Introduction

- I am
  - Lee Howes
  - Senior staff engineer
  - GPU systems team at Qualcomm
  - Co-spec editor for SYCL and OpenCL

- lhowes@qti.qualcomm.com
Introduction

- This tutorial is a joint effort between Qualcomm, AMD and Codeplay

- As a result, the tutorial is accidentally a trilogy in five parts (with thanks to Douglas Adams):

  - Introduction to SYCL
  - Modern C++ and SYCL
  - SYCL and the Parallel-STL
  - triSYCL: Open source SYCL
  - Hands on with Codeplay’s implementation
Introduction

• This tutorial is a joint effort between Qualcomm, AMD and Codeplay

• As a result, the tutorial is accidentally a trilogy in five parts (with thanks to Douglas Adams):

  - Introduction to SYCL
  - Modern C++ and SYCL
  - SYCL and the Parallel-STL
  - triSYCL: Open source SYCL
  - Hands on with Codeplay’s implementation
Background

• Support for C++ has been a common request for OpenCL

• The weak link between host and kernel code is prone to errors
  - People solve using stub generation and stringification scripts
  - Making this interface strongly typed is a better solution

• The C++ standard is continuing to modernize
  - Well-defined memory model
  - Move towards better concurrency and parallelism definitions
What is SYCL™?

- We set out to define a standard for OpenCL that addresses these issues

- SYCL
  - Pronounced SICKLE

- Royalty-free, cross platform C++ programming layer
  - Builds on concepts of portability and efficiency

- A Higher Level Model on top of OpenCL
What is SYCL?

- Single source C++
  - Without language extensions
  - Will build through a standard C++ compiler - though without OpenCL device support

- Single Source Development Cycle
  - C++ source contains functions which can be on both host and device
  - Allows to construct reusable templates for algorithms which use OpenCL for acceleration
  - Type safe interaction between host and device code
The SYCL 1.2 specification

• Here at IWOCL we are launching the SYCL 1.2 specification

• Based on earlier provisional specifications
  - We have incorporated feedback from developers
  - We have streamlined interfaces to better match standard C++ development
  - We have made consistency fixes with a view to quickly incorporating OpenCL 2.1 features

• Specifications are available:
  - https://www.khronos.org/opencl/sycl
Overview

User application code

C++ template libraries

SYCL for OpenCL

OpenCL Devices

CPU

GPU

DSP

FPGA

Other technologies

CPU

Custom Processor

© Copyright Khronos Group 2015 - Page 9
Shared source

- Host code and device code in the same flow
  - Simplifies basic examples

```cpp
#include <CL/sycl.hpp>
#include <iostream>

int main() {
    using namespace cl::sycl;

    int data[1024]; // initialize data to be worked on

    // By including all the SYCL work in a {} block, we ensure all SYCL tasks
    // all SYCL tasks must complete before exiting the block
    {
        // create a queue to enqueue work to queue myQueue;

        // wrap our data variable in a buffer
        buffer<int, 1> resultBuf(data, range<1>(1024));

        // create a command_group to issue commands to the queue
        myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);

            // enqueue a parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
                writeResult[idx] = idx[0];
            }); // end of the kernel function
        }); // end of our commands for this queue
    } // end of scope, so we wait for the queued work to complete

    // print result
    for (int i = 0; i < 1024; i++)
        std::cout << "data[""<<i<<"""] = ""<<data[i]<<std::endl;

    return 0;
}
```
Shared source

- Host code and device code in the same flow
- Simplifies basic examples

```cpp
#include <CL/sycl.hpp>
#include <iostream>

int main() {
    using namespace cl::sycl;

    int data[1024]; // initialize data to be worked on

    // By including all the SYCL work in a {} block, we ensure all SYCL tasks must complete before exiting the block
    {
        // create a queue to enqueue work to queue myQueue;

        // wrap our data variable in a buffer
        buffer<int, 1> resultBuf(data, range<1>(1024));

        // create a command_group to issue commands to the queue
        myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);

            // enqueue a parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
                writeResult[idx] = idx[0];
            }); // end of the kernel function
        }); // end of our commands for this queue
    } // end of scope, so we wait for the queued work to complete

    // print result
    for (int i = 0; i < 1024; i++)
        std::cout << "data[" << i << "] = " << data[i] << std::endl;

    return 0;
}
```
Shared source

- Host code and device code in the same flow
  - Simplifies basic examples

- Device code carries certain restrictions

```cpp
#include <CL/sycl.hpp>
#include <iostream>

int main() {
    using namespace cl::sycl;

    int data[1024]; // initialize data to be worked on

    // By including all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block
    {
        // create a queue to enqueue work to
        // queue myQueue;

        // wrap our data variable in a buffer
        buffer<int, 1> resultBuf(data, range<1>(1024));

        // create a command_group to issue commands to the queue
        myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);

            // enqueue a parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
                writeResult[idx] = idx[0];
            }); // end of the kernel function
        }); // end of our commands for this queue
    } // end of scope, so we wait for the queued work to complete

    // print result
    for (int i = 0; i < 1024; i++)
        std::cout << "data" << i << " = " << data[i] << std::endl;

    return 0;
}
```
Shared source

- Host code and device code in the same flow
  - Simplifies basic examples

- Device code carries certain restrictions

- The big benefit is the tight type system integration
  - `writeResult` here must give access to `int`
  - Type checks correctly against the cast
  - We’ll see this more clearly later

```cpp
#include <CL/sycl.hpp>
#include <iostream>

int main() {
    using namespace cl::sycl;

    int data[1024]; // initialize data to be worked on

    // By including all the SYCL work in a {} block, we ensure all SYCL tasks must complete before exiting the block
    {
        // create a queue to enqueue work to queue myQueue;

        // wrap our data variable in a buffer
        buffer<int, 1> resultBuf(data, range<1>(1024));

        // create a command_group to issue commands to the queue
        myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);

            // enqueue a parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
                writeResult[idx] = idx[0];
            }); // end of the kernel function
        }); // end of our commands for this queue

        // end of scope, so we wait for the queued work to complete

        // print result
        for (int i = 0; i < 1024; i++)
            std::cout << \"data\" << i << \"\" = \" << data[i] << std::endl;
    }

    return 0;
}
Build Process Overview

main.cpp

CPU compiler
(e.g. gcc, llvm, Intel C/C++, Visual C/C++)

CPU object file

SPIR

Binary format?

Executable Object

The SYCL runtime chooses the best binary for the device at runtime

Implementation-defined linking step

Multi-device compilers are not required, but is a possibility
Host fallback

• As it is standard C++ the host compiler path acts as a fallback
  - Exposed to the SYCL host APIs as a true device with or without an OpenCL runtime

• Make use of standard C++ compiler optimisations
int main() {
  using namespace cl::sycl;

  int data[1024];  // initialize data to be worked on

  // By including all the SYCL work in a {} block, we ensure
  // all SYCL tasks must complete before exiting the block
  {
    // create a queue to enqueue work to
    // queue myQueue;

    // wrap our data variable in a buffer
    buffer<int, 1> resultBuf(data, range<1>(1024));

    // create a command_group to issue commands to the queue
    myQueue.submit([&](handler& cgh) {
      // request access to the buffer
      auto writeResult = resultBuf.get_access<access::write>(cgh);

      // enqueue a parallel_for task
      cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
        writeResult[idx] = idx[0];
      });  // end of the kernel function
    });  // end of our commands for this queue

    // end of scope, so we wait for the queued work to complete

    // print result
    for (int i = 0; i < 1024; i++)
      std::cout << "data[" << i << "] = " << data[i] << std::endl;
  }

  return 0;
}
Flexible host code

• Defaults to simplify object construction
  - Default queue here chooses default device, context and platform

• Type safe buffers

• Classes for control where necessary
  - Context, platform, device
  - Selector classes to choose devices
  - Images
  - Programs and Kernels

// create a queue to enqueue work to queue myQueue;

// wrap our data variable in a buffer
buffer<int, 1> resultBuf(data, range<1>(1024));

// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
int main() {
    using namespace cl::sycl;

    int data[1024]; // initialize data to be worked on

    // By including all the SYCL work in a {} block, we ensure
    // all SYCL tasks must complete before exiting the block
    {
        // create a queue to enqueue work to
        queue myQueue;

        // wrap our data variable in a buffer
        buffer<int, 1> resultBuf(data, range<1>(1024));

        // create a command_group to issue commands to the queue
        myQueue.submit([&](handler& cgh) {
            // request access to the buffer
            auto writeResult = resultBuf.get_access<access::write>(cgh);

            // enqueue a parallel_for task
            cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
                writeResult[idx] = idx[0];
            }); // end of the kernel function
        }); // end of our commands for this queue
    } // end of scope, so we wait for the queued work to complete

    // print result
    for (int i = 0; i < 1024; i++)
        std::cout << "data["" << i << "] = " << data[i] << std::endl;

    return 0;
}
Abstracting tasks through handlers

// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024),
        [=](id<1> idx) {
            writeResult[idx] = idx[0];
        }); // end of the kernel function
}); // end of our commands for this queue
Abstracting tasks through handlers

• The handler is designed to match styles developed by the C++ committee

• Traces ordering of parallel operations within the C++ memory model

• Enables clean scope structure for correctly scheduled object destruction

```cpp
// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
  // request access to the buffer
  auto writeResult = resultBuf.get_access<access::write>(cgh);

  // enqueue a parallel_for task
  cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
    writeResult[idx] = idx[0];
  }); // end of the kernel function
}); // end of our commands for this queue
```
Abstracting tasks through handlers

- The handler is designed to match styles developed by the C++ committee

- Traces ordering of parallel operations within the C++ memory model

- Enables clean scope structure for correctly scheduled object destruction

```cpp
// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024),
        [=](id<1> idx) {
            writeResult[idx] = idx[0];
        }); // end of the kernel function
}); // end of our commands for this queue
```
Accessors

- Acquire access to a host buffer via the RAII paradigm

- Access falls out of scope at the end of the queue entry

```cpp
// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024),
        [=](id<1> idx) {
            writeResult[idx] = idx[0];
        }); // end of the kernel function
}); // end of our commands for this queue
```
Accessors

- Acquire access to a host buffer via the RAII paradigm

- Access falls out of scope at the end of the queue entry

```c
// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
        writeResult[idx] = idx[0];
    }); // end of the kernel function
}); // end of our commands for this queue
```
Parallel execution

- OpenCL-style SPMD execution through parallel_for operation
  - Define an iteration space

- Pass a function object/lambda
  - Represents the function executed at each point

```cpp
// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);
    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024),
    [=](id<1> idx) {
        writeResult[idx] = idx[0];
    }); // end of the kernel function
}); // end of our commands for this queue
```
Parallel execution

- OpenCL-style SPMD execution through parallel_for operation
  - Define an iteration space

- Pass a function object/lambda
  - Represents the function executed at each point

- Execution index passed in to each iteration
  - Number of dimensions, style of access well-defined
  - No calling of global built-ins

```cpp
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024), [=](id<1> idx) {
        writeResult[idx] = idx[0];
    }); // end of the kernel function
}); // end of our commands for this queue
```
Naming the kernel

- SYCL does not use language extensions

- SYCL does not require a single source compiler
  - Host code can run through the standard host compiler

- We need to link host and device code together
  - This relies on the type of the kernel body object
  - Lambda types are implementation-defined

- Therefore we have to name them

```cpp
// create a command_group to issue commands to the queue
myQueue.submit([&](handler& cgh) {
    // request access to the buffer
    auto writeResult = resultBuf.get_access<access::write>(cgh);

    // enqueue a parallel_for task
    cgh.parallel_for<class simple_test>(range<1>(1024),
        [=](id<1> idx) {
            writeResult[idx] = idx[0];
        });  // end of the kernel function
});  // end of our commands for this queue
```
Other kernel features

- Vector classes:
  - `vec<T, dims>`

- Subset of iostream functionality
  - Better type safety than `printf`

- Full OpenCL maths library support

- C++-style atomic operations
Variations on dispatch

```cpp
auto command_group = [&](handler & cgh) {
    cgh.single_task<class kernel_name>(
        [=] () {
            // [kernel code]
        });
};

Very simple single instance
Equivalent to clEnqueueTask

Simple map operation over an iteration space
No local memory, no barriers, no information about the range size

```class MyKernel;

```cpp
t auto acc=myBuffer.get_access<read_write>();

 cmdgroup.parallel_for<class MyKernel>(
             range<1>(workItemNo),
             [=] (id<1> index)
             {
                 acc[index] = 42.0f;
             });
```
Variations on dispatch

```cpp
class MyKernel;

myQueue.submit([&](handler &cmdgroup)
{
    auto acc=myBuffer.get_access<read_write>();

    cmdgroup.parallel_for<class MyKernel>(range<1>(workItemNo),
        [=] (item<1> myItem)
        {
            size_t index = item.get_global();
            acc[index] = 42.0f;
        });
};

Adding work-groups
Addition of work-groups allows us to use barriers and communication via local memory

```cpp
auto command_group = [&](handler& cgh) {
    cgh.parallel_for<class example_kernel>(
        nd_range(range(4, 4, 4), range(2, 2, 2)),
        [=](nd_item<3> item) {
            //kernel code
            // Internal synchronization
            item.barrier(access::fence_space::global);
            //kernel code
        });
};
```

Adding information about the range
Item carries more information about the iteration space
Hierarchical parallelism

- Better match the developer’s intent and thought process
  - Reduce the need for a developer to mentally slice the execution into work-items

- Make work-group code explicitly separate from per-work-item code
  - Ease compiler’s job identifying uniform operations and scalar code

- Make barriers implicit
  - Loop fission is no longer necessary to map to the CPU
  - More performance portable

```cpp
auto command_group = [&](handler & cgh) {
  // Issue 8 work-groups of 8 work-items each
cgh.parallel_for_work_group<class example_kernel>(
  range<3>(2, 2, 2), range<3>(2, 2, 2),
  [=](group<3> myGroup) {
    // [workgroup code]
    int myLocal; // this variable shared between workitems
    private_memory<int> myPrivate(myGroup);
    parallel_for_work_item(myGroup, [=](item<3> myItem) {
      // [work-item code]
      myPrivate(myItem) = 0;
    });
    parallel_for_work_item(myGroup, [=](item<3> myItem) {
      // [work-item code]
      output[myGroup.get_local_range() * myGroup.get() + myItem] = myPrivate(myItem);
    });
    // [workgroup code]
  });
};
```
Remainder of the tutorial

- Ronan Keryell - Modern C++ and SYCL

- Ruyman Reyes - SYCL for Parallel STL
  - SYCL is standard C++ and is intended to underpin OpenCL-based implementations of developing C++ concurrency features

- Ronan Keryell - triSYCL: open source SYCL runtime
  - SYCL is standard C++ and triSYCL implements it entirely on the CPU

- Ruyman Reyes and Maria Rovatsou - Hands on with SYCL
  - A hands on tutorial on top of a virtual machine