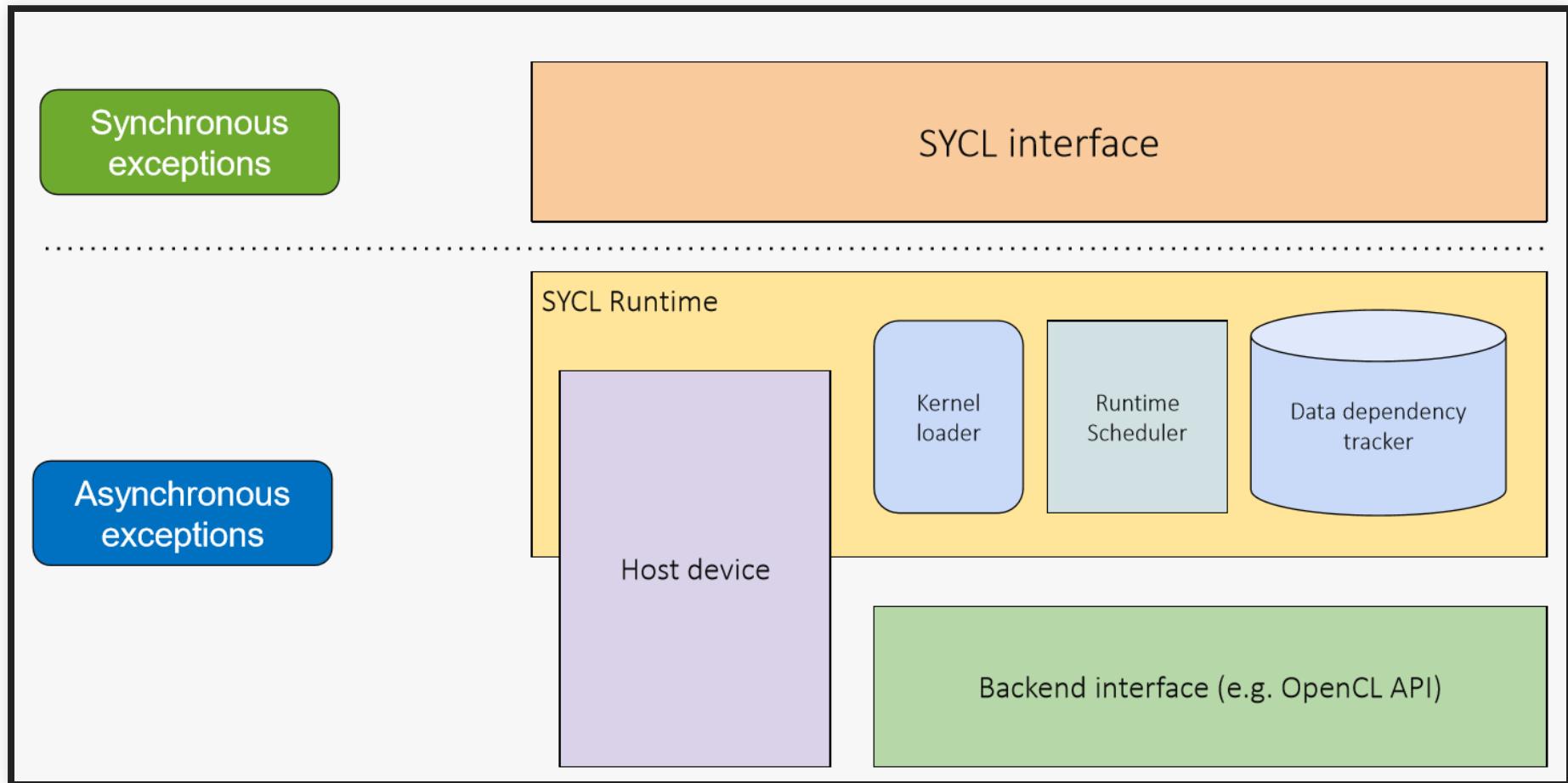


HANDLING SYCL ERRORS

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 different exception types
- Learn about the host device and how to use it

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



HANDLING ERRORS

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
    std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
    queue gpuQueue(gpu_selector{});

    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

    gpuQueue.submit([&] (handler &cgh) {
        auto inA = bufA.get_access<access::mode::read>(cgh);
        auto inB = bufB.get_access<access::mode::read>(cgh);
        auto out = bufO.get_access<access::mode::write>(cgh);

        cgh.parallel_for<add>(range<1>(dA.size()), [=](id<1> i) {
            out[i] = inA[i] + inB[i];
        });
    });
    gpuQueue.wait();
}
```

- If errors are not handled, the application can fail silently

```
int main(int argc, char *argv[]) {
    std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
    try{
        queue gpuQueue(gpu_selector{});

        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

        gpuQueue.submit([&] (handler &cgh) {
            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()), [=](id<1> i){
                out[i] = inA[i] + inB[i];
            });
        });
        gpuQueue.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

```
int main(int argc, char *argv[]) {
    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<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

        gpuQueue.submit([&] (handler &cgh) {
            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()), [=] (id<1> i) {
                out[i] = inA[i] + inB[i];
            });
        });
        gpuQueue.wait_and_throw();
    } 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
- Then you must also call the **throw_asynchronous** or **wait_and_throw** member functions of the queue class
- This will pass the exceptions to the async handler in the user thread so they can be thrown

```
int main(int argc, char *argv[]) {
    std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
    try{
        queue gpuQueue(gpu_selector{}, [=](sycl::exception_list eL) {
            for (auto e : eL) { std::rethrow_exception(e); }
        });
        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

        gpuQueue.submit([&] (handler &cgh){ // Command group submitted to queue
            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()), [=](id<1> i){
                out[i] = inA[i] + inB[i];
            });
        });
        gpuQueue.wait_and_throw(); } catch (...) { /* handle errors */ }
    }
```

- 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
- The exception_ptrs can be rethrown by passing them to **std::rethrow_exception**

```
int main(int argc, char *argv[]) {
    std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
    try {
        queue gpuQueue(gpu_selector{}, [=](sycl::exception_list eL) {
            for (auto e : eL) { std::rethrow_exception(e); }
        });
        ...
        gpuQueue.wait_and_throw();
    } catch (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(int argc, char *argv[]) {
    std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
    try {
        queue gpuQueue(gpu_selector{}, [=](sycl::exception_list eL) {
            for (auto e : eL) { std::rethrow_exception(e); }
        });
        ...
        gpuQueue.wait_and_throw();
    } catch (std::exception e) {
        std::cout << "Exception caught: " << e.what();
        std::cout << " With OpenCL error code: "
        << e.get_cl_code() << std::endl;
    }
}
```

- 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`

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

- In SYCL there are a number of different exception types that inherit from `std::exception`
 - E.g. `runtime_error`, `kernel_error`
- The [SYCL 1.2.1 specification](#) will detail cases where a specific error can be expected

DEBUGGING SYCL KERNEL FUNCTIONS

- Every SYCL implementation is required to provide a host device
 - This device executes native C++ code but is guaranteed to emulate the SYCL execution and memory model
- This means you can debug a SYCL kernel function by switching to the host device and using a standard C++ debugger
 - For example gdb

```
int main(int argc, char *argv[]) {
    std::vector<float> dA{ 7, 5, 16, 8 }, dB{ 8, 16, 5, 7 }, dO{ 0, 0, 0, 0 };
    try{
        queue hostQueue(host_selector{}, async_handler());
        buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
        buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
        buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

        hostQueue.submit([&] (handler &cgh) {
            auto inA = bufA.get_access<access::mode::read>(cgh);
            auto inB = bufB.get_access<access::mode::read>(cgh);
            auto out = bufO.get_access<access::mode::write>(cgh);

            cgh.parallel_for<add>(range<1>(dA.size()),
                [=](id<1> i){out[i] = inA[i] + inB[i];});
        });
        gpuQueue.wait_and_throw();
    } catch (...) { /* handle errors */ }
}
```

- Any SYCL application can be debugged on the host device by switching the queue for a host queue
- By replacing the device selector for the host_selector will ensure that the queue submits all work to the host device