Migration

Migrating from CUDA to SYCL

In this chapter we provide mapping tables for the main nomenclature differences between SYCL, CUDA, and OpenCL. Knowing the equivalent nomenclature for each platform is essential to migrate a CUDA code to a SYCL code. The actual function call for CUDA terminology can be found in CUDA C programming guide. For OpenCL, the function call syntax can be found in OpenCL 1.2 reference pages. The SYCL syntax function call can be found in Khronos SYCL 1.2.1 specification.

Execution model equivalence

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM</td>
<td>CU</td>
<td>CU</td>
</tr>
<tr>
<td>SM core</td>
<td>PE</td>
<td>PE</td>
</tr>
<tr>
<td>thread</td>
<td>work-item</td>
<td>work-item</td>
</tr>
<tr>
<td>block</td>
<td>work-group</td>
<td>work-group</td>
</tr>
</tbody>
</table>

Memory model equivalence

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>private memory</td>
<td>private memory</td>
</tr>
<tr>
<td>shared memory</td>
<td>local memory</td>
<td>local memory</td>
</tr>
<tr>
<td>constant memory</td>
<td>constant memory</td>
<td>constant memory</td>
</tr>
<tr>
<td>global memory</td>
<td>global memory</td>
<td>global memory</td>
</tr>
<tr>
<td>local memory</td>
<td>N/A(device specific)</td>
<td>N/A(device specific)</td>
</tr>
</tbody>
</table>
### Host API equivalence

#### Platform API equivalence

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cudaStreamCreate()</code></td>
<td>queue class</td>
<td><code>clCreateCommandQueue()</code></td>
<td>By default the CUDA stream will be constructed by the CUDA driver.</td>
</tr>
<tr>
<td><code>cudaStreamDestroy()</code></td>
<td>N/A</td>
<td><code>clReleaseCommandQueue()</code></td>
<td>In CUDA this function is required if the stream is created explicitly. In SYCL this is handled by the runtime.</td>
</tr>
<tr>
<td><code>cuStreamSynchronize()</code></td>
<td><code>queue::wait()</code></td>
<td><code>clEnqueueBarrierWithWaitList()</code></td>
<td></td>
</tr>
<tr>
<td><code>cudaDeviceSynchronize()</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>cuStreamWaitEvent()</code></td>
<td><code>event::wait()</code></td>
<td><code>clWaitForEvents()</code></td>
<td></td>
</tr>
<tr>
<td><code>cudaStreamAddCallback()</code></td>
<td>N/A</td>
<td><code>clSetEventCallback()</code></td>
<td>SYCL does not support a callback function.</td>
</tr>
<tr>
<td>N/A</td>
<td><code>platform::get()</code></td>
<td><code>clGetPlatformIDs()</code></td>
<td>This is optional in SYCL and it can be handled by the SYCL runtime via device_selector.</td>
</tr>
<tr>
<td><code>cuCtxCreate()</code></td>
<td><code>context class</code></td>
<td><code>clCreateContext()</code></td>
<td>By default it will be constructed by the CUDA driver.</td>
</tr>
<tr>
<td><code>CUDAGetDevice()</code></td>
<td><code>device class</code></td>
<td><code>clGetDeviceInfo()</code></td>
<td>By default this will be constructed by SYCL via the device selector.</td>
</tr>
<tr>
<td><code>CUDASetDevice()</code></td>
<td><code>device_selector class</code></td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

### Memory management API equivalence

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cudaMalloc()</code></td>
<td></td>
<td><code>clCreateBuffer()</code></td>
<td></td>
</tr>
<tr>
<td><code>cudaMallocHost()</code></td>
<td><code>buffer class</code></td>
<td><code>clEnqueueMapBuffer()</code></td>
<td></td>
</tr>
<tr>
<td><code>cudaHostAlloc()</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA</td>
<td>SYCL</td>
<td>OpenCL</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------------------</td>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>cudaHostGetDevicePointer()</td>
<td>accessor class</td>
<td>N/A</td>
<td>OpenCL does not support a unified memory system.</td>
</tr>
<tr>
<td>cudaMemset()</td>
<td>handler::fill()</td>
<td>clEnqueueFillBuffer()</td>
<td></td>
</tr>
<tr>
<td>cudaMemcpyAsync()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaMemcpy()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cudaFree()</td>
<td></td>
<td>clReleaseMemObject()</td>
<td></td>
</tr>
</tbody>
</table>

Runtime API equivalent

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;&lt;....&gt;&gt;&gt;</td>
<td>nd_range class</td>
<td>global_work_size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local_work_size variables</td>
</tr>
</tbody>
</table>
CUDA | SYCL | OpenCL
---|---|---
"kernel function name"<<<...>>>() | queue::submit() | clCreateProgramWithSource/Binary() clBuildProgram() clCreateKernel() clCreateKernel() clSetKernelArg() clEnqueueNDRangeKernel()

**Device API equivalence**

**Kernel functions qualifiers**

Qualifiers are not needed in SYCL as they are all abstracted by the SYCL runtime classes, but OpenCL ones are provided for reference.

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>global</strong> function</td>
<td>N/A</td>
<td>__kernel function</td>
</tr>
<tr>
<td><strong>device</strong> function</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>constant</strong> variable declaration</td>
<td>N/A</td>
<td>__constant variable declaration</td>
</tr>
<tr>
<td><strong>device</strong> variable declaration</td>
<td>N/A</td>
<td>__global variable declaration</td>
</tr>
<tr>
<td><strong>shared</strong> variable declaration</td>
<td>N/A</td>
<td>__local variable declaration</td>
</tr>
</tbody>
</table>

**Indexing equivalence**

Please note that this is the general mapping between CUDA, OpenCL, and SYCL. Some architectures may not support 3 dimensions.

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>N/A</td>
<td>nd_item class</td>
<td>N/A</td>
</tr>
<tr>
<td>gridDim.{x, y, z}</td>
<td>nd_item::get_num_group({0,1,2})</td>
<td>get_num_group({0,1,2})</td>
</tr>
<tr>
<td>blockDim.{x, y, z}</td>
<td>nd_item::get_local_range({0,1,2})</td>
<td>get_local_size({0,1,2})</td>
</tr>
<tr>
<td>CUDA</td>
<td>SYCL</td>
<td>OpenCL</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>blockIdx.{x, y, z}</td>
<td>nd_item::get_group({0,1,2})</td>
<td>get_group_id({0,1,2})</td>
</tr>
<tr>
<td>threadIdx.{x, y, z}</td>
<td>nd_item::get_local_id({0,1,2})</td>
<td>get_local_id({0,1,2})</td>
</tr>
<tr>
<td>N/A</td>
<td>nd_item::get_global_id({0,1,2})</td>
<td>get_global_id({0,1,2})</td>
</tr>
<tr>
<td>N/A</td>
<td>nd_item::get_linear_group_id()</td>
<td>N/A</td>
</tr>
<tr>
<td>N/A</td>
<td>nd_item::get_linear_local_id()</td>
<td>N/A</td>
</tr>
<tr>
<td>N/A</td>
<td>nd_item::get_linear_global_id()</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Synchronization equivalence

<table>
<thead>
<tr>
<th>CUDA</th>
<th>SYCL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>__syncthread()</td>
<td>nd_item::barrier()</td>
<td>barrier()</td>
</tr>
<tr>
<td>__threadfence_block()</td>
<td>nd_item::mem_fence()</td>
<td>mem_fence()</td>
</tr>
<tr>
<td>N/A</td>
<td>nd_item::mem_fence()</td>
<td>read_mem_fence()</td>
</tr>
<tr>
<td>N/A</td>
<td>nd_item::mem_fence()</td>
<td>write_mem_fence()</td>
</tr>
<tr>
<td>__threadfence()</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>__threadfence_system()</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

How to rewrite CUDA multi-GPU code for SYCL

NVIDIA CUDA supports using multiple GPUs from versions >= 4.0. CUDA can dispatch CUDA kernels, data movement, communication and synchronization among all NVIDIA GPUs.

OpenCL supports using multiple OpenCL-enabled devices for dispatching kernels, data movement, communication and synchronization only if they reside in the same OpenCL context. NVIDIA devices do not have the context restriction as all NVIDIA devices share the same context created for the NVIDIA platform. In general, a computer or a device can accommodate more than one OpenCL-enabled device, provided by different vendors. An example could be a desktop computer with both Intel and AMD GPUs where each of them has its own OpenCL driver. Therefore, each device belongs
to a different OpenCL platform and must have a different context. In such case, two separate OpenCL buffers must be created. Data movement, communication and synchronization must be handled manually. SYCL, on the other hand, not only supports dispatching the kernels to multiple OpenCL devices, but it can also implicitly handle the data movement, synchronization and communication among all OpenCL-enabled devices, irrespective of the platform they belong to.

CUDA multiple GPU

A single CPU thread can be used for handling multiple GPUs in CUDA. The following function is used to get the number of available NVIDIA GPU devices in CUDA:

cudaError_t cudaGetDeviceCount(int* num_gpu);

cudaSetDevice can be used to choose a specific device for dispatching a CUDA kernel. The following pseudo-code demonstrates dispatching a kernel on multiple NVIDIA GPUs:

```c
int num_gpu;
// finding available number of NVIDIA GPU devices
cudaGetDeviceCount(&num_gpu);

//looping over number of devices and dispatching a kernel per device.
for (int i = 0; i < ngpus; i++) {
    // selecting the current device
    cudaSetDevice(i);
    // executing a my_kernel on the selected device
    my_kernel<<<num_blocks, block_size>>>(...);
    // transfering data between the host and the selected device
    cudaMemcpy(...);
}
```

Note that the above code dispatches a kernel by using a single thread. It is possible to parallelize the for loop launching the kernels. However, it is the user's responsibility to handle locking when multiple threads use multiple devices. See the CUDA C programming guide for further information.

SYCL multiple devices

The list of SYCL supported platforms can be obtained with the list of devices for each platform by calling `get_platforms()` and `platform.get_devices()` respectively. Once we have all the devices, we can construct a queue per device and dispatch different kernels to different queues.

When there is only one device, kernels can be submitted to the same device. The following code snippet represents dispatching multiple kernels to a single SYCL device:

```c

auto my_queue = cl::sycl::queue(device_selector);
```
// submitting a kernel to a the sycl queue
my_queue.submit([&](cl::sycl::handler &cgh) {
    // sycl kernel 1
    cgh.parallel_for(...);
});

my_queue.submit([&](cl::sycl::handler &cgh) {
    // sycl kernel 2
    cgh.parallel_for(...);
});

my_queue.submit([&](cl::sycl::handler &cgh) {
    // sycl kernel 3
    cgh.parallel_for(...);
});

...;

Moreover, when there are multiple devices, the kernels can be distributed among all devices. The following code snippet represents dispatching a kernel on multiple SYCL devices:

...;

// getting the list of all supported sycl platforms
auto platform_list = cl::sycl::platform::get_platforms();

// getting the list of devices from the platform
auto device_list = platform.get_devices();

// looping over platforms
for (const auto &platform : platform_list) {
    // looping over devices
    for (const auto &device : device_list) {
        auto queue = cl::sycl::queue(device);
        // submitting a kernel to a the sycl queue
        queue.submit([&](cl::sycl::handler &cgh) {
            // sycl kernel
            cgh.parallel_for(...);
        });
    }
}

...;

Unlike CUDA, dispatching kernels to multiple SYCL devices is a thread safe operation, and can be done from different threads of execution.
Porting CUDA CPU library functions to SYCL

At the time of writing this document, the cuBLAS library, which is part of the CUDA 6.0 (and above) ecosystem, is the only library that can be used as a replacement for a CPU BLAS library directly by replacing the link flags in the build system. In particular, NVBLAS is an interception layer that replaces calls to a CPU-based BLAS library into its cuBLAS counterparts. No source code modification is required.

Although technically possible, at the time of writing this document, such intercept layer for SYCL-BLAS has not been added to the SYCL ecosystem. In general, the interception layer mechanism is suitable for C-based interfaces, where the symbol name is clearly identifiable in the generated binary. C++-based interfaces rely on template meta-programing, which makes the interception of symbols at link time difficult. However, a new BLAS library project with a C interface could implement this feature.

Porting CUDA debug code to SYCL

Error-handling and reporting

In CUDA, all synchronous functions return an error code. Note that asynchronous functions (such as kernel execution) cannot possibly return an error code via its interface, so a later invocation to a function may return an error that comes from a previous API call. Typically, the error code returned by cudaDeviceSynchronize is used to query for the error of a previous kernel execution, although more advanced functionality is available. Developers are encouraged to write their own checking macros to wrap API calls and catch synchronous errors.

In SYCL, error reporting is done via the C++ exception mechanism. Synchronous operations will throw exceptions derived from cl::sycl::exception and can be catch in-place by developers, like in the example below:

```cpp
try {
    mySyclProgram.build_with_kernel_type<class myKernel>()
} catch (cl::sycl::compile_program_error e) {
    std::cout << " The program failed to build " << std::endl;
}
```

Asynchronous operations are captured and stored automatically by the SYCL runtime. Multiple exceptions are stored in the order they are captured in a list, of type cl::sycl::exception_list. On construction of a queue, an optional functor - the asynchronous handler - can be specified. Whenever a user calls the wait_and_throw or async_throw methods of the SYCL queue, this functor is called. Developers can use this function to handle custom behavior for dealing with exceptions, such as reducing the multiple exceptions in a list to the most severe one and re-throwing it. If neither of those methods are called, the exceptions are discarded when the queue object is destroyed. The async_handler example of the ComputeCpp SDK illustrates the usage of this mechanism to rethrow and display exceptions.

In both synchronous and asynchronous cases, when a SYCL-related error occurs, an object of type cl::sycl::exception will be thrown. SYCL exception objects are derived from the standard exception type std::exception. Whenever a specific exception type is available, the SYCL exception type can inherit from the particular exception type and throw a more elaborate report regarding the captured error. Refer to the SYCL specification Section 4.9.2 for details on the Exception
Class Interface. All SYCL exception objects contain a description message, whose contents are implementation defined. Whenever possible, a low-level OpenCL error and/or a pointer to a SYCL context is also provided.

Debugging SYCL code

The CUDA ecosystem includes a CUDA debugger, \texttt{cuda-gdb}, which is based on gdb with some extensions to facilitate debugging of GPU code. Developers can also use printf inside CUDA kernels to display intermediate results. In addition, a functional correctness checking suite called \texttt{cuda-memcheck} is available to check different aspects of CUDA applications.

In SYCL, we have to distinguish two kinds of debugging: (A) Debugging the host application or (B) debugging the device-specific behavior. There is no standard solution for (B), as the code executed on the device can only be debugged within a debugger that contains supports for the given platform. This is inherently device-specific. On the other hand, (A) can be dealt with easily in SYCL. Any code that executes on a device must run on the host using a host queue. Hence, replacing a device-queue with a host queue will have the same expected output (minus device-specific behavior) as the original device queue. Since the host queue is implemented in pure C++, normal C++ debuggers can be used to inspect the behavior of the kernels on host. A host queue can be created on the host simply by creating a queue from a host device. A simple option to enable simple debugging on the host is to use a pre-processor macro when creating a custom debug build, as shown below:

```cpp
#if __DEBUG_IN_HOST_DEVICE
// Construct a host device
host_device hd;
// Create a queue with a host device
queue myQueue(hD);
#else
// Use the default selector of the platform
queue myQueue;
#endif
```

The rest of the code remains unchanged.

Using the same mechanism, any C++ check tool can be used to inspect the behavior of the application on the host. For example, \texttt{valgrind} can be used to detect out of bounds access to arrays when using the host device without requiring special configuration options.

To display intermediate results, SYCL developers can still use the low-level printf function, with equivalent behavior as the OpenCL printf function. For device kernels, C++ \texttt{iostream} operations (e.g., \texttt{cout} or \texttt{cerr}) are not available. However, a replacement SYCL stream class with equivalent functionality is available on the device. The SYCL stream class is defined in Section 4.12 of the SYCL 1.2.1 specification. The example \texttt{hello-world in the ComputeCpp SDK} illustrates the usage of the stream object. The output of the stream object is displayed when the kernel execution completes.

Measuring SYCL application performance

SYCL application performance can be measured by enabling profiling in each SYCL queue. Queue profiling can be enabled by passing the SYCL property list with

\texttt{cl::sycl::property::queue::enable_profiling()} to a SYCL queue.
Once profiling is enabled, the kernel execution submit time, kernel execution start time, and the kernel execution end time can be extracted.

The following code snippet represents how to enable profiling and extract the execution time from a sycl application.

```cpp
int main() {
    // ...

    // enabling SYCL queue profiling
    auto property_list = cl::sycl::property_list{cl::sycl::property::queue::enable_profiling()};

    // adding the property list with profiling enabled option to the sycl queue.
    auto sycl_queue = cl::sycl::queue(property_list);

    //....

    // submitting sycl kernel
    auto event = sycl_queue.submit([&](cl::sycl::handler& cgh){
        //...
    });

    // waiting for the kernel to finish
    event.wait();

    // getting kernel submission time
    auto submit_time =
        event.get_profiling_info<
            cl::sycl::info::event_profiling::command_submit>();

    // getting kernel start time
    auto start_time = event.get_profiling_info<
        cl::sycl::info::event_profiling::command_start>();

    // getting kernel end time
    auto end_time = event.get_profiling_info<
        cl::sycl::info::event_profiling::command_end>();

    // calculating the duration between submitting a sycl kernel and executing
    // sycl kernel in milliseconds.
    auto submission_time = (start_time - submit_time) / 1000000.0f;

    // calculating the kernel execution time in milliseconds
    auto execution_time = (end_time - start_time) / 1000000.0f;
}
```
Porting CUDA build system to SYCL

The CUDA platform offers a single unified compiler that is capable of generating a final application binary from CUDA code, nvcc. Different versions of the CUDA platform offer different capabilities, refer to The nvcc documentation for details. The CUDA compiler can also be used to build only the device code, and use another standard compiler (e.g, gcc) to link and compile on the host.

In the SYCL standard, two mechanisms of compilation are described: (A) Single unified compiler and (B) Multiple compilers for the same source. Different SYCL implementations may implement (A), (B) or both. In this document we focus on Codeplay's ComputeCpp implementation. The details of the build system can be found in the ComputeCpp integration guide. The integration guide contains detailed examples on integration of different build systems and different compilation options.

Switching from nvcc to compute++

The simplest way to switch from the NVIDIA CUDA compiler to the ComputeCpp SYCL compiler is to replace the former with the later. compute++ is a clang driver, hence any standard clang and/or llvm options will be valid. For example, building a simple hello world cuda file:

```
$ nvcc hello_world.cu -o hello_world.exe
```

Is equivalent to using compute++ to build the SYCL C++ source:

```
$ compute++ hello_world.cpp -o hello_world.exe
```

Note compute++ builds C++ files, whereas CUDA builds .cu files. CUDA is not considered standard C++, hence uses a different file type. This implies that CUDA files need to be converted to C++ sources before they can be built with SYCL C++.

The following table shows the equivalence between nvcc-specific options and compute++. nvcc-specific options not listed here are not supported or not applicable to compute++. For all the standard options, refer to the clang command line documentation

<table>
<thead>
<tr>
<th>nvcc</th>
<th>compute++</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(default behavior)</em></td>
<td>-sycl-driver</td>
<td>Compiles both host and device code into a single output binary</td>
</tr>
<tr>
<td>--fatbin, --ptx, --cubin --device-c</td>
<td>-sycl -c or -sycl-device-only</td>
<td>Outputs only the integration header</td>
</tr>
</tbody>
</table>
If the SYCL compiler-driver is used, there is no need to install any host-side compiler. If Single-source multiple-compiler passes is used, the installation of a host-side compiler, like g++, is required.

**Porting CUDA stream to SYCL queue**

**CUDA stream**

A CUDA stream is a sequence of CUDA operations, submitted from host code. These operations are executed asynchronously in order of submission. It is always users' responsibility to synchronize the operation before using the result. If no CUDA stream is given, a default CUDA stream is created and all operations are submitted to the default stream.

It is possible to overlap the execution of multiple CUDA operations by creating multiple CUDA streams and submitting them to different streams. This can be used to overlap submission of kernels with data transfer operations.

The following code snippet demonstrates how to define, create, and release a CUDA stream object. See the **CUDA C programming guide** for more information.

```c
// defining a CUDA stream object
cudaStream_t stream;
// creating a CUDA stream object
cudaError_t err = cudaStreamCreate(&stream);
// releasing a CUDA stream object
cudaError_t err = cudaStreamDestroy(stream);
```

**SYCL queue**

In a similar fashion to CUDA streams, SYCL queues submit command groups for execution asynchronously. However, SYCL is a higher-level programming model, and data transfer operations are implicitly deduced from the dependencies of the kernels submitted to any queue. Furthermore, SYCL queues can map to multiple OpenCL queues, enabling transparent overlapping of data-transfer and kernel execution. The SYCL runtime handles the execution order of the different command groups (kernel + dependencies) automatically across multiple queues in different devices.

We can create a SYCL queue by instantiating the queue class.

```c
cl::sycl::queue myQueue{device_selector}
```

The `device_selector` determines the SYCL device we are going to use. The system resources required by the queue are released automatically after it goes out of scope following C++ RAII rules.
Event mechanism

CUDA event function

A CUDA event is a marker associated with a certain point in the stream. A CUDA event can be used either to synchronize the stream execution or to monitor the progress in the device. The following code snippet represents the declaration, construction and releasing of a CUDA event. More information can be found in CUDA C programming guide.

```c
// declaration of a CUDA event
cudaEvent_t event;

// creation of a CUDA event
cudaError_t cudaEventCreate(&event);

// using the event here ...

// releasing a CUDA event
cudaError_t cudaEventDestroy(event);
```

SYCL event object

An event in SYCL is an abstraction of the OpenCL cl_event object. An OpenCL event is used to synchronize memory transfers, dispatch of kernels and signaling barriers. As an abstraction layer on OpenCL events, SYCL events accommodate synchronization between different contexts, devices and platforms. The submit method in a SYCL queue returns a cl::sycl::event. SYCL events can be used to manually define synchronization points irrespective of the underlying device and, thus, can be used also to synchronize operations on host queues.

```c
{
    // SYCL events are returned from a command group submission
    auto event = syclQueue.submit([&](cl::sycl::handler &cgh) {
        /** SYCL command group **/
    });
    // Users can wait on a specific event, regardless on where is submitted
    event.wait();
    // Event is released automatically at the end of the scope
}
```

Triggering user functions from device events

CUDA callback function

A stream callback function enables the execution of a host function provided by the application after all the previous operations in the stream have completed. A stream callback can be submitted to a CUDA stream by calling the following function:

```c
cudaError_t cudaStreamAddCallback(cudaStream_t stream, cudaStreamCallback_t callback, void *userData, unsigned int flags);
```
Where the callback is the callback function and the userData is the parameter passed to the callback function.

A CUDA stream callback has the following restrictions:

- A callback function cannot call any CUDA API function
- A callback function cannot contain any synchronization function.

The following code snippet represents the usage of the CUDA callback example.

```c
void CUDART_CB callback_func(cudaStream_t stream, cudaError_t status, void *) {
    printf("callback function is called");
}
// defining a CUDA stream object
cudaStream_t stream;
// creating a CUDA stream object
cudaError_t err = cudaStreamCreate(&stream);
// launching the first kernel
first_kernel<<<numBlock, blockSize, 0, stream>>>();
second_kernel<<<numBlock, blockSize, 0, stream>>>();
third_kernel<<<numBlock, blockSize, 0, stream>>>();
cudaStreamAddCallback(stream, callback_func, NULL, 0);
// releasing a CUDA stream object
cudaError_t err = cudaStreamDestroy(stream);
```

SYCL callback function

There is no direct functionality in SYCL to trigger a host callback from a device function. Developers can manually wait on an event and call the callback function afterwards. The following code snippet represents an example of a callback function implementation using this mechanism.

```c
std::function<void(std::string)> my_callback =
    [](std::string my_string) { std::cout << my_string << std::endl; }
// constructing the SYCL queue
auto queue = cl::sycl::queue{default_selector};
// data on host
auto vec = std::vector<float>{100, 1};
// constructing a SYCL buffer
auto buffer = cl::sycl::buffer<float, 1>{vec.data(),
    cl::sycl::range<1>{100}};
// submitting the kernel to the queue
auto event = queue.submit([&](cl::sycl::handler &cgh) {
    // getting a read_write access over SYCL buffer
    auto acc = buffer.get_access<cl::sycl::access::mode::read_write>(cgh);
    // SYCL device kernel
cgh.parallel_for<class kernel>(
        cl::sycl::range<1>{100}),
```
This achieves the same result as a CUDA callback, using pure C++ features.

Codeplay has proposed to the SYCL standard a new type of handler, the `host_handler`, that is capable of executing a single task on the host and can be submitted directly to the SYCL device queue. It ensures that this task is executed asynchronously and in-order w.r.t the other command groups on the same queue. See the **Enqueuing host tasks on SYCL queues** proposal in SYCL for further information. This feature is available since ComputeCpp 0.5.1 and above as a vendor extension.