SYCL™ for OpenCL™

Andrew Richards, CEO Codeplay & Chair SYCL Working group

GDC, March 2014
Where is OpenCL today?

- OpenCL: supported by a very wide range of platforms
  - Huge industry adoption
- Provides a C-based kernel language
- NEW: SPIR provides ability to build other languages on top
- Now, we need to provide languages and libraries
- Topic for today: C++

OpenCL C kernels

Other language kernels

Device X

Device Y

Device Z

OpenCL

© Copyright Khronos Group 2014 - Page 2

OpenCL and the OpenCL logo are trademarks of Apple Inc.
SYCL for OpenCL

• Pronounced ‘sickle’ to go with ‘spear’ (SPIR)
• Royalty-free, cross-platform C++ programming layer
  – Builds on concepts portability & efficiency of OpenCL
  – Ease of use and flexibility of C++
• Single-source C++ development
  – C++ template functions can contain host & device code
  – Construct complex reusable algorithm templates that use OpenCL for acceleration
SYCL Roadmap

• Today
  – Releasing a provisional specification to enable feedback
  – Developers can provide input into standardization process
  – Feedback via Khronos forums

• Next steps
  – Full specification, based on feedback
  – Conformance testsuite to ensure compatibility between implementations
  – Release of implementations
What we want to achieve

- We want to enable a C++ on OpenCL ecosystem
  - With C++ libraries supported on OpenCL
  - C++ tools supported on OpenCL
  - Aim to achieve long-term support for OpenCL features with C++
  - Good performance of C++ software on OpenCL
  - Multiple sources of implementations
  - Enable future innovation
Where can I get SYCL?

Codeplay is working on an implementation

It’s an open, royalty-free standard
Anyone can implement it
Simple example

Does everything* expected of an OpenCL program: compilation, startup, shutdown, host fall-back, queue-based parallelism, efficient data movement.

* (this sample doesn’t catch exceptions)
FEATURES OF SYCL
Default synchronization

- Uses C++ RAII
  - Simple to use
  - Clear, obvious rules
  - Common in C++

```cpp
int my_array [20];
{
    cl::sycl::buffer my_buffer (my_array, 20); // creates the buffer
    // my_array is now taken over by the SYCL system and its contents undefined

    { // access to my_buffer is now free to other threads/queues
        auto my_access = my_buffer.get_access<cl::sycl::access::read_write,
            cl::sycl::access::host_buffer> ();

        /* The host now has access to the buffer via my_access.
           This is a synchronizing operation - it blocks until access is ready.
           Access is released when my_access is destroyed
           */
    }

    /* my_buffer is destroyed. Waits for all threads/queues to complete work on
       my_buffer. Then writes any modified contents of my_buffer back to
       my_array, if necessary.
       */
} // creates the buffer
```

© Copyright Khronos Group 2014 · Page 9
Task Graphs

SYCL separates data storage (buffers) from data access (accessors). This allows easy, safe, efficient scheduling.

```cpp
buffer<int> my_buffer(data, 10);
command_group(myqueue, [&]() {
    auto in_access = my_buffer.access<cl::sycl::access:read_only>();
    auto out_access = my_buffer.access<cl::sycl::access:write_only>();
    parallel_for_workgroup(nd_range(range(size), range(groupsize)),
        lambda<class hierarchical>([=](group_id group) {
            parallel_for_workitem(group, [=](tile_id tile) {
                out_access[tile.global()] = in_access[tile.global()] * 2;
            });
        });
});
```
Hierarchical Data Parallelism

```cpp
buffer<int> my_buffer(data, 10);

command_group(my_queue, [&](){
    auto in_access = my_buffer.access<cl::sycl::access:read_only>();
    auto out_access = my_buffer.access<cl::sycl::access:write_only>();

    parallel_for_workgroup(nd_range(range(size), range(groupsize)),
        lambda<class hierarchical>([=](group_id group)
            {
                parallel_for_workitem(group, [=](tile_id tile)
                    {
                        out_access[tile] = in_access[tile] * 2;
                    });
            });
});
```

Advantages:
1. Easy to understand the concept of work-groups
2. Performance-portable between CPU and GPU
3. No need to think about barriers (automatically deduced)
4. Easier to compose components & algorithms
   - e.g. Kernel fusion
Single source

• Developers want to write templates, like:
  \[
  \text{parallel\_sort}\langle\text{MyClass}\rangle\ (\text{myData});
  \]

• This requires a single template function that includes both host and device code
  – The host code ensures the right data is in the right place
  – Type-checking (and maybe conversions) required
Choose your own host compiler

• Why?
  – Developers use a lot of CPU-compiler-specific features (OS integration, for example) - *SYCL supports this*
  – The kind of developer that wants to accelerate code with OpenCL will often use CPU-specific optimizations, intrinsic functions etc. - *SYCL supports this*
  – For example, a developer will think it reasonable to accelerate CPU code with OpenMP and GPU code with OpenCL, but want to share source between the 2. - *SYCL supports this*

• OpenCL C supports this approach, but without single source
  – SYCL additionally allows single source
Choose your own host compiler

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

int main()
{
    int result; // this is where we will write our result
    {
        // by sticking all the SYCL work in a {} block, we ensure
        // all SYCL tasks must complete before exiting the block

        // create a queue to work on
        cl::sycl::queue myQueue;

        // wrap our result variable in a buffer
        cl::sycl::buffer<int> resultBuf(&result, 1);

        // create some 'commands' for our 'queue'
        cl::sycl::command_group(myQueue, [&](){
            // request access to our buffer
            auto writeResult = resultBuf.access<cl::sycl::access:write_only>();

            // enqueue a single, simple task
            cl::sycl::single_task(kernel_lambda<class simple_test>([]()
            {
                writeResult[0] = 1234;
            })); // end of our commands for this queue
        }); // end of our commands for this queue

    } // end scope, so we wait for the queue to complete

    printf("Result = %d\n", result);
}
```
Support multiple device compilers

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

int main()
{
    int result; // this is where we will write our result
    {
        // by sticking all the SYCL work in a {} block, we ensure
        // all SYCL tasks must complete before exiting the block
        // create a queue to work on
        cl::sycl::queue myQueue;
        // wrap our result variable in a buffer
        cl::sycl::buffer<int> resultBuf(&result, 1);
        // create some ‘commands’ for our ‘queue’
        cl::sycl::command_group(myQueue, [&]() {
            // request access to our buffer
            auto writeResult = resultBuf.access<cl::sycl::access:write_only>();
            // enqueue a single, simple task
            cl::sycl::single_task(kernel_lambda<class simple_test>([]()
            {
                writeResult[0] = 1234;
            })); // end of our commands for this queue
        }); // end scope, so we wait for the queue to complete
        printf("Result = %d\n", result);
    }
}
```

Support multiple device compilers

- Support multiple device compilers
- The SYCL runtime chooses the best binary for the device at runtime

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

**CPU object file**

**SYCL device compiler**

**SYCL device compiler**

**Binary format?**

Multi-device compilers is not required, but it is a possibility
Can use a common library

• Can use #ifdefs to implement common libraries differently on different compilers/devices
  – e.g. defining domain-specific maths functions that use OpenCL C features on device and CPU-specific intrinsics on host
  – Or, define your own `parallel_for` templates that use (for example) OpenMP on host and OpenCL on device
  – The C++ code that calls the library function is shared across platforms, but the library is compiled differently depending on host/device
Asynchronous operation

- A `command_group` is:
  - an atomic operation that includes all memory object creation, copying, mapping, and execution of kernels
  - enqueued, non-blocking
  - scheduling dependencies tracked automatically
  - thread-safe
- Only blocks on return of data to host
Low-latency error-handling

- We use exception-handling to catch errors
- We use the standard C++ RAII approach
  - However, some developers require destructors to return immediately, even on error.
  - But the error-causing code may be asynchronously running. So such a developer needs to leave code running and resources released later. We support with 'storage objects'
Relationship to core OpenCL

• Built on top of OpenCL
• Runs kernels on OpenCL devices

• Can construct SYCL objects from OpenCL objects and OpenCL objects obtained from SYCL objects
OpenCL/OpenGL interop

• Built directly on top of OpenCL interop extension
  – SYCL uses the same structures, macros, enums etc
• Lets developers share OpenGL images/textures etc with SYCL as well as OpenCL
• Only runs on OpenCL devices that support one of the CL/GL interop extensions
• Users can query a device for extension support first
The specification

An overview of the specification itself
The specification itself

• http://www.khronos.org/opencl/sycl
Structure

(Similar to OpenCL structure)

– Section 1: Introduction
– Section 2: SYCL Architecture
  • Very similar to OpenCL architecture
– Section 3: SYCL Programming Interface
  • This is the C++ interface that works across host and device
– Section 4: Support of non-core OpenCL features
– Section 5: Device compiler
  • This is the C++ compiler that compiles kernels
– Appendix A: Glossary
Architecture 1

• SYCL has *queues* and *command-groups*
  – Queues are identical to OpenCL C
  – Command-groups enqueue multiple OpenCL commands to handle data movement, access, synchronization etc

• SYCL has *buffers* and *images*
  – Built on top of OpenCL buffers and images, but abstracts away the different queues, devices, platforms maps, copying etc.
  – Can create SYCL buffers/images from OpenCL buffers/images, or obtain OpenCL buffers/images from SYCL buffers/images (but need to specify context).
Architecture 2

• In SYCL, access to data is defined by *accessors*
  – Users constructs within command-groups: used to define types of access and create data movement and synchronization commands

• Error handling
  – Synchronous errors handled by C++ exceptions
  – Asynchronous errors handled via user-supplied error-handler based on C++14 proposal [n3711]
Architecture: kernels

- Kernels can be:
  - Single task: A non-data-parallel task
  - Basic data parallel: NDRange with no workgroup
  - Work-group data parallel: NDRange with workgroup
  - Hierarchical data parallel: compiler feature to express workgroups in more template-friendly form

- Restrictions on language features \textit{in kernels}, no:
  - function pointers, virtual methods, exceptions, RTTI ...

- Vector classes can work efficiently on host & device
- OpenCL C kernel library available in kernels
Architecture: advanced features

• Storage objects
  – used to define complex ownership/synchronization
• All OpenCL C features supported in kernels
  – (but maybe in a namespace)
  – Including swizzles
• All host compiler features supported in host code
• Wrappers for: programs, kernels, samplers, events
  – Allows linking OpenCL C kernels with SYCL kernels
SYCL Programming Interface

• Defined as a C++ templated class library
• Some classes host-only, some (e.g. accessors, vectors) host-and-device
• Only uses standard C++ features, so code written to this library sharable between host and device.
• Classes have methods to construct from OpenCL C objects and obtain OpenCL C objects wherever possible
• Events, buffers, images work across multiple devices and contexts
SYCL Extensions

- Defines how OpenCL device extensions (e.g. CL/GL interop) are available within SYCL
- Availability is based on device support
- Host can also support extensions
- Queries are provided to allow users to query for device and host support for extensions
- OpenCL extensions not in the SYCL spec are still usable within SYCL due to deep OpenCL integration and interop
SYCL Device Compiler

• Defines the language features available in kernels
• Supports restricted standard C++11 feature-set
  – Restricted by capabilities of OpenCL 1.2 devices
  – Would be enhanced for OpenCL 2.0 in the future
• Defines how OpenCL kernel language features are available within SYCL
  – Users using OpenCL kernel language features need to ensure their code is compilable for host. May need #ifdef
What now?

• We are releasing this *provisional* specification to get feedback from developers
  – So please give feedback! Khronos forums are best place
  – [http://www.khronos.org/opencl/sycl](http://www.khronos.org/opencl/sycl)

• Next steps
  – Full specification, based on feedback
  – Conformance test suite to ensure compatibility between implementations
  – Release of implementations