SYCL Academy

Hands-On HPC Application Development Using C++ and SYCL

James Reinders, Phuong Nguyen, Thomas Applencourt, Rod Burns, Alastair Murray

HANDLING ERRORS AND DEBUGGING

LEARNING OBJECTIVES

- Learn about how SYCL handles errors
- Learn about the difference between synchronous and asynchronous exceptions
- Learn how to handle exceptions and retrieve further information
- Learn about the host device and how to use it

SYCL EXCEPTIONS

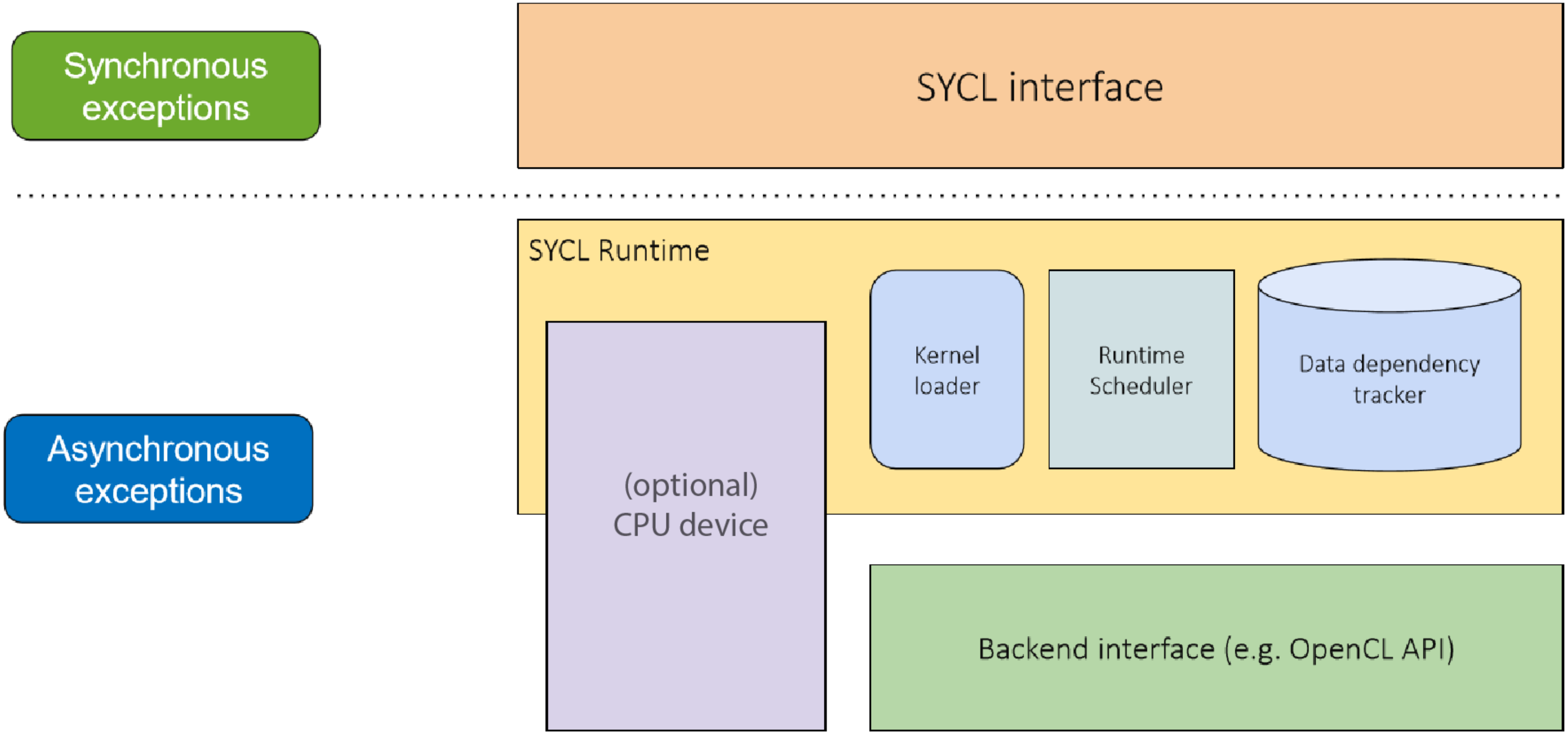
- In SYCL errors are handled by throwing exceptions.
- It is crucial that these errors are handled, otherwise your application could fail in unpredictable ways.
- In SYCL there are two kinds of error:
    - Synchronous errors (thrown in user thread).
    - Asynchronous errors (thrown by the SYCL scheduler).

HANDLING ERRORS

```
int main() {
  queue q();

  /* Synchronous code */

  q.submit([&](handler &cgh) {

    /* Synchronous code */

    cgh.single_task<add>(buf0.get_range(), [=](id<1> i) {

      /* Asynchronous code */

    });
  });
}
```

- Kernels run asynchronously on the device, and will throw asynchronous errors.
- Everything else runs synchronously on the host, and will throw synchronous errors.

SYCL EXCEPTIONS



HANDLING ERRORS

```
class add;

int main() {
  queue q();

  /* Synchronous code */

  q.submit([&](handler &cgh) {
    /* Synchronous code */

    cgh.single_task<add>([=](id<1> i) {
      /* Asynchronous code */
    });
  }).wait();
}
```

- Code on the device runs asynchronously
- If errors are not handled, the application can fail.
- SYCL 2020 provides a default async handler that will call std::terminate when an asynchronous error is thrown.

```
class add;

int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
  try {
    queue gpuQueue(gpu_selector{});

    buffer bufA{dA};
    buffer bufB{dB};
    buffer bufO{dO};

    gpuQueue.submit([&](handler &cgh) {
      auto inA = accessor{bufA, cgh, read_only};
      auto inB = accessor{bufB, cgh, read_only};
      auto out = accessor{bufO, cgh, write_only};

      cgh.single_task<add>(bufO.get_range(), [=](id<1> i) {
        out[i] = inA[i] + inB[i];
      });
    }).wait();

  } catch (...) { /* handle errors */ }
}
```

- Synchronous errors are typically thrown by SYCL API functions.
- In order to handle all SYCL errors you must wrap everything in a try-catch block.

```
class add;

int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
  try{
    queue gpuQueue(gpu_selector{}, async_handler{});

    buffer bufA{dA};
    buffer bufB{dB};
    buffer bufO{dO};

    gpuQueue.submit([&](handler &cgh) {
      auto inA = accessor{bufA, cgh, read_only};
      auto inB = accessor{bufB, cgh, read_only};
      auto out = accessor{bufO, cgh, write_only};

      cgh.single_task<add>(bufO.get_range(), [=](id<1> i) {
        out[i] = inA[i] + inB[i];
      });
    }).wait();

    gpuQueue.throw_asynchronous();
  } catch (...) { /* handle errors */
}
```

- Asynchronous errors errors that may have occurred will be thrown after a command group has been submitted to a queue.
    - To handle these errors you must provide an async handler when constructing the queue object.
    - To call the throw_asynchronous or wait_and_throw member functions of the queue class.

```
class add;

int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0
  try{
    queue gpuQueue(gpu_selector{}, [=](exception_list eL) {
      for (auto e : eL) { std::rethrow_exception(e); }
    });

    buffer bufA{dA};
    buffer bufB{dB};
    buffer bufO{dO};

    gpuQueue.submit([&](handler &cgh) {
      auto inA = accessor{bufA, cgh, read_only};
      auto inB = accessor{bufB, cgh, read_only};
      auto out = accessor{bufO, cgh, write_only};
```

- The async handler is a C++ lambda or function object that takes as a parameter an exception_list
- The exception_list class is a wrapper around a list of exception_ptrs which can be iterated over
- …eption_ptrs can be rethrown by passing them to std::rethrow_exception

```
int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0
  try {
    queue gpuQueue(gpu_selector{}, [=](exception_list eL) {
      for (auto e : eL) { std::rethrow_exception(e); }
    });

    ...

    gpuQueue.throw_asynchronous();
  } catch (const std::exception& e) {
    std::cout << "Exception caught: " << e.what()
     << std::endl;
  }
}
```

- Once rethrown and caught, a SYCL exception can provide information about the error
- The what member function will return a string with more details

```
int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0
  try {
    queue gpuQueue(gpu_selector{}, [=](exception_list eL) {
      for (auto e : eL) { std::rethrow_exception(e); }
    });

    ...

    gpuQueue.throw_asynchronous();
  } catch (const sycl::exception& e) {
    std::cout << "Exception caught: " << e.what();
    std::: cout << " With OpenCL error code: "
     << e.get_cl_code() << std::endl;
  }
}
```

- In SYCL 1.2.1, if the exception has an OpenCL error code associated with it this can be retrieved by calling the get_cl_code member function
- If there is no OpenCL error code this will return CL_SUCCESS
- SYCL 2020 provides the error_category_for templated free function that allows
g for the category of the exception depending on the backend used (e.g.
backend::opencl), and e.code().value() will correspond to the backend error

```
int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };

  queue gpuQueue(gpu_selector{}, [=](exception_list eL) {
    for (auto e : eL) { std::rethrow_exception(e); }
  });
  context gpuContext = gpuQueue.get_context();

  try {
    ...
    gpuQueue.wait_and_throw();
  } catch (const sycl::exception& e) {
    if (e.has_context()) {
      if (e.get_context() == gpuContext) {
        /* handle error */
      }
    }
  }
}
```

- The has_context member function will tell you if there is a SYCL context associated with the error
- If that returns true then the get_context member function will return the associated SYCL context object

EXCEPTION TYPES

- SYCL 2020 has a single sycl::exception type which provides different error codes
  - e.g., errc::runtime, errc::kernel

DEBUGGING SYCL KERNEL FUNCTIONS

- Top debugging tip: use CPU devices during development as much as is appropriate.
- SYCL 2020 only guarantees that a device will always be available.
- We can query the host_debuggable device aspect to check for host-level type debugging support. Such devices allow us to debug a SYCL kernel function using a standard C++ debugger (e.g., gdb).

SYCL Academy

```
class add;

int main() {
  std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
  try{
    queue hostQueue(aspect_selector<aspect::host_debuggable>(), async_handler{});

    buffer bufA{dA};
    buffer bufB{dB};
    buffer bufO{dO};

    hostQueue.submit([&](handler &cgh) {
      auto inA = accessor{bufA, cgh, read_only};
      auto inB = accessor{bufB, cgh, read_only};
      auto out = accessor{bufO, cgh, write_only};

      cgh.single_task<add>(bufO.get_range(), [=](id<1> i) {
        out[i] = inA[i] + inB[i];
      });
    });
    hostQueue.wait_and_throw();
  } catch (...) { /* handle errors */ }
}
```

- In general, a SYCL application can be debugged on the CPU device by switching the queue for a CPU queue
- Replacing the device selector for the aspect_selector will ensure that the queue submits all work to the device with the requested aspects, in this case a host_debuggable device

QUESTIONS

EXERCISE

Lesson_Materials/Lecture_03_Error_Handling

- Introduce a synchronous error, and
- Introduce an asynchronous error.
- Catch them and report them without aborting the program (so we get to see both error messages).
- The try/catch framework is already in place.