SYCL 2020; more than meets the eye

Gordon Brown – Principal Software Engineer, SYCL & C++

SYCLCON2020 - April 2020
• The story so far...
  • SYCL emerged in 2015 with the ratification of SYCL 1.2
  • Early implementations ComputeCpp and triSYCL allowed people to try it out
  • After a couple of years of feedback came a major update in SYCL 1.2.1
  • The following year ComputeCpp became the first conformant implementation
  • Over the next two years came hipSYCL and Intel’s DPC++
  • Various extensions were proposed to improve SYCL
  • DPC++ was extended to support CUDA
• Today...
  • We are pleased to be able to share a preview of the next revision of SYCL
Backend generalization
Unified shared memory
Group algorithms

Modules
In-order queues
Host tasks

Specialization constants
Sub-groups
Improved address space inference

Indicative only, still subject to change!
Backend generalization
Unified shared memory
Group algorithms

Modules
In-order queues
Host tasks

Specialization constants
Sub-groups
Improved address space inference

Indicative only, still subject to change!
Backend generalization
• SYCL 1.2.1 is based entirely on OpenCL 1.2
  • Supports OpenCL 1.2 devices

• SYCL 2020 will be generalized to support additional backends
  • SYCL runtime will be able to support devices from other backends
  • Execution and memory model will remain the same
  • API will be largely unaffected

• Backends
  • OpenCL 1.2 will remain a core backend in SYCL 2020
  • Implementations will be able to support later revisions of OpenCL
  • Implementations can support other backends as well
What the SYCL Ecosystem looks like now

- A single SYCL runtime can target multiple backends
- The rest of the stack remains the same
• What the SYCL Ecosystem looks like now
  • A single SYCL runtime can target multiple backends
  • The rest of the stack remains the same

Every SYCL impl must have a host backend
• What the SYCL Ecosystem looks like now
  • A single SYCL runtime can target multiple backends
  • The rest of the stack remains the same

Every SYCL impl
must have at least
one non-host
backend
• The SYCL topology has been extended to include backends
  • Device selectors can be used to find devices from any backend
• The SYCL topology has been extended to include backends
  • Device selectors can be used to find devices from any backend
The SYCL topology has been extended to include backends

- Device selectors can be used to find devices from any backend

The same device can be in two backends
Customizability and portability

Generic SYCL
- Standard SYCL programming model
- Portable across all SYCL implementations

Interoperability
- Interoperability with the native objects of the backend programming models
- Portable across implementations which support the same backend

Backend/vendor-specific
- Backend or vendor specific extensions
- Non-portable, but can be used conditionally
Customizability and portability

**Generic SYCL**
- Standard SYCL programming model
- Portable across all SYCL implementations

**Interoperability**
- Interoperability with the native objects of the backend programming models
- Portable across implementations which support the same backend

**Backend/vendor-specific**
- Backend or vendor specific extensions
- Non-portable, but can be used conditionally
Customizability and portability

**Generic SYCL**
- Standard SYCL programming model
- Portable across all SYCL implementations

**Interoperability**
- Interoperability with the native objects of the backend programming models
- Portable across implementations which support the same backend

**Backend/vendor-specific**
- Backend or vendor specific extensions
- Non-portable, but can be used conditionally
using namespace sycl;

void algorithm(queue q, buffer in, buffer out) {

    // For OpenCL backend
    #ifdef SYCL_BACKEND_OPENCL
        auto cntx = q.get_context();
        auto clQueue = get_native<backend::opencl>(q, cntx);
        auto clEvent = opencl_algorithm(clQueue, in, out);
        make_event<backend::opencl>(clEvent, cntx).wait();
    #endif

    // For generic SYCL
    #else
        generic_sycl_algorithm(q, in, out).wait();
    #endif
}
Example of backend interoperability

The free functions `get_native` can be used to retrieve native backend objects from a SYCL object. You must also provide the `context` you want to interoperate on for some of these.

```cpp
using namespace sycl;

void algorithm(queue q, buffer in, buffer out) {
  // For OpenCL backend
  #ifdef SYCL_BACKEND_OPENCL
    auto cntx = q.get_context();
    auto clQueue = get_native<backend::opencl>(q, cntx);
    auto clEvent = opencl_algorithm(clQueue, in, out);
    make_event<backend::opencl>(clEvent, cntx).wait();
  // For generic SYCL
  #else
    generic_sycl_algorithm(q, in, out).wait();
  #endif
}
```

*Indicative only, still subject to change!*
Example of backend interoperability

```
using namespace sycl;

void algorithm(queue q, buffer in, buffer out) {
    // For OpenCL backend
    #ifdef SYCL_BACKEND_OPENCL
        auto cntx = q.get_context();
        auto clQueue = get_native<backend::opencl>(q, cntx);
        auto clEvent = opencl_algorithm(clQueue, in, out);
        make_event<backend::opencl>(clEvent, cntx).wait();
    #else
        generic_sycl_algorithm(q, in, out).wait();
    #endif
}
```

Indicative only, still subject to change!

The native objects can be passes to a native algorithm

In this case say it returns native event to synchronize with
Example of backend interoperability

The free functions `make_<sycl_type>` can be used to construct SYCL objects from a native backend object.

You also have to provide the `context` you want to interoperate on to these functions.

```cpp
using namespace sycl;

void algorithm(queue q, buffer in, buffer out) {
    // For OpenCL backend
    #ifdef SYCL_BACKEND_OPENCL
        auto cntx = q.get_context();
        auto clQueue = get_native<backend::opencl>(q, cntx);
        auto clEvent = opencl_algorithm(clQueue, in, out);
        make_event<backend::opencl>(clEvent, cntx).wait();
    #else
        generic_sycl_algorithm(q, in, out).wait();
    #endif
}
```

Indicative only, still subject to change!

The free functions `make_<sycl_type>` can be used to construct SYCL objects from a native backend object.

You also have to provide the `context` you want to interoperate on to these functions.
Modules
• The program class in SYCL 1.2.1 has a number of issues
  • The API was tied closely to OpenCL
  • The compile/link/build APIs are not thread-safe
  • There is no representation of different file formats
  • There is no association with the kernels in the translation unit

• So it was decided to redesign the API for SYCL 2020
  • The result of this redesign is SYCL modules
So what is a SYCL module?

- Represents all kernels in a translation unit
- Contains one or more device images
- Each device image must contain the symbols for each kernel
• **Input module**
  - Source such as OpenCL C, SPIR-V or some other IR

• **Object module**
  - Compiled binary in any file format, can be linked with other object modules

• **Executable module**
  - Compiled and linked binary in any format, can be executed
• APIs such as `parallel_for` still work as before
  • Modules will be compiled implicitly if not explicitly created
• Module state and transformations have changed
  • Modules have their state in their type
  • Compile, link and build operations are now thread-safe free functions
• There is now a module associated with the translation unit
  • You can retrieve this module by calling `this_module::get`
• There is now more control over selection of the file format
  • The module to be used for a command group is chosen by calling `use_module`
  • This can be passed a device image selector for manually selection
Example of using modules

Indicative only, still subject to change!

A module for the current translation unit can be retrieved by calling `this_module::get`

You must provide the `module_state` that you want and the `context` you want the module for
Example of using modules

Modules can be compiled, linked and built using free functions such as `build`

These free functions can optionally take properties to customize the compilation and linking

using namespace sycl;
class foo;
void vector_add(queue q, buffer a, buffer b, buffer o) {
    auto cntx = q.get_context();
    auto inputModule = this_module::get<module_state::input>(cntx);
    auto execModule = build(inputModule);
    auto kernelFooName = this_module::kernel_name_v<foo>;
    auto kernelFoo = execModule.get_kernel(kernelFooName);
    q.submit([&](handler &cgh){
        cgh.use_module(execModule);
        cgh.parallel_for<foo>(range<1>(1024), [=](id<1> idx){
           /* kernel code */
        }, kernelFoo);
    });
}
Example of using modules

```cpp
using namespace sycl;

class foo;

void vector_add(queue q, buffer a, buffer b, buffer o) {
    auto cntx = q.get_context();

    auto inputModule = this_module::get<module_state::input>(cntx);
    auto execModule = build(inputModule);

    auto kernelFooName = this_module::kernel_name_v<foo>;
    auto kernelFoo = execModule.get_kernel(kernelFooName);

    q.submit([&](handler &cgh) {
        cgh.use_module(execModule);

        cgh.parallel_for<foo>(range<1>(1024), [=](id<1> idx) {
            /* kernel code */
            kernelFoo);
    });
}
```

A kernel can be retrieved from the module of `module_state::executable` by calling `get_kernel`

get_kernel only takes kernel names by string so you can retrieve the string form of a kernel name using `this_module::kernel_name_v`
Example of using modules

using namespace sycl;
class foo;
void vector_add(queue q, buffer a, buffer b, buffer o) {
  auto cntx = q.get_context();
  auto inputModule = this_module::get<module_state::input>(cntx);
  auto execModule = build(inputModule);
  auto kernelFooName = this_module::kernel_name_v<foo>;
  auto kernelFoo = execModule.get_kernel(kernelFooName);
  q.submit([&](handler &cgh) {
    cgh.use_module(execModule);
    cgh.parallel_for<foo>(range<1>(1024), [=](id<1> idx) {
      /* kernel code */
    }, kernelFoo);
  });
}

Indicative only, still subject to change!

The kernel can then be passed to kernel invocation functions such as `parallel_for`.

The module containing the kernel must be selected by calling the `use_module` member function of the handler.

`use_module` can optionally take a device image selector for choosing between multiple device images.
Host tasks
• SYCL 1.2.1 is missing some useful work scheduling features
  • There is no way to asynchronously schedule arbitrary C++ code
  • There is no way to asynchronously interoperate with SYCL memory objects

• So it was decided to introduce the host task to SYCL 2020
  • Host tasks have been around as an extension for a while
  • A redesign of the feature for SYCL 2020 now includes interoperability
Example of using a host task

```cpp
using namespace sycl;

queue gpuQueue(gpu_selector_v, async_handler{});

gpuQueue.submit([&](handler &cgh){
    auto inAcc = outBuf.get_access< access::mode::write, access::target::host_buffer>(cgh);
    auto outAcc = inBuf.get_access<access::mode::write, access::target::host_buffer>(cgh);

    cgh.host_task([&](){
        /* arbitrary C++ code */
    });
});

gpuQueue.wait_and_throw();
```

A **host_task** can execute arbitrary C++ code so is not limited by kernel function restrictions.

A **host_task** is scheduled following the same memory model rules as kernels.
Example of using a host task

```
using namespace sycl;
queue gpuQueue(gpu_selector_v, sync_handler{});
gpuQueue.submit([&](handler &cgh){
    auto inAcc = outBuf.get_access<access::mode::write,
                      access::target::host_buffer>(cgh);
    auto outAcc = inBuf.get_access<access::mode::write,
                      access::target::host_buffer>(cgh);
    cgh.host_task([=](){
        /* arbitrary C++ code */
    });
});
gpuQueue.wait_and_throw();
```

Indicative only, still subject to change!

Accessors created with `access::mode::host_buffer` create a dependency for the data to be available to the host task on the host.

These accessors can be accessed directly in the host task function.
Example of using a host task for interop

```
using namespace sycl;

queue gpuQueue(gpu_selector{}, async_handler{});

gpuQueue.submit([&](handler &cgh){
    auto inAcc = outBuf.get_access<access::mode::write,
                access::target::global_buffer>(cgh);
    auto outAcc = inBuf.get_access<access::mode::write,
                access::target::global_buffer>(cgh);

cgh.host_task([=] (interop_handle kh){
    auto clQueue = kh.get_native_queue<backend::opencl>();
    auto clIn = kh.get_native_mem<backend::opencl>(inAcc);
    auto clOut = kh.get_native_mem<backend::opencl>(outAcc);

    /* interop code */
});
});

gpuQueue.wait_and_throw();
```

Indicative only, still subject to change!

A **host_task** can also be used for interoperability by taking an optional **interop_handle** parameter.
Example of using a host task for interop

```cpp
using namespace sycl;

queue gpuQueue(gpu_selector{}, async_handler{});

gpuQueue.submit([&](handler &cgh){
    auto inAcc = outBuf.get_access< access::mode::write,
    access::target::global_buffer>(cgh);
    auto outAcc = inBuf.get_access<access::mode::write,
    access::target::global_buffer>(cgh);

cgh.host_task([&](interop_handle kh){
    auto clQueue = kh.get_native_queue<backend::opencl>();
    auto clIn = kh.get_native_mem<backend::opencl>(inAcc);
    auto clOut = kh.get_native_mem<backend::opencl>(outAcc);

    /* interop code */

});
});

gpuQueue.wait_and_throw();
```

Indicative only, still subject to change!

Accessors created with access::mode::global_buffer create a dependency for the data to be available to the queue’s device.

These accessors cannot be accessed directly inside the host task.

Instead they can be used to retrieve their native objects.
using namespace sycl;

queue gpuQueue(gpu_selector{}, async_handler{});

gpuQueue.submit([&](handler &cgh){
    auto inAcc = outBuf.get_access<access::mode::write,
               access::target::global_buffer>(cgh);
    auto outAcc = inBuf.get_access<access::mode::write,
                      access::target::global_buffer>(cgh);

    cgh.host_task([=](interop_handle kh){
        auto clQueue = kh.get_native_queue<backend::opencl>();
        auto clIn = kh.get_native_mem<backend::opencl>(inAcc);
        auto clOut = kh.get_native_mem<backend::opencl>(outAcc);

        /* interop code */
    });
});

gpuQueue.wait_and_throw();

**Indicative only, still subject to change!**

You can do this using the `get_native_`* member functions of the `interop_handle` class

You must specify the **backend** and in the case of memory objects provide the accessor and they return the native objects
SYCL 2020 will be a significant improvement to the standard with many great new features.

The SYCL working group are aiming to ratify a provisional specification summer 2020.

We look forward to people’s feedback when it comes.
Thank you!

gordon@codeplay.com

@codeplaysoft  info@codeplay.com  codeplay.com