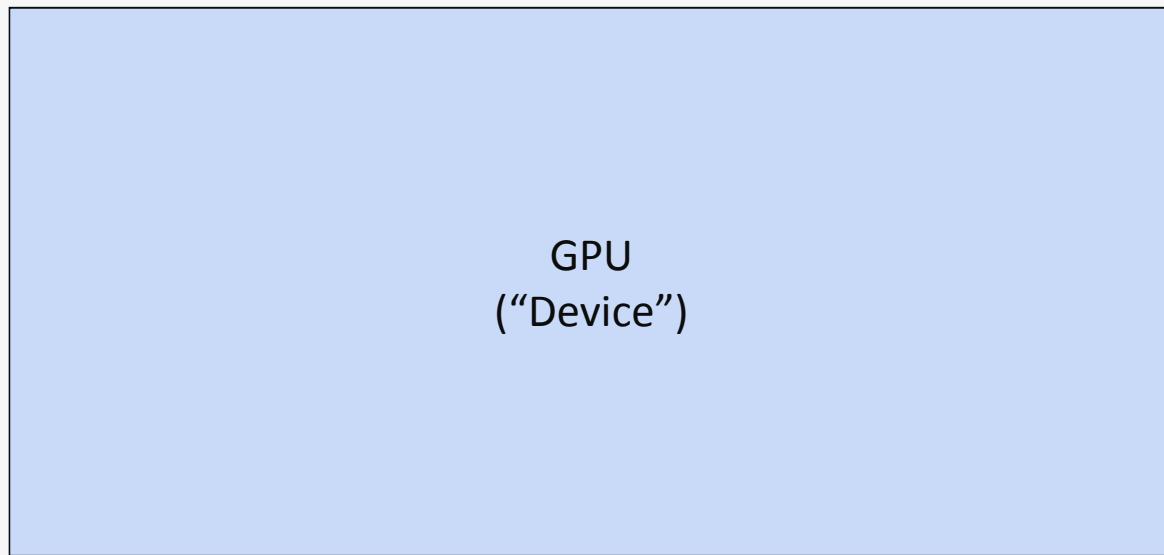
Now let's take a look at the GPU...



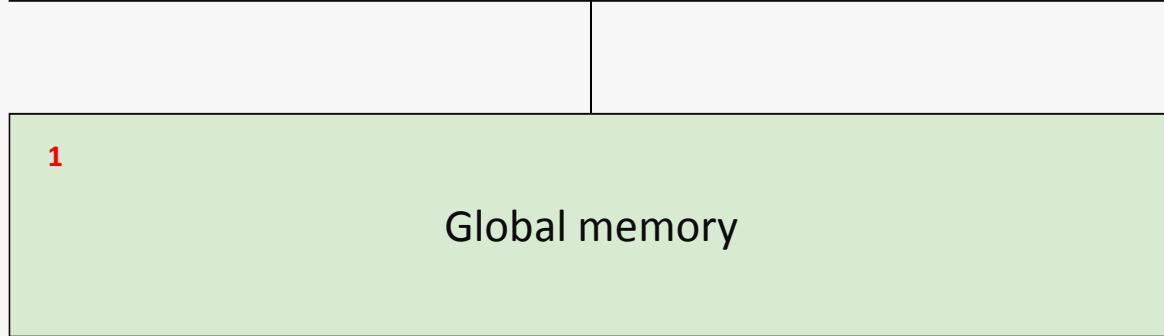
Now let's take a look at the GPU...



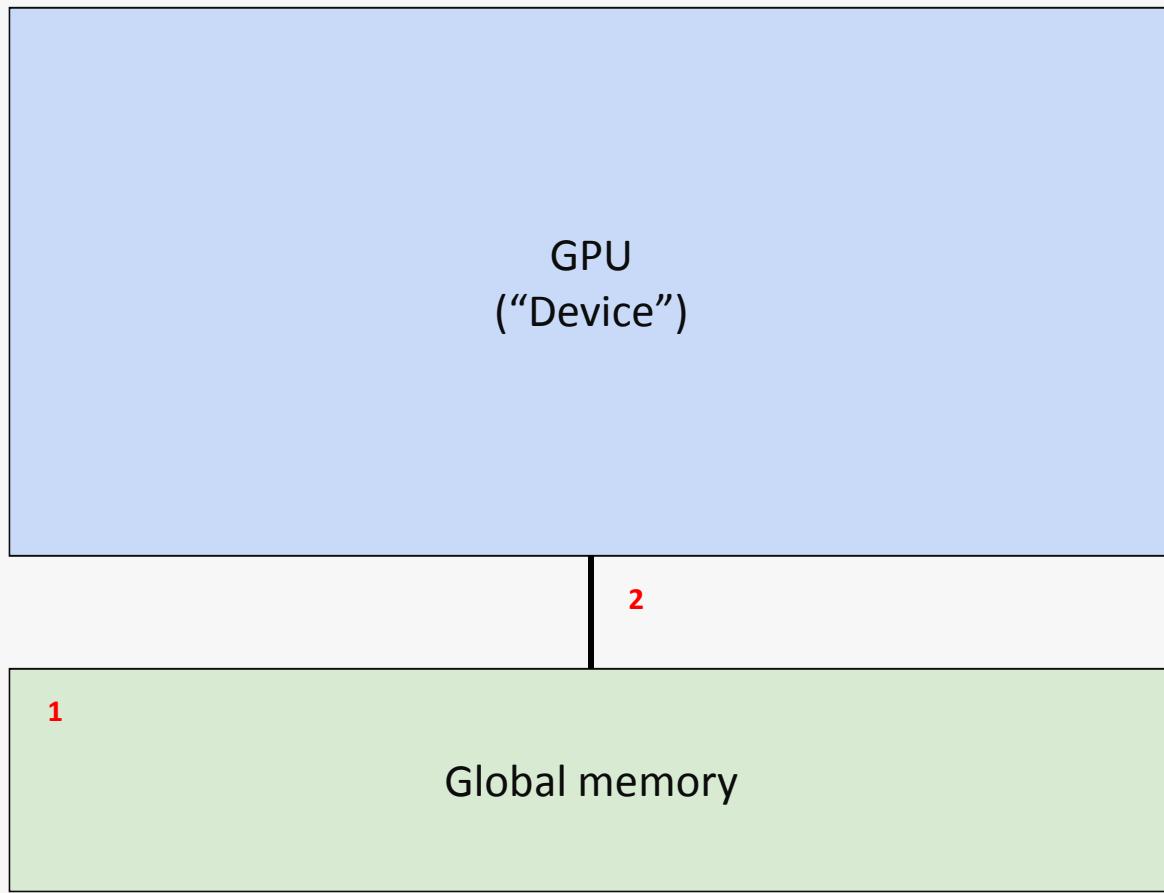
Now let's take a look at the GPU...



1. A GPU has a region of dedicated global memory

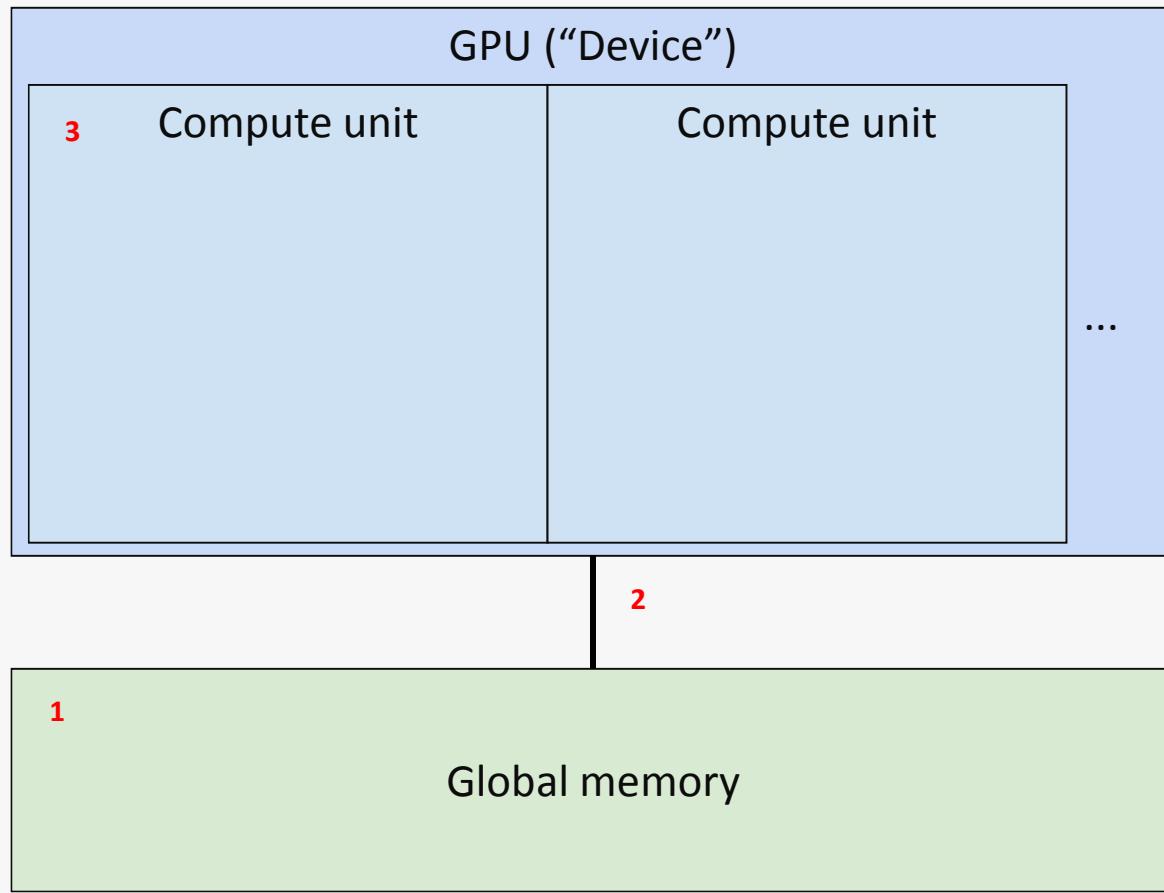


Now let's take a look at the GPU...



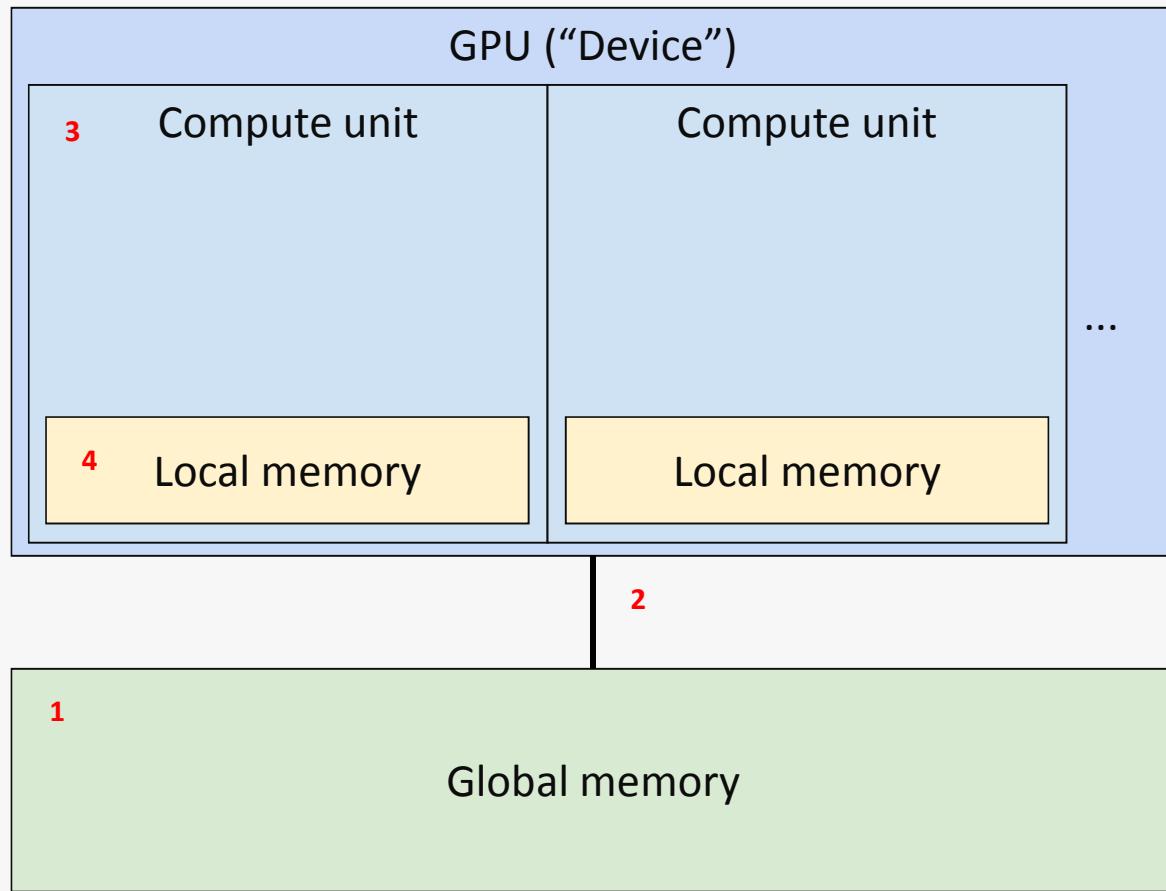
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus

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



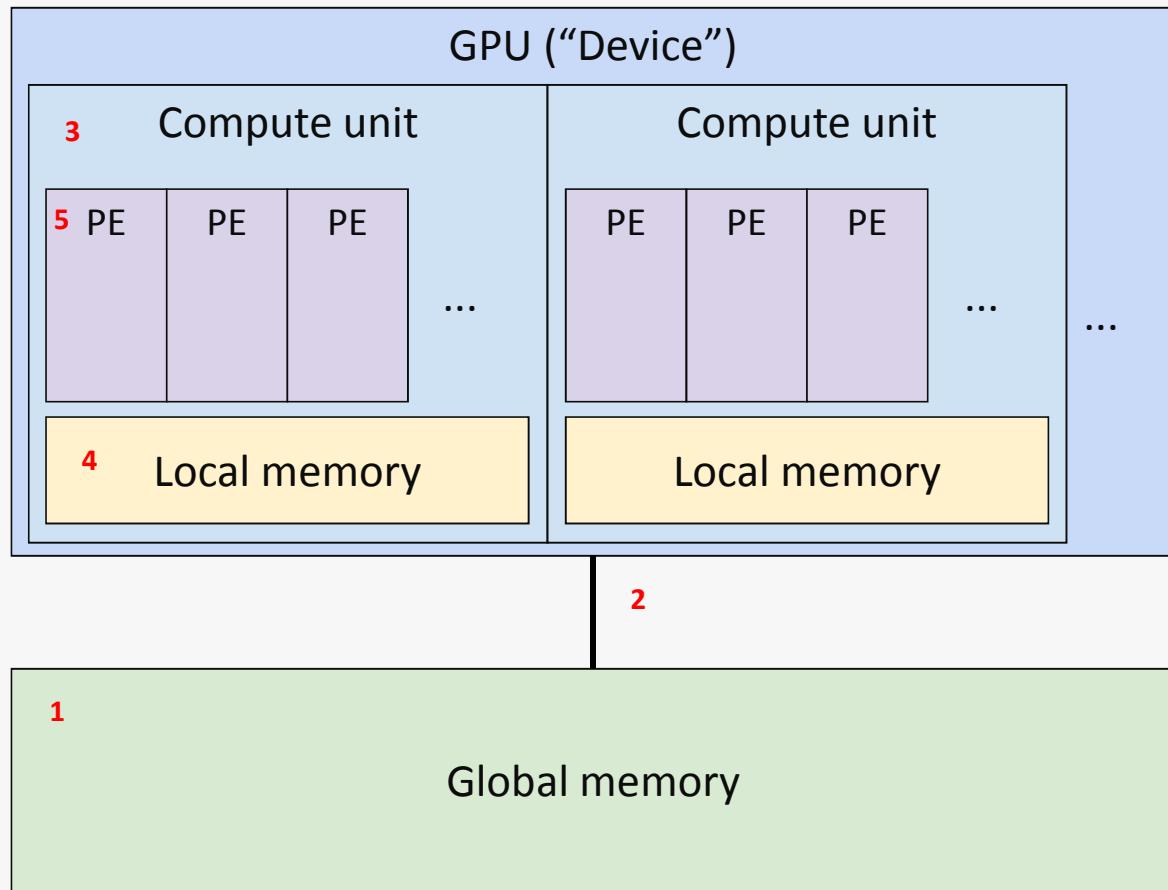
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units

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



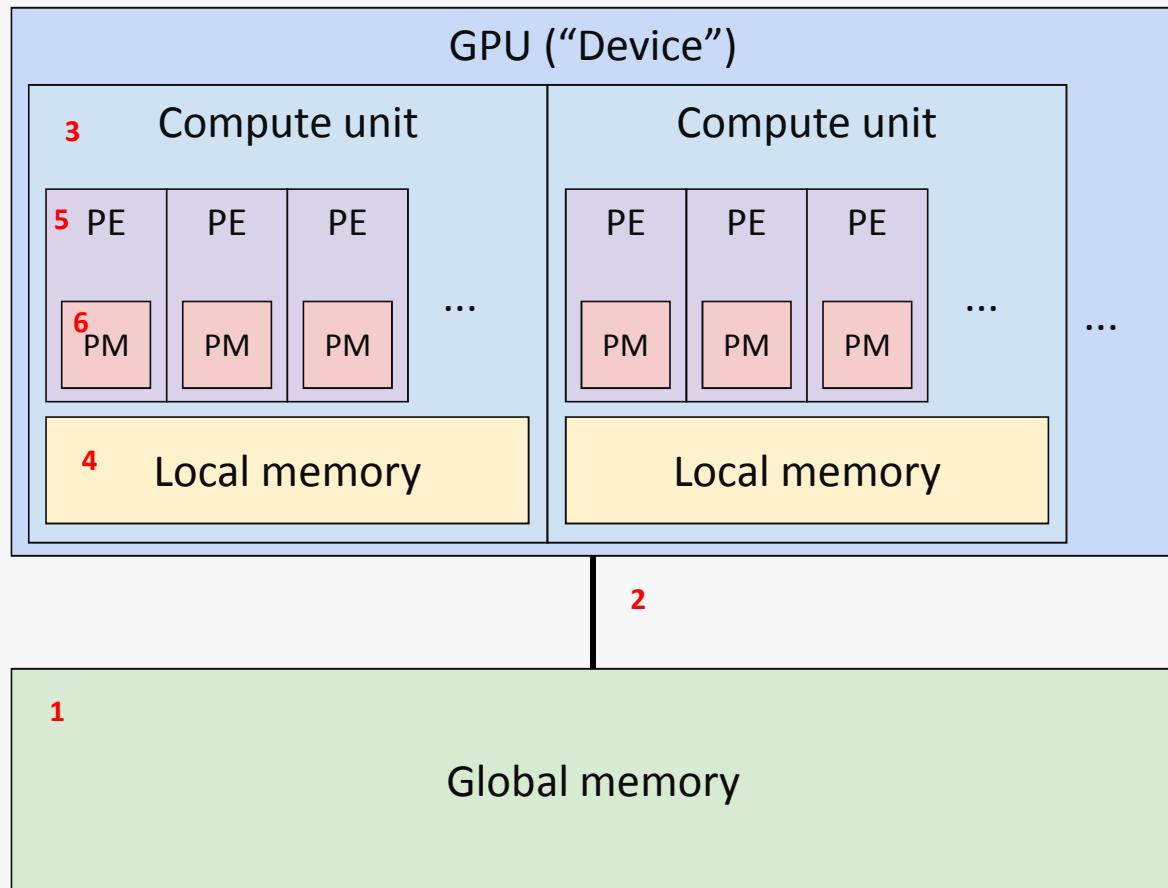
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory

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



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

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



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

Key takeaways

CPUs are optimized for latency and GPUs are optimized for throughout

When programming for a co-processor such as a GPU you must compile and offload functions and manage data movement



Questions?



Chapter 7: Configuring a Queue

Gordon Brown

CppCon 2019 – Sep 2019

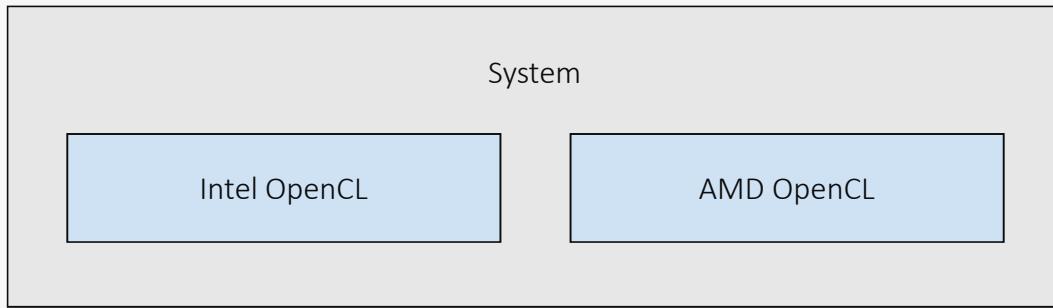
- Learning objectives:
 - Learn about the SYCL topology discovery
 - Learn how to query information about a platform or device
 - Learn how to select a device using a device selector
 - Learn about the SYCL queue and what it does
 - Learn about command groups and the command group handler
 - Learn about the SYCL scheduler model

SYCL system topology

System

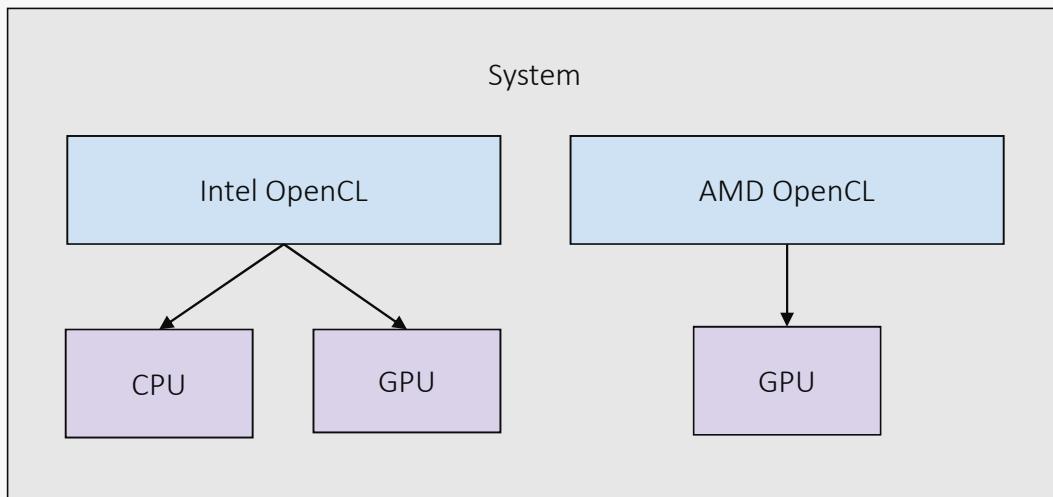
A SYCL application can execute work across a range of different heterogeneous devices

The devices that are available in any given system are determined at runtime through topology discovery



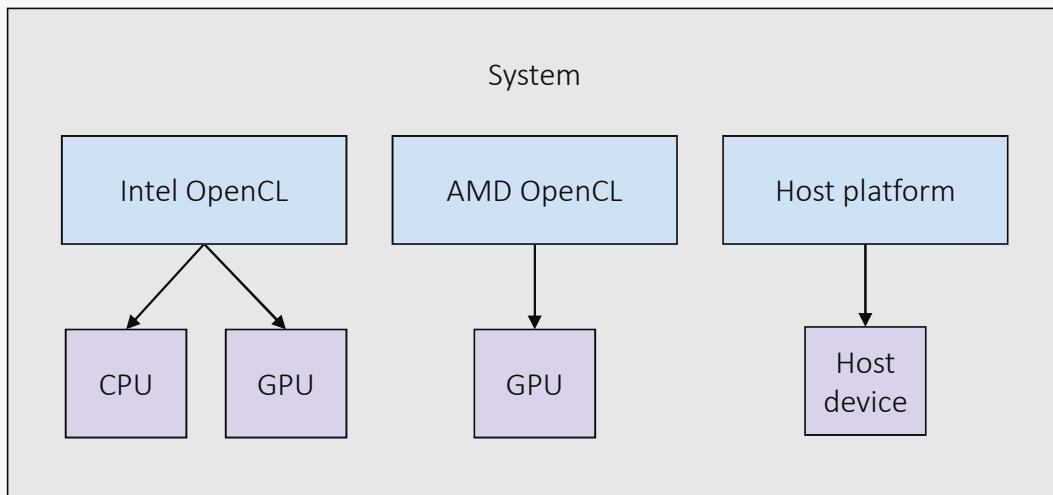
In the SYCL topology a system has a number of platforms

A platform refers to a backend implementation, e.g. Intel OpenCL



Each platform has one or more devices associated with it; CPU, GPU, accelerator, etc

Each device is only associated with a single platform

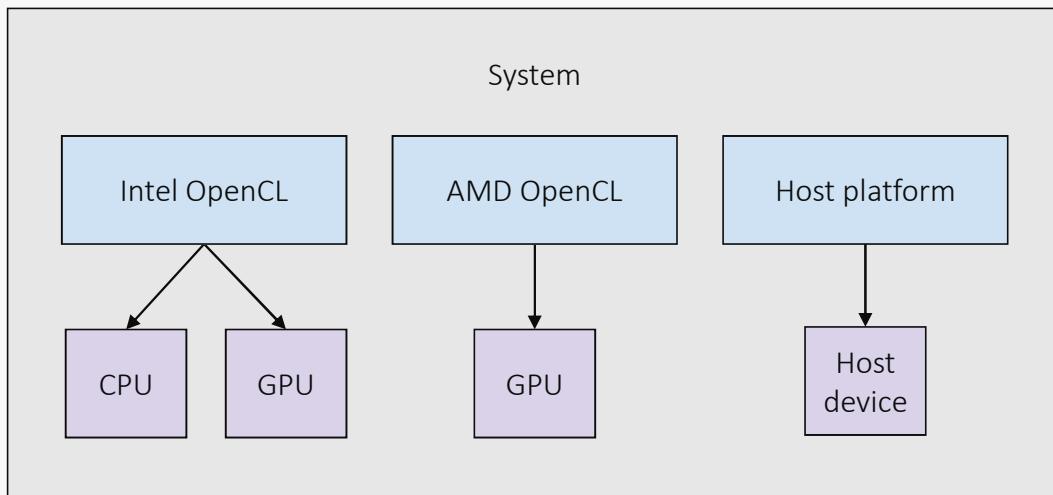


In SYCL there is also a host device which executes SYCL kernels as native C++

The host device emulates the execution and memory model of an OpenCL device

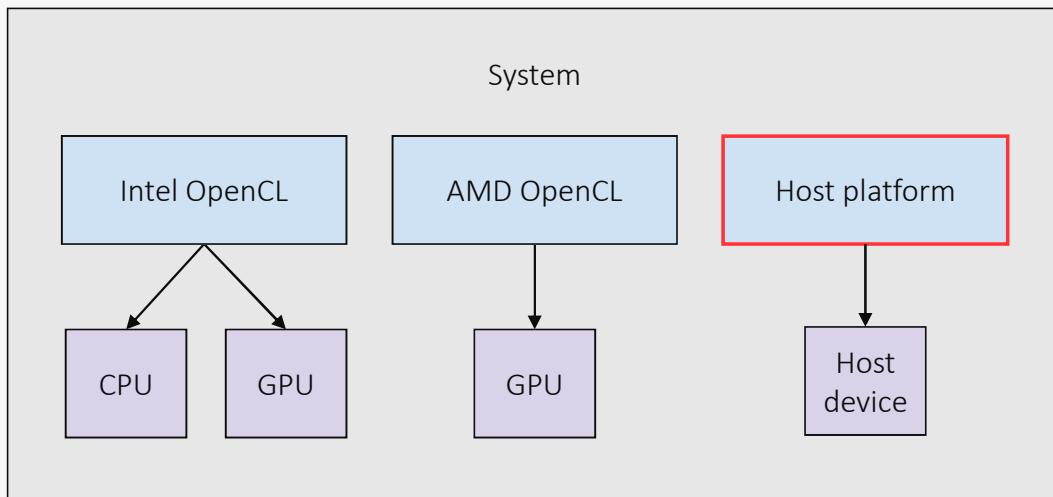
This is very useful for debugging SYCL kernel

There is only ever one host device and that device is associated with a host platform



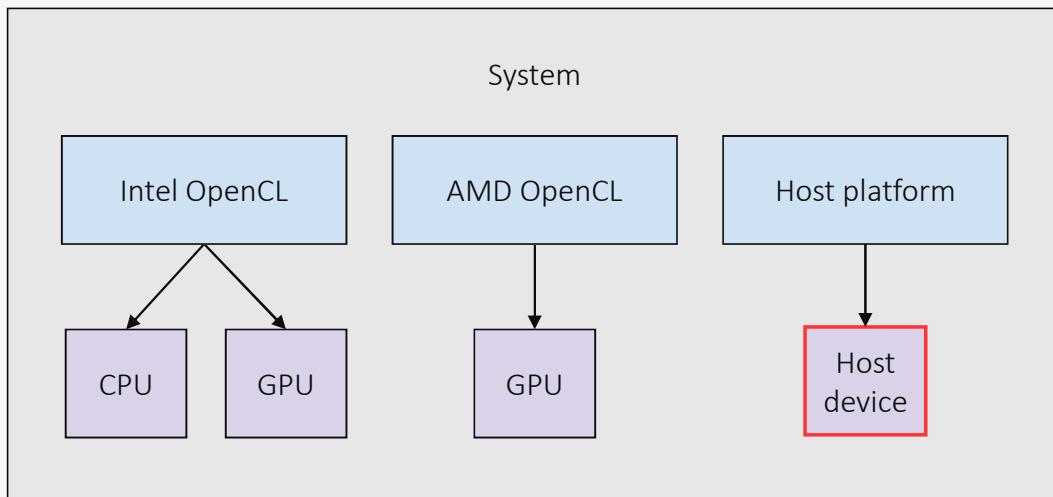
A platform is represented by the
platform class

A device is represented by the
device class



```
auto hostPlatform = platform{};
```

A default constructed platform object represents the host platform

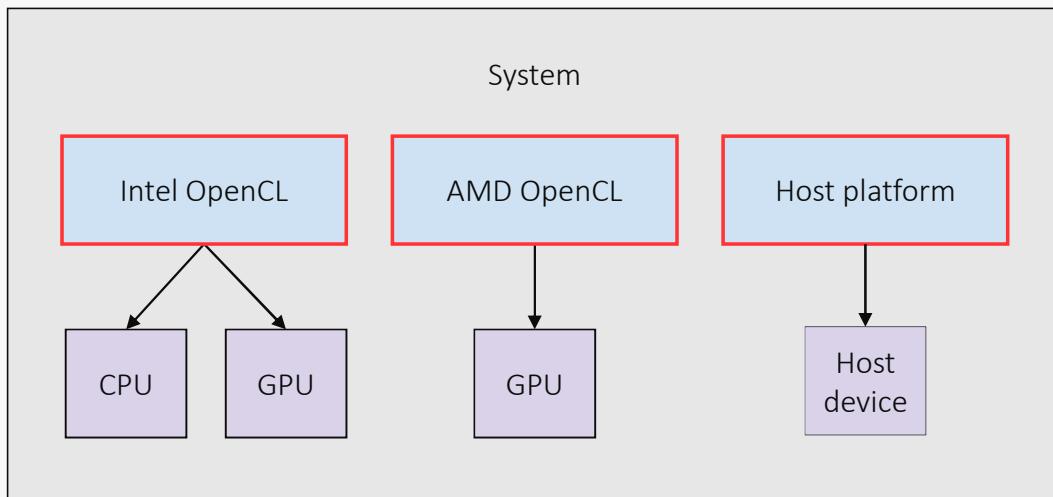


```
auto hostDevice = device{};
```

A default constructed device object represents the host device

- In SYCL there are two ways to query a system's topology
 - The topology can be manually queried and iterated over via APIs of the platform and device classes
 - The topology can be automatically queried and iterated over using a user specified heuristic by a device selector object

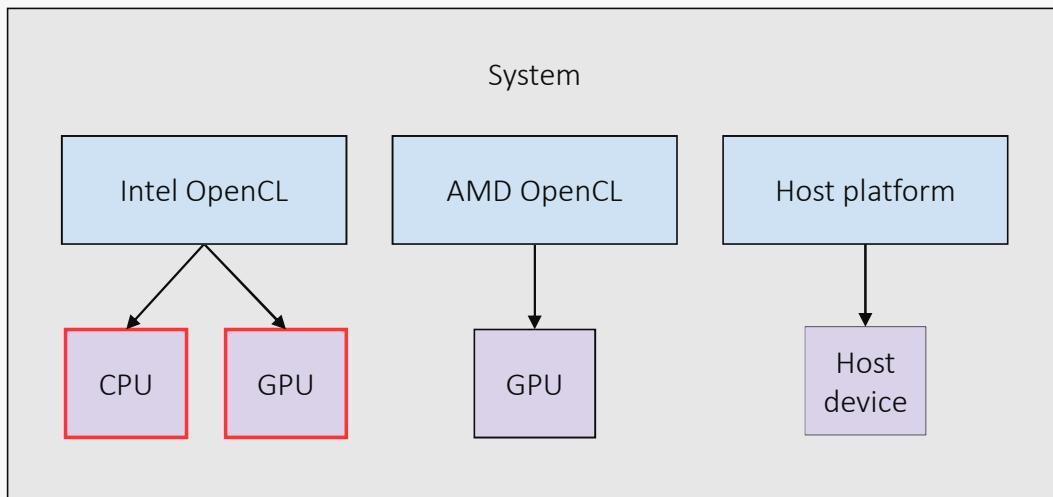
Querying the topology manually



```
auto platforms =
    platform::get_platforms();
```

The platform class provides the static function **get_platforms** for retrieving a vector of all available platforms in the system

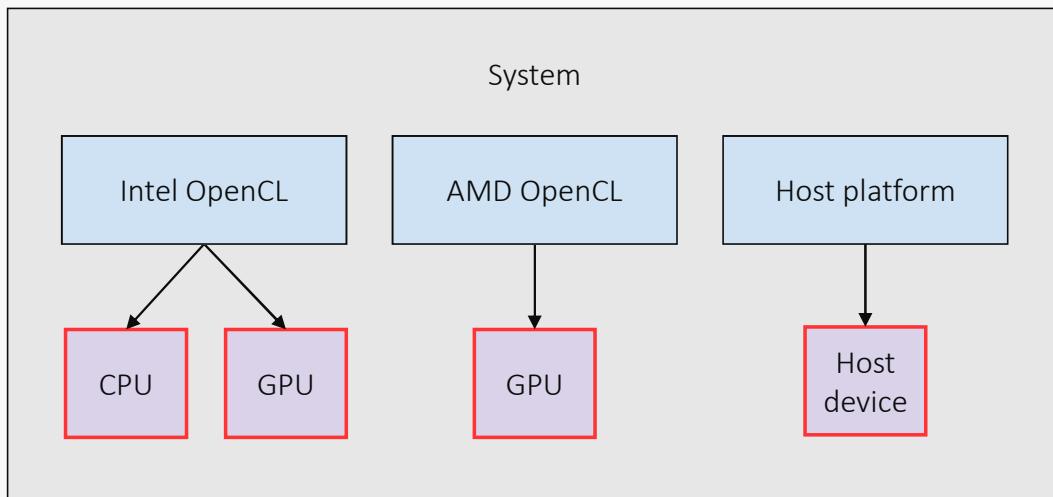
This includes the host platform



```
auto intelDevices =
    intelPlatform.get_devices();
```

The platform class provides the member function **get_devices** that will return a vector of all devices associated with that platform

This includes the host device if the platform object represents a host platform

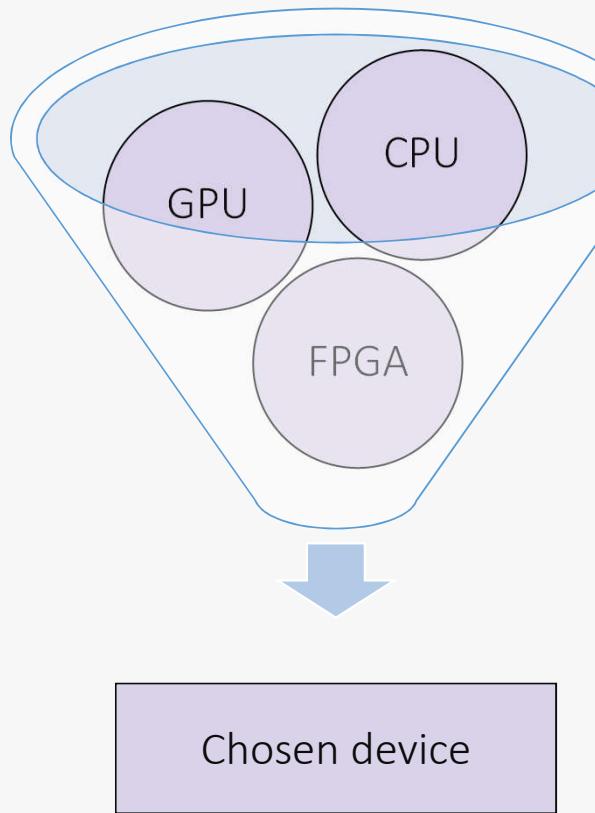


```
auto devices =
    device::get_devices();
```

The device class also provides the static function **`get_devices`** for retrieving a vector of all available devices in the system

This includes the host device

Querying the topology using a device selector

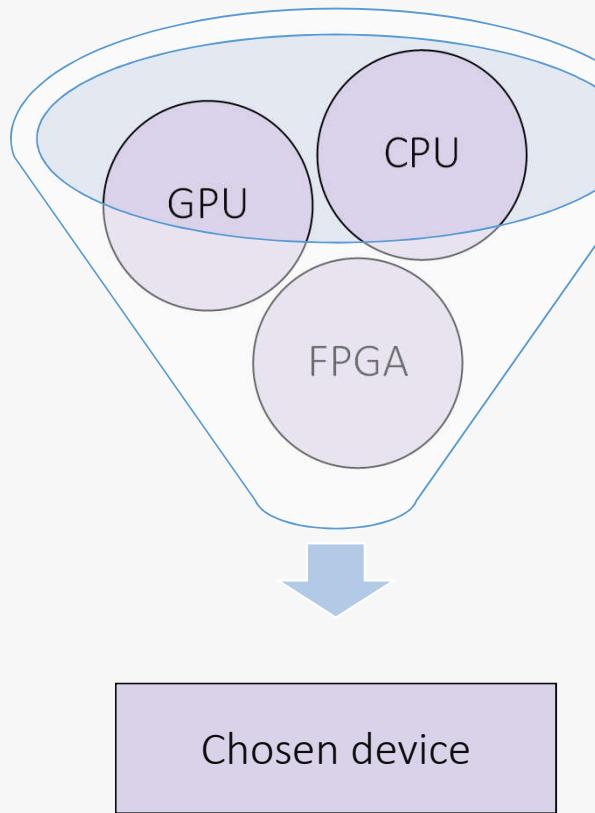


To simplify the process of traversing the system topology SYCL provides device selectors

A from the **device_selector** class device selector is a C++ function object, inherited , which defines a heuristic for scoring devices

SYCL provides a number of standard device selectors, e.g. **default_selector**, **host_selector**, **gpu_selector**, etc

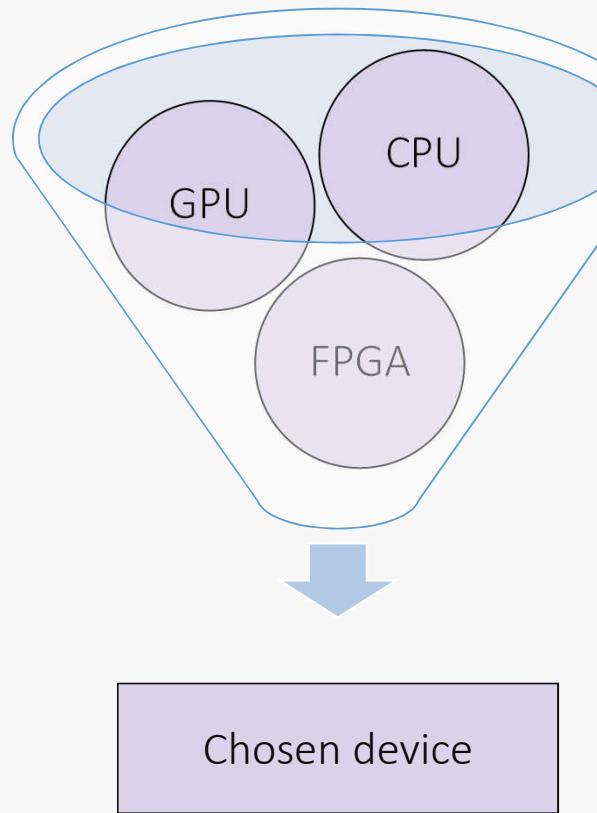
Users can also create their own device selectors



```
auto gpuSelector =  
    gpu_selector{};  
  
auto gpuDevice =  
    gpuSelector.select_device();
```

The `device_selector` class provides the member function `select_device` which queries all devices in the system and scores them and returns the device with the highest score

A device with a negative score will never be chosen



```
auto defSelector =  
    default_selector{};  
  
auto chosenDevice =  
    defSelector.select_device();
```

The **default_selector** is a standard device selector type which will choose a device based on an implementation defined heuristic

Custom device selectors

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

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

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

struct gpu_selector : public device_selector {

    int operator()(const device& dev) const override {
        return dev.get_device_type() == device_type::gpu;
    }
};

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

A device selector must inherit from the default selector

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

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

struct gpu_selector : public device_selector {

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

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

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

    }
}
```

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

This is where you write the logic for scoring each device

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

struct gpu_selector : public device_selector {

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

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

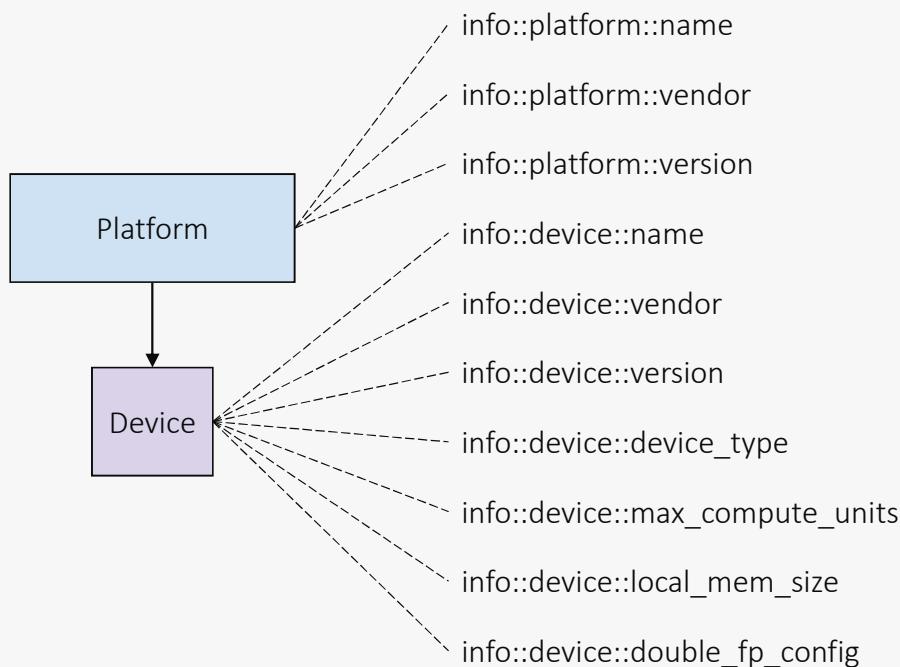
int main(int argc, char *argv[]) {

    auto gpuQueue = queue{gpu_selector{}};
}
```

Now that there is a device selector that chooses a specific device we can use that to construct a queue

Selecting a device

- Whether you are querying the topology manually or automatically
 - You need information about the devices to you are choosing between



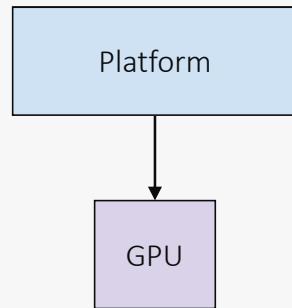
Information about platforms and devices can be queried using the template member function **get_info**

The info that you are querying is specified by the template parameter

You can also query a device for its associated platform with the **get_platform** member function

```

auto plt = dev.get_platform();
auto platformName =
  dev.get_info<info::device::name>();
  
```



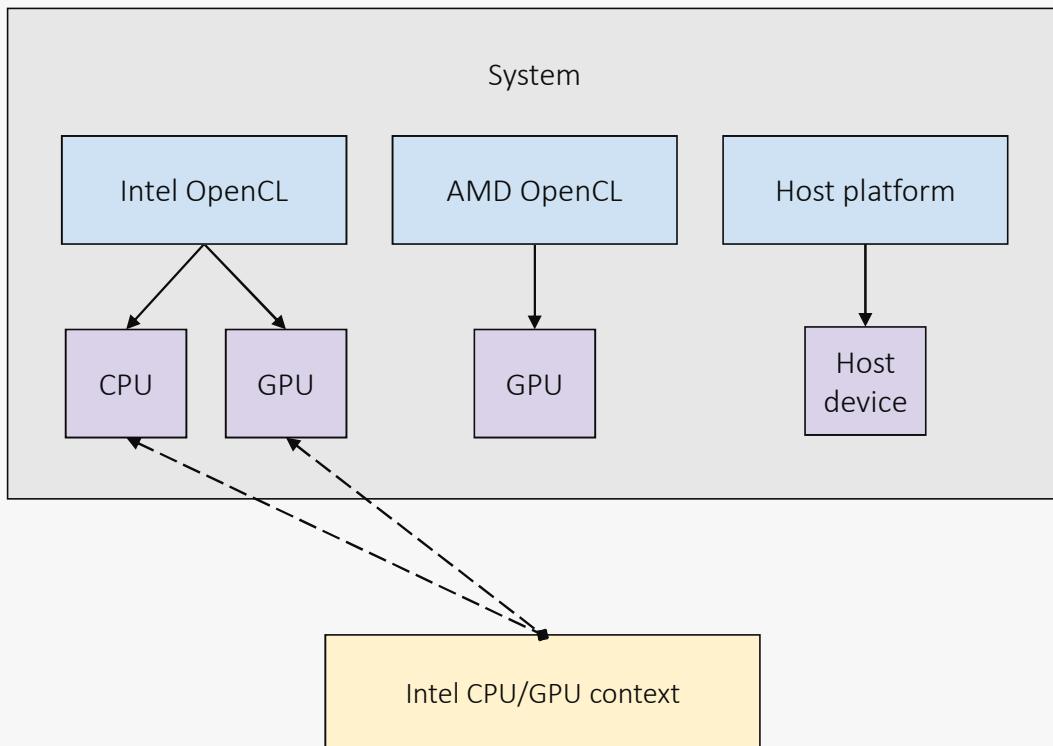
A platform or device can also be queried for supporting OpenCL extensions using the **has_extension** member function

The extension is specified as a string parameter

This checks for both KHR and vendor extensions

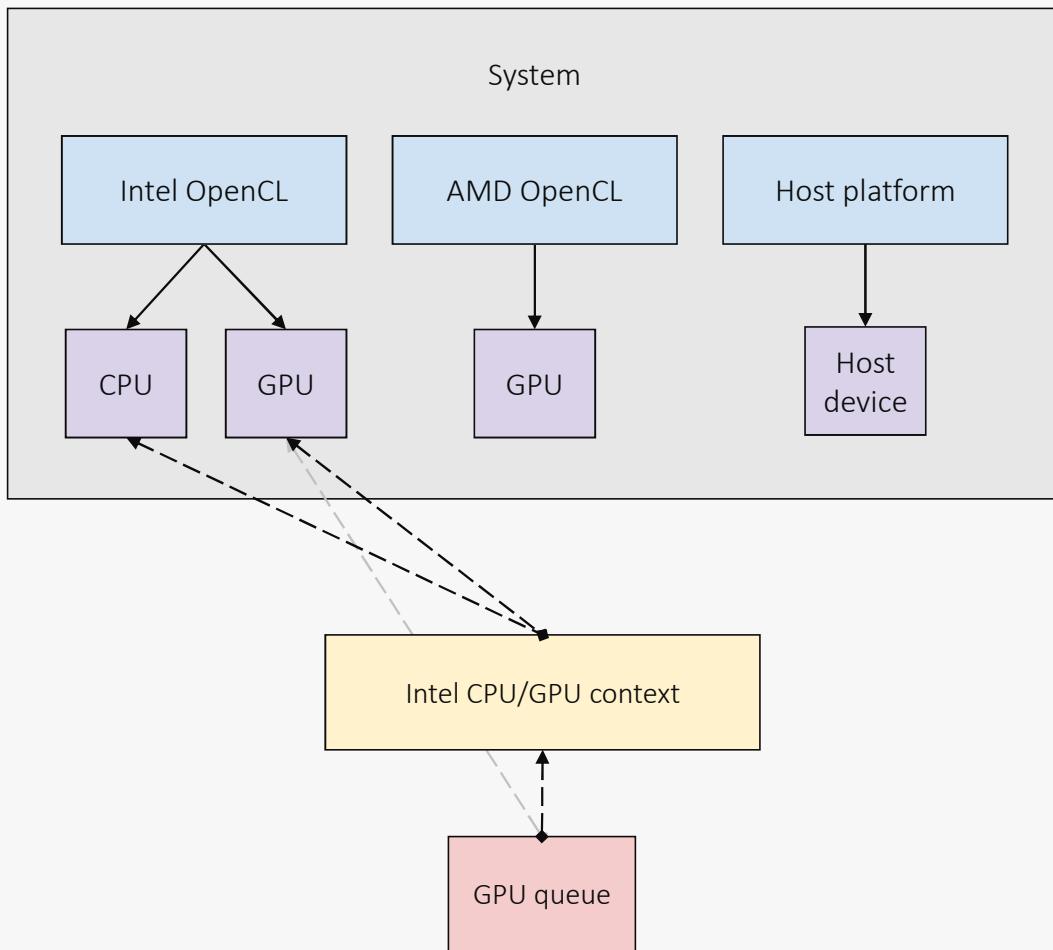
```
auto platformSupportsSpir =  
plt.has_extension("cl_khr_spir");
```

Configuring a queue



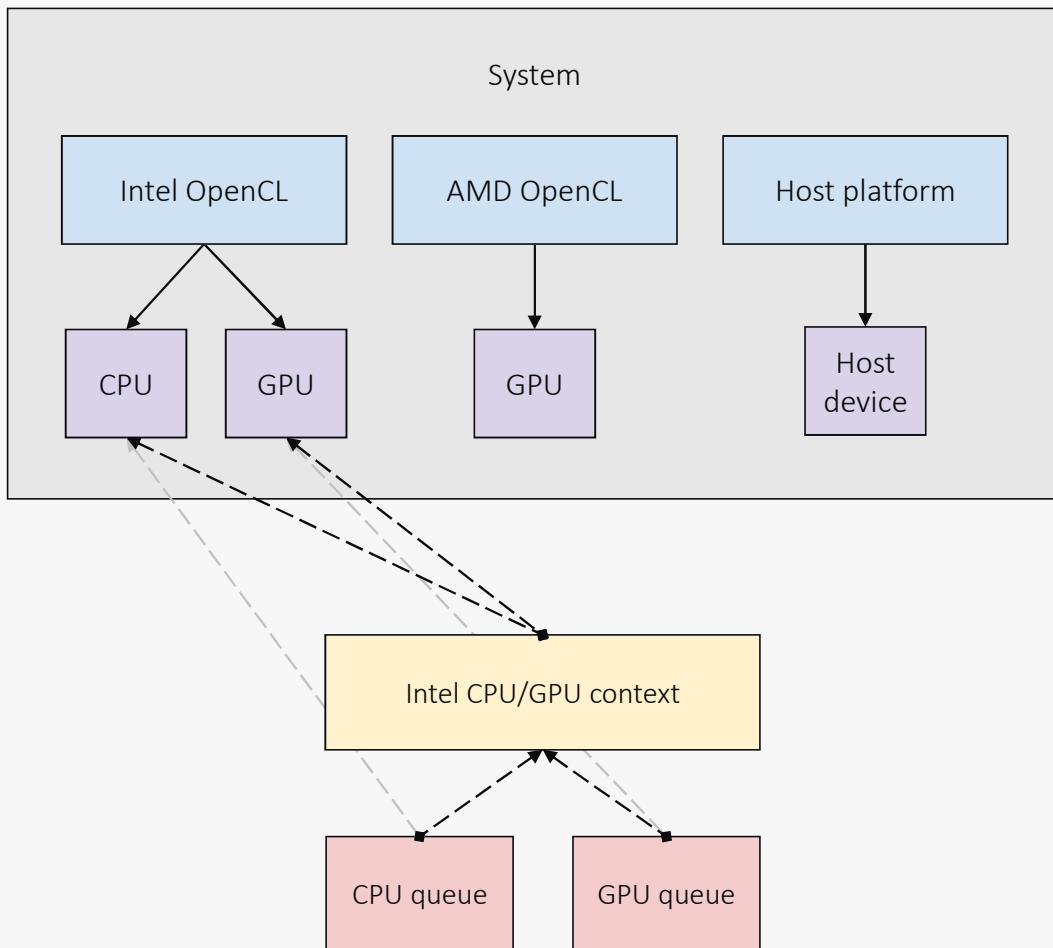
In SYCL the underlying execution and memory resources of a platform and its devices is managed by creating a context

A context represents one or more devices, but all devices must be associated with the same platform



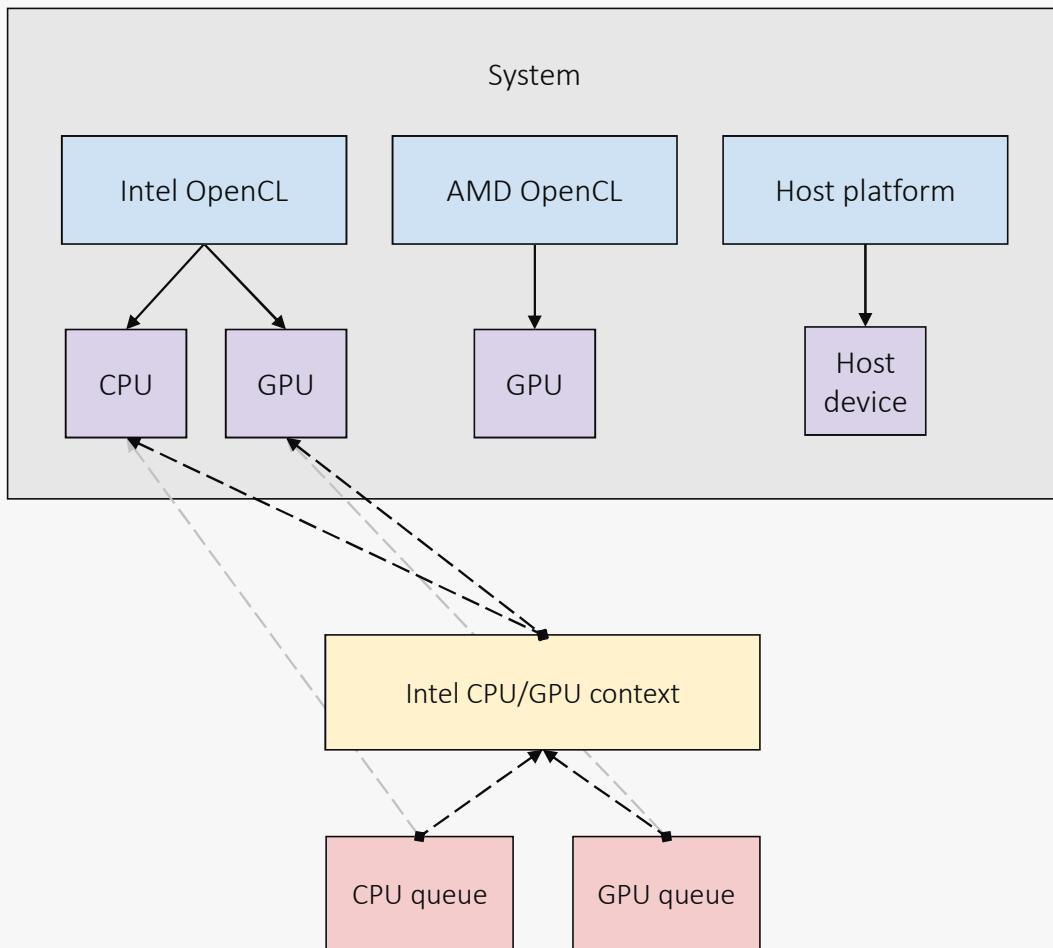
In SYCL the object which is used to submit work is the queue

A queue is associated with a context and a specific device



A single SYCL application will often want to target several different devices

This can be useful for task level parallelism and load balancing

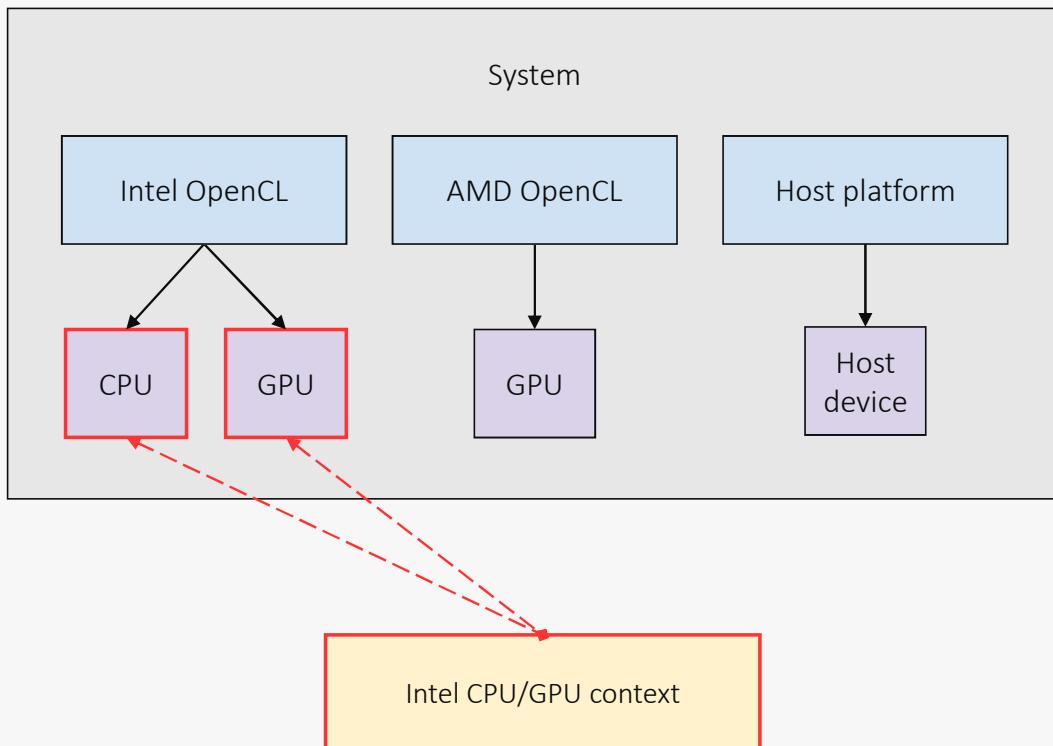


A context is created using the
context class

A queue is created using the
queue class

- There are several ways to construct a queue
 - Ranging in levels of control over the configuration you require

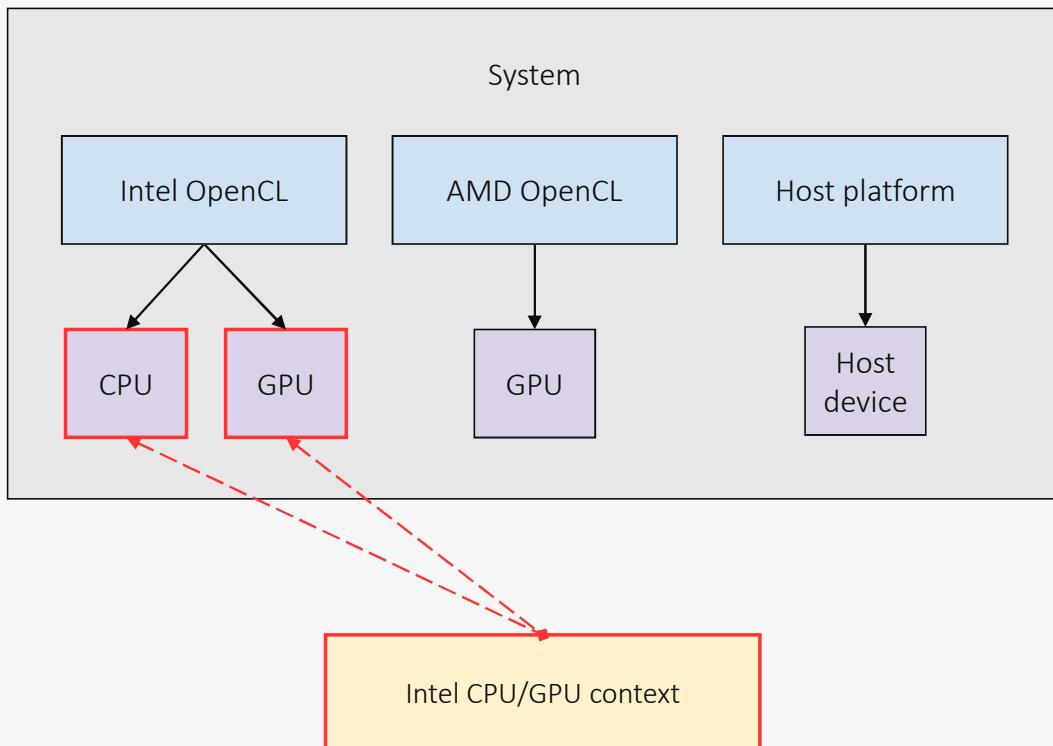
Creating a Context



```
auto defaultContext =  
    context{};
```

A default constructed context object will use the **default_selector** to choose a device

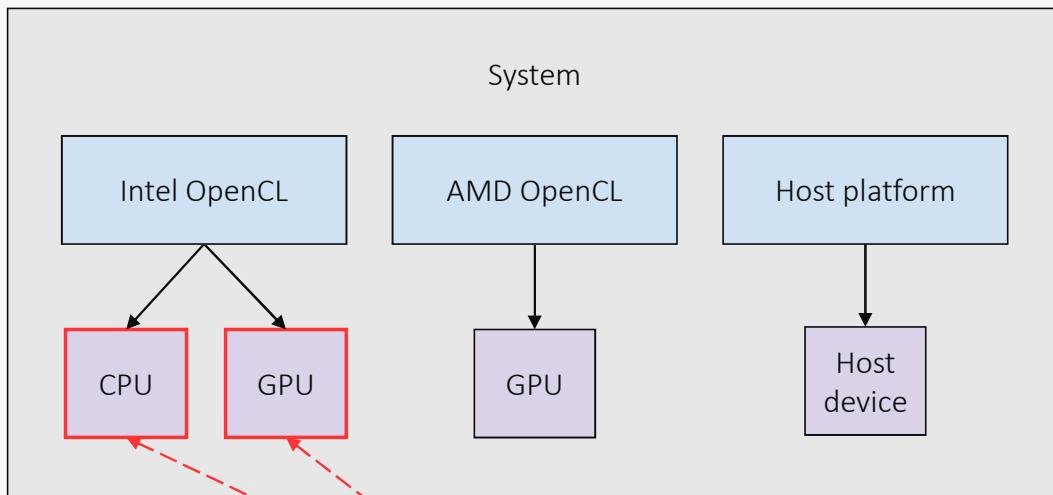
It will then create a context for all devices that share a platform with the chosen device



```
auto intelSelector =  
    intel_selector{};  
  
auto intelContext =  
    context{intelSelector};
```

A context object can be constructed from a device selector

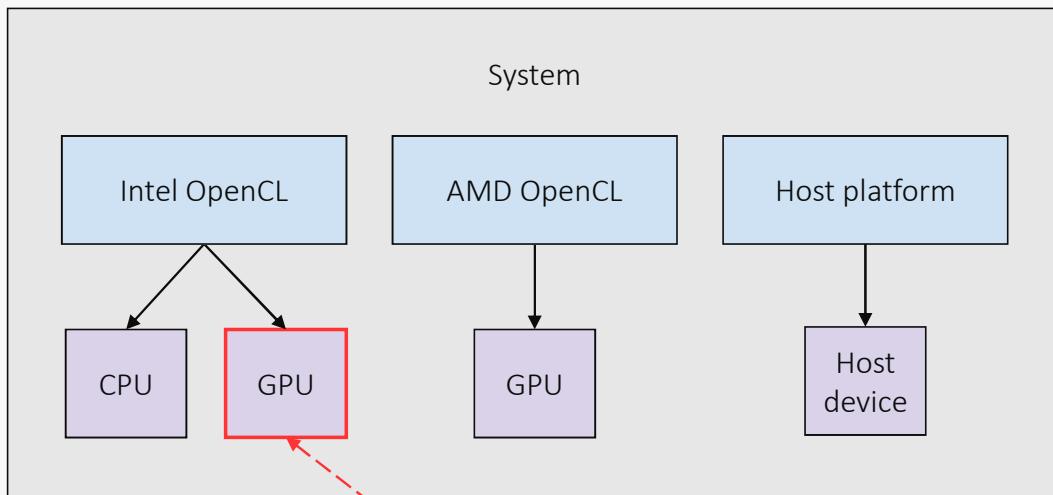
This will use the device selector to choose a device and the create a context for all devices that share a platform with the chosen device



```
auto intelContext =  
    context{intelPlatform};
```

A context object can be constructed from a platform

This will create a context for all devices associated with that platform

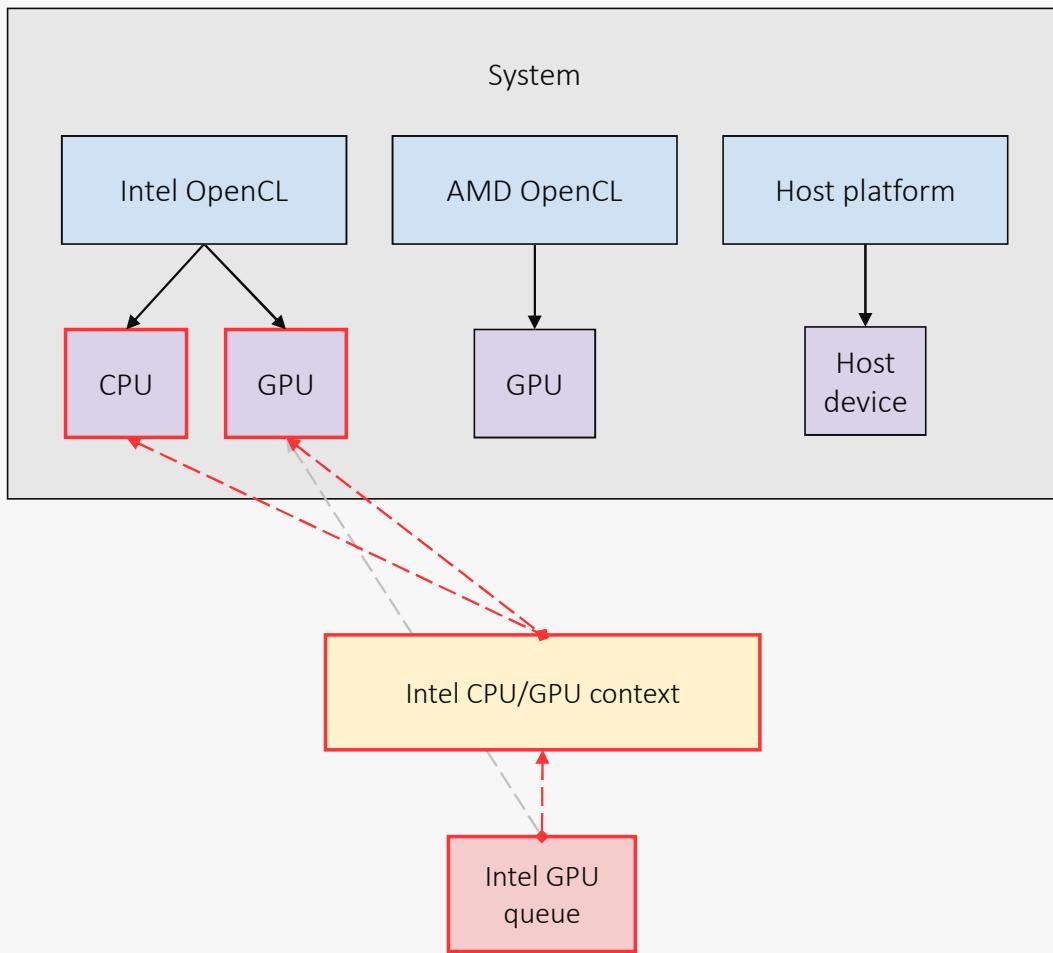


```
auto intelGPUContext =
    context{intelGPUDevice};
```

A context object can be constructed from a device

This will create a context for only that device

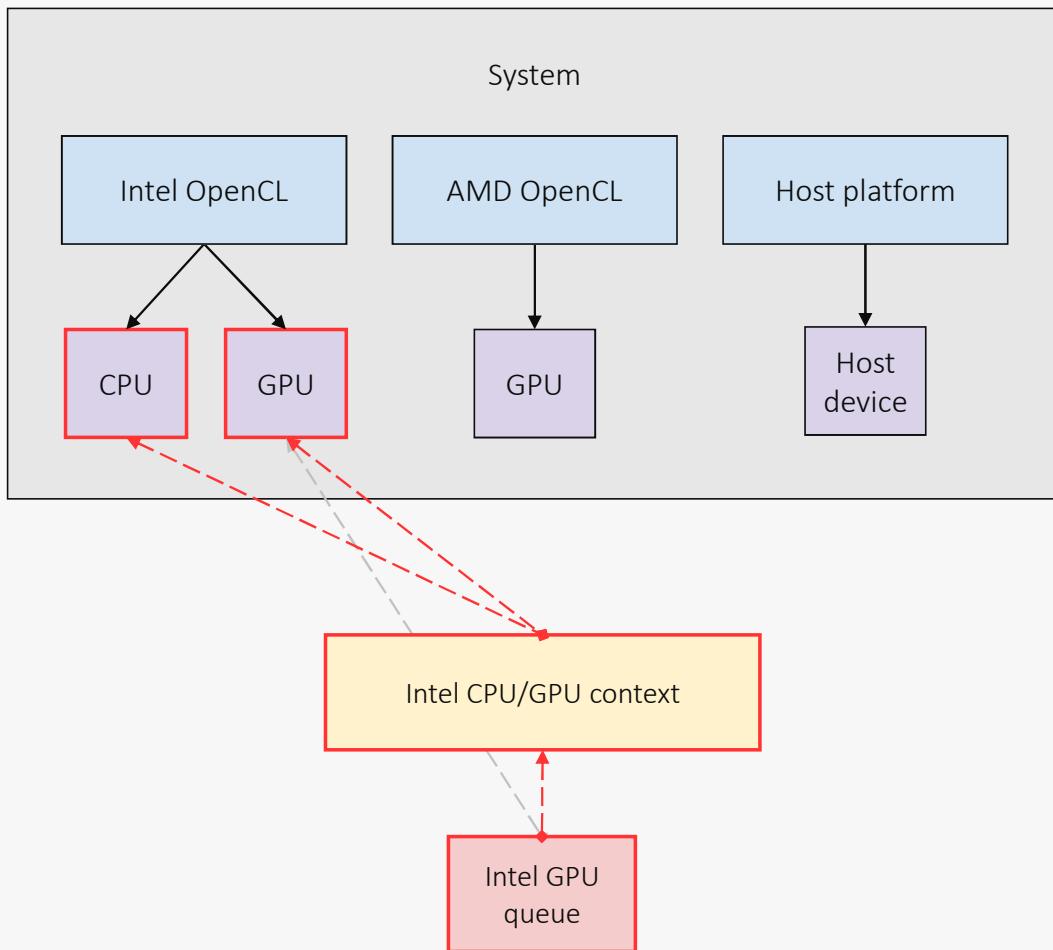
Creating a Queue



```
auto defaultQueue = queue{};
```

A default constructed queue object will use the **default_selector** to choose a device

It will then create a queue for the chosen device, creating an implicit context for all devices that share a platform with the chosen device

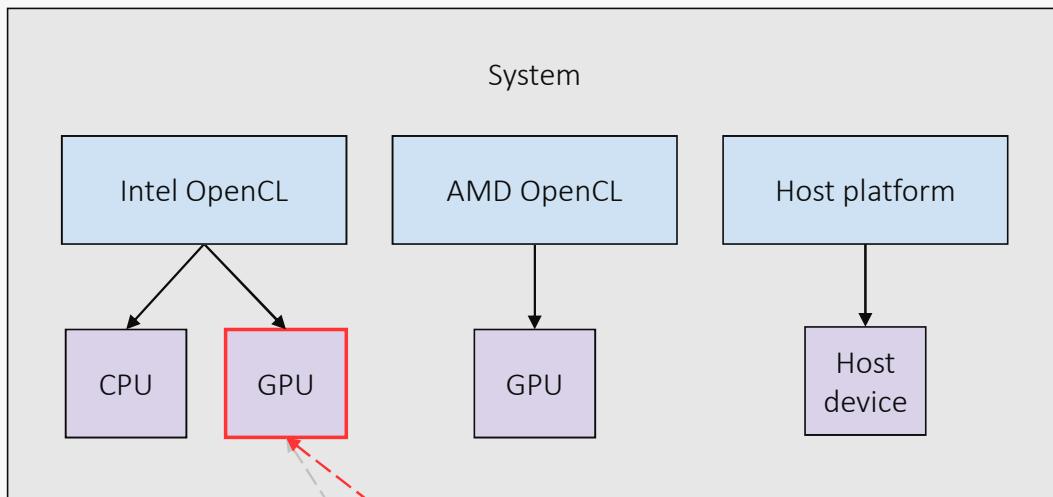


```
auto intelGPUSelector =
    intel_gpu_selector{};

auto intelGPUQueue =
    queue{intelGPUSelector};
```

A queue object can be constructed from a device selector which is used to choose a device

It will then create a queue for the chosen device, creating an implicit context for all devices that share a platform with the chosen device



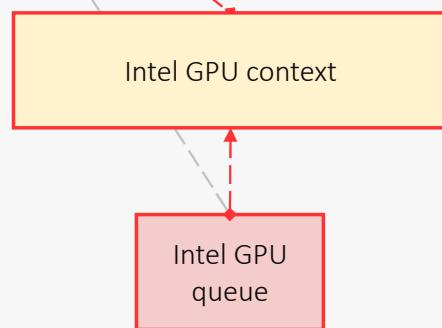
```
auto intelGPUSel =
    intel_gpu_selector{};

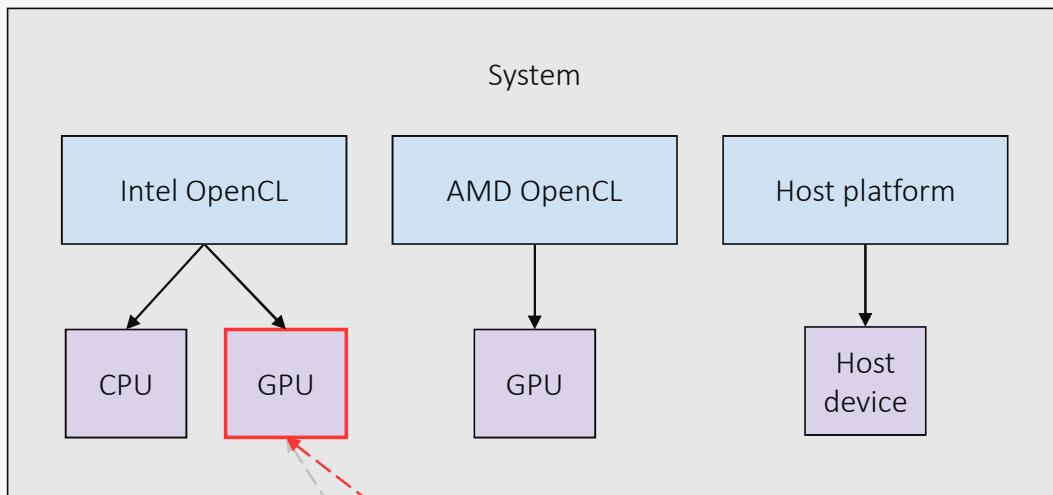
auto intelGPUQueue =
    queue{intelContext,
        intelGPUSel};
```

A queue object can be constructed from a context

A device selector must also be provided which is used to choose a device

It will then create a queue for the chosen device, using the context provided





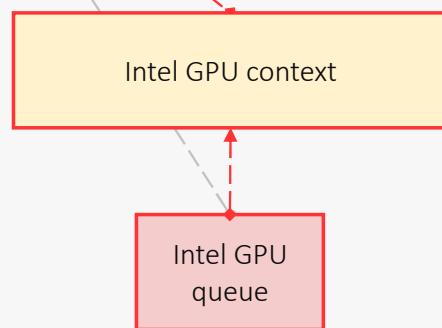
```

auto intelGPUSel =
    intel_gpu_selector{};

auto intelGPUDevice =
    intelGPUSel.select_device();

auto intelGPUQueue =
    queue{intelGPUDevice};

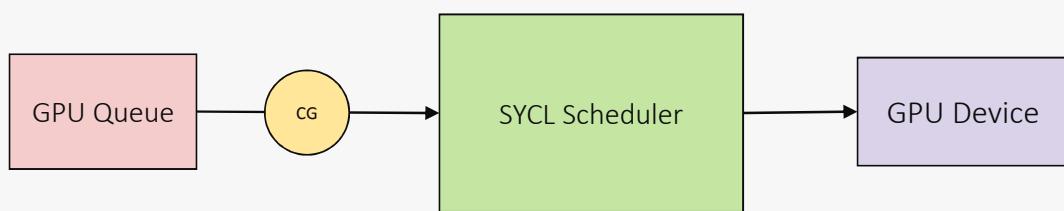
```



A queue object can be constructed from a device

This will create a queue for only that device

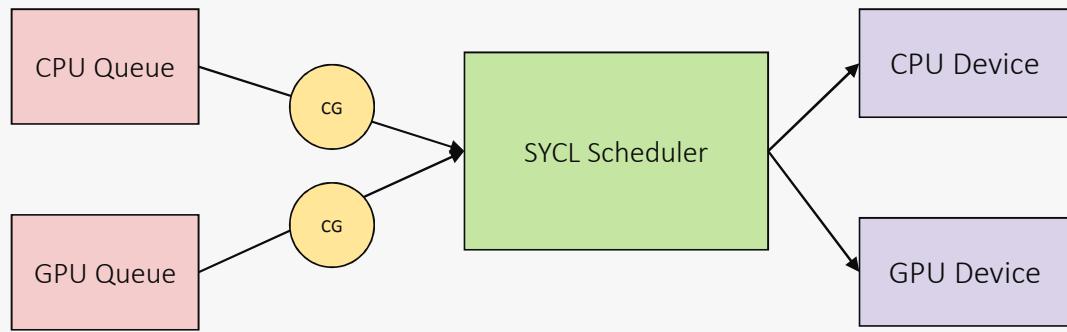
Submitting work to a queue



In SYCL work is submitted via a queue object

This is done using the **submit** member function

This will submit a command group to the SYCL scheduler for execution on the device associated with the queue



The same scheduler is used for all queues in order to share dependency information

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

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

    gpuQueue.submit([&] (handler &cgh) {
        // Command group
    });

}
```

The **submit** member function takes a C++ function object, which takes a reference to a **handler** object

The function object can be a lambda or a class with a function call operator

The body of the function object represents the command group that is being submitted

The handler object is created by the SYCL runtime and is used to link commands and requirements declared inside the command group

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

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

    gpuQueue.submit([&] (handler &cgh) {
        // Command group
    });
}
```

The command group is processed exactly once when **submit** is called

At this point all the commands and requirements declared inside the command group are collected together, processed and passed on to the scheduler

The work is then enqueued to the device asynchronously by the SYCL scheduler, potentially in another thread

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

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

    gpuQueue.submit([&] (handler &cgh) {
        // Command group
    });

    gpuQueue.wait();
}
```

The queue object will not wait for work to complete on destruction

You must wait on the queue to complete if there are no data dependencies

Key takeaways

SYCL allows you to discover the topology of your system

A SYCL queue is used to enqueue work to a device such as a GPU

A SYCL device selector allows you to specify how a device is chosen

SYCL provides the **get_info** API for querying information about devices



Questions?

Exercise 1:

Configuring a queue

- How to construct a queue using a device selector
- How to define and use your own device selector
- How to query information about a device