INTRODUCTION TO SYCL

SYCL and the SYCL logo are trademarks of
the Khronos Group Inc.

LEARNING OBJECTIVES

- Learn about the SYCL 1.2.1 specification and it's implementations
- Learn about the major features that SYCL provides
- Learn about the components of a SYCL implementation
- Learn about the anatomy of a typical SYCL application
- Learn where to find useful resources for SYCL

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

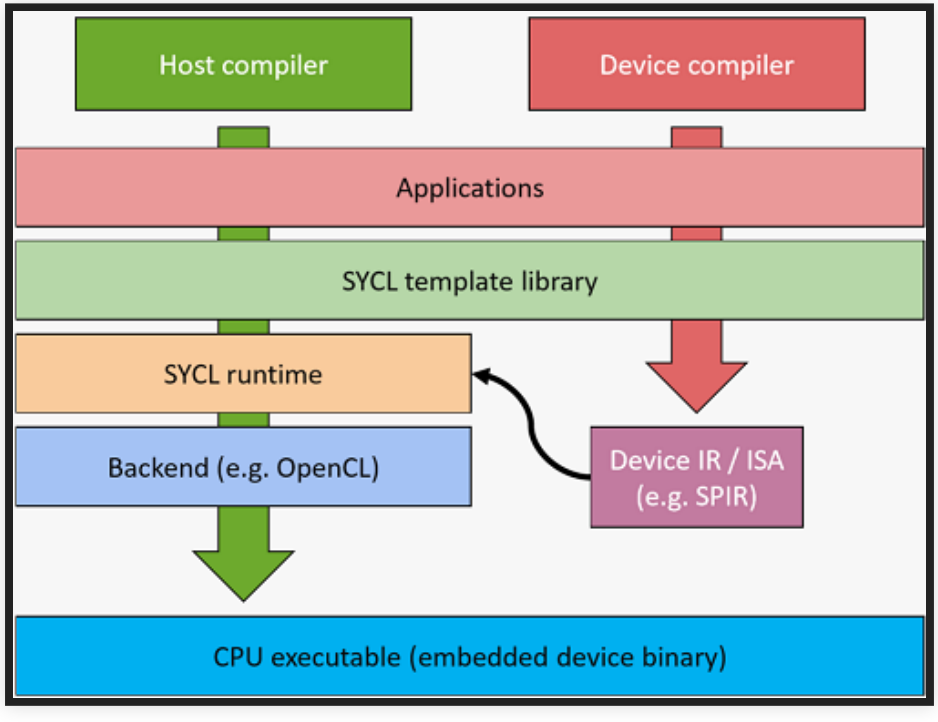
WHAT IS SYCL?

SYCL and the SYCL logo are trademarks of
the Khronos Group Inc.

SYCL is a single source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

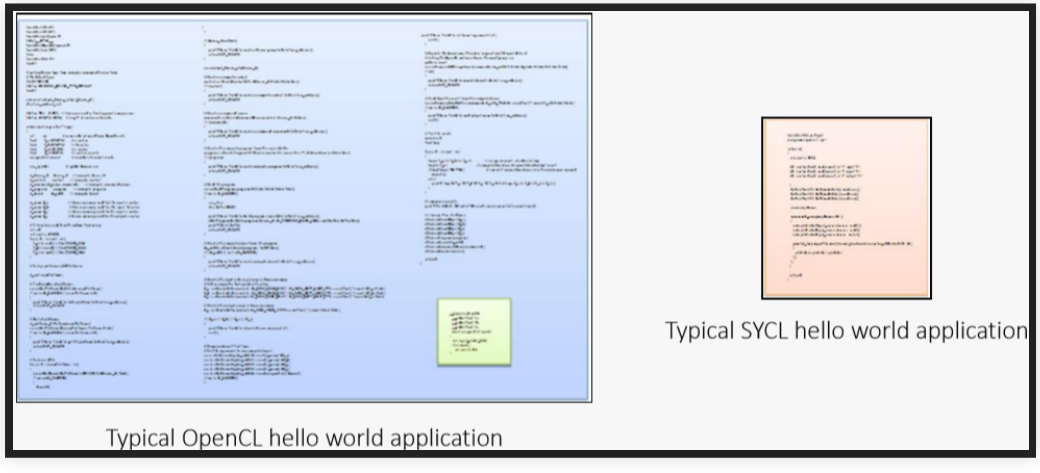
SYCL is a **single source**, high-level, standard C++ programming model, that can target a range of heterogeneous platforms



- SYCL allows you to write both host CPU and device code in the same C++ source file
- This requires two compilation passes; one for the host code and one for the device code

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

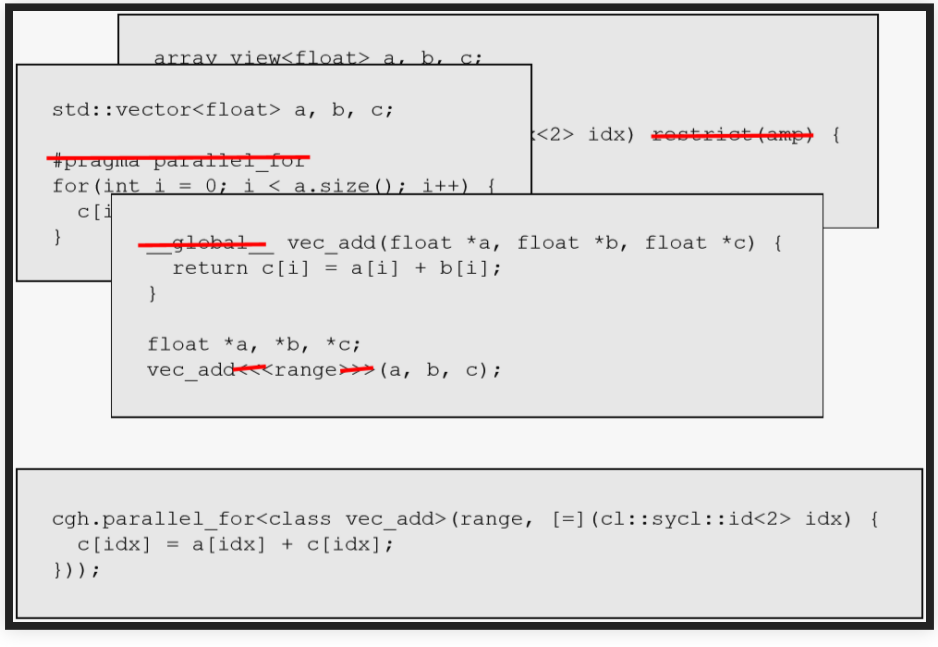
SYCL is a single source, **high-level**, standard C++ programming model, that can target a range of heterogeneous platforms



- SYCL provides high-level abstractions over common boilerplate code
  - Platform/device selection
  - Buffer creation
  - Kernel compilation
  - Dependency management and scheduling

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

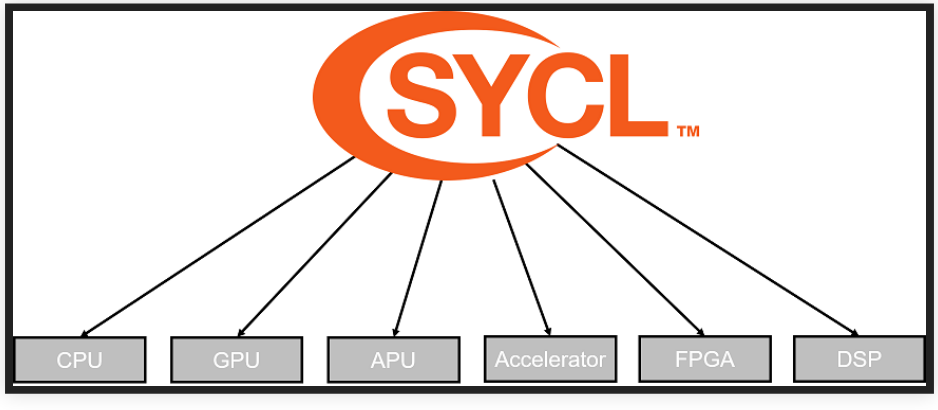
SYCL is a single source, high-level **standard C++** programming model, that can target a range of heterogeneous platforms



- SYCL allows you to write standard C++
- Unlike the other implementations shown on the left there are:
  - No language extensions
  - No pragmas
  - No attributes

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

SYCL is a single source, high-level standard C++ programming model, that can **target a range of heterogeneous platforms**



- SYCL can target any device supported by its back-end
- SYCL can target a number of different backends

While the current specification is limited to OpenCL, some implementations are already supporting other non-OpenCL back-ends.

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

WHAT SYCL IMPLEMENTATIONS ARE AVAILABLE?

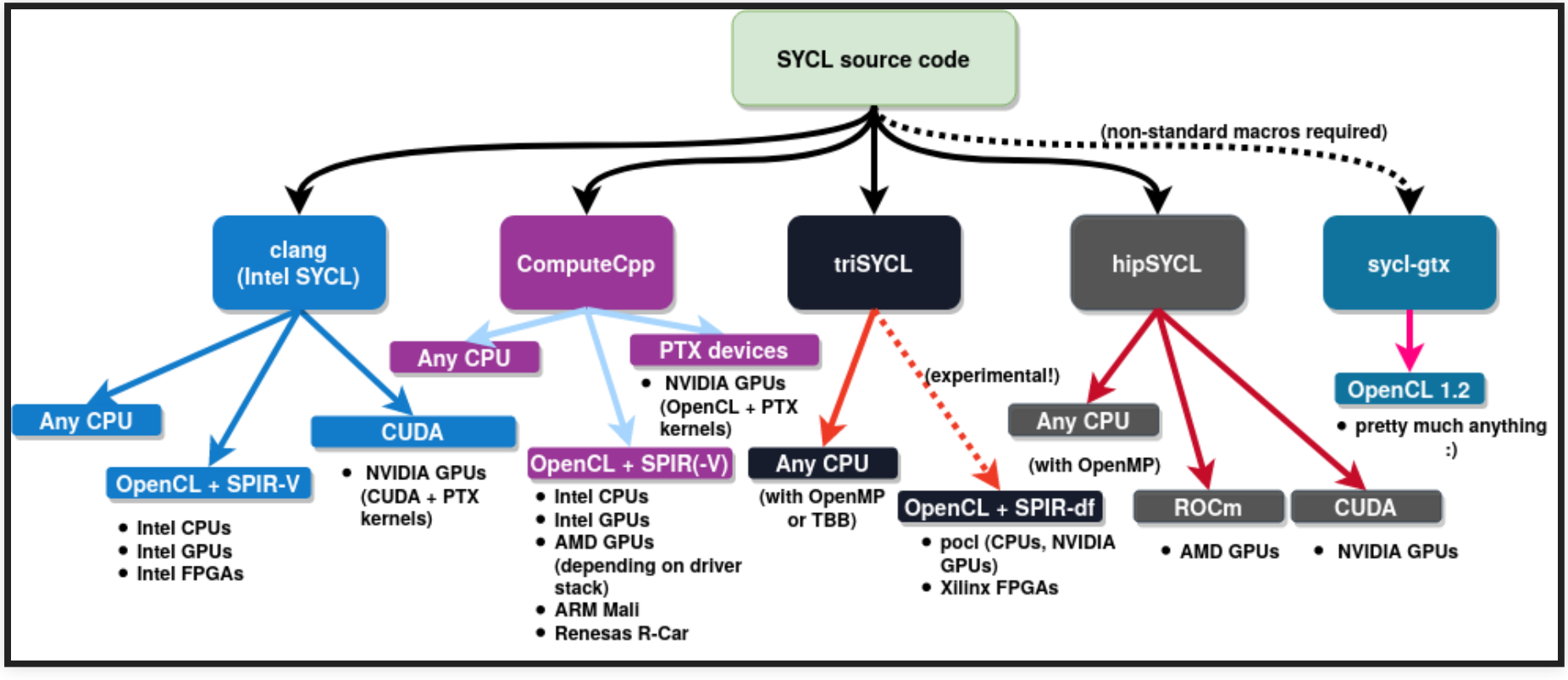
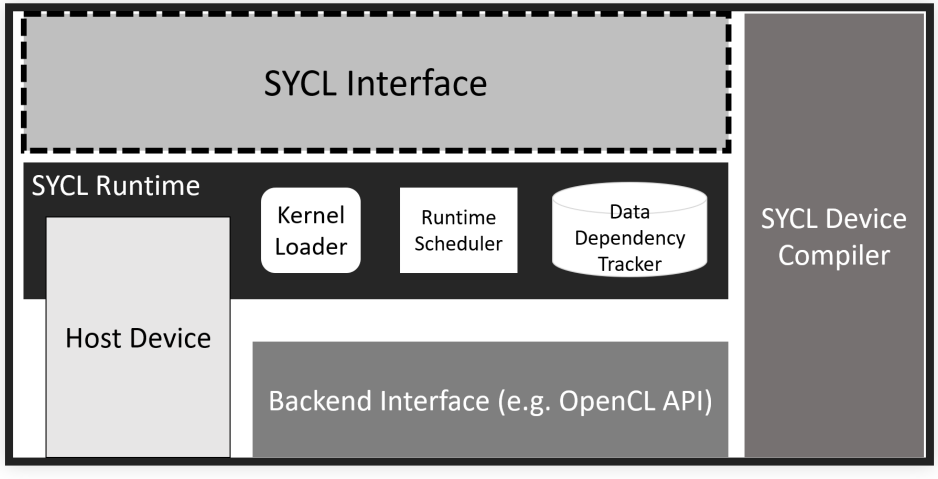


Image referenced from the hipSYCL project (https://github.com/illuhad/hipSYCL)

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

WHAT IS IN A SYCL IMPLEMENTATION?

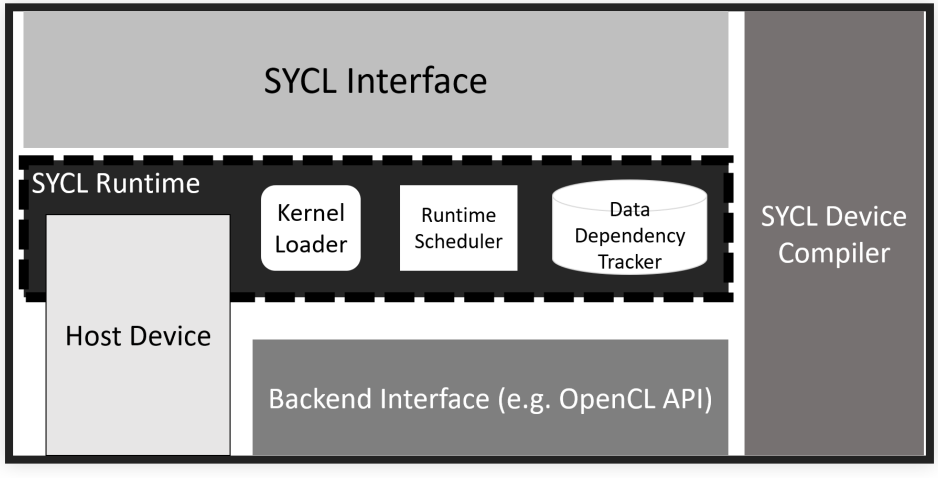
SYCL and the SYCL logo are trademarks of the Khronos Group Inc.



- The SYCL interface is a C++ template library that developers can use to access the features of SYCL
- The same interface is used for both the host and device code

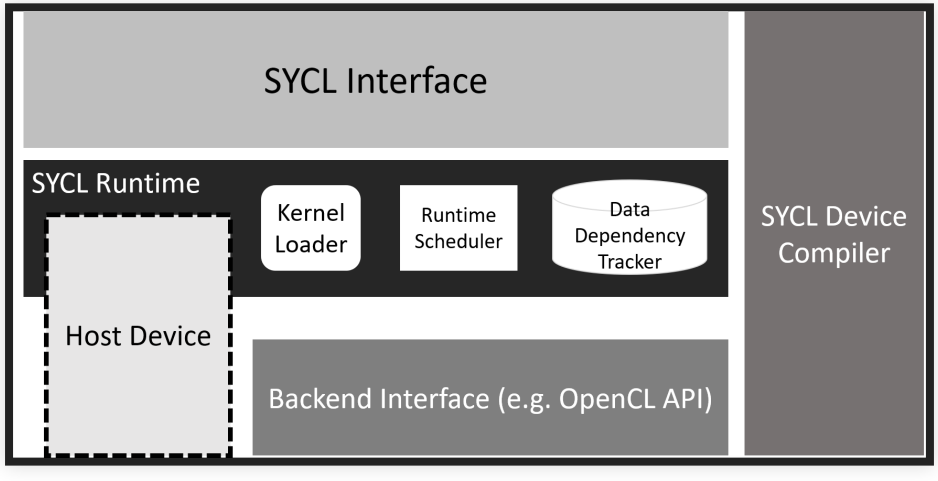
- The host is generally the CPU and is used to dispatch the parallel execution of kernels
- The device is the parallel unit used to execute the kernels, such as a GPU

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.



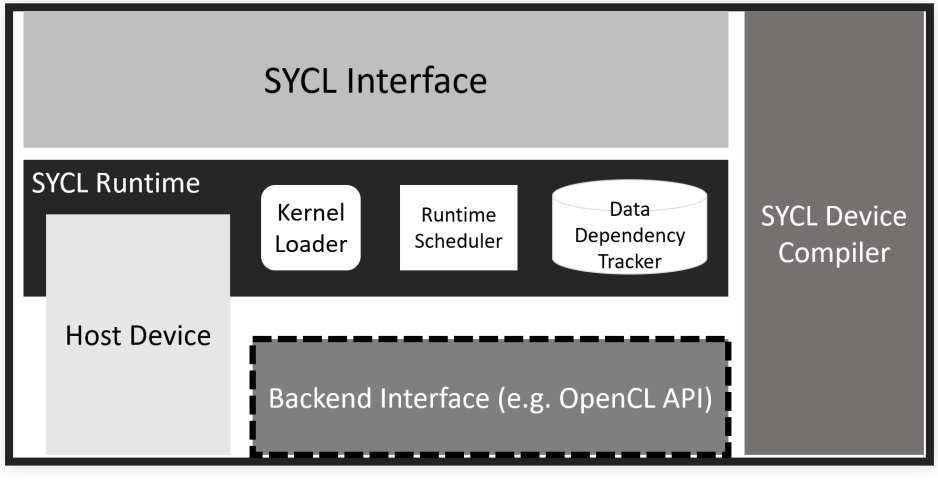
- The SYCL runtime is a library that schedules and executes work
  - It loads kernels, tracks data dependencies and schedules commands

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.



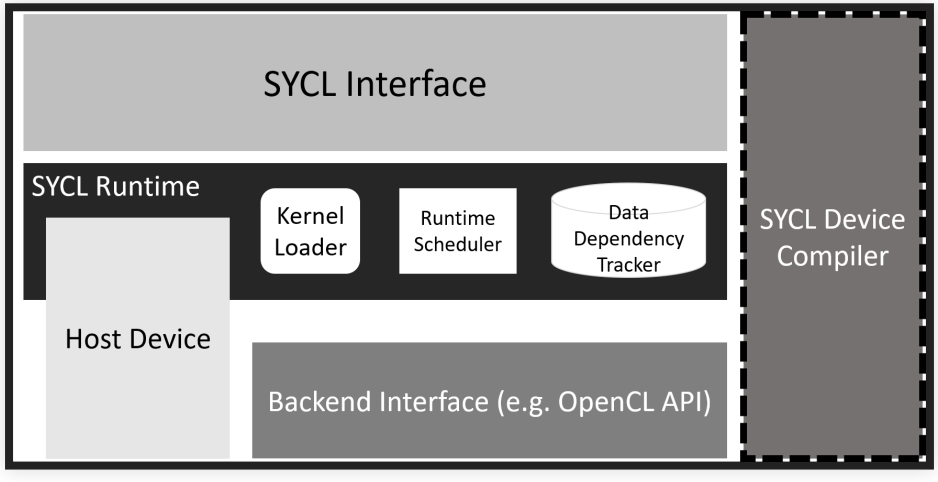
- The host device is an emulated backend that is executed as native C++ code and emulates the SYCL execution and memory model
- The host device can be used to execute kernels without backend drivers and for debugging purposes

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.



- The back-end interface is where the SYCL runtime calls down into a back-end in order to execute on a particular device
- The standard back-end is OpenCL but some implementations support other interfaces

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.



- The SYCL device compiler is a C++ compiler which can identify SYCL kernels and compile them down to an IR or ISA
  - This can be SPIR, SPIR-V, GCN, PTX or any proprietary vendor ISA

IR = Intermediate Representation **ISA** = Instruction Set Architecture

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

WHAT DOES A SYCL APPLICATION LOOK LIKE?

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

INCLUDE THE SYCL HEADER FILE

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

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

}
```

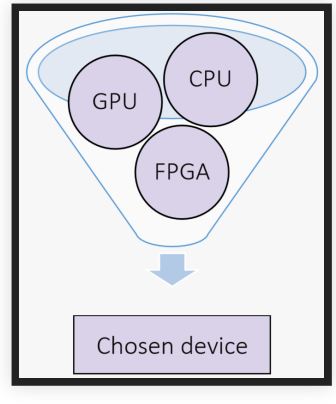
- First we include the SYCL header which contains the runtime API
- We also import the **cl::sycl** namespace here, this reduces the amount of code we need to write

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

DEVICE SELECTORS AND QUEUES

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

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

  queue gpuQueue{gpu_selector{}};

}
```

- Device selectors allow you to choose a device based on a custom configuration
- The queue default constructor uses a the default_selector, which allows the runtime to select a device for you



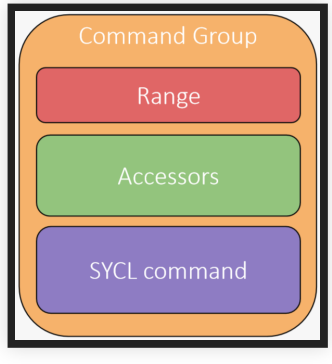
SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

QUEUES AND COMMAND GROUPS

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

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

  queue gpuQueue{gpu_selector{}};

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

  });
}
```

With a queue we can submit a command group; a command group contains

- A SYCL command (e.g. a SYCL kernel function)
- Execution range
- Accessors



SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

SET UP VECTORS

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

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};

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

  });
}
```

We initialize three vectors, two inputs (**dA**, **dB**) and an output (**dO**)

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

CREATE BUFFERS

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

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … }; // psuedo code

  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){

  });
}
```

We create a buffer for each vector to manage the data across host and device

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

BUFFERS ON LEAVING SCOPE

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

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};

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

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

    });
  }
}
```

Buffers synchronize on destruction via RAII So adding this scope means that all kernels writing to the buffers will wait and the data will be copied back to the vectors on leaving this scope

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

CREATE ACCESSORS

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

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQueue{gpu_selector{}};

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

        });
  }
}
```

We create an accessor for each of the buffers Read access for the two input buffers and write access for the output buffer

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

SYCL KERNEL

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

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  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]; });

      });
  }
```

- We define a SYCL kernel function for the command group using the parallel_for API
- The first argument here is a range, specifying the iteration space
- The second argument is a lambda function that represents the entry point for the SYCL kernel
- This lambda takes an id parameter that describes the current iteration being executed

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

TEMPLATE PARAMETER

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

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  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(cgh);
      auto inB = bufB.get_access(cgh);
      auto out = bufO.get_access(cgh);

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

    });
  }
```

- The template parameter to parallel_for is used to name the lambda
- The reason for this is that C++ does not have a standard ABI for lambdas so they are represented differently across the host and device compiler
- SYCL kernel functions follow C++ ODR rules, which means that if a SYCL kernel is in a template context, the kernel name needs to reflect that context, so must contain the same template arguments

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

ERROR HANDLING

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

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  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(cgh);
        auto inB = bufB.get_access(cgh);
        auto out = bufO.get_access(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 */ }
  }
}
```

- In SYCL errors are handled using exception handling, so you should always wrap SYCL code in a try-catch block
  - Some exceptions are thrown synchronously at the point of using a SYCL API
  - Other exceptions are asynchronous and are stored by the runtime and passed to an **async handler** when the queue is told to throw

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

WHERE TO GET STARTED WITH SYCL

- Visit https://www.khronos.org/sycl/ to find the latest SYCL specifications
- Checkout the documentation provided with one of the SYCL implementations.
- Visit https://sycl.tech to find out about all the SYCL implementations, news and videos

SYCL and the SYCL logo are trademarks of the Khronos Group Inc.

QUESTIONS

SYCL and the SYCL logo are trademarks of
the Khronos Group Inc.