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

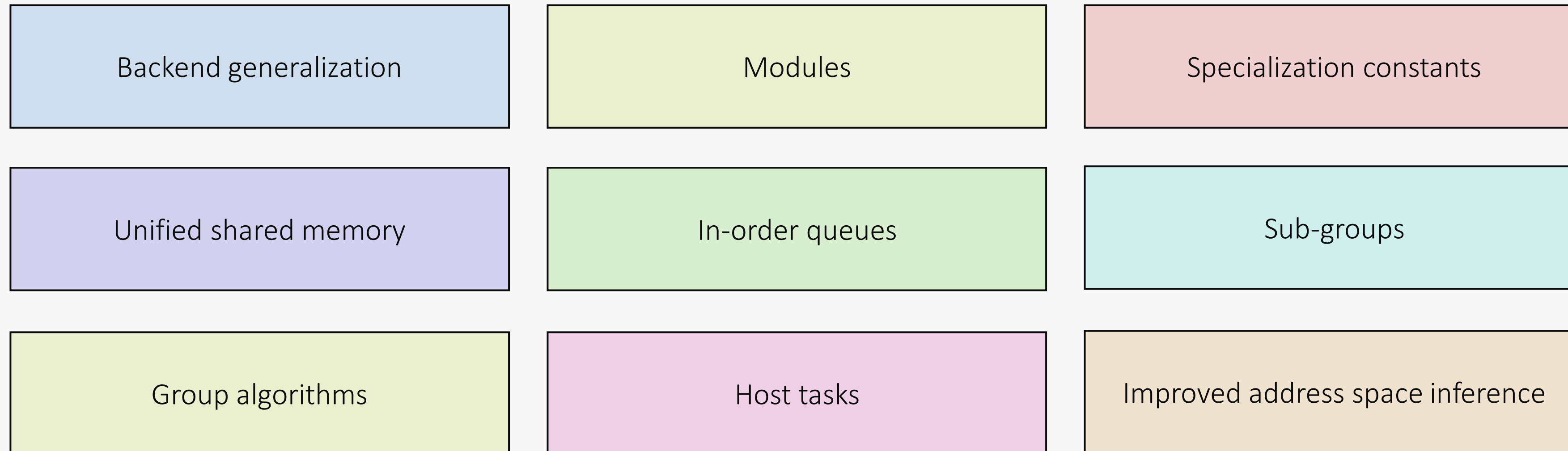
© 2020 Codeplay Software Ltd.

- Today…
  - We are pleased to be able to share a preview of the next revision of SYCL

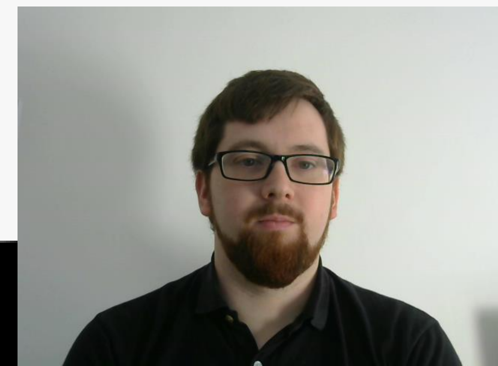
© 2020 Codeplay Software Ltd.

UNDER CONSTRUCTION

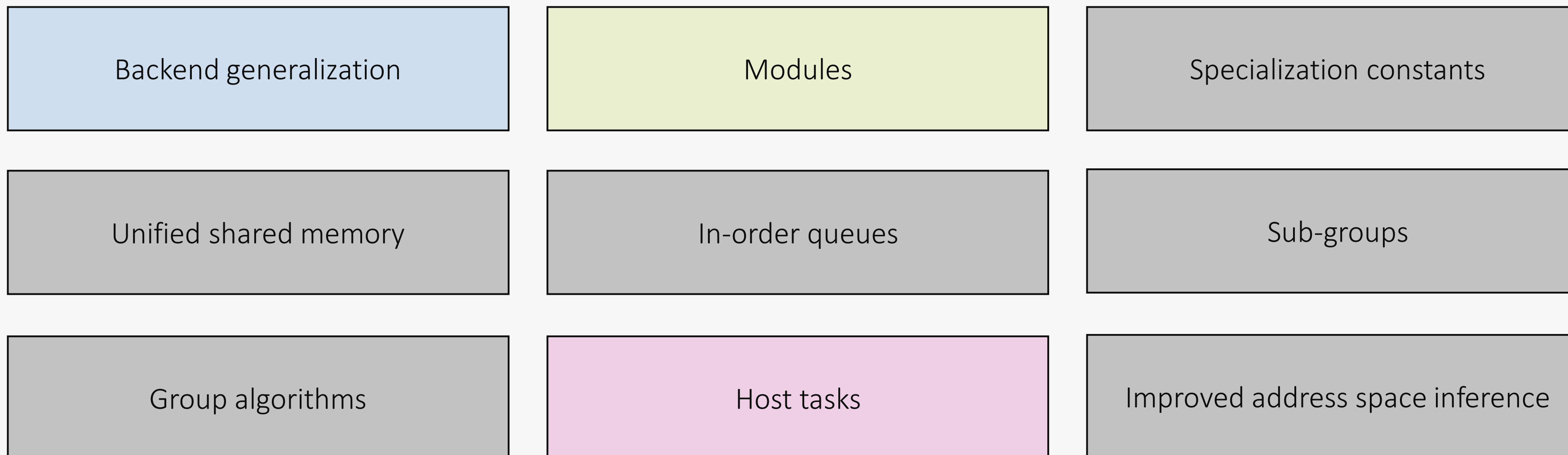
© 2020 Codeplay Software Ltd.



Indicative only, still subject to change!



© 2020 Codeplay Software Ltd.



Indicative only, still subject to change!

© 2020 Codeplay Software Ltd.

Backend generalization

© 2020 Codeplay Software Ltd.

- 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

© 2020 Codeplay Software Ltd.

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

SYCL applications

SYCL 2020 API

SYCL runtime

Host	OpenCL	Vulkan	NVIDIA CUDA

codeplay
© 2020 Codeplay Software Ltd.

- 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

SYCL applications

SYCL 2020 API

SYCL runtime

Host	OpenCL	Vulkan	NVIDIA CUDA

© 2020 Codeplay Software Ltd.

- 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

SYCL applications

SYCL 2020 API

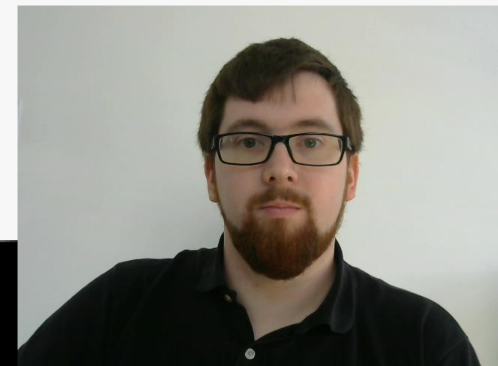
SYCL runtime

Host

OpenCL

Vulkan

NVIDIA CUDA



© 2020 Codeplay Software Ltd.

- The SYCL topology has been extended to include backends
  - Device selectors can be used to find devices from any backend

© 2020 Codeplay Software Ltd.

- The SYCL topology has been extended to include backends
  - Device selectors can be used to find devices from any backend

The host device exists in the host backend

System

Backend: OpenCL, CUDA, Host

Platform: Intel, Nvidia, Nvidia, Host

Device: CPU, GPU, GPU, GPU, Host

© 2020 Codeplay Software Ltd.

- The SYCL topology has been extended to include backends
  - Device selectors can be used to find devices from any backend

	System			
Backend	OpenCL		CUDA	Host
Platform	Intel	Nvidia	Nvidia	Host
Device	CPU, GPU	GPU	GPU	Host

The same device can be in two backends

© 2020 Codeplay Software Ltd.

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

© 2020 Codeplay Software Ltd.

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

© 2020 Codeplay Software Ltd.

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

© 2020 Codeplay Software Ltd.

Example of backend interoperability

Indicative only, still subject to change!

```
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
}
```

SYCL applications can be specialized for specific backends

This is considered non-generic SYCL code so it's recommended to always use the backend macros to guard the code and have fallback

© 2020 Codeplay Software Ltd.

Example of backend interoperability

Indicative only, still subject to change!

```
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
}
```

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

© 2020 Codeplay Software Ltd.

Example of backend interoperability

Indicative only, still subject to change!

```
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
}
```

The native objects can be passes to a native algorithm

In this case say it returns native event to synchronize with

© 2020 Codeplay Software Ltd.

Example of backend interoperability

Indicative only, still subject to change!

```
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
}
```

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

© 2020 Codeplay Software Ltd.

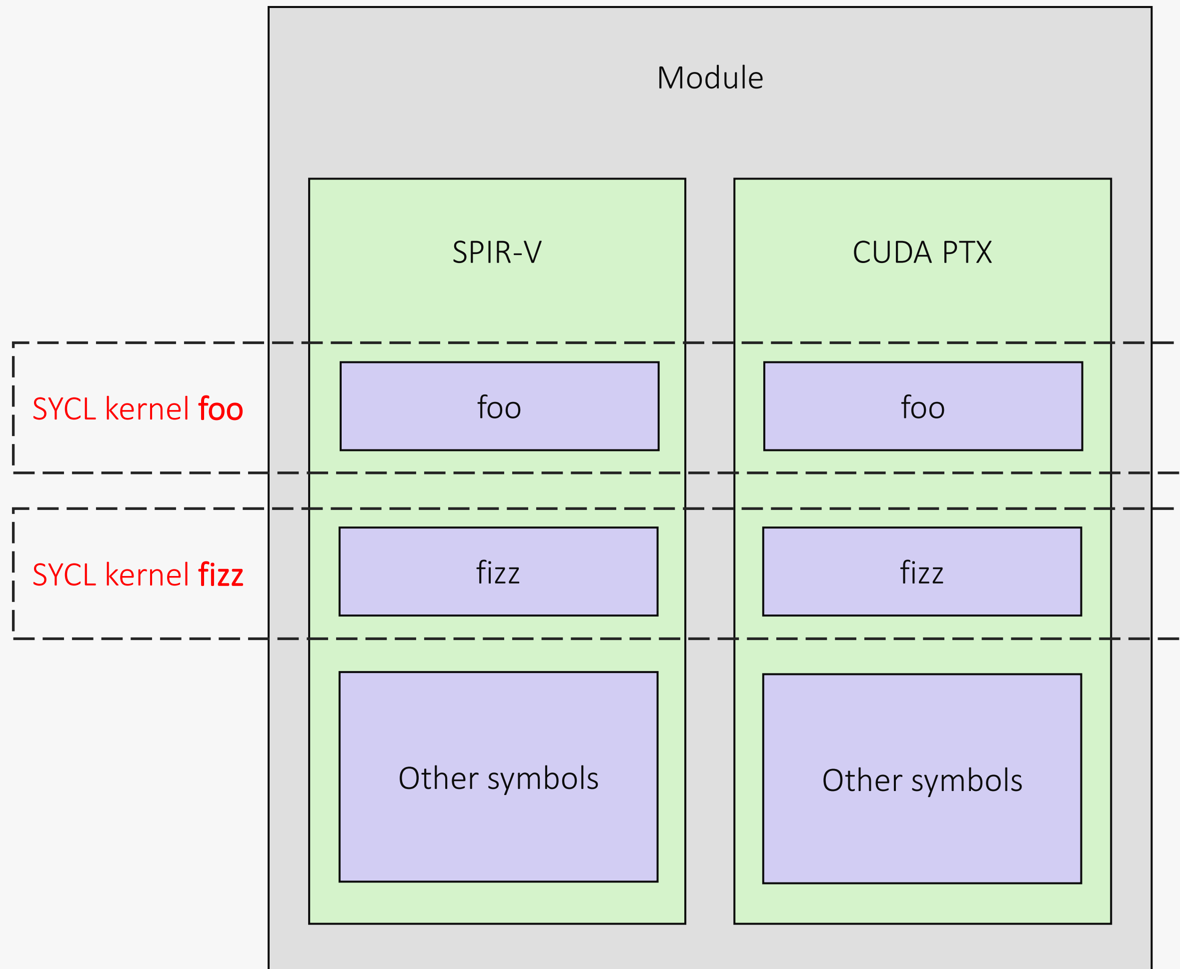
Modules

© 2020 Codeplay Software Ltd.

- 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

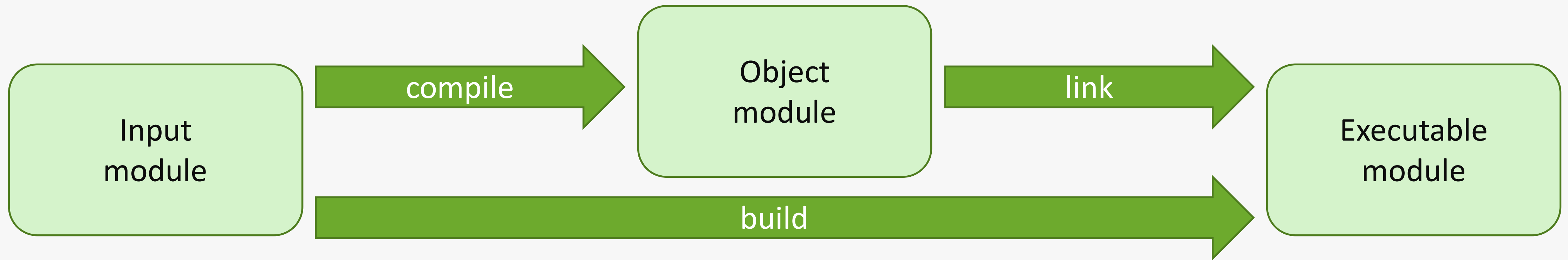
- So it was decided to redesign the API for SYCL 2020
  - The result of this redesign is SYCL modules

© 2020 Codeplay Software Ltd.



- 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

© 2020 Codeplay Software Ltd.



- 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

© 2020 Codeplay Software Ltd.

- 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

© 2020 Codeplay Software Ltd.

Example of using modules

Indicative only, still subject to change!

```
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 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

© 2020 Codeplay Software Ltd.

Example of using modules

Indicative only, still subject to change!

```
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);
  });
}
```

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

© 2020 Codeplay Software Ltd.

Example of using modules

Indicative only, still subject to change!

```
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

© 2020 Codeplay Software Ltd.

Example of using modules

Indicative only, still subject to change!

```
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);
  });
}
```

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

© 2020 Codeplay Software Ltd.

Host tasks

© 2020 Codeplay Software Ltd.

- 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

© 2020 Codeplay Software Ltd.

Example of using a host task

Indicative only, still subject to change!

```
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

© 2020 Codeplay Software Ltd.

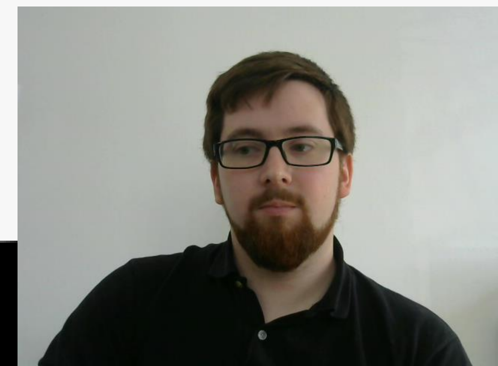
Example of using a host task

Indicative only, still subject to change!

```
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();
```

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



© 2020 Codeplay Software Ltd.

Example of using a host task for interop

Indicative only, still subject to change!

```
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();
```

A host_task can also be used for interoperability by taking an optional interop_handle parameter

© 2020 Codeplay Software Ltd.

Example of using a host task for interop

Indicative only, still subject to change!

```
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();
```

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

© 2020 Codeplay Software Ltd.

Example of using a host task for interop

Indicative only, still subject to change!

```
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();
```

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

© 2020 Codeplay Software Ltd.

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

© 2020 Codeplay Software Ltd.

We're Hiring!

codeplay.com/careers/

Thank you!

gordon@codeplay.com

@codeplaysoft

info@codeplay.com

codeplay.com