SYCL for OpenCL - Single-source C++

• 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
    - e.g. parallel_sort<MyType> (myData);
  - Construct complex reusable algorithm templates using OpenCL for acceleration

• SYCL 1.2 Final spec released!
  - At IWOCL in May 2014

• Multiple implementations
  - Including open source triSYCL from AMD
  - https://github.com/amd/triSYCL
OpenCL / SYCL Stack

User application code

C++ template libraries

SYCL for OpenCL

OpenCL Devices

CPU

GPU

DSP

FPGA

Other technologies

CPU

Custom Processor
Example SYCL Code #1

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

int main ()
{
    …
    // Device buffers
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        // Data accessors
        auto a = buf_a.get_access<access::read>(cgh);
        auto b = buf_b.get_access<access::read>(cgh);
        auto c = buf_c.get_access<access::read>(cgh);
        auto r = buf_r.get_access<access::write>(cgh);
        // Kernel
        cgh.parallel_for<class three_way_add>(count, [=](id<> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
    …
}``
```cpp
#include <CL/sycl.hpp>

int main ()
{
    ...
    // Device buffers
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
    queue myQueue;
    myQueue.submit([&](handler& cgh)
    {
        // Data accessors
        auto a = buf_a.get_access<access::read>(cgh);
        auto b = buf_b.get_access<access::read>(cgh);
        auto c = buf_c.get_access<access::read>(cgh);
        auto r = buf_r.get_access<access::write>(cgh);
        // Kernel
        cgh.parallel_for<class three_way_add>(count, [=](id<> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
    ...
}
```

Some language restrictions within kernels
Build Process Overview

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

int main ()
{
    // Device buffers
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));

    queue myQueue;
    command_group(myQueue, [&]()
    {
        // Data accessors
        auto a = buf_a.get_access<access::read>();
        auto b = buf_b.get_access<access::read>();
        auto c = buf_c.get_access<access::read>();
        auto r = buf_r.get_access<access::write>();

        // Kernel
        parallel_for<class three_way_add>(count, [&](id<> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
}
```

---

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

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**
- **SPIR**
- **Binary format?**
- **Executable Object**

Implementation-defined linking step

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

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

CPU object file

SPIR

Binary format?
Major talking points

- SYCL uses standard C++ with no language extensions
- Tasks, such as memory object creations, mapping, copies and kernel execution, are scheduled automatically
  - Using SYCL allows for dependencies to be tracked automatically
  - Specifying data access rules reduces overheads, allows for efficient scheduling
- Hierarchical Parallelism
const int n_items = 32;
range<1> r(n_items);
int array_a[n_items] = { 0 };
int array_b[n_items] = { 0 };
buffer<int, 1> buf_a(array_a, range<1>(r));
buffer<int, 1> buf_b(array_b, range<1>(r));

queue q;
command_group(q, [&](){
    auto acc_a = buf_a.get_access<read_write>();
    algorithm_a s(acc_a);
    parallel_for(n_items, s);
});

command_group(q, [&](){
    auto acc_b = buf_b.get_access<read_write>();
    algorithm_b s(acc_b);
    parallel_for(n_items, s);
});

command_group(q, [&](){
    auto acc_a = buf_a.get_access<read_write>();
    algorithm_c s(acc_a);
    parallel_for(n_items, s);
});
Hierarchical Data Parallelism

```cpp
buffer<int> my_buffer(data, 10);
auto in_access = my_buffer.access<cl::sycl::access:read_only>();
auto out_access = my_buffer.access<cl::sycl::access:write_only>();

command_group(my_queue, [&](){
    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
What Does This Mean for Developers?

• Enables a standard C++11 codebase targeting multiple OpenCL devices
  - As it’s C++, a host CPU device implementation can be provided in headers

• SYCL is cross-toolchain as well as cross-platform
  - No language extensions, standard C++ compilers can build SYCL source code

• Device compilers enable SYCL running on OpenCL devices
  - Can have multiple device compilers linking into final executable
  - Doesn’t affect original source build

• You could implement SYCL simply using C++ threads
  - All the synchronization, parallelism wins remain - but running on CPU
  - No external dependencies
What Does This Mean for Developers?

- Enables developers to move quickly into writing SYCL code
  - Provides methods for dealing with targets that do not have OpenCL(yet!)
  - Has other development benefits...

- A fallback CPU implementation is debuggable!
  - Using normal C++ debuggers
  - Profiling tools also work on CPU device

- Huge bonus for productivity and adoption
  - Cost of entry to use SYCL very low
OpenCL Features within SYCL

- Can access OpenCL objects from SYCL objects
- Can construct SYCL objects from OpenCL object
- Interop with OpenGL remains in SYCL
  - Uses the same structures/types
- Developers still have the ability to optimize at a low level should they need to
In Summary

• SYCL: a royalty-free, cross platform C++ programming layer
  - Very low cost to entry

• An OpenCL ecosystem based on standard C++
  - Single source development without language extensions
  - C++ developers can easily utilize OpenCL in code

• Final SYCL 1.2 available online
  - https://www.khronos.org/opencl/sycl
  - Together with conformance tests!

• Please use the SYCL forum thread for feedback!
  - http://www.khronos.org/opencl/sycl