SYCL TOPOLOGY DISCOVERY AND QUEUE CREATION

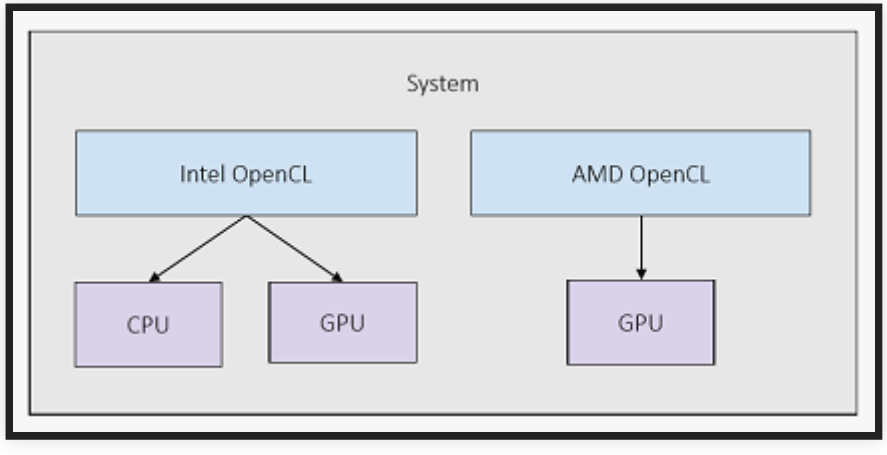
LEARNING OBJECTIVES

- Learn about the SYCL system topology and how to traverse it
- Learn how to query information about a platform or device
- Learn how to select a device; both manually and using device selectors
- Learn how to configure a queue
- Learn about the SYCL scheduler and how to enqueue work

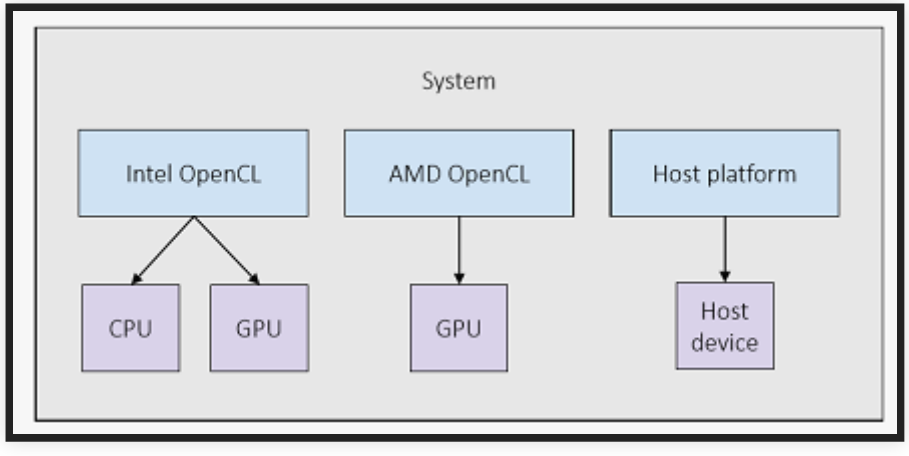
SYCL SYSTEM TOPOLOGY

- A SYCL application can execute work across a range of different heterogeneous devices
- The devices that are available in any given system are determined at runtime through topology discovery

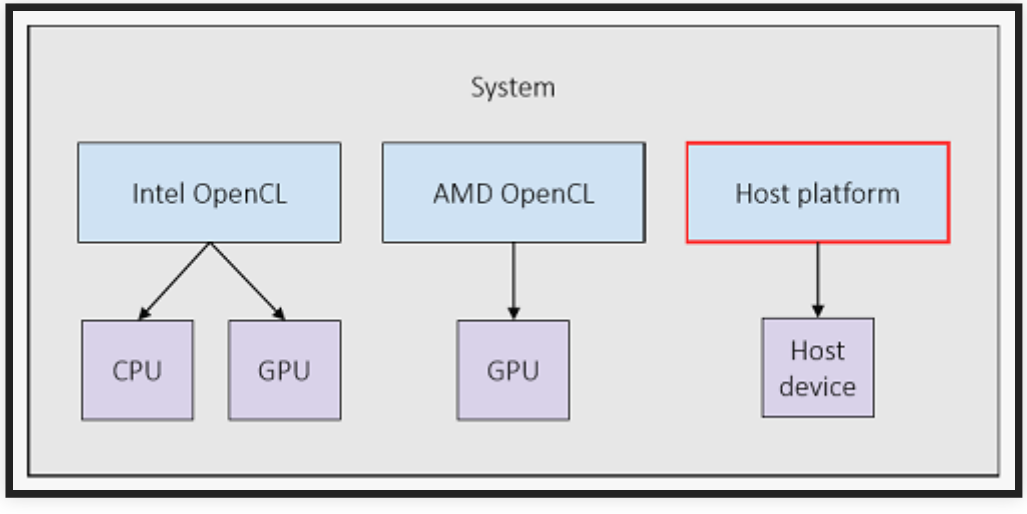
- The SYCL runtime will discover a set of platforms that are available in the system
  - Each platform represents a backend implementation such as Intel OpenCL and Nvidia OpenCL
- The SYCL runtime will also discover all the devices available for each of those platforms
  - CPU, GPU, FPGA, and other kinds of accelerators



- In SYCL there is also a host device which executes SYCL kernels as native C++
  - The host device emulates the execution and memory model of a SYCL device
- This is very useful for debugging SYCL kernels
- There is only ever one host device and that device is associated with a host platform
  - This is generally a CPU implementation



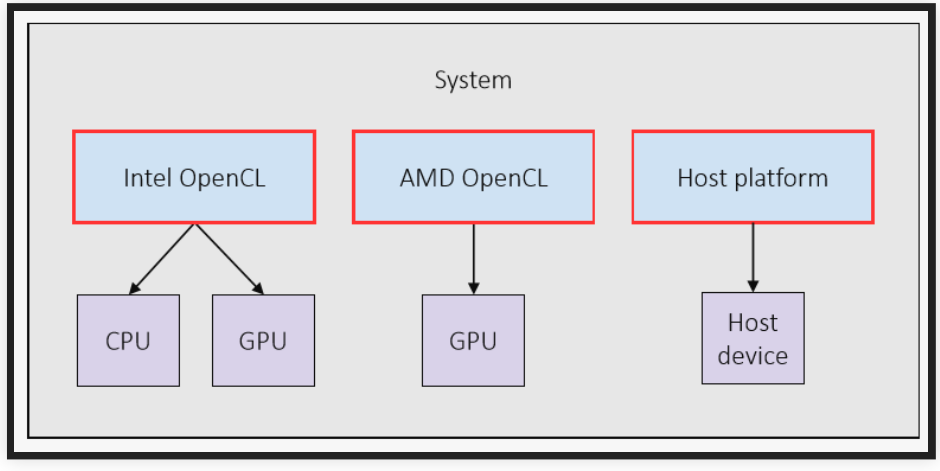
- Platforms and devices are represented by the **platform** and **device** classes respectively
- A default constructed platform object represents the host platform
- A default constructed device object represents the host device



- In SYCL there are two ways to query a system's topology
  - The topology can be manually queried and iterated over via APIs of the platform and device classes
  - The topology can be automatically queried and iterated over using a use specified heuristic by a device selector object

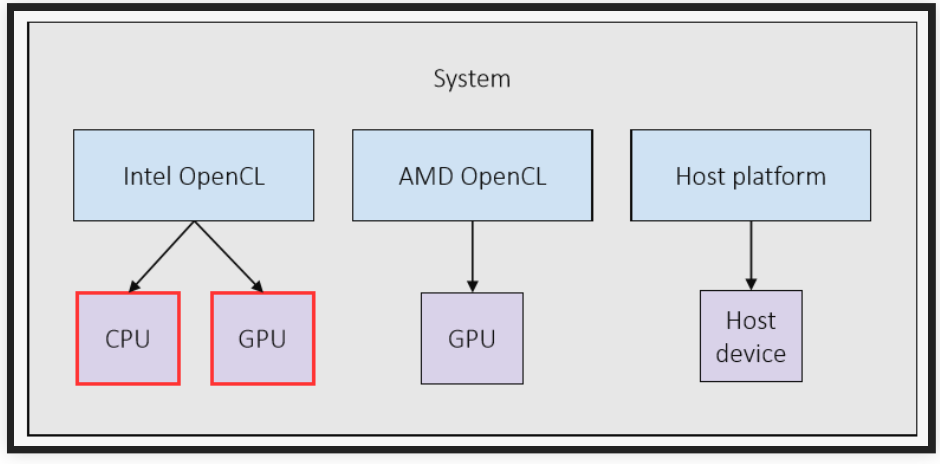
QUERYING THE TOPOLOGY MANUALLY

```
auto platforms = platform::get_platforms();
```



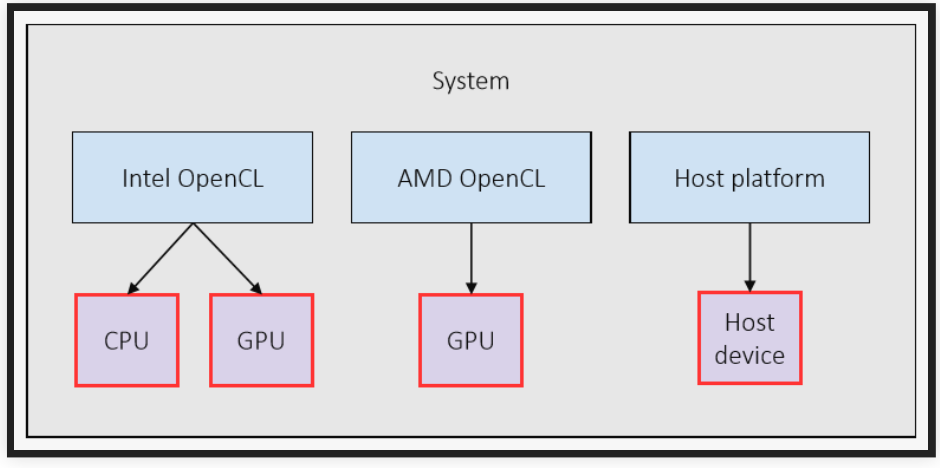
- The platform class provides the static function **get_platforms**
  - It retrieves a vector of all available platforms in the system
- This includes the host platform

```
auto intelDevices = intelPlatform.get_devices();
```



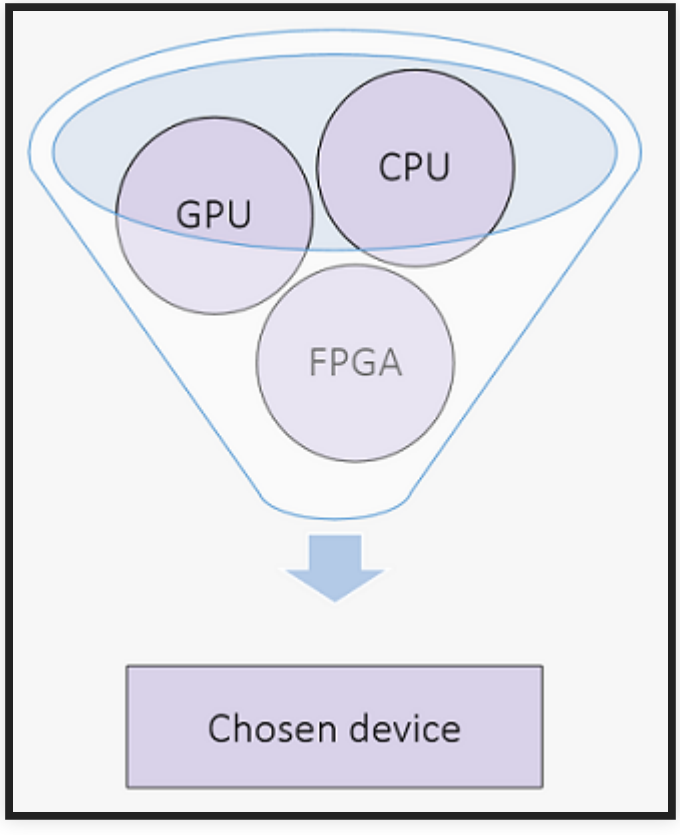
- The platform class provides the member function **get_devices** that
  - It returns a vector of all devices associated with that platform
- This includes the host device if the platform object represents a host platform

```
auto devices = device::get_devices();
```



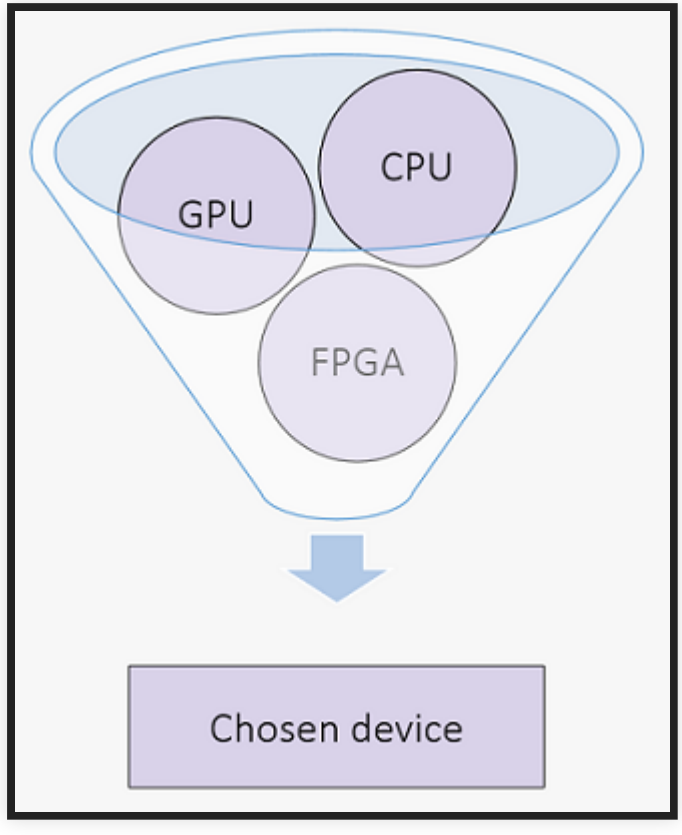
- The device class also provides the static function **get_devices**
  - It retrieves a vector of all available devices in the system
- This includes the host device

QUERYING THE TOPOLOGY USING A DEVICE SELECTOR



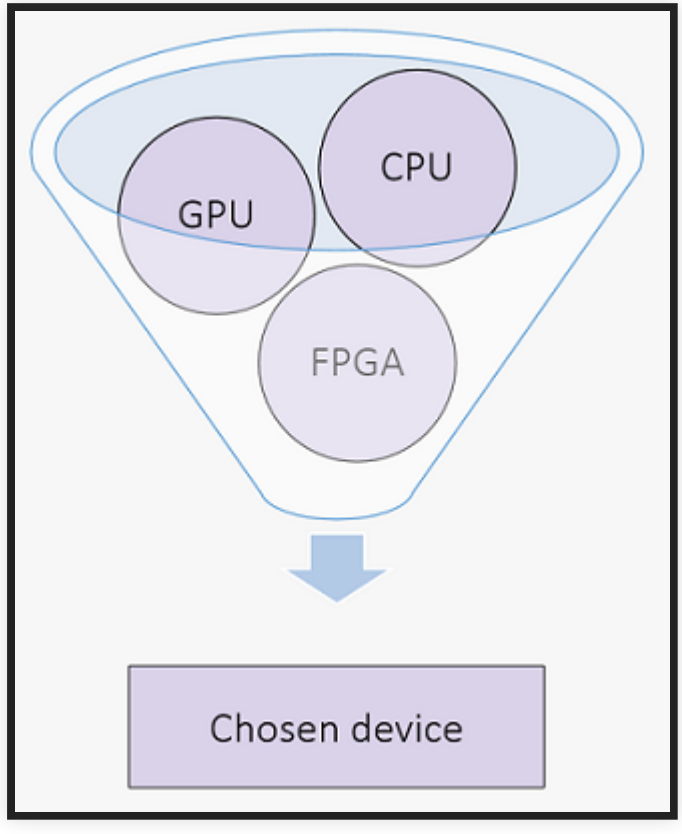
- To simplify the process of traversing the system topology SYCL provides device selectors
- A device selector is is a C++ function object, derived from the **device_selector** class, which defines a heuristic for scoring devices
- SYCL provides a number of standard device selectors, e.g. **default_selector**, **gpu_selector**, etc
- Users can also create their own device selectors

```
auto gpuSelector = gpu_selector{};
auto gpuDevice = gpuSelector.select_device();
```



- The **device_selector** class provides the member function **select_device**
  - Queries all devices and returns the one with the highest "score"
- A device with a negative score will never be chosen

```
auto defSelector = default_selector{};
auto chosenDevice = defSelector.select_device();
```



- The **default_selector** is a standard device selector type
  - Chooses a device based on an implementation defined heuristic

CUSTOM DEVICE SELECTORS

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

struct gpu_selector : public device_selector {

  int operator()(const device& dev) const override {

  }
};

int main(int argc, char *argv[]) {


}
```

- A device selector must inherit from the **device_selector** class
- A device selector must have a function call operator which takes a reference to a device

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

struct gpu_selector : public device_selector {

  int operator()(const device& dev) const override {
    if (dev.is_gpu()){
      return 1;
    }
    else {
      return -1;
    }
  }
};

int main(int argc, char *argv[]) {


}
```

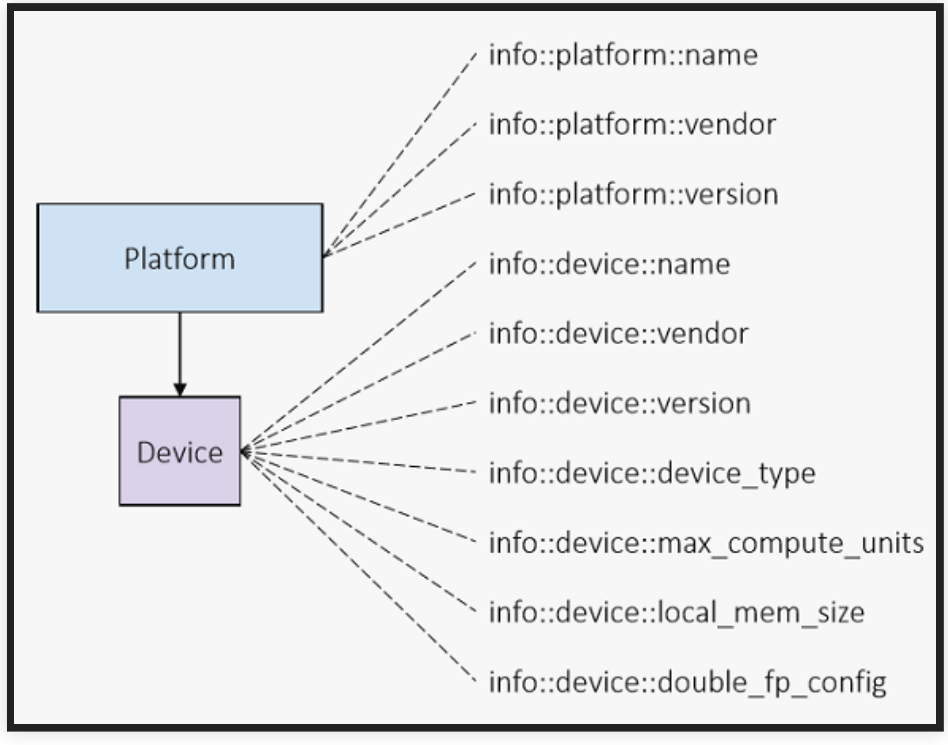
- The body of the function call operator defines the heuristic for selecting devices
- This is where you write the logic for scoring each device

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

struct gpu_selector : public device_selector {

  int operator()(const device& dev) const override {
    if (dev.is_gpu()){
          return 1;
        }
        else {
          return -1;
        }



  }
};

int main(int argc, char *argv[]) {
  auto gpuQueue = queue{gpu_selector{}};

}
```

- Now that there is a device selector that chooses a specific device we can use that to construct a queue

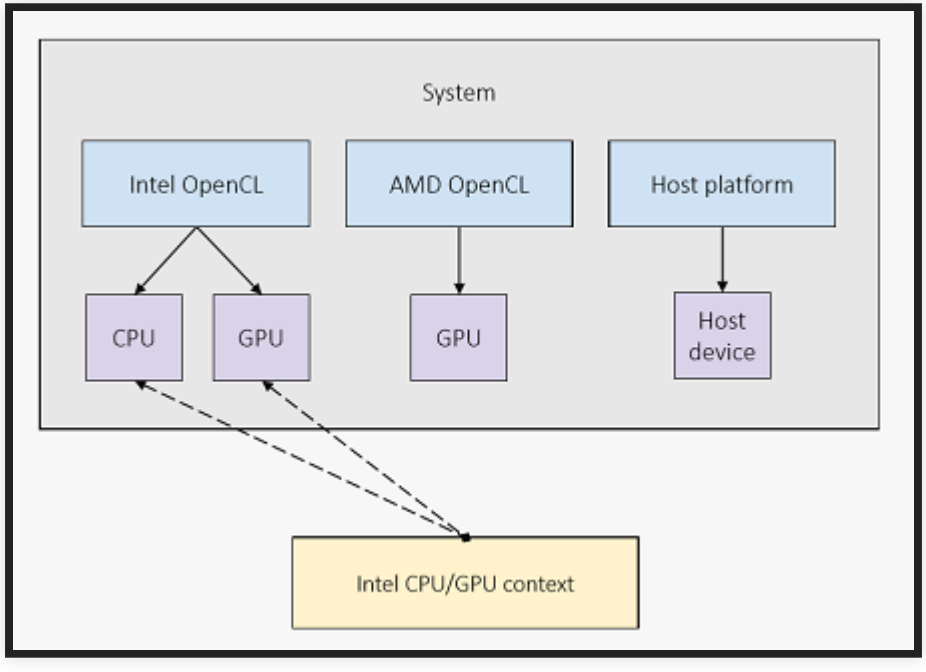
SELECTING A SYCL DEVICE

```
auto plt = dev.get_platform();
auto platformName = dev.get_info<info::device::name
```

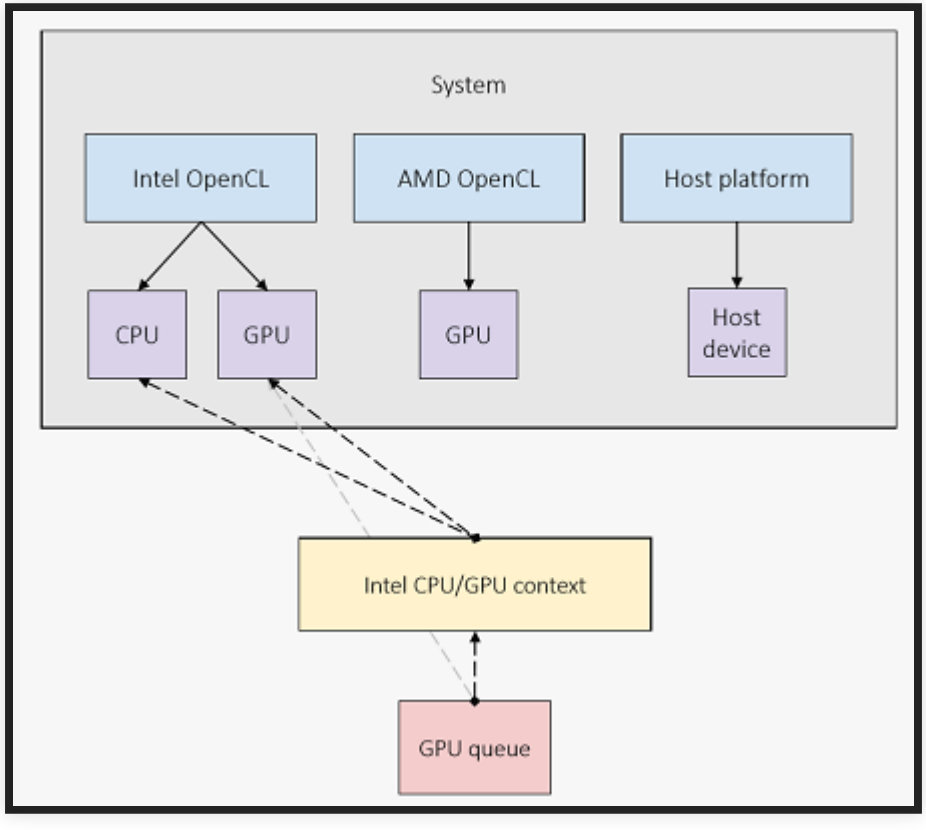


- Information about platforms and devices can be queried using the template member function **get_info**
- The info that you are querying is specified by the template parameter
- You can also query a device for its associated platform with the **get_platform** member function

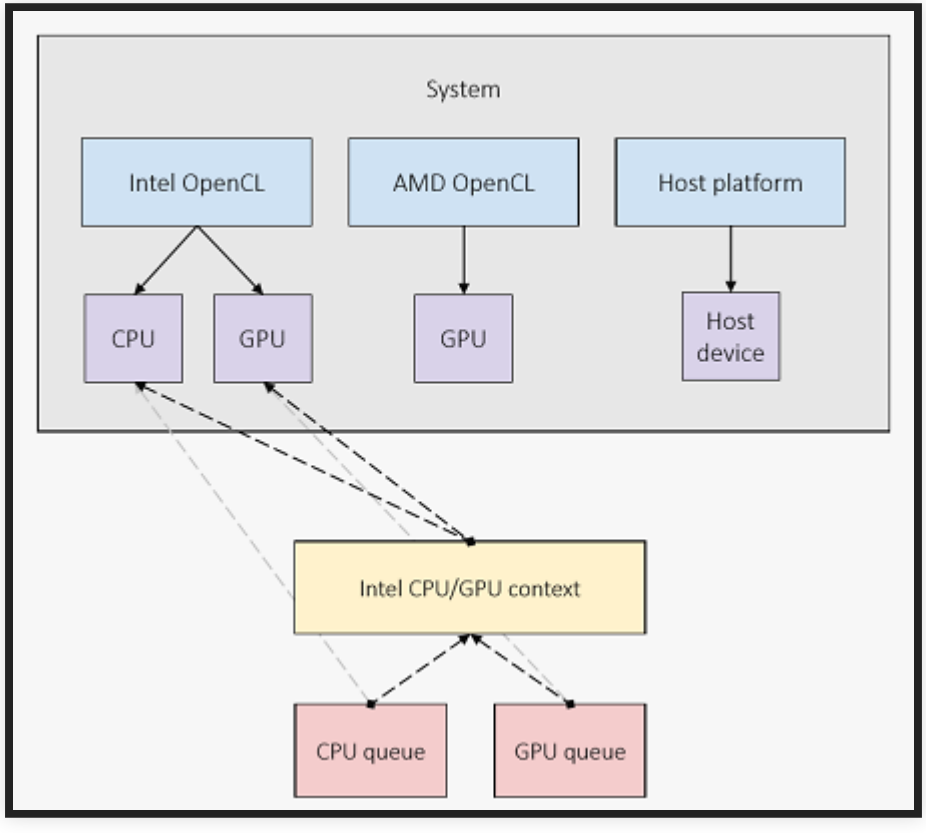
WHAT IS A SYCL QUEUE



- In SYCL the underlying execution and memory resources of a platform and its devices is managed by creating a context
- A context represents one or more devices, but all devices must be associated with the same platform



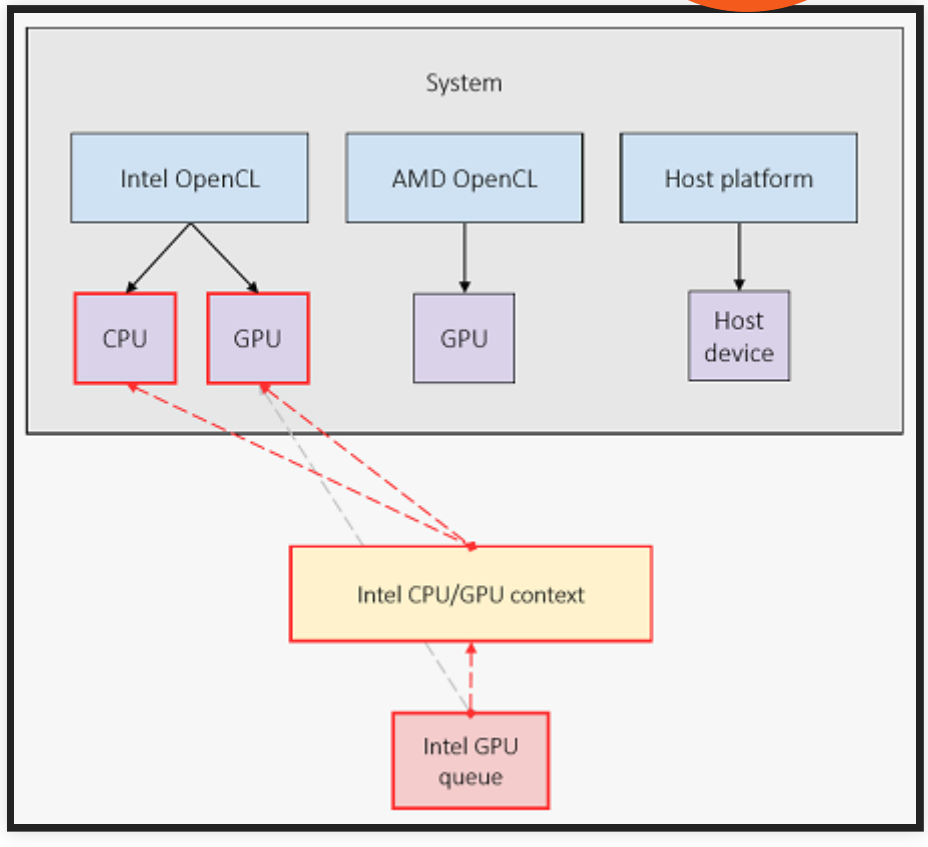
- In SYCL the object which is used to submit work is the queue
- A queue processes command groups and submits commands to the SYCL scheduler for a particular context and device



- A single SYCL application will often want to target multiple different devices
- This can be useful for task level parallelism and load balancing

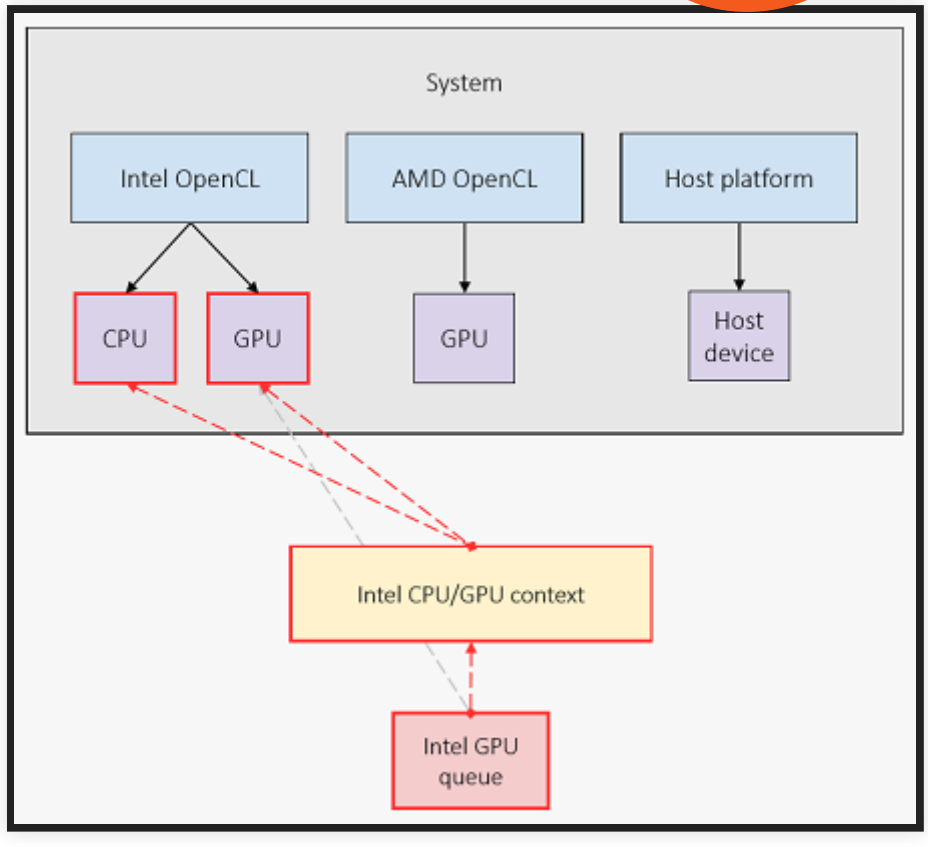
CREATING A QUEUE

```
auto defaultQueue = queue{};
```



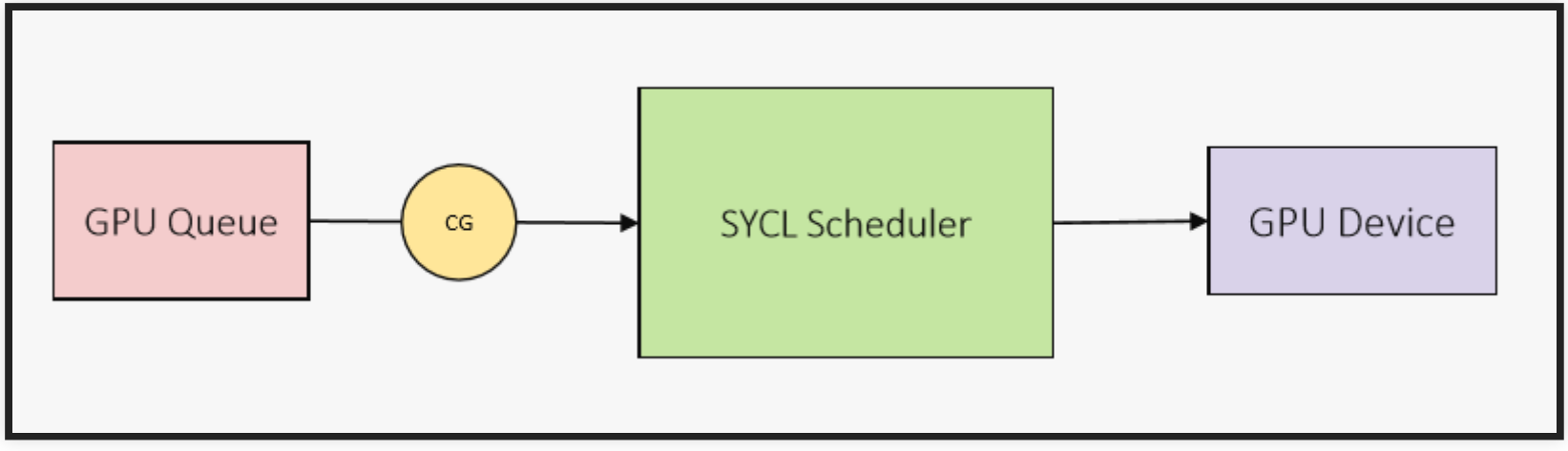
- A default constructed queue object will use the **default_selector** to choose a device and create an implicit context

```
auto intelGPUSelector = intel_gpu_selector{};
auto intelGPUQeuue = queue{intelGPUSelector};
```

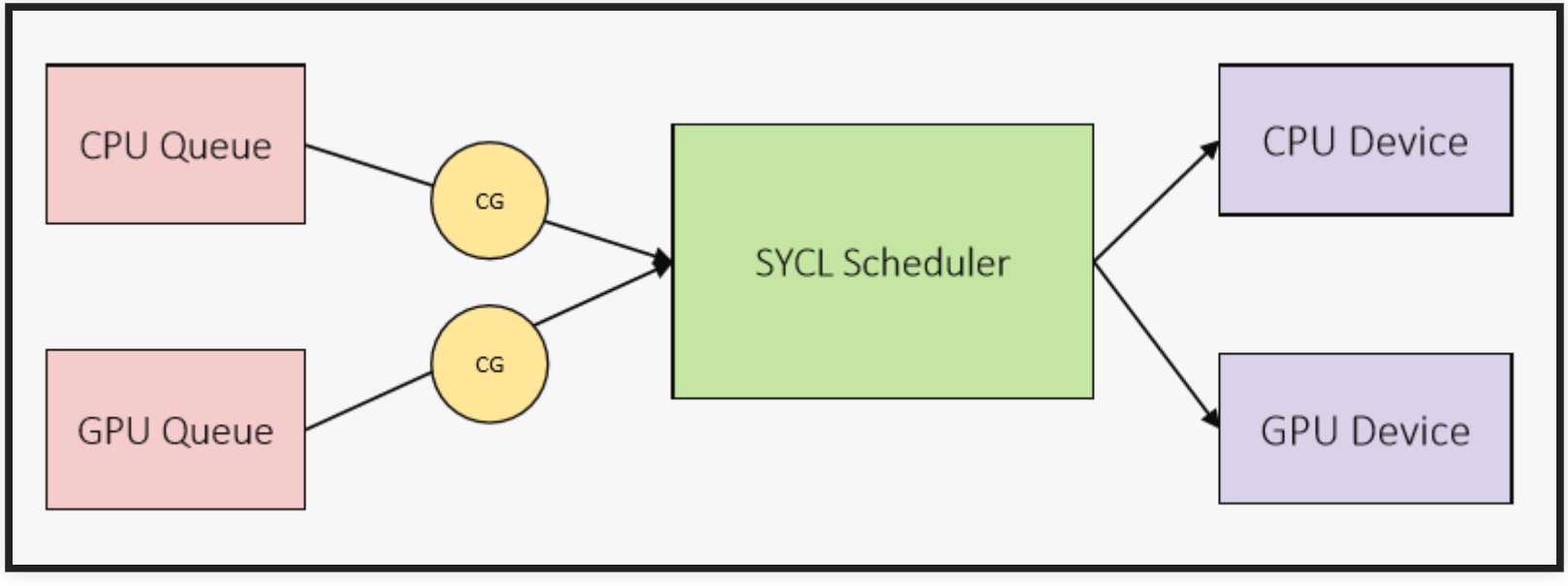


- A queue object can be constructed from a device selector which is used to choose a device and create an implicit context

SUBMITTING WORK TO A QUEUE



- In SYCL work is submitted via a queue object
- This is done using the **submit** member function
- This will process the command group and submit commands to the SYCL scheduler for the context and device associated with the queue



- The same scheduler is used for all queues in order to share dependency information

```
#include <CL/sycl.hpp> using namespace cl::sycl;
int main(int argc, char *argv[]) {
  queue gpuQueue(gpu_selector{});
  gpuQueue.submit([&](handler &cgh){

    // Command group

  });


}
```

- The **submit** member function takes a C++ function object, which takes a reference to a **handler** object
- The function object can be a lambda or a class with a function call operator
- The body of the function object represents the command group that is being submitted
- The handler object is created by the SYCL runtime and is used to link commands and requirements declared inside the command group

```
#include <CL/sycl.hpp> using namespace cl::sycl;
int main(int argc, char *argv[]) {
  queue gpuQueue(gpu_selector{});
  gpuQueue.submit([&](handler &cgh){

    // Command group

  });


}
```

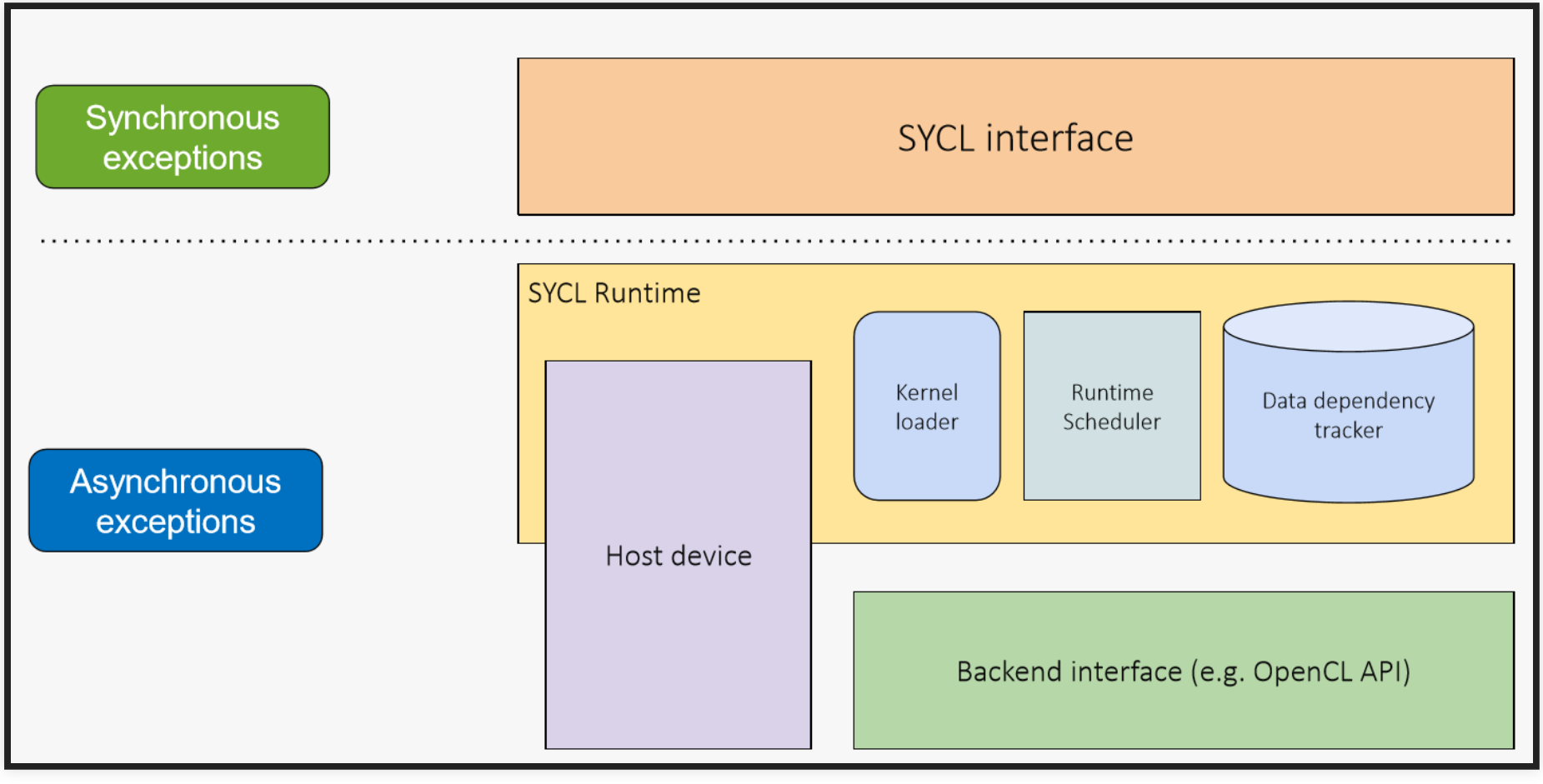
- The command group is processed exactly once when **submit** is called
- At this point all the commands and requirements declared inside the command group are collected together, processed and passed on to the scheduler
- The work is then enqueued to the device asynchronously by the SYCL scheduler, potentially in another thread

```
#include <CL/sycl.hpp> using namespace cl::sycl;
int main(int argc, char *argv[]) {
  queue gpuQueue(gpu_selector{});
  gpuQueue.submit([&](handler &cgh){

    // Command group

  });

  gpuQueue.wait();

}
```

- The queue object will not wait for work to complete on destruction
- There are other ways to wait for work to complete if you have data dependencies
- But it's often useful to be able to explicitly wait on a queue to complete any outstanding work

HANDLING ERRORS IN SYCL

- 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 runtime)



```
#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 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

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

QUESTIONS