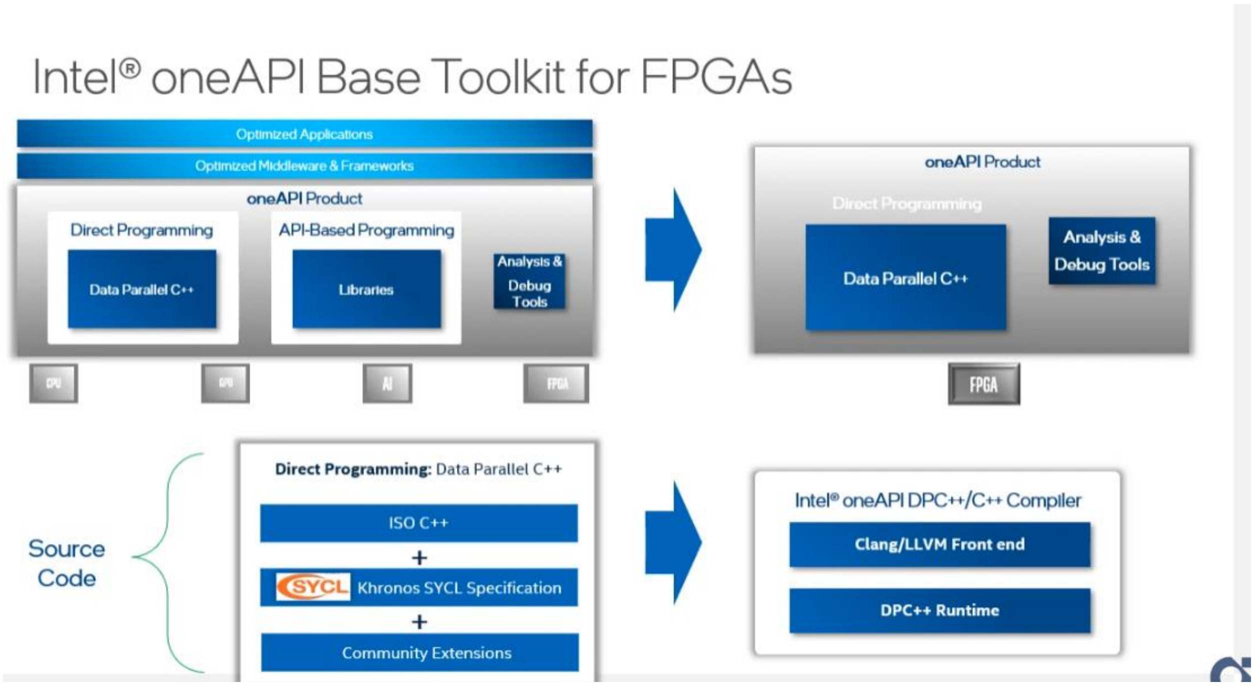
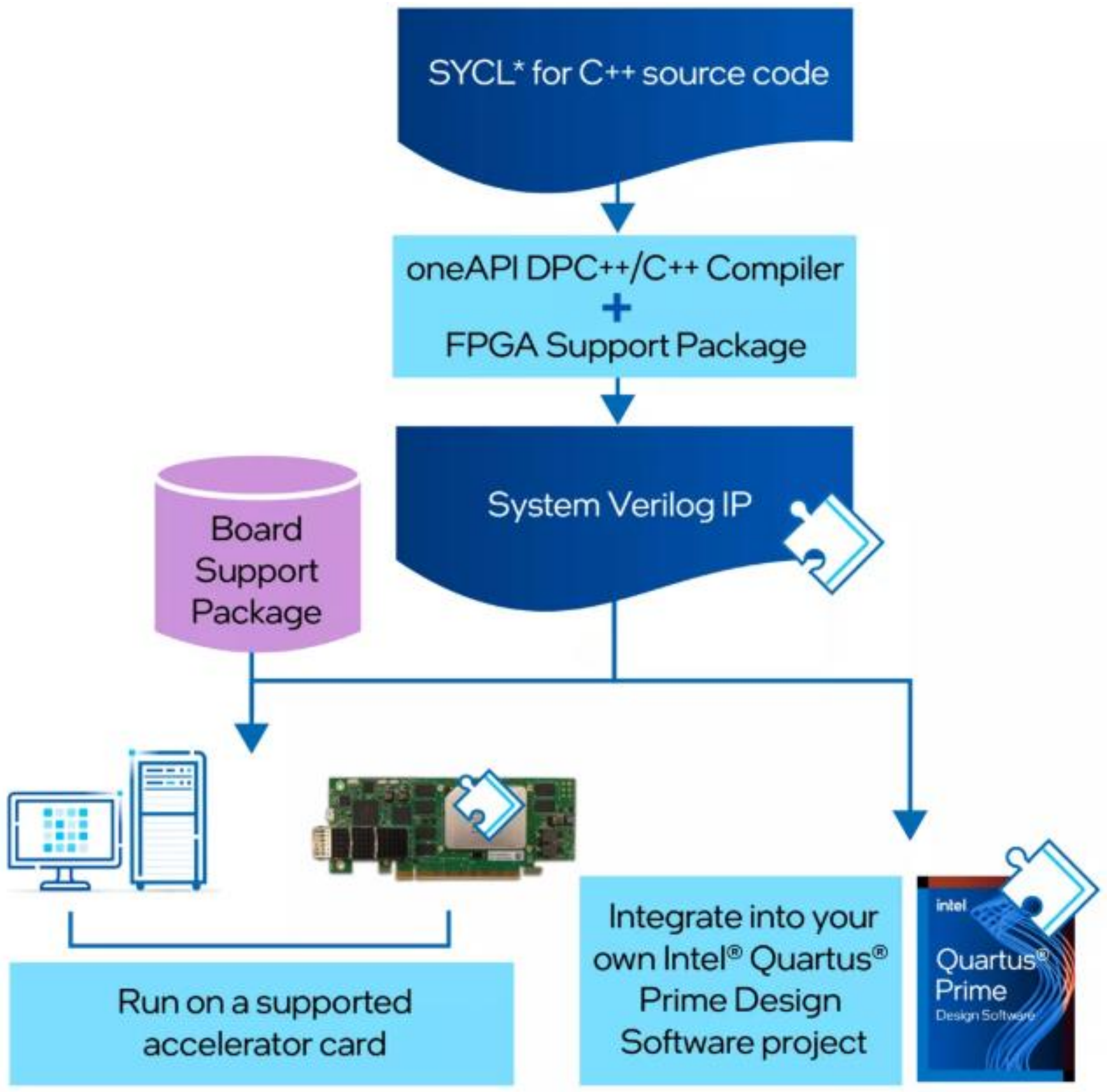
Here we are taking FPGA’s, compiling and running form CPU is straight forward and GPU is very similar too because the accelerator device which is the CPU and GPU are available in the host device but the FPGA are a little bit different

To understand the general structure of DPC++ you can refer to the image below

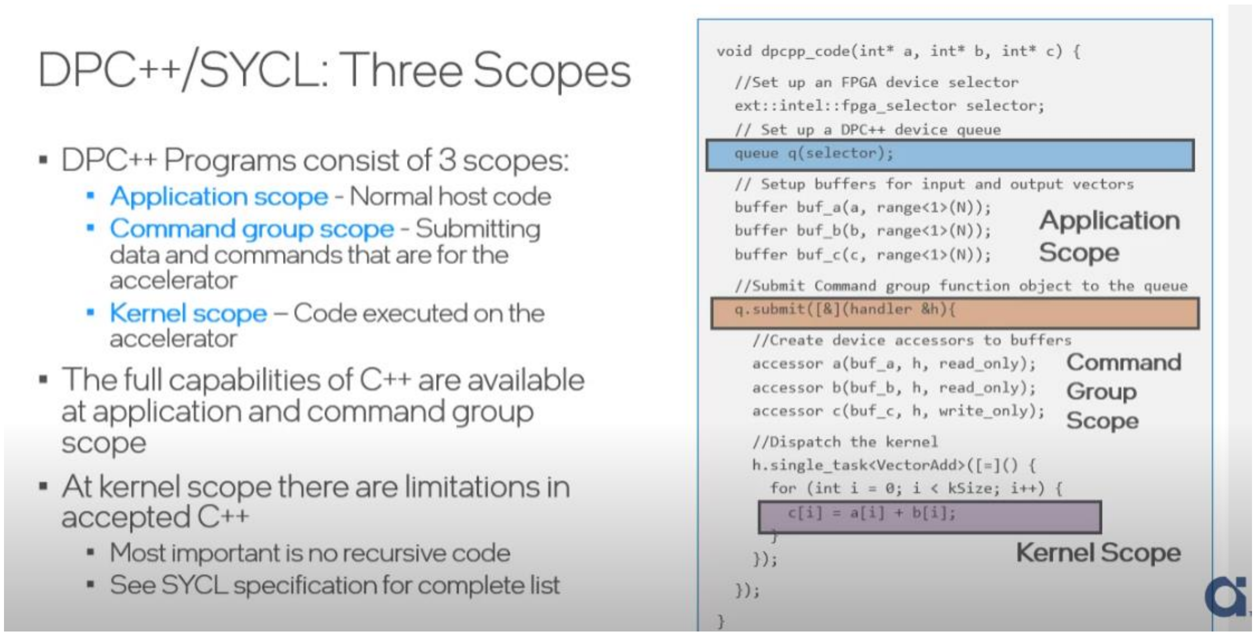



SYCL* for C++ source code
oneAPI DPC++/C++ Compiler
+
FPGA Support Package
System Verilog IP
Board Support Package
Run on a supported accelerator card
Integrate into your own Intel® Quartus® Prime Design Software project
intel
Quartus® Prime
Design Software

Some general information about DPC++/Sycl:

- DPC++ is based on C++ and Sycl, and Sycl is based on OpenCL
- Even though it meant to target xPU's but you do still need to 'tune'

In writing DPC++/Sycl code there are three main scopes to consider

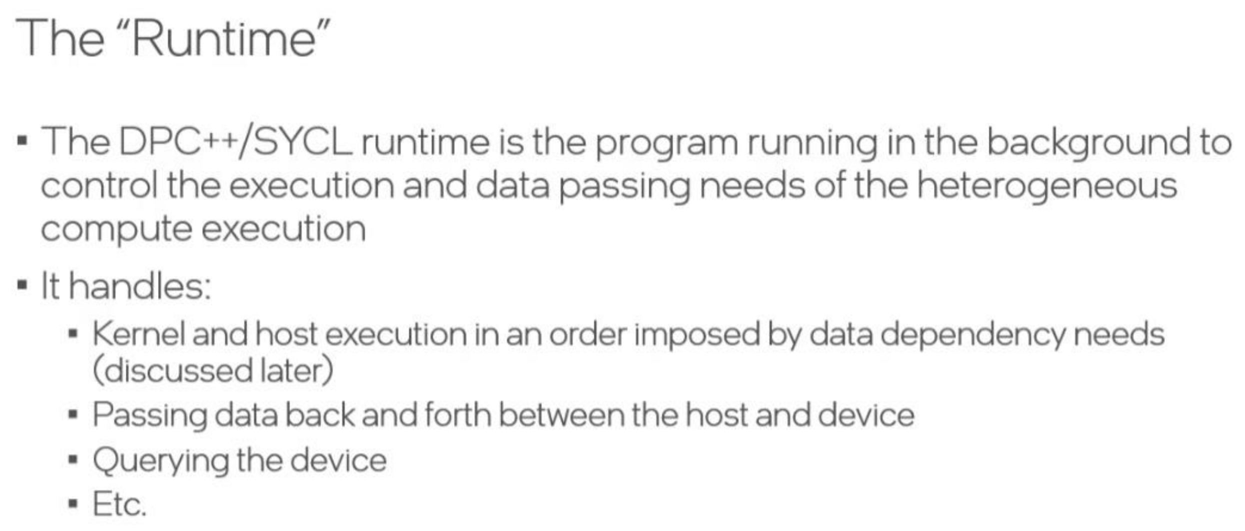


DPC++/SYCL: Three Scopes

- DPC++ Programs consist of 3 scopes:
  - Application scope - Normal host code
  - Command group scope - Submitting data and commands that are for the accelerator
  - Kernel scope – Code executed on the accelerator
- The full capabilities of C++ are available at application and command group scope
- At kernel scope there are limitations in accepted C++
  - Most important is no recursive code
  - See SYCL specification for complete list

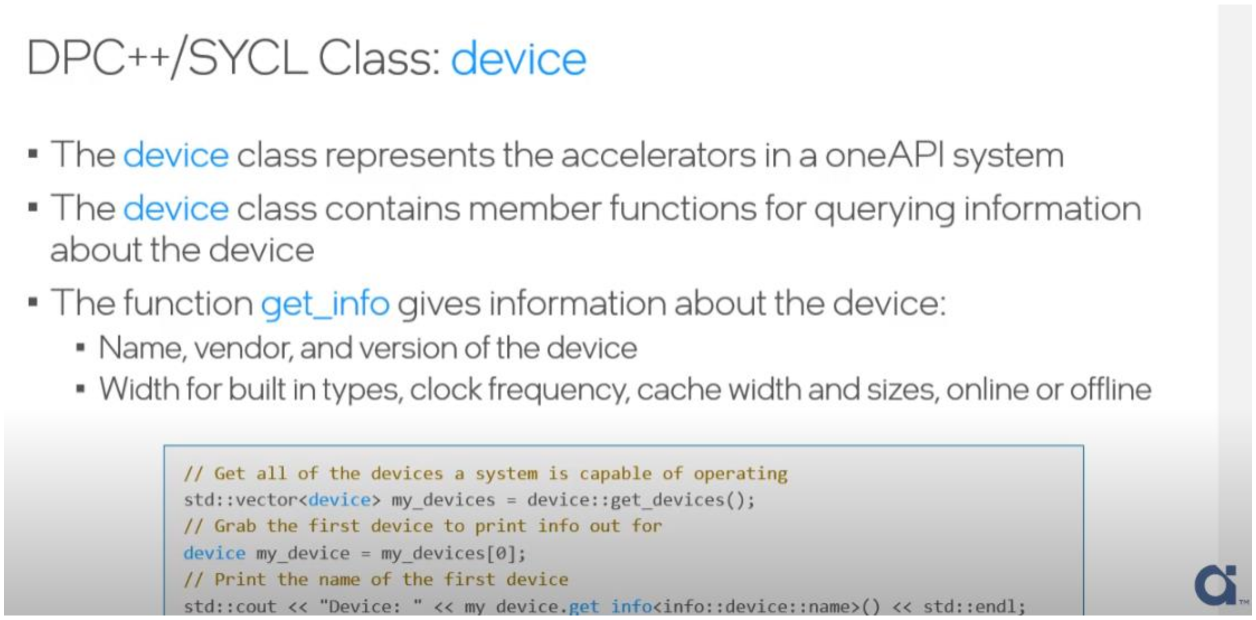
```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  ext::intel::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

Application Scope
Command Group Scope
Kernel Scope



The "Runtime"

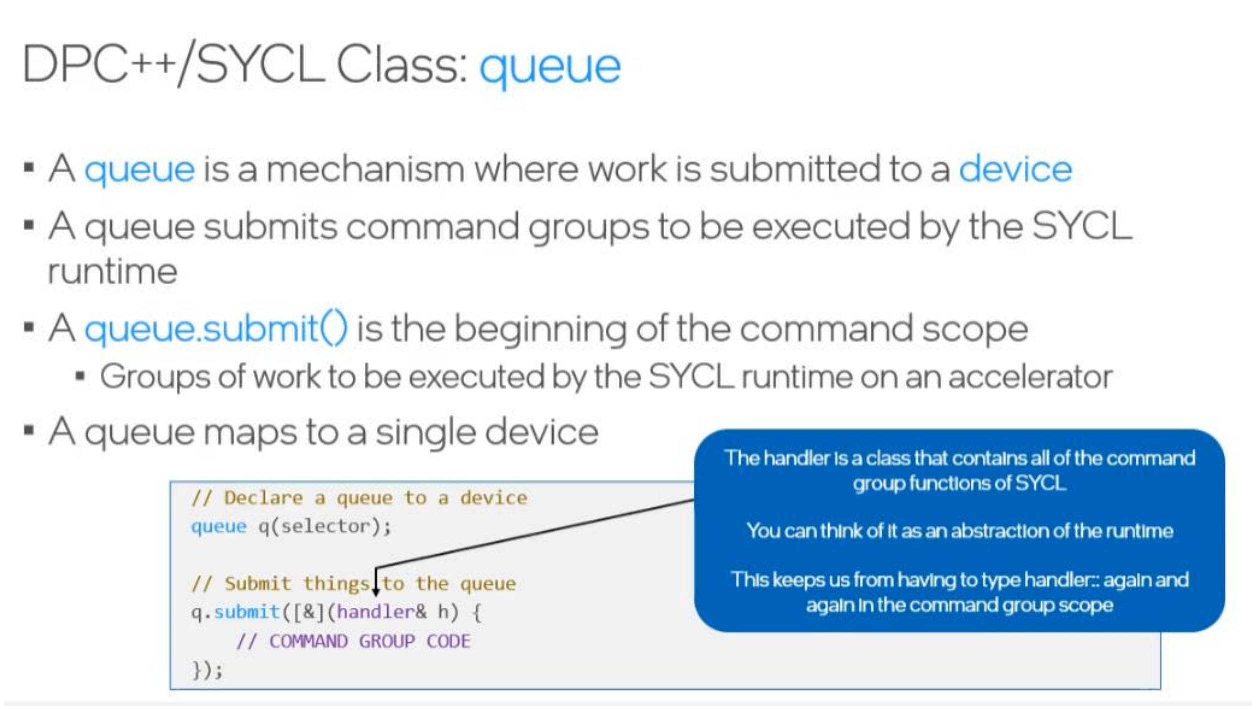
- The DPC++/SYCL runtime is the program running in the background to control the execution and data passing needs of the heterogeneous compute execution
- It handles:
  - Kernel and host execution in an order imposed by data dependency needs (discussed later)
  - Passing data back and forth between the host and device
  - Querying the device
  - Etc.



DPC++/SYCL Class: device

- The device class represents the accelerators in a oneAPI system
- The device class contains member functions for querying information about the device
- The function get_info gives information about the device:
  - Name, vendor, and version of the device
  - Width for built in types, clock frequency, cache width and sizes, online or offline

```
// Get all of the devices a system is capable of operating
std::vector<device> my_devices = device::get_devices();
// Grab the first device to print info out for
device my_device = my_devices[0];
// Print the name of the first device
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```



DPC++/SYCL Class: queue

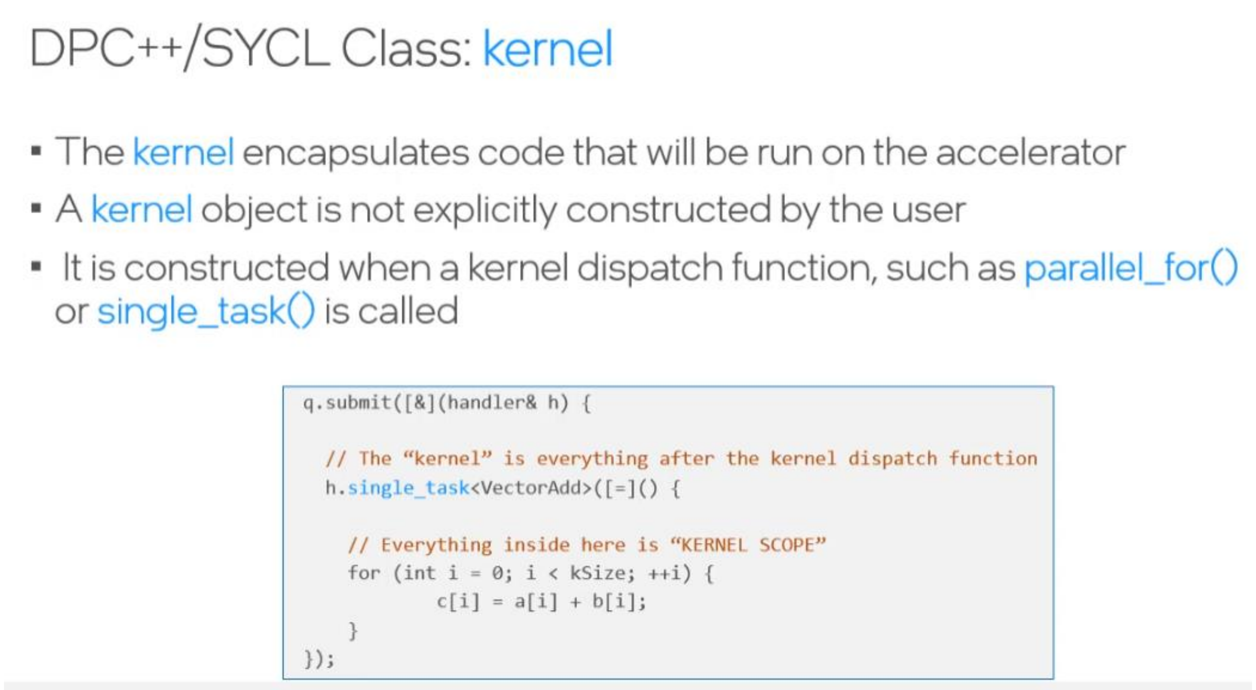
- A queue is a mechanism where work is submitted to a device
- A queue submits command groups to be executed by the SYCL runtime
- A queue.submit() is the beginning of the command scope
  - Groups of work to be executed by the SYCL runtime on an accelerator
- A queue maps to a single device

```
// Declare a queue to a device
queue q(selector);

// Submit things to the queue
q.submit([&](handler& h) {
    // COMMAND GROUP CODE
});
```

The handler is a class that contains all of the command group functions of SYCL

You can think of it as an abstraction of the runtime

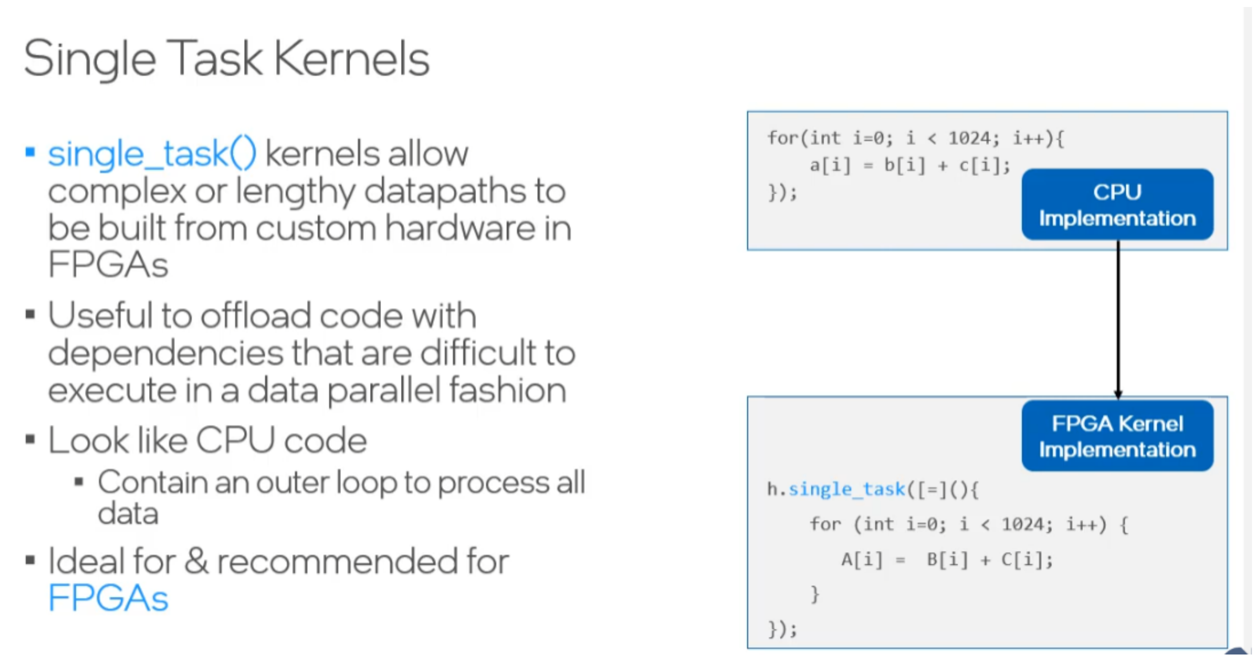
This keeps us from having to type handler:: again and again in the command group scope



DPC++/SYCL Class: kernel

- The kernel encapsulates code that will be run on the accelerator
- A kernel object is not explicitly constructed by the user
- It is constructed when a kernel dispatch function, such as parallel_for() or single_task() is called

```
q.submit([&](handler& h) {

  // The “kernel” is everything after the kernel dispatch function
  h.single_task<VectorAdd>([=]() {

    // Everything inside here is “KERNEL SCOPE”
    for (int i = 0; i < kSize; ++i) {
            c[i] = a[i] + b[i];
    }
});
```

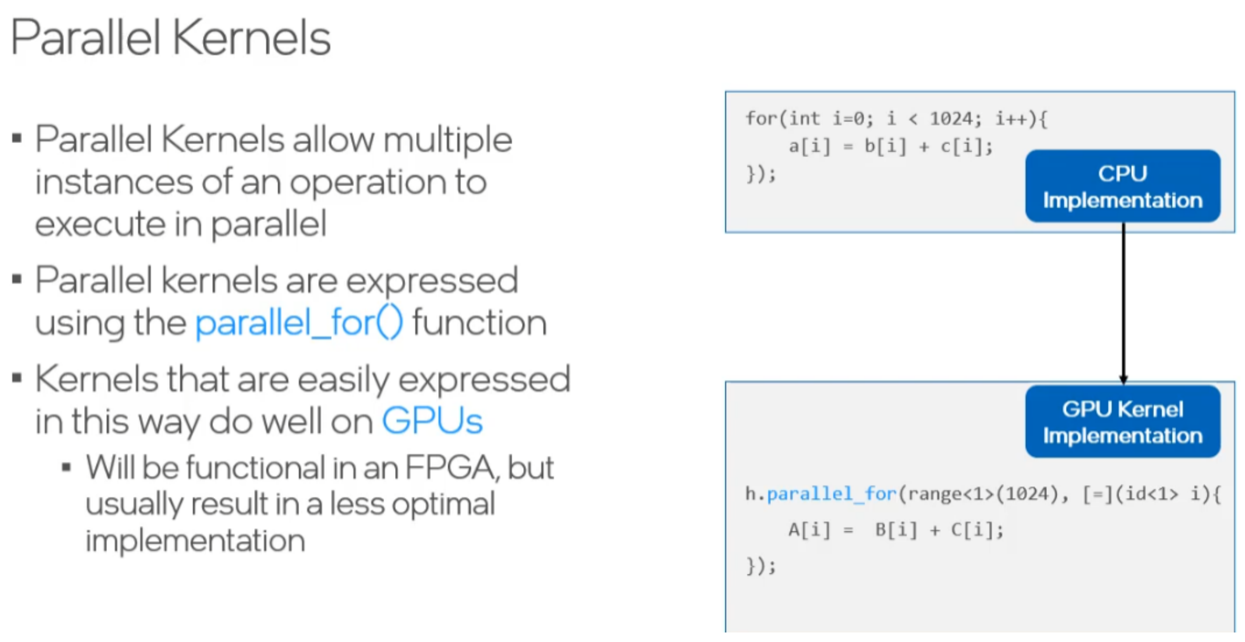


Single Task Kernels

- single_task() kernels allow complex or lengthy datapaths to be built from custom hardware in FPGAs
- Useful to offload code with dependencies that are difficult to execute in a data parallel fashion
- Look like CPU code
  - Contain an outer loop to process all data
- Ideal for & recommended for FPGAs

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

```
h.single_task([=](){
    for (int i=0; i < 1024; i++) {
       A[i] =  B[i] + C[i];
    }
});
```



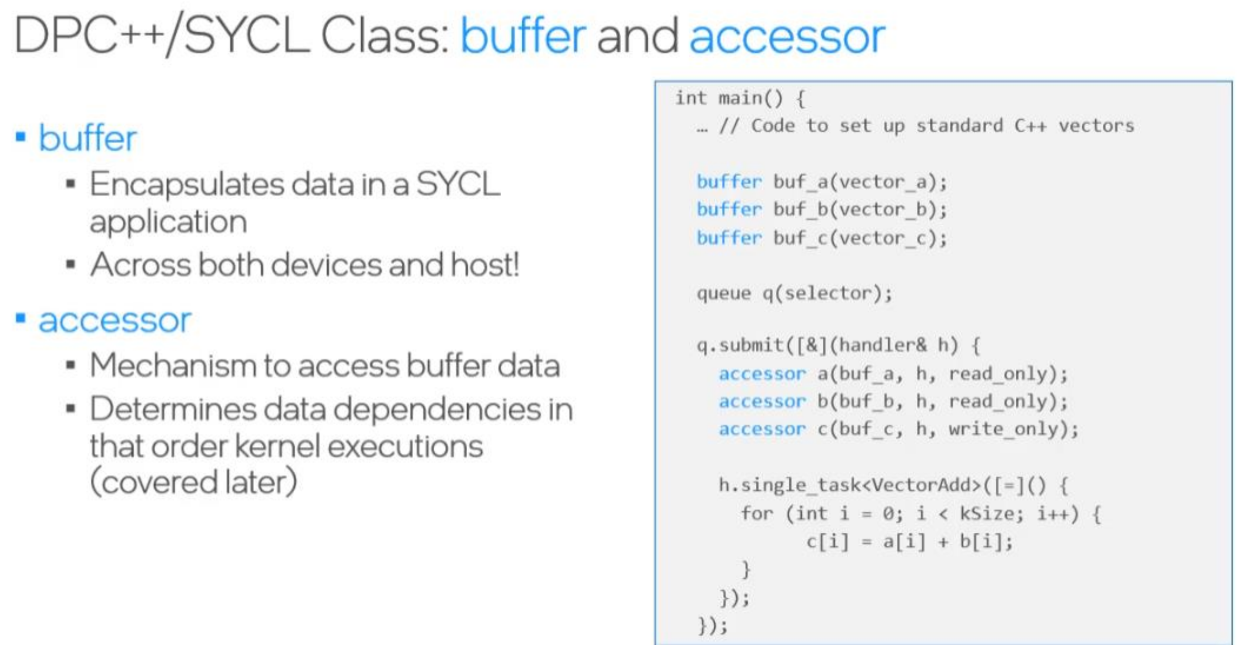
Parallel Kernels

- Parallel Kernels allow multiple instances of an operation to execute in parallel
- Parallel kernels are expressed using the parallel_for() function
- Kernels that are easily expressed in this way do well on GPUs
  - Will be functional in an FPGA, but usually result in a less optimal implementation

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

CPU Implementation
GPU Kernel Implementation

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    A[i] =  B[i] + C[i];
});
```

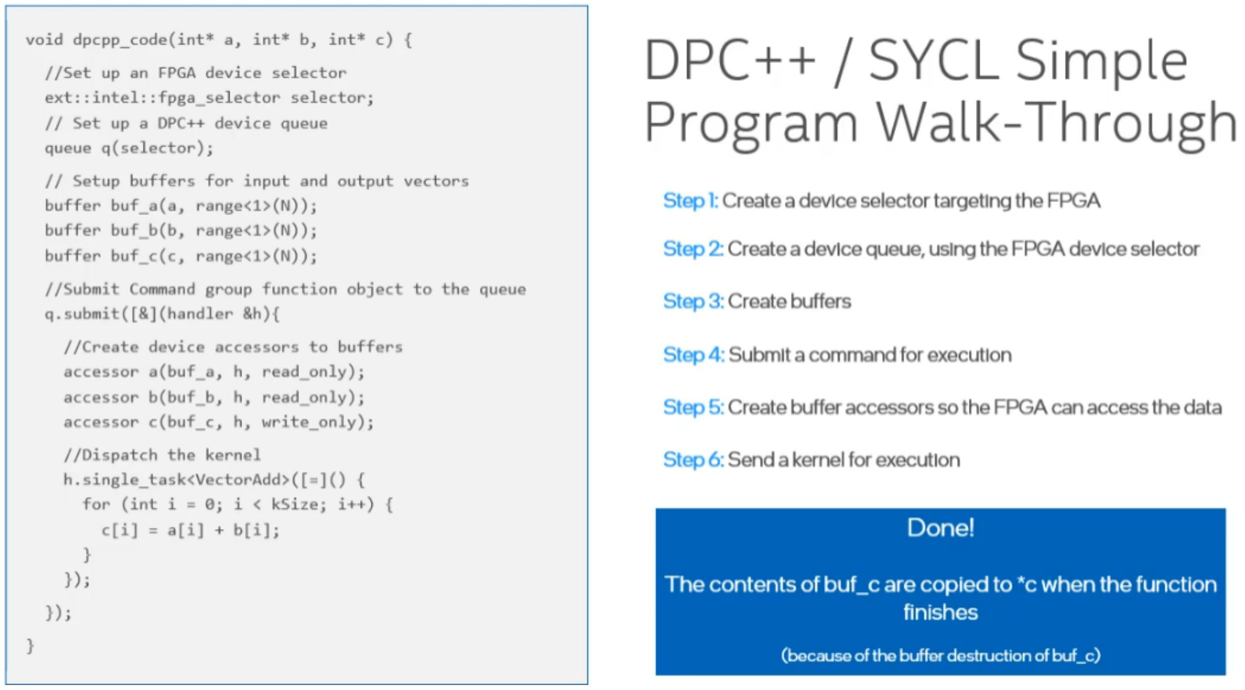


DPC++/SYCL Class: buffer and accessor

- buffer
  - Encapsulates data in a SYCL application
  - Across both devices and host!
- accessor
  - Mechanism to access buffer data
  - Determines data dependencies in that order kernel executions (covered later)

```
int main() {
  … // Code to set up standard C++ vectors

  buffer buf_a(vector_a);
  buffer buf_b(vector_b);
  buffer buf_c(vector_c);

  queue q(selector);

  q.submit([&](handler& h) {
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);

    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
            c[i] = a[i] + b[i];
      }
    });
  });
```

You have to include “sycl.hpp” and “fpga_extenstions.hpp”



```
void dpcpp_code(int* a, int* b, int* c) {
  //Set up an FPGA device selector
  ext::intel::fpga_selector selector;
  // Set up a DPC++ device queue
  queue q(selector);
  // Setup buffers for input and output vectors
  buffer buf_a(a, range<1>(N));
  buffer buf_b(b, range<1>(N));
  buffer buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
    //Create device accessors to buffers
    accessor a(buf_a, h, read_only);
    accessor b(buf_b, h, read_only);
    accessor c(buf_c, h, write_only);
    //Dispatch the kernel
    h.single_task<VectorAdd>([=]() {
      for (int i = 0; i < kSize; i++) {
        c[i] = a[i] + b[i];
      }
    });
  });
}
```

DPC++ / SYCL Simple Program Walk-Through

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Step 6: Send a kernel for execution

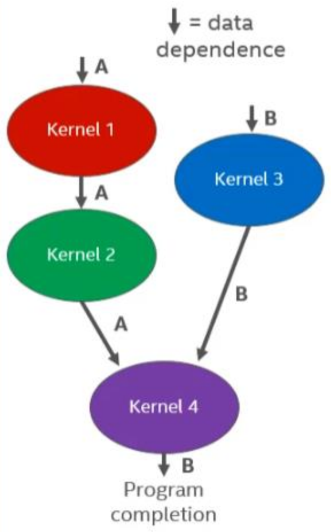
Done!

The contents of buf_c are copied to *c when the function finishes

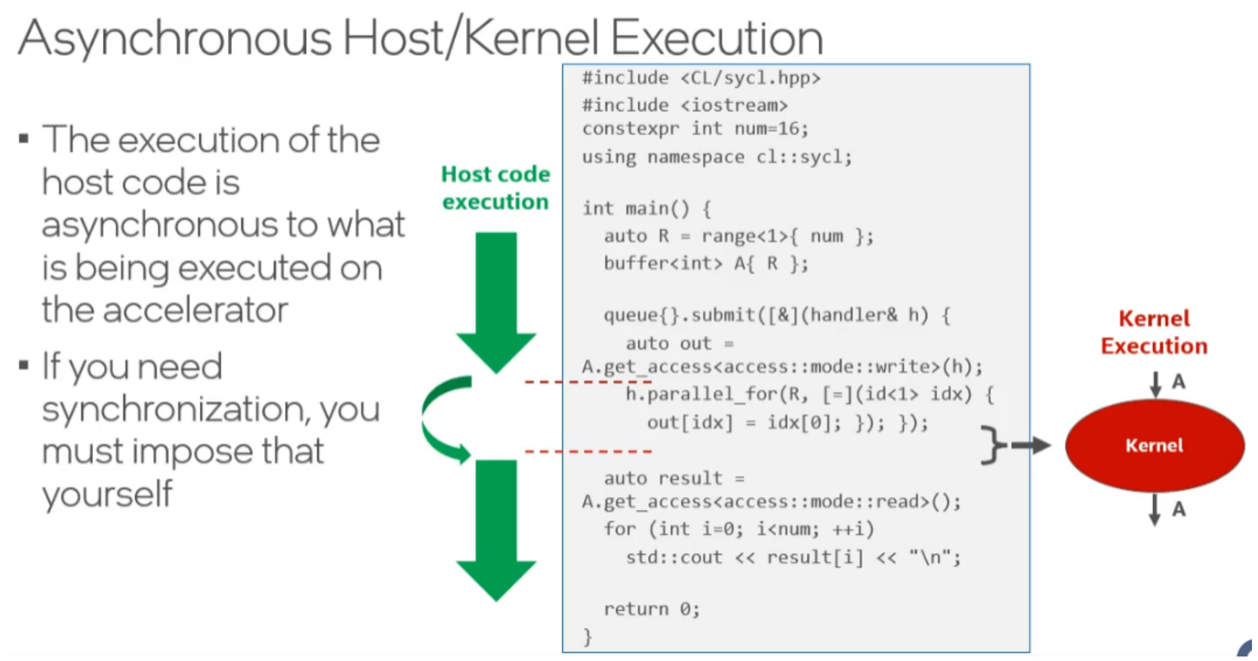
(because of the buffer destruction of buf_c)

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                         } Kernel 1

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                         } Kernel 2

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });                         } Kernel 3

  Q.submit([&](handler& h) {
  auto in = A.get_access<access::mode::read>(h);
  auto inout =
    B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      inout[idx] *= in[idx]; }); });                     } Kernel 4
}
```

Kernel Execution Order



- Kernels can execute at the same time
  - If no data dependencies
- Accessors are used to determine dependencies
- Execution ordering is automatically determined



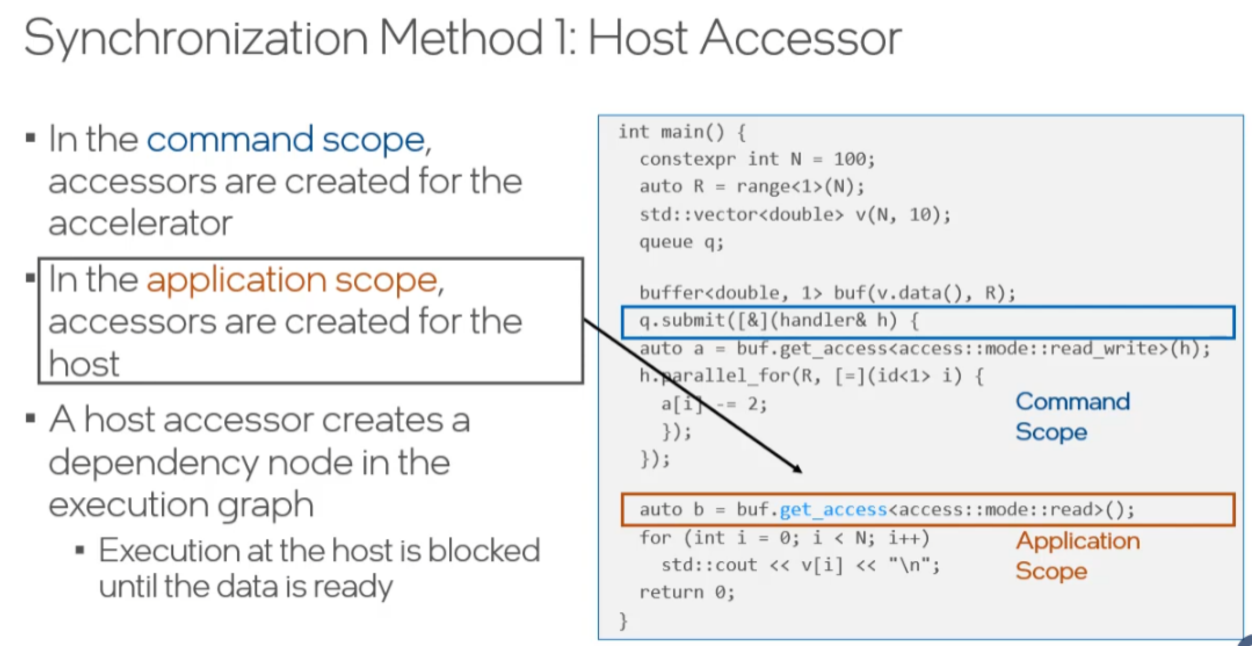
Asynchronous Host/Kernel Execution

- The execution of the host code is asynchronous to what is being executed on the accelerator
- If you need synchronization, you must impose that yourself

```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R };

  queue{}.submit([&](handler& h) {
    auto out =
A.get_access<access::mode::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

  auto result =
A.get_access<access::mode::read>();
  for (int i=0; i<num; ++i)
    std::cout << result[i] << "\n";

  return 0;
}
```

There are multiple ways to synchronize between the host and the kernel

You can make a dummy dependency in the application scope

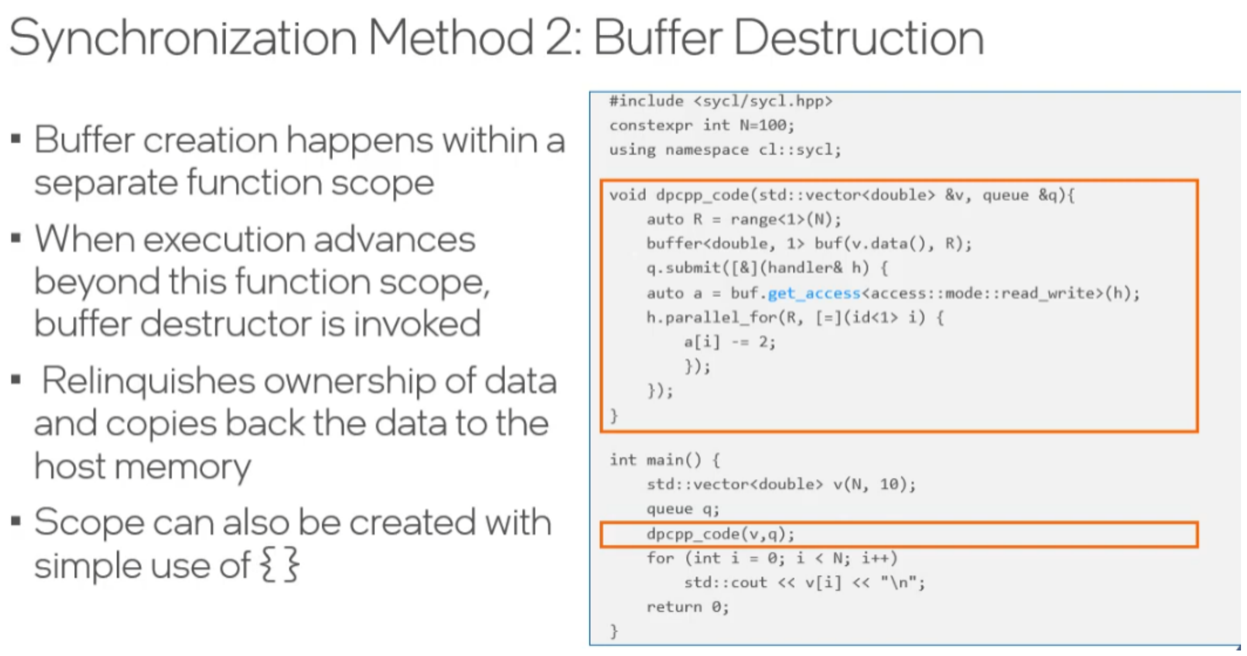


Synchronization Method 1: Host Accessor

- In the command scope, accessors are created for the accelerator
- In the application scope, accessors are created for the host
- A host accessor creates a dependency node in the execution graph
  - Execution at the host is blocked until the data is ready

```
int main() {
  constexpr int N = 100;
  auto R = range<1>(N);
  std::vector<double> v(N, 10);
  queue q;

  buffer<double, 1> buf(v.data(), R);
  q.submit([&](handler& h) {
  auto a = buf.get_access<access::mode::read_write>(h);
  h.parallel_for(R, [=](id<1> i) {
    a[i] -= 2;
    });
  });

  auto b = buf.get_access<access::mode::read>();
  for (int i = 0; i < N; i++)
    std::cout << v[i] << "\n";
  return 0;
}
```

Or you can depend on the buffer destruction because it has to wait until the kernel finishes execution to destruct the buffer



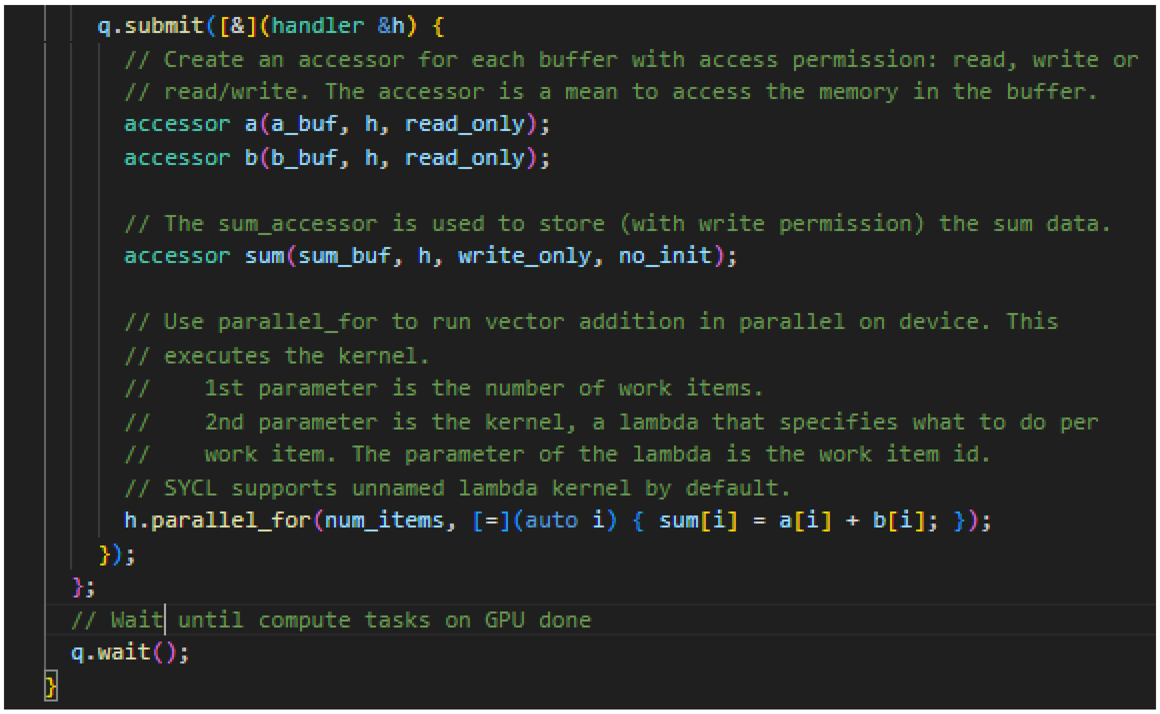
Synchronization Method 2: Buffer Destruction

- Buffer creation happens within a separate function scope
- When execution advances beyond this function scope, buffer destructor is invoked
- Relinquishes ownership of data and copies back the data to the host memory
- Scope can also be created with simple use of { }

```
#include <sycl/sycl.hpp>
constexpr int N=100;
using namespace cl::sycl;

void dpcpp_code(std::vector<double> &v, queue &q){
    auto R = range<1>(N);
    buffer<double, 1> buf(v.data(), R);
    q.submit([&](handler& h) {
    auto a = buf.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> i) {
        a[i] -= 2;
        });
    });
}

int main() {
    std::vector<double> v(N, 10);
    queue q;
    dpcpp_code(v,q);
    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```

Or you can do it using a different syntax, wrapping it in an inner scope using curly braces

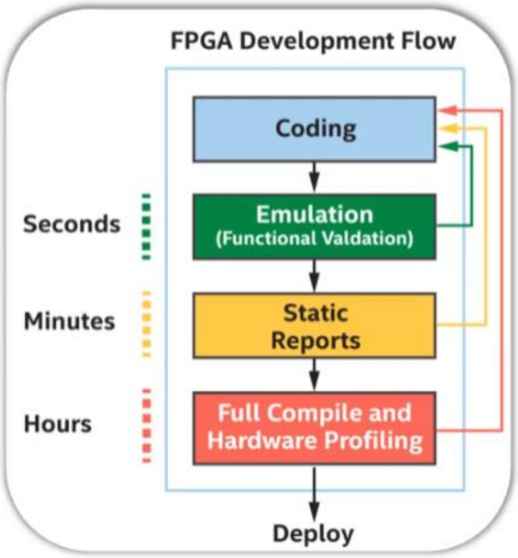
Or simply use the q.wait();



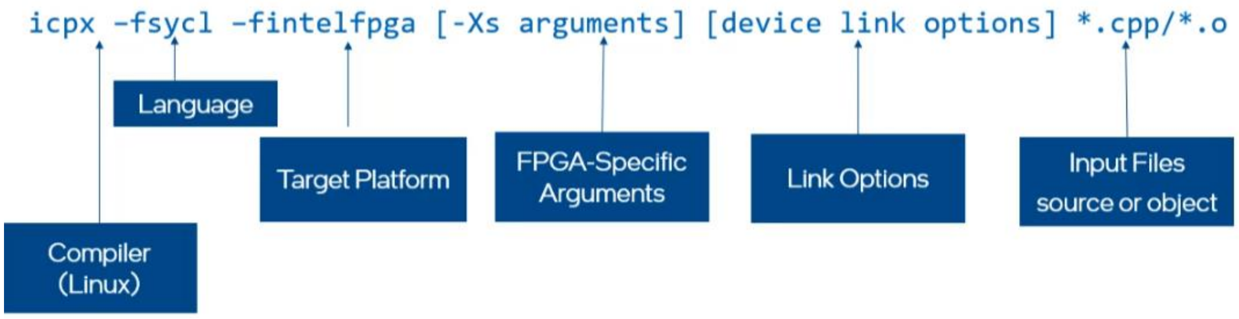
```
    q.submit([&](handler &h) {
      // Create an accessor for each buffer with access permission: read, write or
      // read/write. The accessor is a mean to access the memory in the buffer.
      accessor a(a_buf, h, read_only);
      accessor b(b_buf, h, read_only);

      // The sum_accessor is used to store (with write permission) the sum data.
      accessor sum(sum_buf, h, write_only, no_init);

      // Use parallel_for to run vector addition in parallel on device. This
      // executes the kernel.
      //    1st parameter is the number of work items.
      //    2nd parameter is the kernel, a lambda that specifies what to do per
      //    work item. The parameter of the lambda is the work item id.
      // SYCL supports unnamed lambda kernel by default.
      h.parallel_for(num_items, [=](auto i) { sum[i] = a[i] + b[i]; });
    });
  };
  // Wait until compute tasks on GPU done
  q.wait();
}
```

FPGA Development Flow for oneAPI Projects

- FPGA Emulator target (Emulation)
  - Compiles in seconds
  - Runs completely on the host
- Optimization report generation
  - Compiles in seconds to minutes
  - Identify bottlenecks
- FPGA bitstream compilation
  - Compiles in hours
  - Enable profiler to get runtime analysis



Anatomy of the SYCL* Command Targeting FPGAs



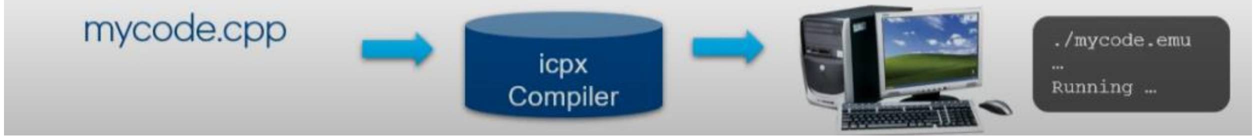
```
icpx -fsycl -fintelfpga [-Xs arguments] [device link options] *.cpp/*.o
```

Emulation Command

```
#ifdef FPGA_EMULATOR
  ext::intel::fpga_emulator_selector device_selector;
#else
  ext::intel::fpga_selector device_selector;
#endif
```

Include this construct in your code

```
icpx –fyscl -fintelfpga –DFPGA_EMULATOR <source_file>.cpp
```



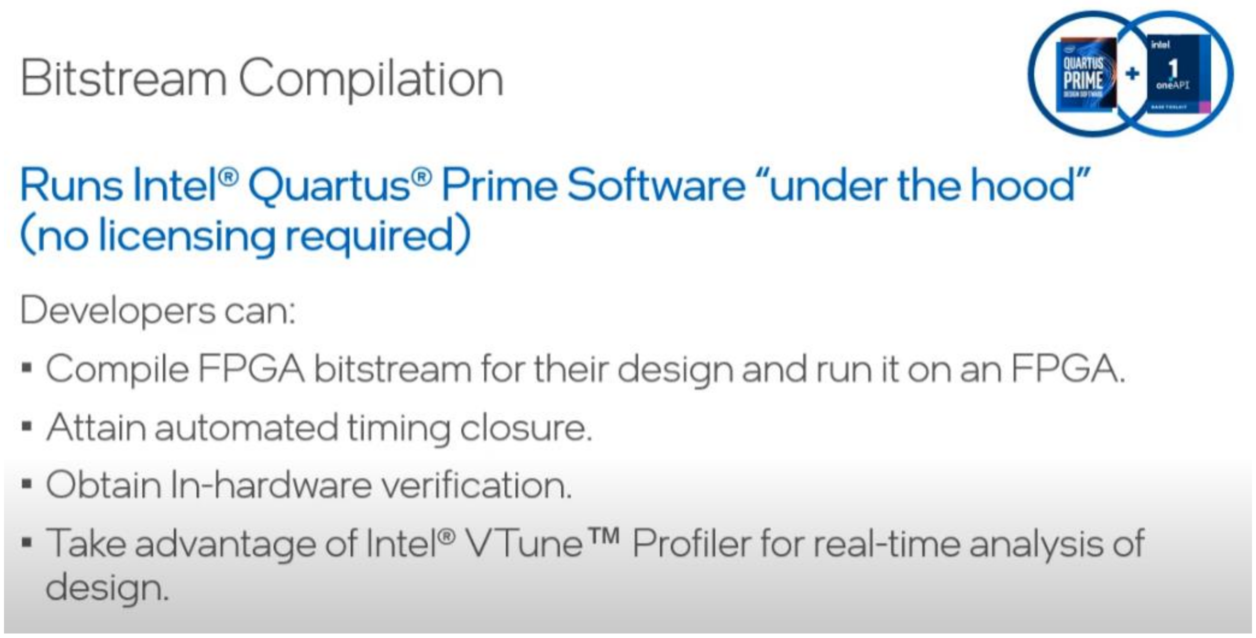
```
./mycode.emu
…
Running …
```

Command to Produce an Optimization Report

```
icpx -fyscl –fintelfpga –Xshardware -fsycl-link <source_file>.cpp
```

The default value for –fsycl-link is -fsycl-link=early which produces an early image object file and report

- A report showing optimization, area, and architectural information will be produced in <file_name>.prj/reports/

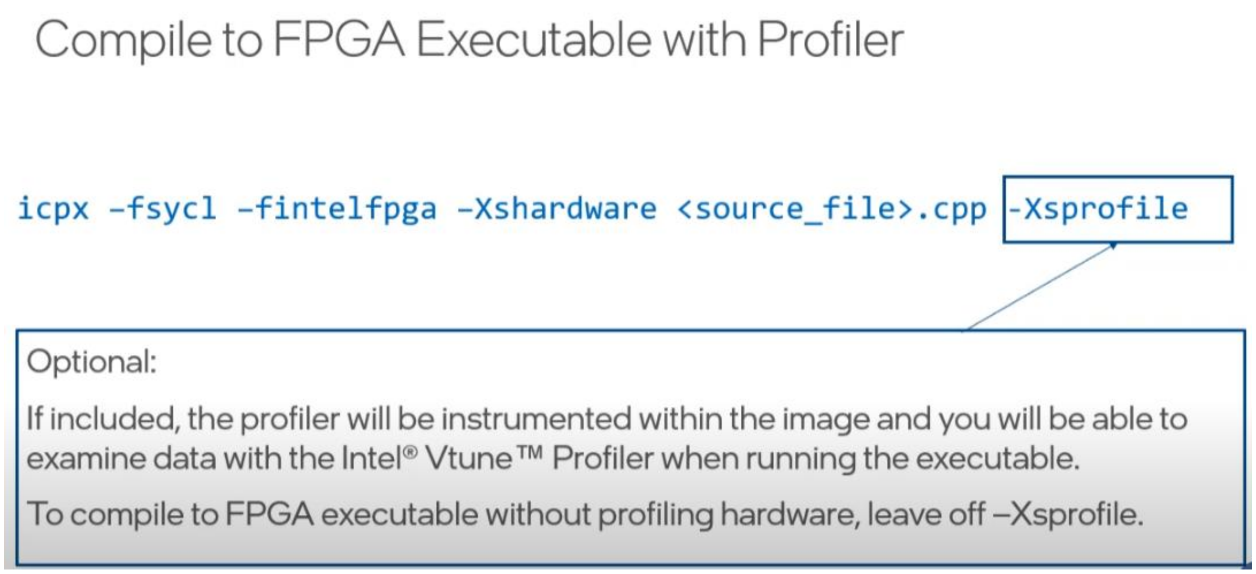


Bitstream Compilation

Runs Intel® Quartus® Prime Software "under the hood" (no licensing required)

Developers can:

- Compile FPGA bitstream for their design and run it on an FPGA.
- Attain automated timing closure.
- Obtain In-hardware verification.
- Take advantage of Intel® VTune™ Profiler for real-time analysis of design.



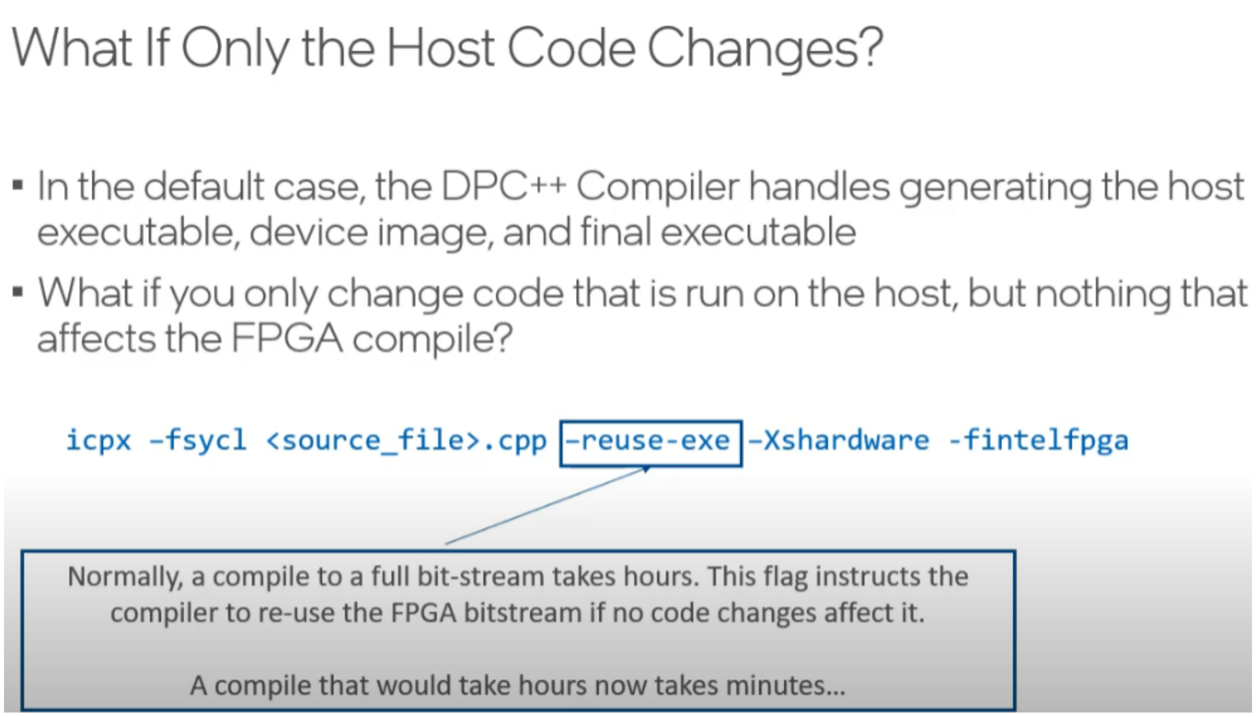
Compile to FPGA Executable with Profiler

```
icpx -fsycl -fintelfpga -Xshardware <source_file>.cpp -Xsprofile
```

Optional:

If included, the profiler will be instrumented within the image and you will be able to examine data with the Intel® Vtune™ Profiler when running the executable.

To compile to FPGA executable without profiling hardware, leave off –Xsprofile.



What If Only the Host Code Changes?

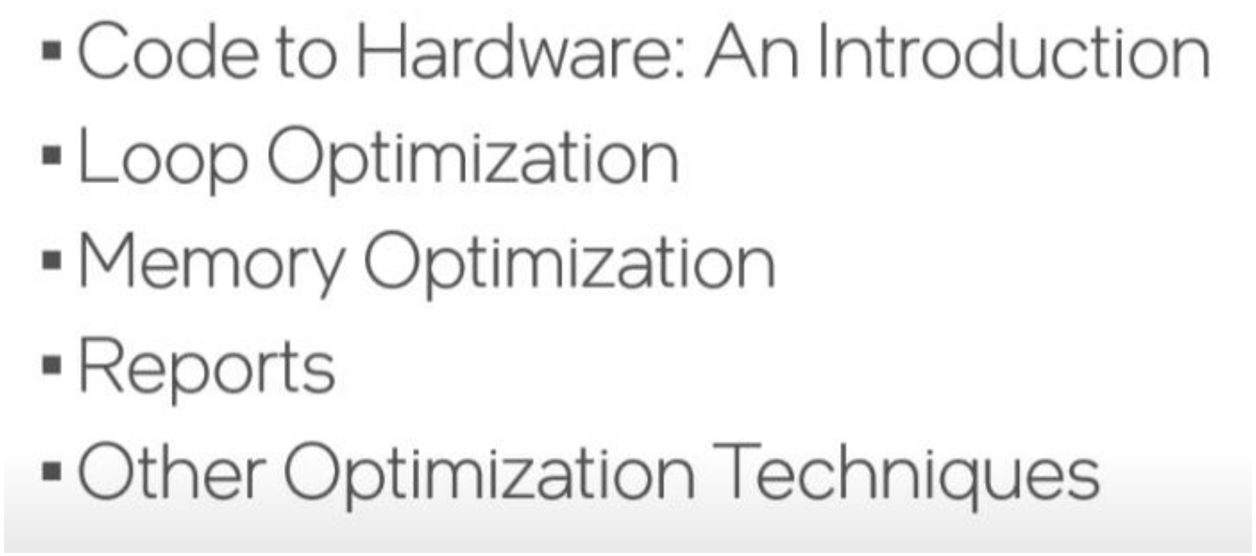
- In the default case, the DPC++ Compiler handles generating the host executable, device image, and final executable
- What if you only change code that is run on the host, but nothing that affects the FPGA compile?

```
icpx -fsycl <source_file>.cpp -reuse-exe -Xshardware -fintelfpga
```

Normally, a compile to a full bit-stream takes hours. This flag instructs the compiler to re-use the FPGA bitstream if no code changes affect it.

A compile that would take hours now takes minutes…

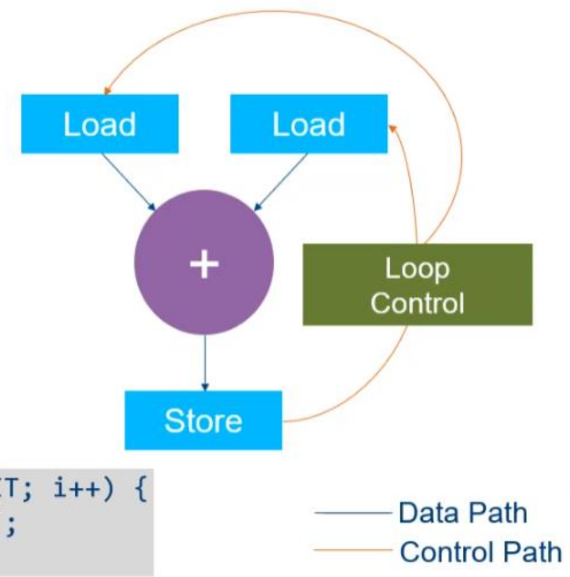
Optimizations



- Code to Hardware: An Introduction
- Loop Optimization
- Memory Optimization
- Reports
- Other Optimization Techniques

How Is a Pipeline Built?

- Hardware is added for
  - Computation
  - Memory Loads and Stores
  - Control and scheduling
    - Loops & Conditionals



```
for (int i=0; i<LIMIT; i++) {
  c[i] = a[i] + b[i];
}
```