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

Michael Wong, Gordon Brown

ACCU 2018
VP of R&D of Codeplay

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

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

We build GPU compilers for semiconductor companies

- Now working to make AI/ML heterogeneous acceleration safe for autonomous vehicle

Who am I? Who are we?

Ported TensorFlow to open standards using SYCL

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

Build LLVM-based compilers for accelerators

Implement OpenCL and SYCL for accelerator processors
Gordon Brown

- Background in C++ programming models for heterogeneous systems
- Developer with Codeplay Software for 6 years
- Worked on ComputeCpp (SYCL) since its inception
- Contributor to the Khronos SYCL standard for 6 years
- Contributor to C++ executors and heterogeneity for 2 years
Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.

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

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

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

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

OTHER COMPANY, PRODUCT, AND SERVICE NAMES MAY BE TRADEMARKS OR SERVICE MARKS OF OTHERS.
Codeplay - Connecting AI to Silicon

Products

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

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

Addressable Markets

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

Technologies: Vision Processing
Machine Learning
Artificial Intelligence
Big Data Compute

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

Customers
3 Act Play

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

1. What’s still missing from C++?
What have we achieved so far for C++20?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Depends on</th>
<th>Current target (estimated, could slip)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concepts</td>
<td></td>
<td>C++20 (adopted, including convenience syntax)</td>
</tr>
<tr>
<td>Contracts</td>
<td></td>
<td>C++20 (adopted)</td>
</tr>
<tr>
<td>Ranges</td>
<td></td>
<td>C++20 (adopted)</td>
</tr>
<tr>
<td>Coroutines</td>
<td></td>
<td>C++20</td>
</tr>
<tr>
<td>Modules</td>
<td></td>
<td>C++20</td>
</tr>
<tr>
<td>Reflection</td>
<td></td>
<td>TS in C++20 timeframe, IS in C++23</td>
</tr>
<tr>
<td>Executors</td>
<td></td>
<td>Lite in +12 timeframe, Full in C++23</td>
</tr>
<tr>
<td>Networking</td>
<td>Executors, and possibly Coroutines</td>
<td>C++23</td>
</tr>
<tr>
<td>future.then, async2</td>
<td>Executors</td>
<td></td>
</tr>
</tbody>
</table>
Use the Proper Abstraction with C++

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

Task parallelism:

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

Data parallelism:

- Many small tasks with same operations on multiple data
- Optimized for throughput
Review of Latency, bandwidth, throughput

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

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

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

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

Example

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

The latency is: 8 hours.

The throughput is: 120 cars / day or 5 cars / hour.
Flynn’s Taxonomy

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

Most common parallel computer systems
What kind of processors should we build

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

**GPU**
- Large number of small processors
- Less control structures and more processing units
  - Can do less complex logic
  - Lower power consumption
- Optimised for throughput
  - Maximising the amount of work done per unit of time
Multicore CPU vs Manycore GPU

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

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

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

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

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

- Virtually all processors have SIMD memory subsystems

- This has 2 effects
  - Sparse access wastes bandwidth
    - 2 words used, 8 words loaded: \( \frac{1}{4} \) effective bandwidth
  - Unaligned access wastes bandwidth
    - 4 words used, 8 words loaded: \( \frac{1}{2} \) effective bandwidth
Data Structure Padding

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

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

• 1998, when C++ 98 was released
  • Intel Pentium II: 0.45 GFLOPS
  • No SIMD: SSE came in Pentium III
  • No GPUs: GPU came out a year later

• 2011: when C++11 was released
  • Intel Core-i7: 80 GFLOPS
  • AVX: 8 DP flops/Hz*4 cores *4.4 GHz = 140 GFlops
  • GTX 670: 2500 GFLOPS

• Computers have gotten so much faster, how come software have not?
  • Data structures and algorithms
In 1998, a typical machine had the following flops.

0.45 GFLOPS, 1 core

Single threaded C++98/C99/Fortran dominated this picture.
In 2011, a typical machine had the following flops

80 GFLOPS 4 cores

To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, OpenCL
In 2011, a typical machine had the following flops

80 GFLOPS 4 cores+140 GFLOPS AVX

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

To program the vector unit, you have to use Intrinsics, OpenCL, CUDA, or auto-vectorization
In 2011, a typical machine had the following flops

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

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

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

To program the GPU, you have to use CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP.
In 2017, a typical machine had the following flops

140 GFLOPS + 560 GFLOPS AVX + 4600 GFLOPS GPU

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

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

To program the GPU, you have to use SYCL, CUDA, OpenCL, OpenGL, DirectX, Intrinsics, OpenMP
“The end of Moore’s Law”

“The free lunch is over”

“The future is parallel and heterogeneous”

“GPUs are everywhere”
Take a typical Intel chip

- **Intel Core i7 7th Gen**
  - 4x CPU cores
    - Each with hyperthreading
    - Each with 8-wide AVX instructions
  - **GPU**
    - With 1280 processing elements
Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip.
Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip.

Using vectorisation allows you to fully utilise the resources of a single hyperthread.
## Use the Proper Abstraction with C++

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>How is it supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>C++11/14/17 threads, async</td>
</tr>
<tr>
<td>HW threads</td>
<td>C++11/14/17 threads, async, hw_concurrency</td>
</tr>
<tr>
<td>Vectors</td>
<td>Parallelism TS2-&gt;</td>
</tr>
<tr>
<td>Atomic, Fences, lockfree, futures, counters, transactions</td>
<td>C++11/14/17 atomics, Concurrency TS1-&gt; Transactional Memory TS1</td>
</tr>
<tr>
<td>Parallel Loops</td>
<td>Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each</td>
</tr>
<tr>
<td>Heterogeneous offload, fpga</td>
<td>OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja</td>
</tr>
<tr>
<td>Distributed</td>
<td>HPX, MPI, UPC++</td>
</tr>
<tr>
<td>Caches</td>
<td>C++17 false sharing support</td>
</tr>
<tr>
<td>Numa</td>
<td></td>
</tr>
<tr>
<td>TLS</td>
<td></td>
</tr>
<tr>
<td>Exception handling in concurrent environment</td>
<td></td>
</tr>
</tbody>
</table>
Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip.

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

Using multi-threading allows you to fully utilise all CPU cores.
## Use the Proper Abstraction with C++

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>How is it supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>C++11/14/17 threads, async</td>
</tr>
<tr>
<td>HW threads</td>
<td>C++11/14/17 threads, async, hw_concurrency</td>
</tr>
<tr>
<td>Vectors</td>
<td>Parallelism TS2-&gt;</td>
</tr>
<tr>
<td>Atomic, Fences, lockfree, futures, counters, transactions</td>
<td>C++11/14/17 atomics, Concurrency TS1-&gt; Transactional Memory TS1</td>
</tr>
<tr>
<td>Parallel Loops</td>
<td>Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each</td>
</tr>
<tr>
<td>Heterogeneous offload, fpga</td>
<td>OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja</td>
</tr>
<tr>
<td>Distributed</td>
<td>HPX, MPI, UPC++</td>
</tr>
<tr>
<td>Caches</td>
<td>C++17 false sharing support</td>
</tr>
<tr>
<td>Numa</td>
<td></td>
</tr>
<tr>
<td>TLS</td>
<td></td>
</tr>
<tr>
<td>Exception handling in concurrent environment</td>
<td></td>
</tr>
</tbody>
</table>
Serial C++ code alone only takes advantage of a very small amount of the available resources of the chip

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

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

Using heterogeneous dispatch allows you to fully utilise the entire chip
GPGPU programming was once a niche technology

- Limited to specific domain
- Separate source solutions
-Verbose low-level APIs
- Very steep learning curve
## Coverage after C++11

<table>
<thead>
<tr>
<th>Asynchronous Agents</th>
<th>Concurrent collections</th>
<th>Mutable shared state</th>
<th>Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>summary</td>
<td>tasks that run independently and communicate via messages</td>
<td>operations on groups of things, exploit parallelism in data and algorithm structures</td>
<td>avoid races and synchronizing objects in shared memory</td>
</tr>
<tr>
<td>examples</td>
<td>GUI, background printing, disk/net access</td>
<td>trees, quicksorts, compilation</td>
<td>locked data(99%), lock-free libraries (wizards), atomics (experts)</td>
</tr>
<tr>
<td>key metrics</td>
<td>responsiveness</td>
<td>throughput, many core scalability</td>
<td>race free, lock free</td>
</tr>
<tr>
<td>requirement</td>
<td>isolation, messages</td>
<td>low overhead</td>
<td>composability</td>
</tr>
<tr>
<td>today's abstractions</td>
<td>C++11: thread, lambda function, TLS</td>
<td>C++11: Async, packaged tasks, promises, futures, atomics</td>
<td>C++11: locks, memory model, mutex, condition variable, atomics, static init/term</td>
</tr>
</tbody>
</table>
Top500 contenders
Internet of Things

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

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

C++AMP  
SYCL  
CUDA Agency  
Kokkos  
HPX  
Raja
C++ Executors: Unified interface for execution

- **SYCL / OpenCL / CUDA / HCC**
- **OpenMP / MPI**
- **C++ Thread Pool**
- **Boost.Asio / Networking TS**
Act 2

1. What’s still missing from C++?

2. What makes GPU work so fast?
The way of CPU and GPU

The diagram shows the relationship between CPU and GPU memory. The host code is connected to both the CPU and GPU, which are referred to as "Host" and "Device" respectively. The CPU and GPU memory are connected by arrows indicating data transfer. The diagram highlights the co-processor relationship between the CPU and GPU.
The way of CPU and GPU

1. The CPU allocates memory on the GPU
The way of CPU and GPU

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

CPU
(“Host”)

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

GPU
(“Device”)

GPU memory
1. A GPU has a region of dedicated global memory

GPU
(“Device”)

Global memory
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements
1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements
6. Each processing element has dedicated private memory
1. A processing element executes a single work-item
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
1. A processing element executes a single work-item.

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

3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit.
1. A processing element executes a single work-item.
2. Each work-item can access private memory, a dedicated memory region for each processing element.
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit.
4. Each work-item can access local memory, a dedicated memory region for each compute unit.
1. A processing element executes a single work-item.
2. Each work-item can access private memory, a dedicated memory region for each processing element.
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit.
4. Each work-item can access local memory, a dedicated memory region for each compute unit.
5. A GPU executes multiple work-groups.
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. A compute unit executes a work-group, composed of multiple work-items, one for each processing element in the compute unit
4. Each work-item can access local memory, a dedicated memory region for each compute unit
5. A GPU executes multiple work-groups
6. Each work-item can access global memory, a single memory region available to all processing elements on the GPU
1. Multiple work-items will execute concurrently
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs do execute a number of work-items uniformly (lock-step), but that number is unspecified
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs do execute a number of work-items uniformly (lock-step), but that number is unspecified
4. A work-item can share results with other work-items via local and global memory
1. Multiple work-items will execute concurrently
2. They are not guaranteed to all execute uniformly
3. Most GPUs do execute a number of work-items uniformly (lock-step), but that number is unspecified
4. A work-item can share results with other work-items via local and global memory
5. However this means that it’s possible for a work-item to read a result that hasn’t yet been written to yet, you have a data race
1. This problem can be solved by a synchronisation primitive called a work-group barrier.
1. This problem can be solved by a synchronisation primitive called a work-group barrier.

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

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

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

3. So now you can be sure that all of the results that you want to read from have been written to.
1. This problem can be solved by a synchronisation primitive called a work-group barrier.

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

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

4. However this does not apply across work-group boundaries, and you have a data race again.
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)

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

2. Again you can be sure that all of the results that you want to read from have been written to

3. However kernel barriers have a higher overhead as they require you to launch another kernel
1. This problem can be solved by a synchronisation primitive called a kernel barrier (launching separate kernels)

2. Again you can be sure that all of the results that you want to read from have been written to

3. However kernel barriers have a higher overhead as they require you to launch another kernel

4. And kernel barriers require results to be stored in global memory, local memory is not persistent across kernels
## CUDA vs OpenCL terminology

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread</td>
<td>work-item</td>
</tr>
<tr>
<td>warp</td>
<td>wavefront</td>
</tr>
<tr>
<td>thread block</td>
<td>work-group</td>
</tr>
<tr>
<td>grid</td>
<td>computation domain</td>
</tr>
<tr>
<td>global memory</td>
<td>global memory</td>
</tr>
<tr>
<td>shared memory</td>
<td>local memory</td>
</tr>
<tr>
<td>local memory</td>
<td>private memory</td>
</tr>
<tr>
<td>streaming multiprocessor (SM)</td>
<td>compute unit</td>
</tr>
<tr>
<td>scalar core</td>
<td>processing element</td>
</tr>
</tbody>
</table>

© 2018 Codeplay Software Ltd.
Sequential CPU code

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

calc(in, out);
```

SPMD GPU code

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

/* specify degree of parallelism */
parallel_for(calc, in, out, 1024);
```
SPMD: Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data.

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

SPMD is a parallel execution model and assumes multiple cooperating processors executing a program.
Kernels are launched in the form of an nd-range.
An nd-range can be 1, 2 or 3 dimensions.
An nd-range describes a number of work-items divided into equally sized work-groups.
An nd-range is constructed from the total number of work-items (global range) and the number of work-items in a work-group (local range).

nd-range \{\{12, 12\}, \{4, 4\}\}
An nd-range is mapped to the underlying hardware
  ○ Work-groups are mapped to compute units
  ○ Work-items are mapped to processing units
The kernel is executed once per work-item in the nd-range.
Each work item knows its index within the nd-range:
- global range \{12, 12\}
- local range \{4, 4\}
- group range \{3, 3\}
- global id \{6, 5\}
- local id \{2, 1\}
- group id \{1, 1\}
Act 3

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

Cross-platform, single-source, high-level, C++ programming layer
Built on top of OpenCL and based on standard C++11
Delivering a heterogeneous programming solution for C++
Why use SYCL to program a GPU?

- Enables programming heterogeneous devices such as GPUs using standard C++
- Provides a high-level abstraction for development of complex parallel software applications
- Provides efficient data dependency analysis and task scheduling and synchronisation
__global__ vec_add(float *a, float *b, float *c) {
    return c[i] = a[i] + b[i];
}

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

vector<float> a, b, c;
#pragma parallel_for
for(int i = 0; i < a.size(); i++) {
    c[i] = a[i] + b[i];
}

cgh.parallel_for<vec_add>(range, [=](cl::sycl::id<2> idx) {
    c[idx] = a[idx] + c[idx];
});
SYCL separates the storage and access of data through the use of buffers and accessors.

SYCL provides data dependency tracking based on accessors to optimise the scheduling of tasks.
Buffers and accessors are type safe access across host and device.

Buffers manage data across the host and one or more devices.

Accessors are used to describe access requirements.
Buffer

- host_buffer accessor: Request access to a buffer immediately on the host
- global_buffer accessor: Request access to a buffer in the global memory region
- constant_buffer accessor: Request access to a buffer in the constant memory region
- local accessor: Allocate memory in the local memory region
Implicit vs Explicit Data Movement

Examples:
- SYCL, C++ AMP

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

Examples:
- OpenCL, CUDA, OpenMP

Implementation:
- Data is moved to the device via explicit copy APIs

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

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

Here we’re using CUDA as an example

```c++
float *h_a = { ... }, d_a;
cudaMalloc(void **&d_a, size);
cudaMemcpy(d_a, h_a, size,
cudaMemcpyHostToDevice);
vec_add<<<64, 64>>>(a, b, c);
cudaMemcpy(d_a, h_a, size,
cudaMemcpyDeviceToHost);
```
Benefits of data dependency task graphs

- Allows you to describe your problems in terms of relationships
  - Removes the need to en-queue explicit copies
  - Removes the need for complex event handling

- Allows the runtime to make data movement optimizations
  - Preemptively copy data to a device before kernels
  - Avoid unnecessarily copying data back to the host after execution on a device
  - Avoid copies of data that you don’t need
## Coverage after C++17

<table>
<thead>
<tr>
<th>summary</th>
<th>Asynchronous Agents</th>
<th>Concurrent collections</th>
<th>Mutable shared state</th>
<th>Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tasks that run independently and communicate via messages</td>
<td>operations on groups of things, exploit parallelism in data and algorithm structures</td>
<td>avoid races and synchronizing objects in shared memory</td>
<td>Dispatch/offload to other nodes (including distributed)</td>
</tr>
<tr>
<td>today’s abstractions</td>
<td>C++11: thread, lambda function, TLS, async</td>
<td>C++11: Async, packaged tasks, promises, futures, atomics, C++ 17: ParallelSTL, control false sharing</td>
<td>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</td>
<td>C++17: progress guarantees, TOE, execution policies</td>
</tr>
</tbody>
</table>
C++17 introduces a number of parallel algorithms and new execution policies which dictate how they can be parallelized.

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

Execution policies:
- sequenced_execution_policy (seq)
- parallel_execution_policy (par)
- parallel_unsequenced_execution_policy (par_unseq)
result accumulate(first, last, init, [binary_op])

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

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

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

first acc = init
then for each it in [first, last) in order
   acc = binary_op(acc, *it)
then return acc
result \ reduce([\text{execution\_policy},]
\text{first}, \text{last},
\text{init},
[\text{binary\_op}])

\text{first acc} = \text{GSUM}(\text{binary\_op}, \text{init},
\text{first}, ...,
*(\text{last}-1))

\text{then return acc}
result reduce([execution_policy,]
  first, last,
  init,
  [binary_op])

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

then return acc
result reduce([execution_policy,]
    first, last,
    init,
    [binary_op])

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

then return acc
result reduce([execution_policy,]
first, last,
init,
[binary_op])

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

then return acc
result reduce([execution_policy,]
first, last,
init,
[binary_op])

first acc = GSUM(binary_op, init, *first,...,
*(last-1))
then return acc
Due to the requirements of GSUM reduce is allowed to be unordered

However this means that `binary_op` is required to be both **commutative** and **associative**
Commutativity means changing the order of operations does not change the result

<table>
<thead>
<tr>
<th>Integer operations</th>
<th>Floating-point operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y == y + x$</td>
<td>$x + y == y + x$</td>
</tr>
<tr>
<td>$x * y == y * x$</td>
<td>$x * y == y * x$</td>
</tr>
<tr>
<td>$x - y != y - x$</td>
<td>$x - y != y - x$</td>
</tr>
<tr>
<td>$x / y != y / x$</td>
<td>$x / y != y / x$</td>
</tr>
</tbody>
</table>
Associativity means changing the grouping of operations does not change the result

**Integer operations**

(x + y) + z == x + (y + z)
(x * y) * z == x * (y * z)
(x - y) - z != x - (y - z)
(x / y) / z != x / (y / z)

**Floating-point operations**

(x + y) + z != x + (y + z)
(x * y) * z != x * (y * z)
(x - y) - z != x - (y - z)
(x / y) / z != x / (y / z)
So how do we parallelise this on a GPU?

- We want to utilize the available hardware
- We want to keep dependencies to a minimum
- We want to make efficient use of local memory and work-group synchronization
Here we have the standard prototype for the reduce parallel algorithm, taking a SYCL execution policy.

There is an assumption here that the iterators are contiguous.
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
}

SYCL separates memory storage and access using buffers and accessors

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

Accessors represent an instance of access to a particular buffer
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
}

We create a buffer to manage the input data

We call set_final_data with nullptr in order to tell the runtime not to copy back to the original host address on destruction
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
}

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

So in this case, on returning from the reduce function...
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
}

In SYCL devices are selected using a device selector

A device selector picks the best device based on a particular heuristic
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
}
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
    size_t data_size = std::distance(first, last);
    auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
}

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

These are important for determining how work is distributed across work-groups.
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
    size_t dataSize = std::distance(first, last);
    auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
    do {
        dataSize /= maxWorkGroupSize;
    } while (dataSize > 1);
}
template <class It, class T, class BinOp>
The function reduce is defined as follows:

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

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

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

We create an accessor for the input buffer
The access mode is read_write because we want to be able to write back a result
template <class It, class T, class BinOp>
T reduce(sycl_execution_policy_t policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
    size_t dataSize = std::distance(first, last);
    auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
    do {
        q.submit([&](handler& cgh) {
            auto global = dataSize;
            auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
            auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
            accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
        });
        dataSize /= maxWorkGroupSize;
    } while (dataSize > 1);
}

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

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

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

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

We copy each element from global memory to local memory of their respective work-group
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
    size_t data_size = std::distance(first, last);
    auto max_work_group_size = q.get_device().get_info<info::device::max_work_group_size>();
    do {
        q.submit([&](handler& cgh) {
            auto global = data_size;
            auto local = range<1>(std::min(data_size, max_work_group_size));
            auto input_acc = bufI.template get_access<access::mode::read_write>(cgh);
            accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
            cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
                scratch[it.get_local_id(0)] = input_acc[it.get_global_id(0)];
                it.barrier(access::fence_space::local_space);
            });
        });
        data_size /= max_work_group_size;
    } while (data_size > 1);
}

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

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

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

We branch on the first half of the work-items per loop by only executing work-items before the offset.
We call the binary_op with the elements in local memory of the current work-item and the respective work-item on the other side of the offset and assign the result to the element in local memory of the current work-item.
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
    using value_t = typename std::iterator_traits<It>::value_type;
    buffer<value_t, 1> bufI(first, last);
    bufI.set_final_data(nullptr);
    queue q(gpu_selector{});
    size_t dataSize = std::distance(first, last);
    auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
    do {
        q.submit([&](handler& cgh) {
            auto global = dataSize;
            auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
            auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
            accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
            cgh.parallel_for<KernelName>(nd_range<1>(global, local), [=](nd_item<1> it) {
                scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
                it.barrier(access::fence_space::local_space);
                for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
                    if (it.get_local_id(0) < offset) {
                        scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                            scratch[it.get_local_id(0) + offset]);
                    }
                }
                it.barrier(access::fence_space::local_space);
            });
        });
        dataSize /= maxWorkGroupSize;
    } while (dataSize > 1);
}
template <class It, class T, class BinOp, class KernelName>
T reduce(sycl_execution_policy_t<KernelName> policy, It first, It last, T init, BinOp binary_op) {
  using value_t = typename std::iterator_traits<It>::value_type;
  buffer<value_t, 1> bufI(first, last);
  bufI.set_final_data(nullptr);
  queue q(gpu_selector{});
  size_t dataSize = std::distance(first, last);
  auto maxWorkGroupSize = q.get_device().get_info<info::device::max_work_group_size>();
  do {
    q.submit([&](handler& cgh) {
      auto global = dataSize;
      auto local = range<1>(std::min(dataSize, maxWorkGroupSize));
      auto inputAcc = bufI.template get_access<access::mode::read_write>(cgh);
      accessor<value_t, 1, access::mode::read_write, access::target::local>(local, cgh);
      cgh.parallel_for<KernelName>(nd_range<1>(global, local), [&](nd_item<1> it) {
        scratch[it.get_local_id(0)] = inputAcc[it.get_global_id(0)];
        it.barrier(access::fence_space::local_space);
        for (size_t offset = local[0] / 2; offset > 0; offset /= 2) {
          if (it.get_local_id(0) < offset) {
            scratch[it.get_local_id(0)] = binary_op(scratch[it.get_local_id(0)],
                                                   scratch[it.get_local_id(0) + offset]);
          }
          it.barrier(access::fence_space::local_space);
          }
        if (it.get_local_id(0) == 0) {
          inputAcc[it.get_group(0)] = scratch[it.get_local_id(0)];
        }
      });
    });
    dataSize /= maxWorkGroupSize;
  } while (dataSize > 1);
}

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

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

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

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

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

We looked at how this applies to the GPU architecture

We looked at why this is so important in modern C++
## Use the Proper Abstraction with C++

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>How is it supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>C++11/14/17 threads, async</td>
</tr>
<tr>
<td>HW threads</td>
<td>C++11/14/17 threads, async</td>
</tr>
<tr>
<td>Vectors</td>
<td>Parallelism TS2-&gt;C++20</td>
</tr>
<tr>
<td>Atomic, Fences, lockfree, futures, counters, transactions</td>
<td>C++11/14/17 atomics, Concurrency TS1-&gt;C++20, Transactional Memory TS1</td>
</tr>
<tr>
<td>Parallel Loops</td>
<td>Async, TBB:parallel_invoke, C++17 parallel algorithms, for_each</td>
</tr>
<tr>
<td>Heterogeneous offload, fpga</td>
<td>OpenCL, SYCL, HSA, OpenMP/ACC, Kokkos, Raja P0796 on affinity</td>
</tr>
<tr>
<td>Distributed</td>
<td>HPX, MPI, UPC++ P0796 on affinity</td>
</tr>
<tr>
<td>Caches</td>
<td>C++17 false sharing support</td>
</tr>
<tr>
<td>Numa</td>
<td>Executors, Execution Context, Affinity, P0443-Executor TS</td>
</tr>
<tr>
<td>TLS</td>
<td>EALS, P0772</td>
</tr>
<tr>
<td>Exception handling in concurrent environment</td>
<td>EH reduction properties P0797</td>
</tr>
</tbody>
</table>
Oh, and one more thing
What can I do with a Parallel For Each?

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

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

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

Traditional for each uses only one core, rest of the die is unutilized!
What can I do with a Parallel For Each?

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

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

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

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)
```
What can I do with a Parallel For Each?

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

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

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

Workload is distributed across cores!

(mileage may vary, implementation-specific behaviour)
What can I do with a Parallel For Each?

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

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

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

Workload is distributed on the GPU cores

(mileage may vary, implementation-specific behaviour)
What can I do with a Parallel For Each?

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

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

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

Workload is distributed on all cores!

(mileage may vary, implementation-specific behaviour)
# Demo Results - Running std::sort
(Running on Intel i7 6600 CPU & Intel HD Graphics 520)

<table>
<thead>
<tr>
<th>size</th>
<th>$2^{16}$</th>
<th>$2^{17}$</th>
<th>$2^{18}$</th>
<th>$2^{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::seq</td>
<td>0.27031s</td>
<td>0.620068s</td>
<td>0.669628s</td>
<td>1.48918s</td>
</tr>
<tr>
<td>std::par</td>
<td>0.259486s</td>
<td>0.478032s</td>
<td>0.444422s</td>
<td>1.83599s</td>
</tr>
<tr>
<td>std::unseq</td>
<td>0.24258s</td>
<td>0.413909s</td>
<td>0.456224s</td>
<td>1.01958s</td>
</tr>
<tr>
<td>sycl_execution_policy</td>
<td>0.273724s</td>
<td>0.269804s</td>
<td>0.277747s</td>
<td>0.399634s</td>
</tr>
</tbody>
</table>
SYCL Ecosystem

- ComputeCpp - https://codeplay.com/products/computesuite/computecpp
- triSYCL - https://github.com/triSYCL/triSYCL
- SYCL - http://sycl.tech
- SYCL ParallelSTL - https://github.com/KhronosGroup/SyclParallelSTL
- Eigen http://eigen.tuxfamily.org
Eigen Linear Algebra Library

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

https://bitbucket.org/eigen/eigen/
TensorFlow

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

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

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

Developer Choice

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

Standards bodies

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

Research

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

Open source

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

Presentations

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

Company

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

Codeplay build the software platforms that deliver massive performance
What our ComputeCpp users say about us

Benoit Steiner – Google TensorFlow engineer

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

ONERA

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

Hartmut Kaiser - HPX

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

WIGNER Research Centre for Physics

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

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

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

All of this and more at: http://sycl.tech
Thank you for listening

@codeplaysoft  /codeplaysoft  codeplay.com