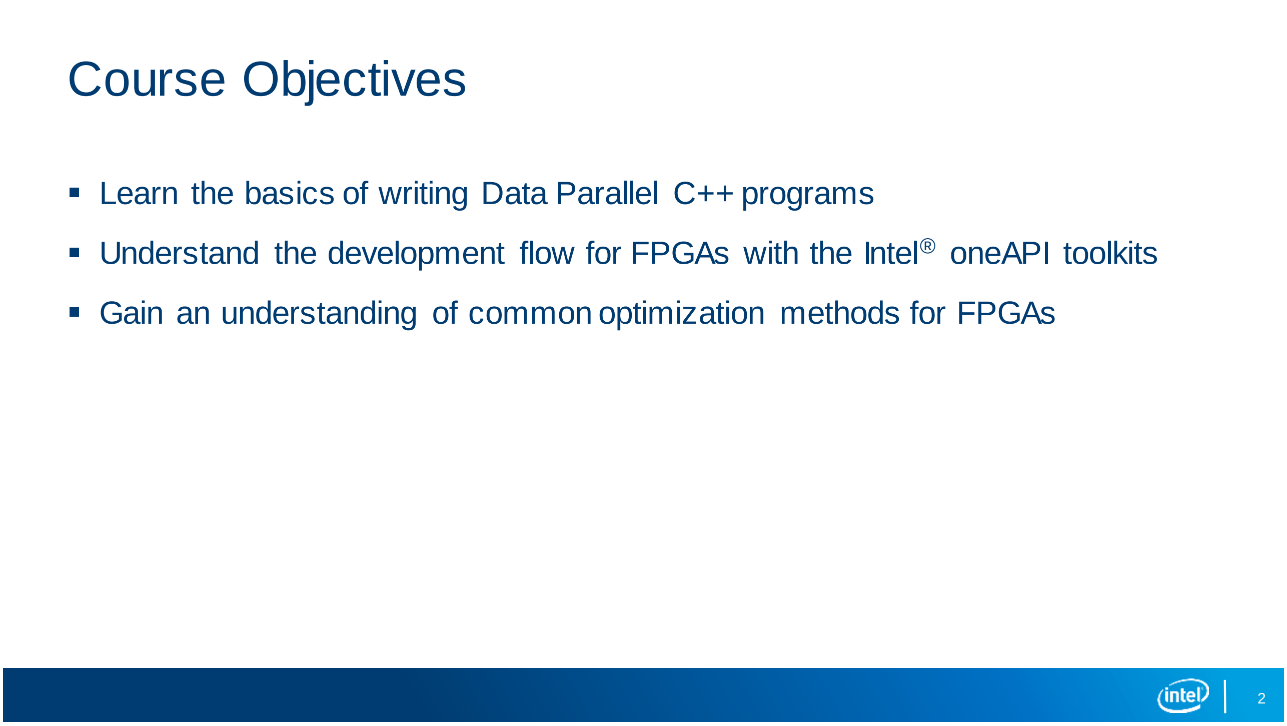
Using Intel® oneAPI Toolkits with FPGAs



Course Objectives

- Learn the basics of writing Data Parallel C++ programs
- Understand the development flow for FPGAs with the Intel® oneAPI toolkits
- Gain an understanding of common optimization methods for FPGAs

Awareness of **Beta**

Intel® oneAPI Toolkits are Beta.

COURSE AGENDA

The Basics

The oneAPI Toolset

Introduction to Data Parallel C++

Lab: Overview of DPC++

Using FPGAs with the Intel® oneAPI Toolkits

What are FPGAs and Why Should I Care About Programming Them?

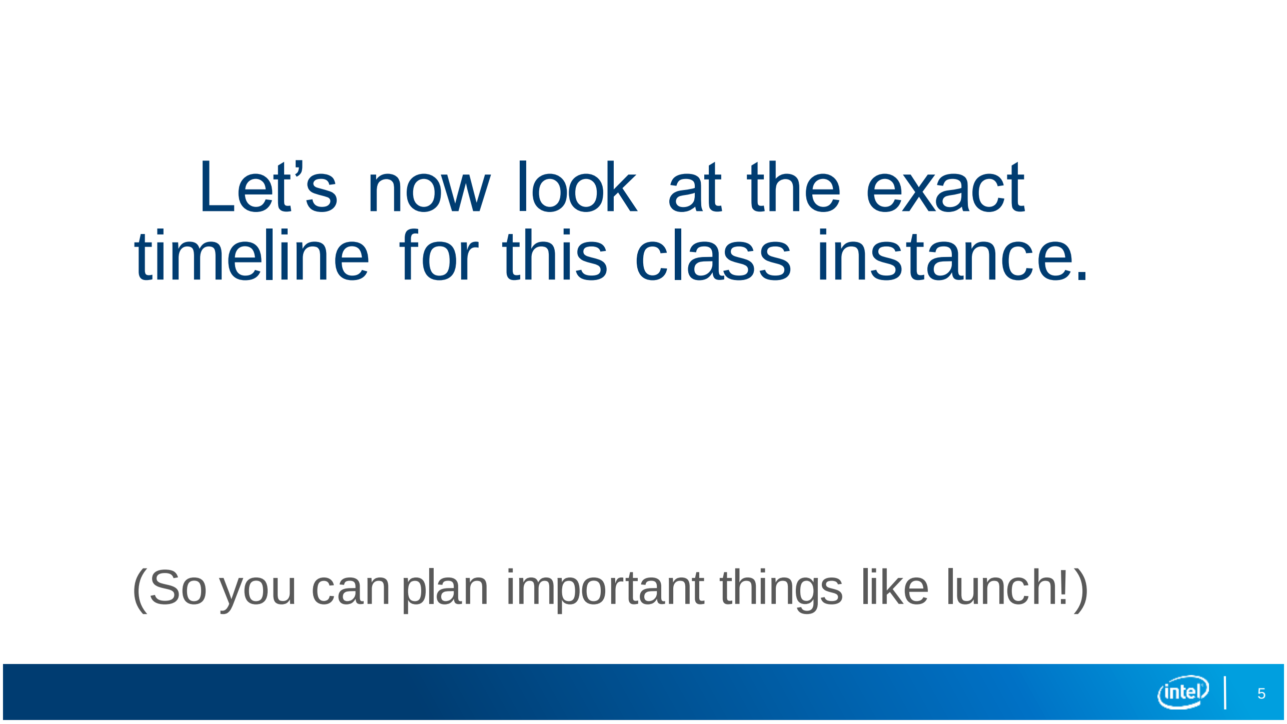
Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

Lab: Practice the FPGA Development Flow

Optimizing Your Code for FPGAs

Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

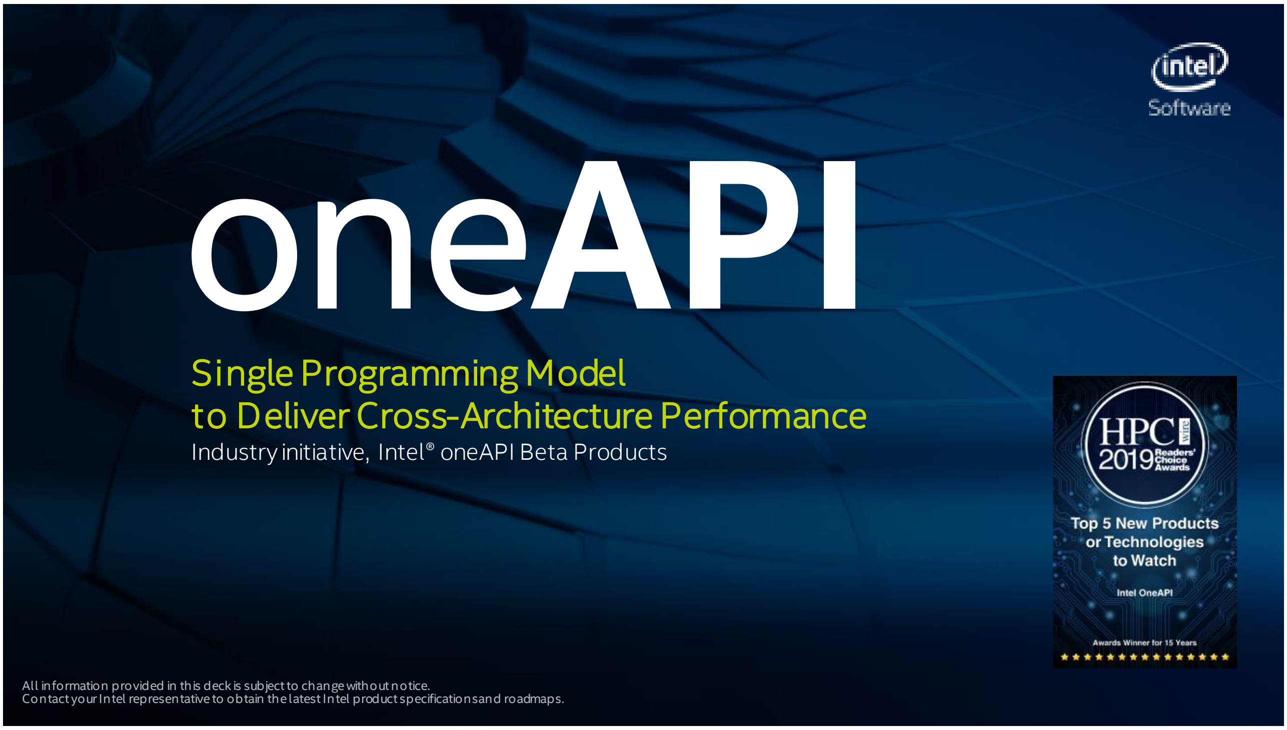
Lab: Optimizing the Hough Transform Kernel



Let’s now look at the exact timeline for this class instance.

(So you can plan important things like lunch!)

THE BASICS:
THE TOOLSET



Software

oneAPI

Single Programming Model to Deliver Cross-Architecture Performance

Industry initiative, Intel® oneAPI Beta Products

HPC wire 2019 Readers' Choice Awards

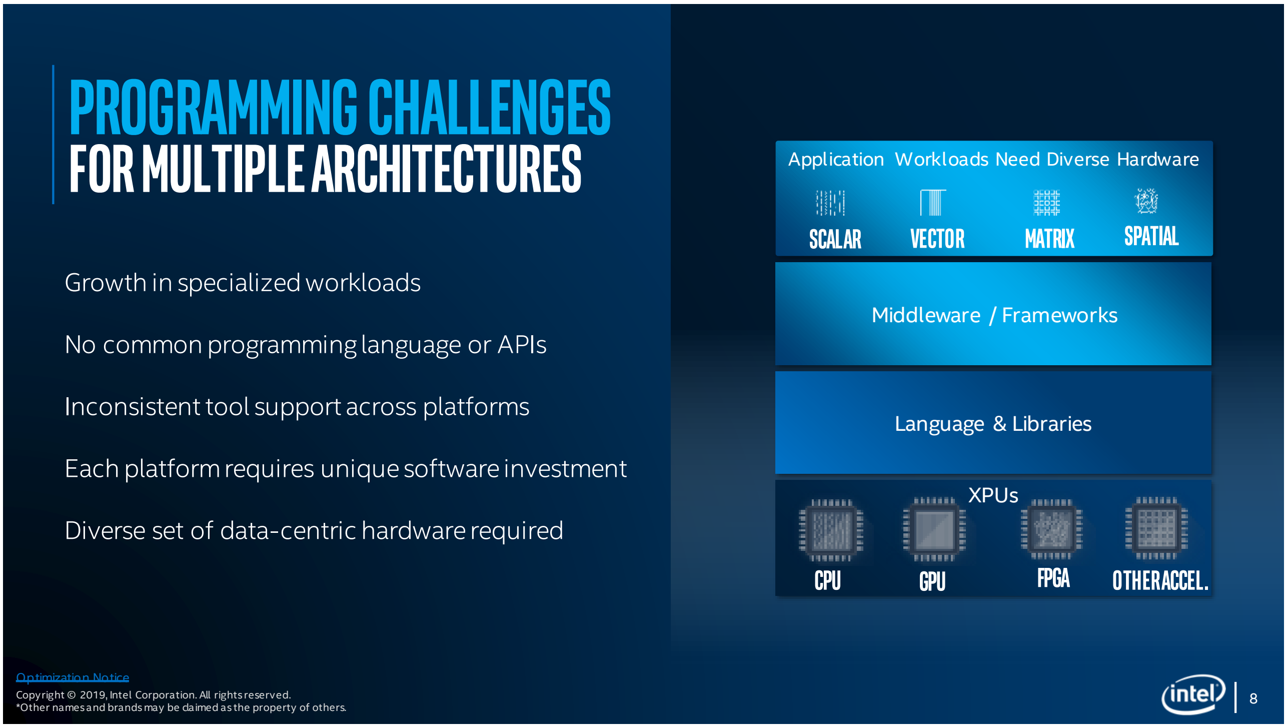
Top 5 New Products or Technologies to Watch

Intel OneAPI

Awards Winner for 15 Years

All information provided in this deck is subject to change without notice.
Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

7



PROGRAMMING CHALLENGES FOR MULTIPLE ARCHITECTURES

Growth in specialized workloads

No common programming language or APIs

Inconsistent tool support across platforms

Each platform requires unique software investment

Diverse set of data-centric hardware required

Application Workloads Need Diverse Hardware

SCALAR VECTOR MATRIX SPATIAL

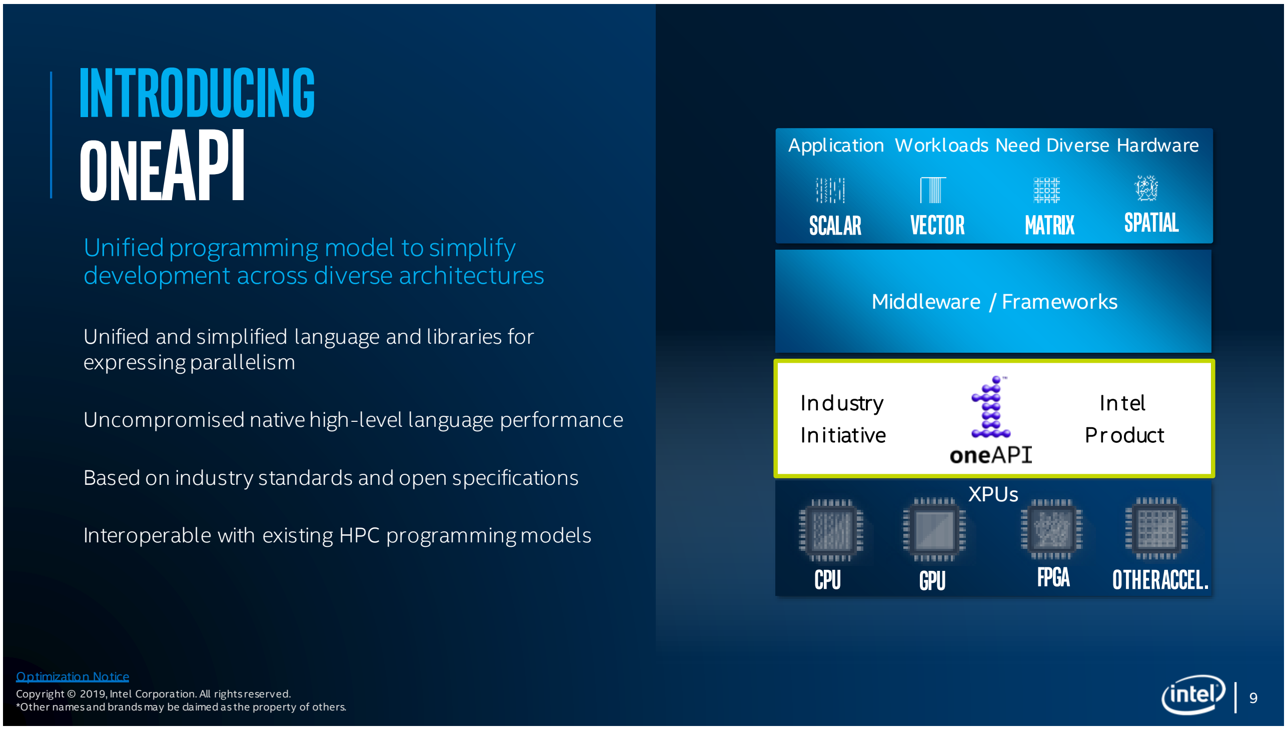
Middleware / Frameworks

Language & Libraries

XPUs

CPU GPU FPGA OTHER ACCEL.

8



INTRODUCING ONEAPI

Unified programming model to simplify development across diverse architectures

Unified and simplified language and libraries for expressing parallelism

Uncompromised native high-level language performance

Based on industry standards and open specifications

Interoperable with existing HPC programming models

Application Workloads Need Diverse Hardware

SCALAR VECTOR MATRIX SPATIAL

Middleware / Frameworks

Industry Initiative

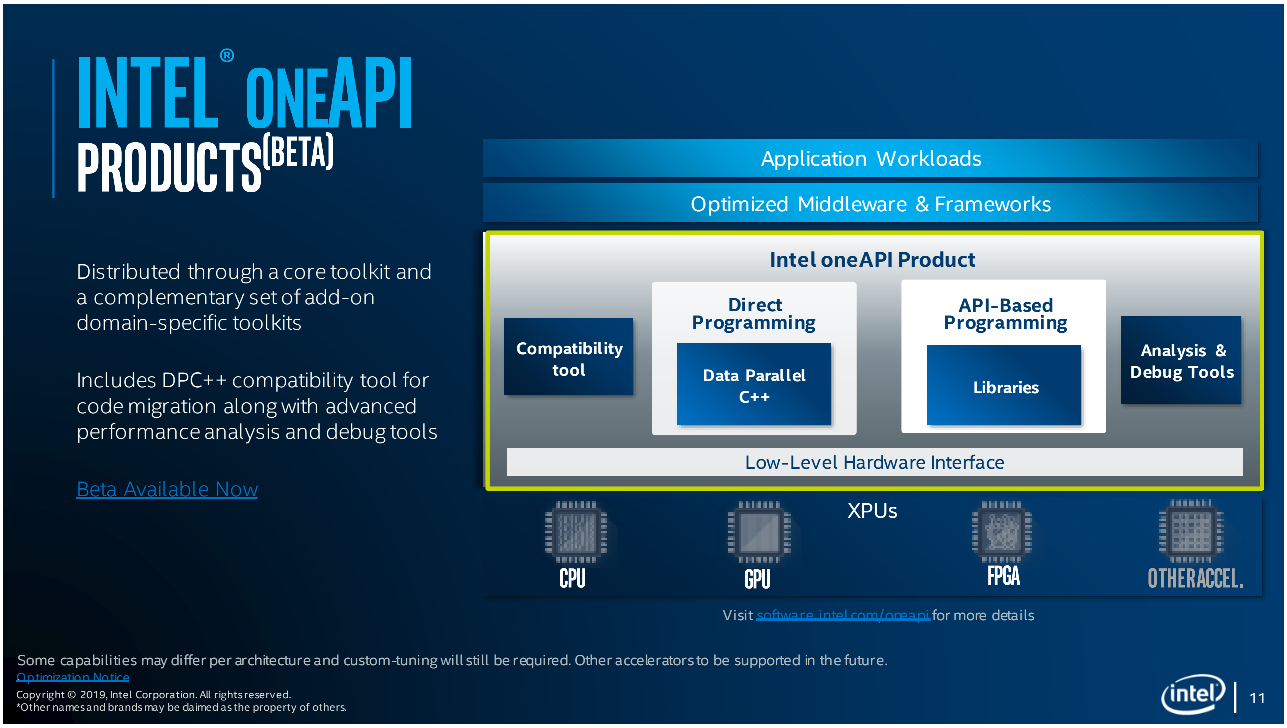
oneAPI

Intel Product

XPUs

CPU GPU FPGA OTHER ACCEL.

Optimization Notice



INTEL® ONEAPI PRODUCTS(BETA)

Distributed through a core toolkit and a complementary set of add-on domain-specific toolkits

Includes DPC++ compatibility tool for code migration along with advanced performance analysis and debug tools

Beta Available Now

Application Workloads

Optimized Middleware & Frameworks

Intel oneAPI Product

Compatibility tool

Direct Programming

Data Parallel C++

API-Based Programming

Libraries

Analysis & Debug Tools

Low-Level Hardware Interface

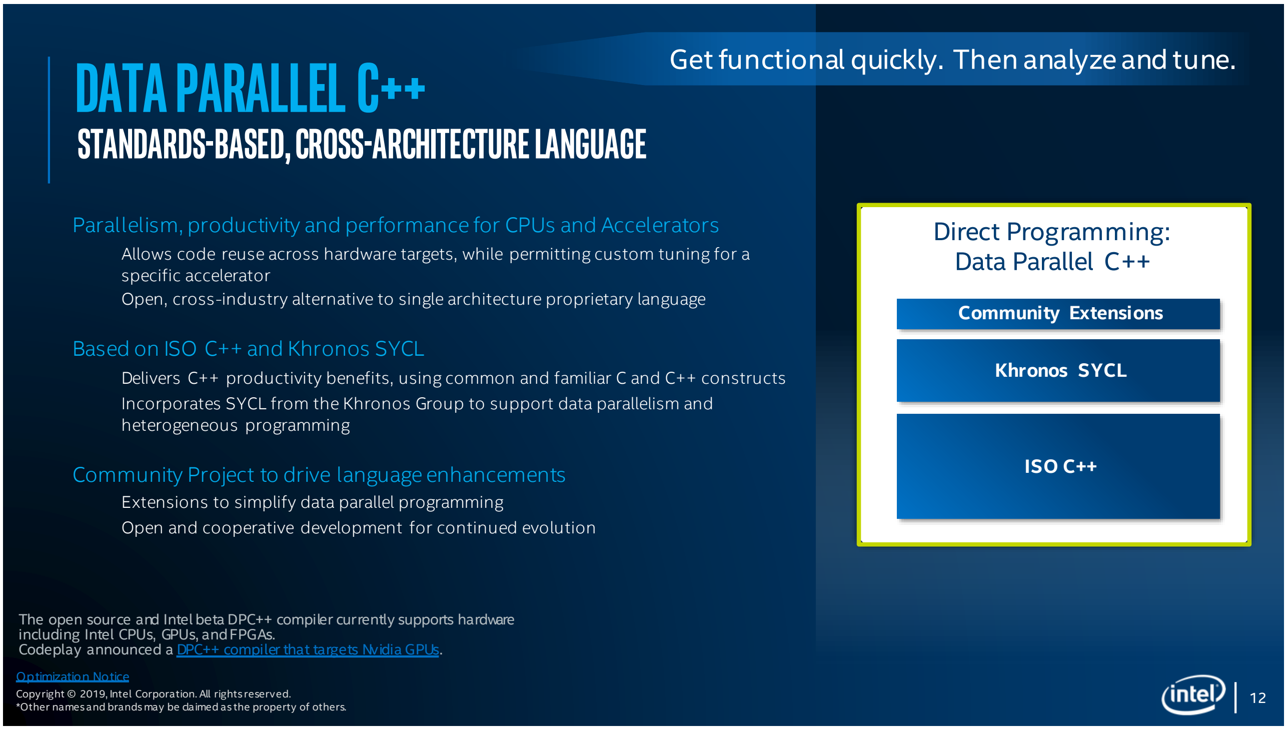
XPUs

CPU GPU FPGA OTHER ACCEL.

Visit software.intel.com/oneapi for more details

Some capabilities may differ per architecture and custom-tuning will still be required. Other accelerators to be supported in the future.

Optimization Notice



Get functional quickly. Then analyze and tune.

DATA PARALLEL C++

STANDARDS-BASED, CROSS-ARCHITECTURE LANGUAGE

Parallelism, productivity and performance for CPUs and Accelerators

Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator

Open, cross-industry alternative to single architecture proprietary language

Based on ISO C++ and Khronos SYCL

Delivers C++ productivity benefits, using common and familiar C and C++ constructs

Incorporates SYCL from the Khronos Group to support data parallelism and heterogeneous programming

Community Project to drive language enhancements

Extensions to simplify data parallel programming

Open and cooperative development for continued evolution

Direct Programming:
Data Parallel C++

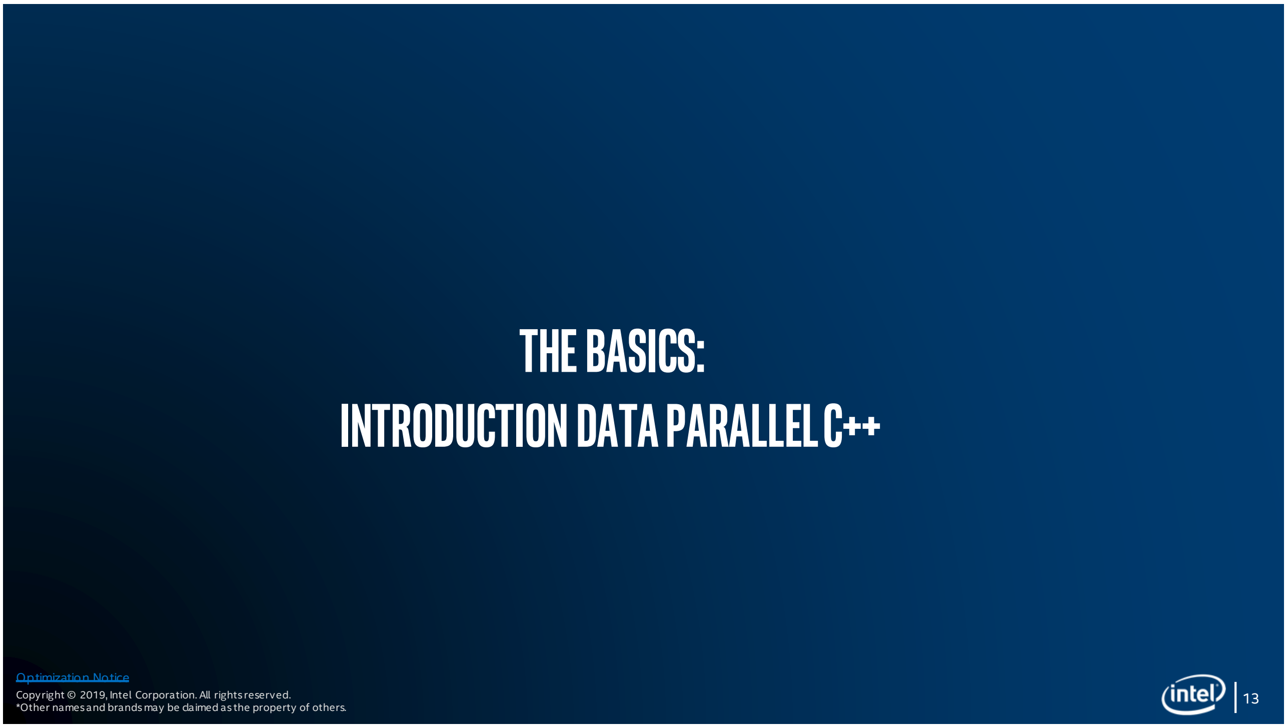
Community Extensions

Khronos SYCL

ISO C++

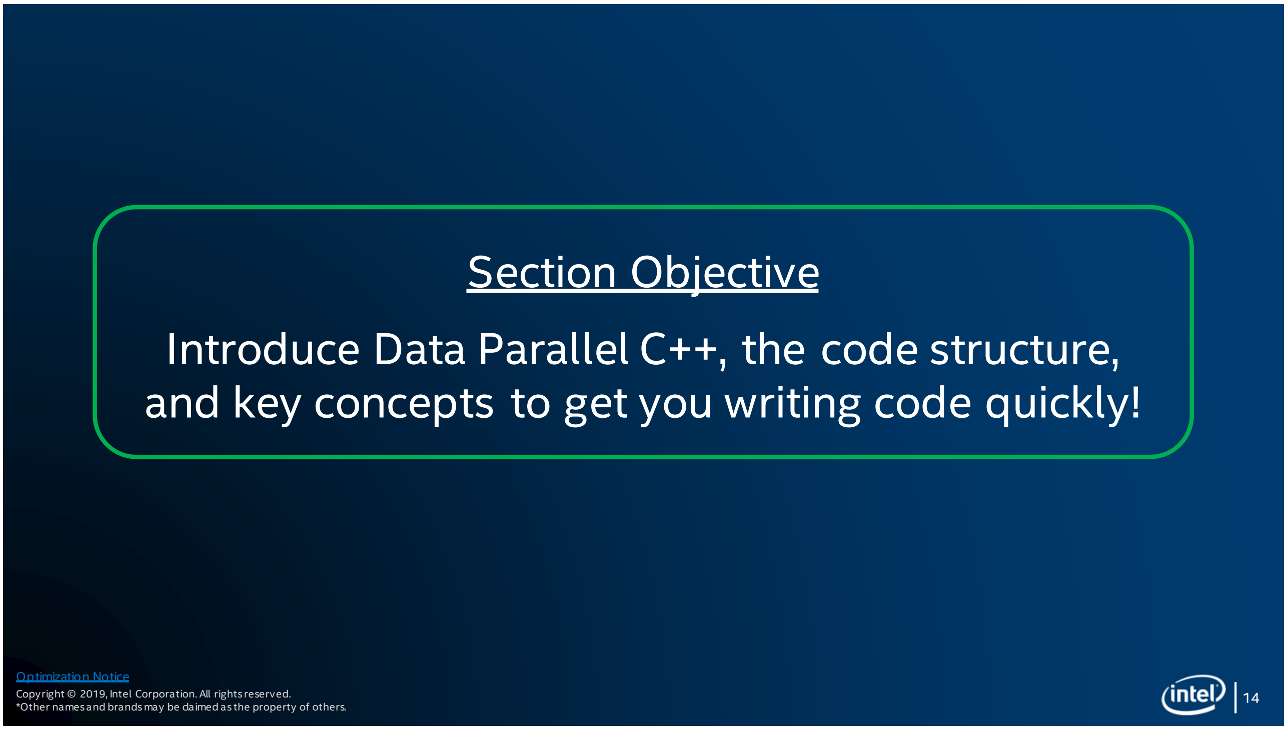
The open source and Intel beta DPC++ compiler currently supports hardware including Intel CPUs, GPUs, and FPGAs.
Codeplay announced a DPC++ compiler that targets Nvidia GPUs.

Optimization Notice



THE BASICS:
INTRODUCTION DATA PARALLEL C++

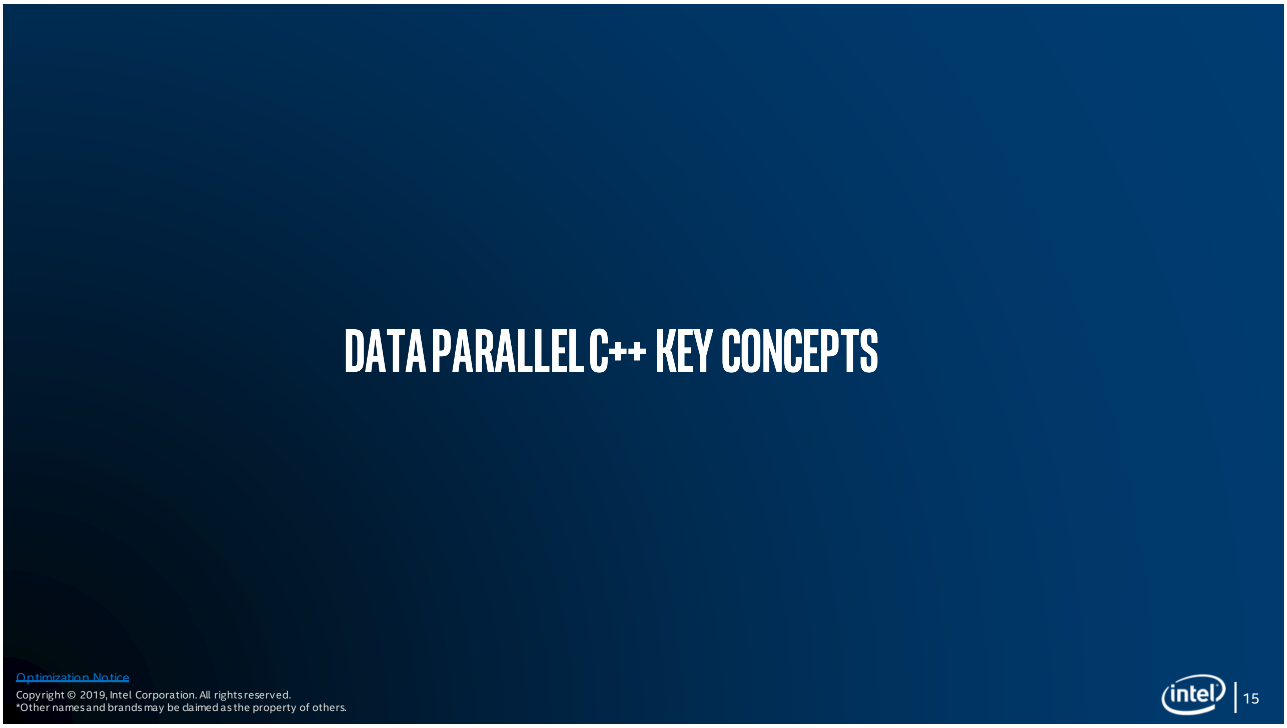
Optimization Notice



Section Objective

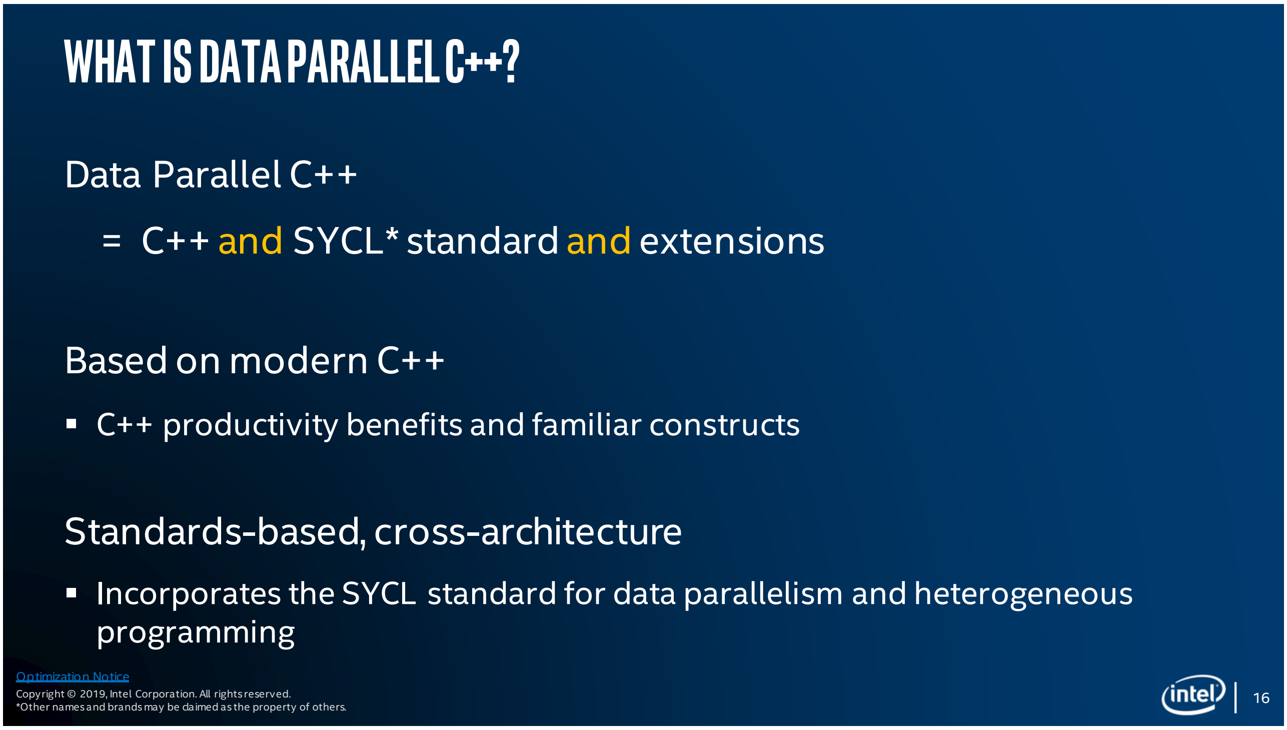
Introduce Data Parallel C++, the code structure, and key concepts to get you writing code quickly!

Optimization Notice



DATA PARALLEL C++ KEY CONCEPTS

Optimization Notice



WHAT IS DATA PARALLEL C++?

Data Parallel C++

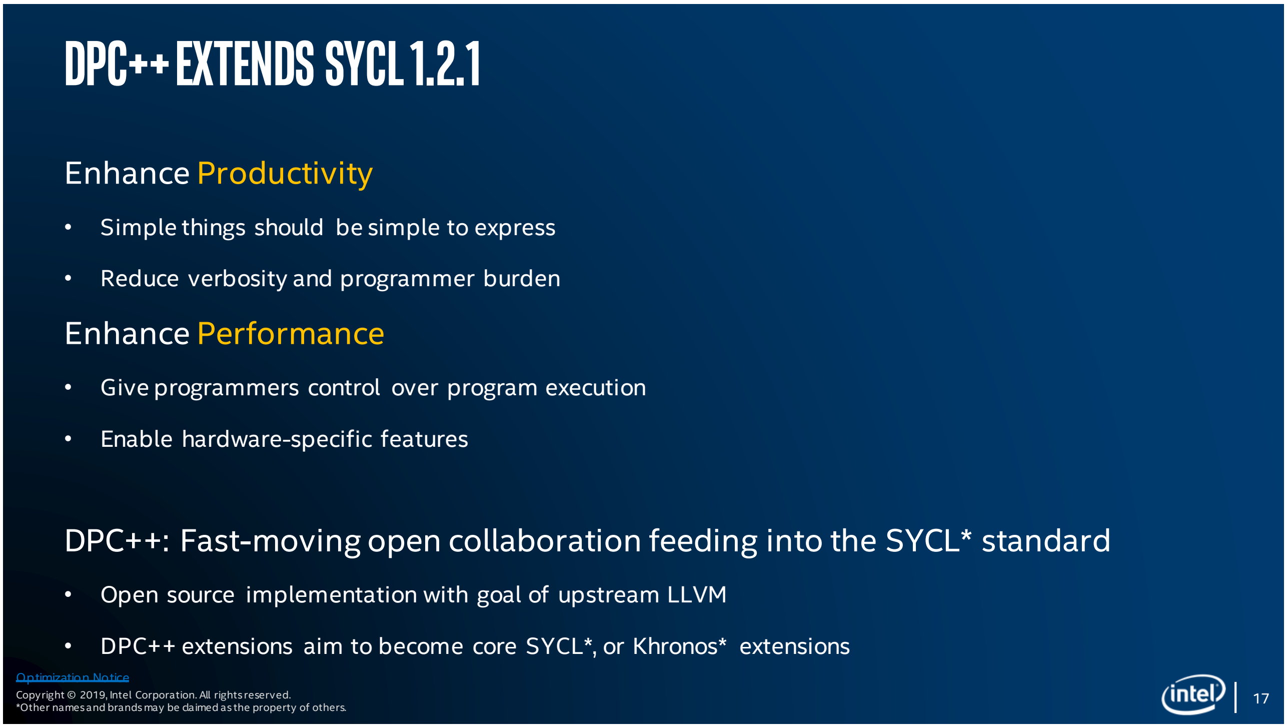
= C++ and SYCL* standard and extensions

Based on modern C++

- C++ productivity benefits and familiar constructs

Standards-based, cross-architecture

- Incorporates the SYCL standard for data parallelism and heterogeneous programming



DPC++ EXTENDS SYCL 1.2.1

Enhance Productivity

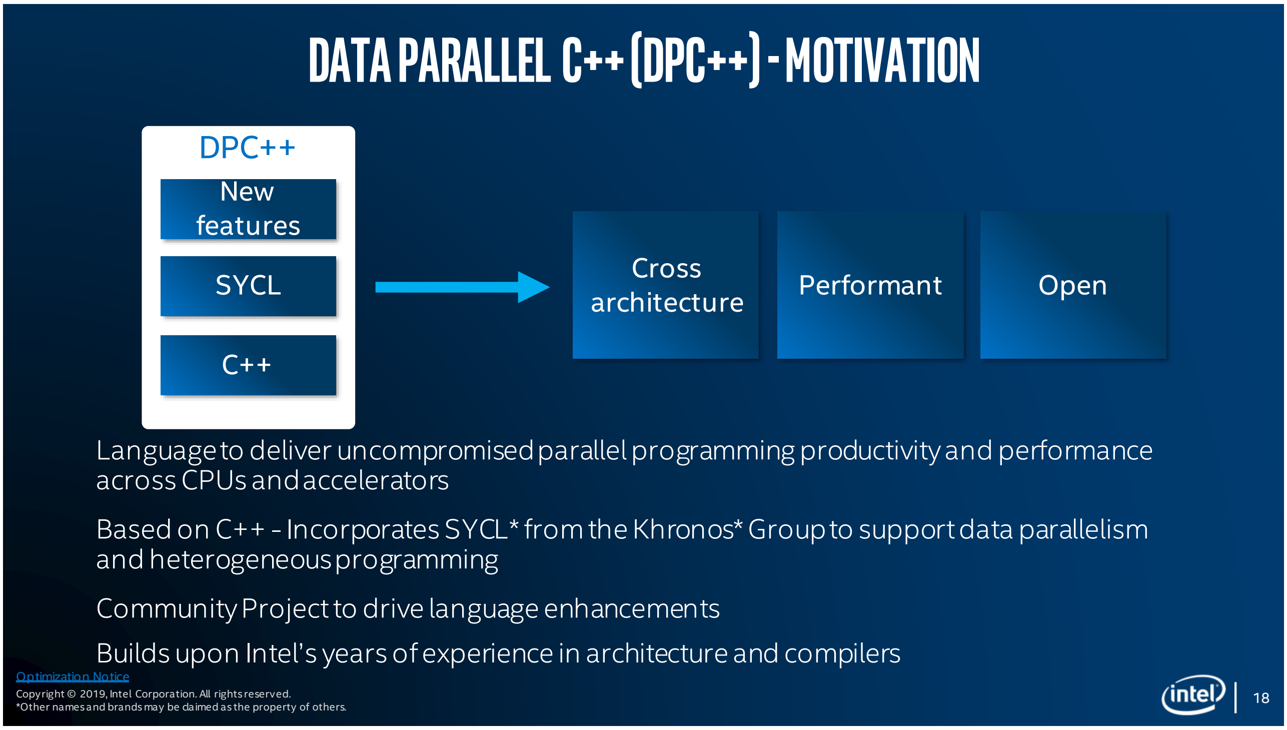
- Simple things should be simple to express
- Reduce verbosity and programmer burden

Enhance Performance

- Give programmers control over program execution
- Enable hardware-specific features

DPC++: Fast-moving open collaboration feeding into the SYCL* standard

- Open source implementation with goal of upstream LLVM
- DPC++ extensions aim to become core SYCL*, or Khronos* extensions



DATA PARALLEL C++ (DPC++) - MOTIVATION

DPC++

New features

SYCL

C++

Cross architecture

Performant

Open

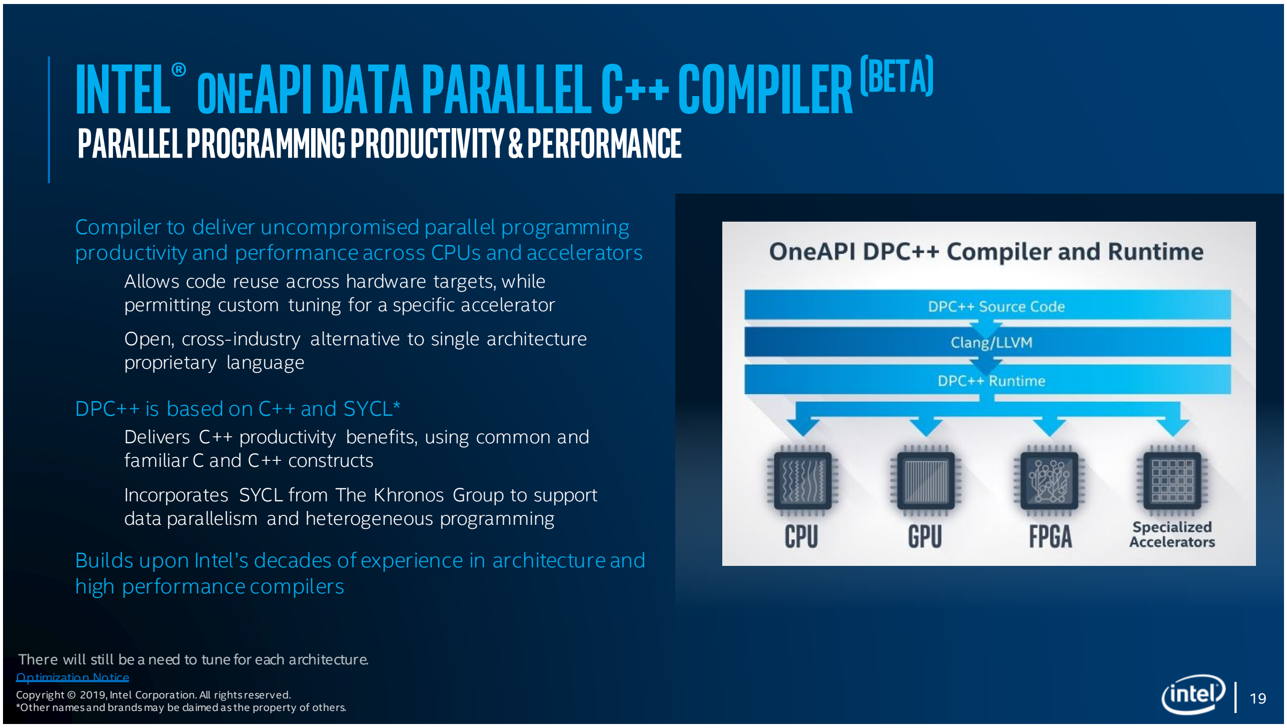
Language to deliver uncompromised parallel programming productivity and performance across CPUs and accelerators

Based on C++ - Incorporates SYCL* from the Khronos* Group to support data parallelism and heterogeneous programming

Community Project to drive language enhancements

Builds upon Intel's years of experience in architecture and compilers

Optimization Notice



INTEL® ONEAPI DATA PARALLEL C++ COMPILER (BETA)

PARALLEL PROGRAMMING PRODUCTIVITY & PERFORMANCE

Compiler to deliver uncompromised parallel programming productivity and performance across CPUs and accelerators

- Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator
- Open, cross-industry alternative to single architecture proprietary language

DPC++ is based on C++ and SYCL*

- Delivers C++ productivity benefits, using common and familiar C and C++ constructs
- Incorporates SYCL from The Khronos Group to support data parallelism and heterogeneous programming

Builds upon Intel's decades of experience in architecture and high performance compilers

OneAPI DPC++ Compiler and Runtime

DPC++ Source Code

Clang/LLVM

DPC++ Runtime

CPU

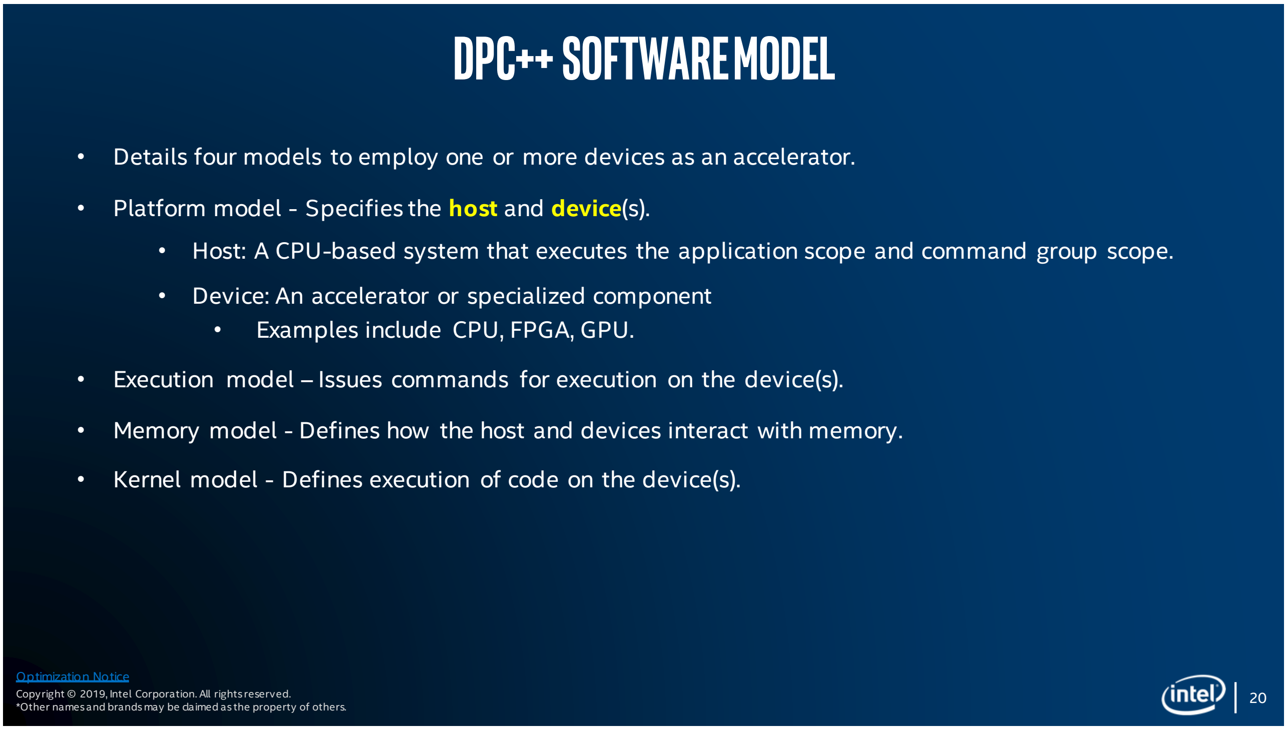
GPU

FPGA

Specialized Accelerators

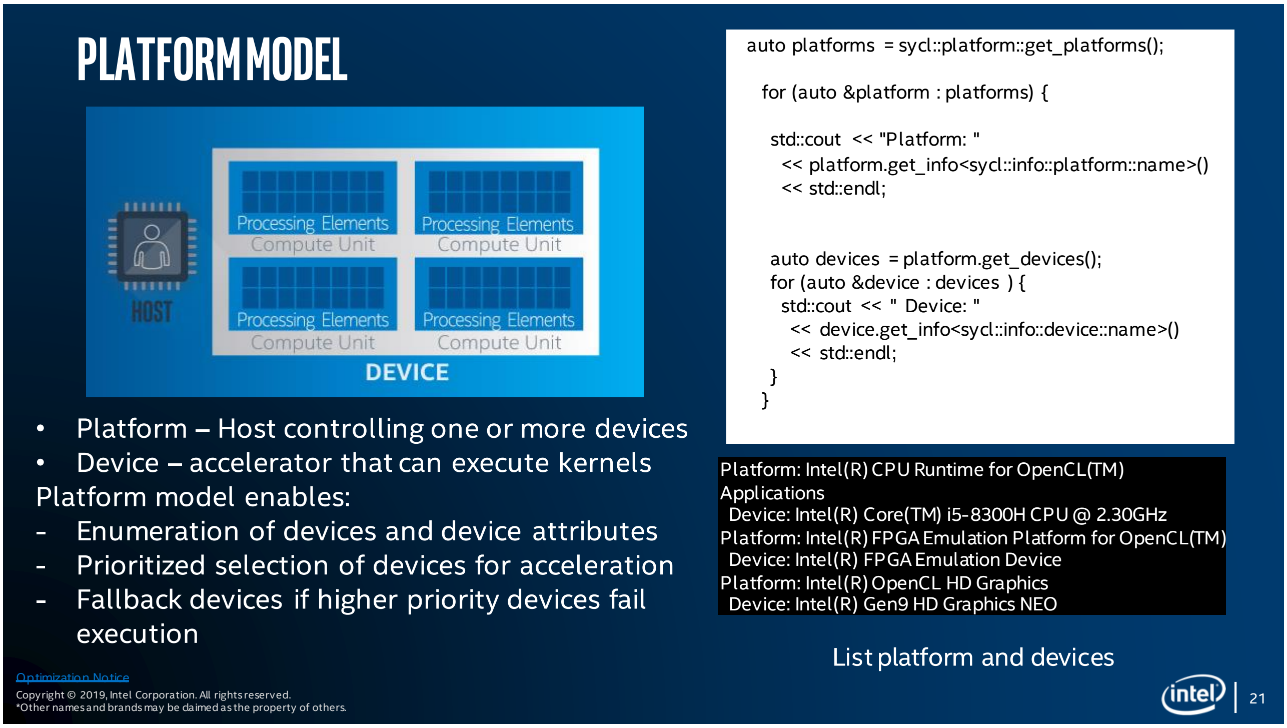
There will still be a need to tune for each architecture.

Optimization Notice



DPC++ SOFTWARE MODEL

- Details four models to employ one or more devices as an accelerator.
- Platform model - Specifies the **host** and **device**(s).
    - Host: A CPU-based system that executes the application scope and command group scope.
    - Device: An accelerator or specialized component
        - Examples include CPU, FPGA, GPU.
- Execution model – Issues commands for execution on the device(s).
- Memory model - Defines how the host and devices interact with memory.
- Kernel model - Defines execution of code on the device(s).



PLATFORM MODEL

- Platform – Host controlling one or more devices
- Device – accelerator that can execute kernels

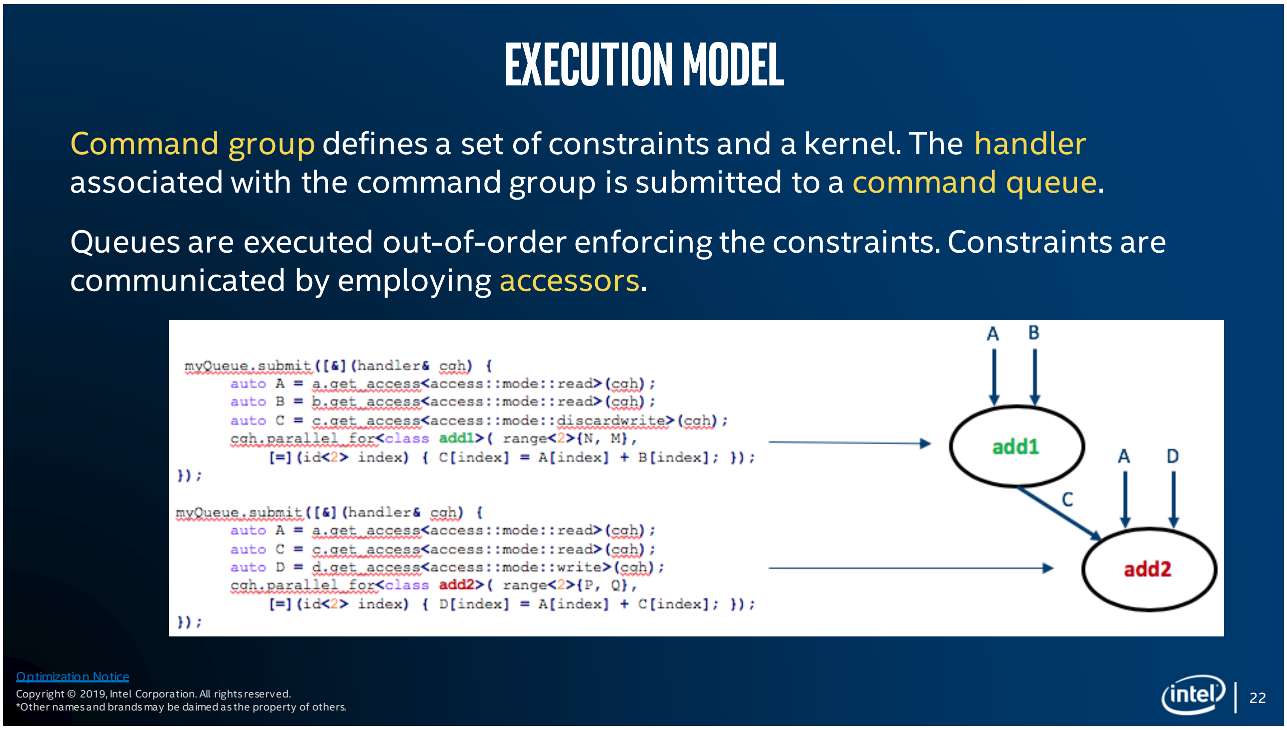
Platform model enables:

- Enumeration of devices and device attributes
- Prioritized selection of devices for acceleration
- Fallback devices if higher priority devices fail execution

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

 for (auto &platform : platforms) {

  std::cout << "Platform: "
    << platform.get_info<sycl::info::platform::name>()
    << std::endl;


  auto devices = platform.get_devices();
  for (auto &device : devices ) {
    std::cout << " Device: "
      << device.get_info<sycl::info::device::name>()
      << std::endl;
  }
 }
```

```
Platform: Intel(R) CPU Runtime for OpenCL(TM)
Applications
 Device: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Platform: Intel(R) FPGA Emulation Platform for OpenCL(TM)
 Device: Intel(R) FPGA Emulation Device
Platform: Intel(R) OpenCL HD Graphics
 Device: Intel(R) Gen9 HD Graphics NEO
```

List platform and devices



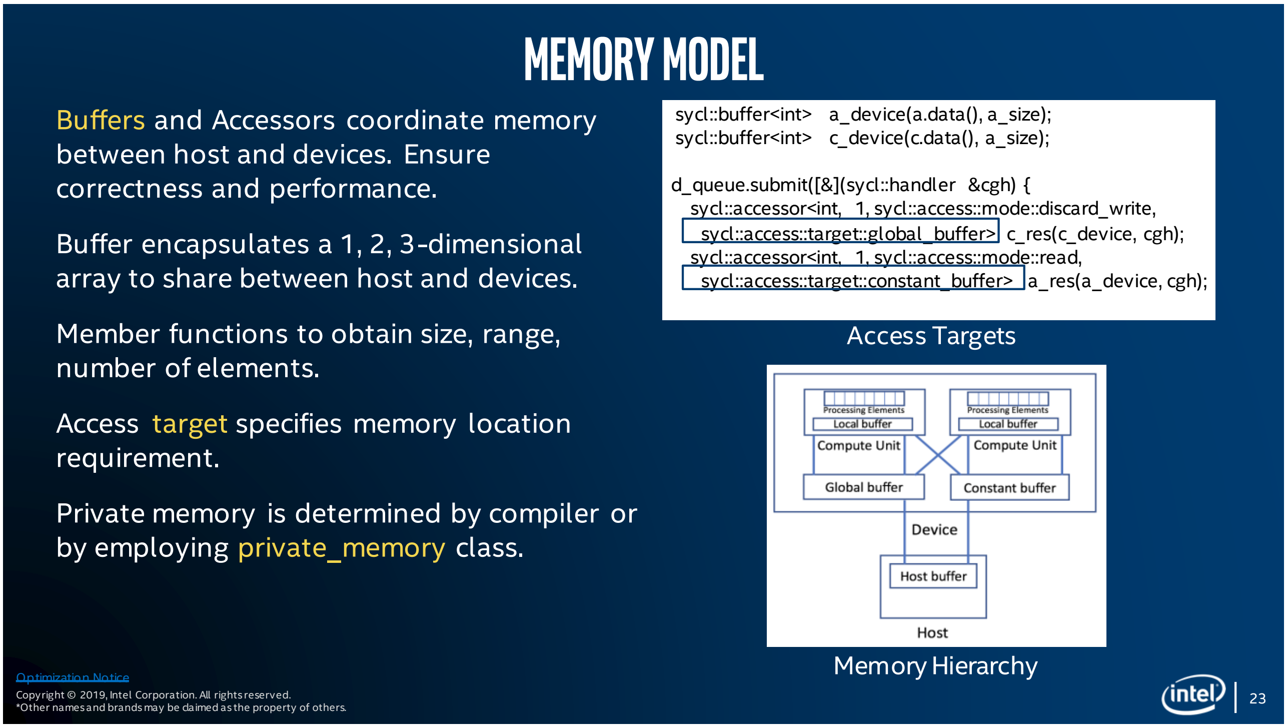
EXECUTION MODEL

Command group defines a set of constraints and a kernel. The handler associated with the command group is submitted to a command queue.

Queues are executed out-of-order enforcing the constraints. Constraints are communicated by employing accessors.

```
 myQueue.submit([&](handler& cgh) {
        auto A = a.get_access<access::mode::read>(cgh);
        auto B = b.get_access<access::mode::read>(cgh);
        auto C = c.get_access<access::mode::discardwrite>(cgh);
        cgh.parallel_for<class add1>( range<2>{N, M},
              [=](id<2> index) { C[index] = A[index] + B[index]; });
});

myQueue.submit([&](handler& cgh) {
        auto A = a.get_access<access::mode::read>(cgh);
        auto C = c.get_access<access::mode::read>(cgh);
        auto D = d.get_access<access::mode::write>(cgh);
        cgh.parallel_for<class add2>( range<2>{P, Q},
              [=](id<2> index) { D[index] = A[index] + C[index]; });
});
```

Optimization Notice



MEMORY MODEL

Buffers and Accessors coordinate memory between host and devices. Ensure correctness and performance.

Buffer encapsulates a 1, 2, 3-dimensional array to share between host and devices.

Member functions to obtain size, range, number of elements.

Access target specifies memory location requirement.

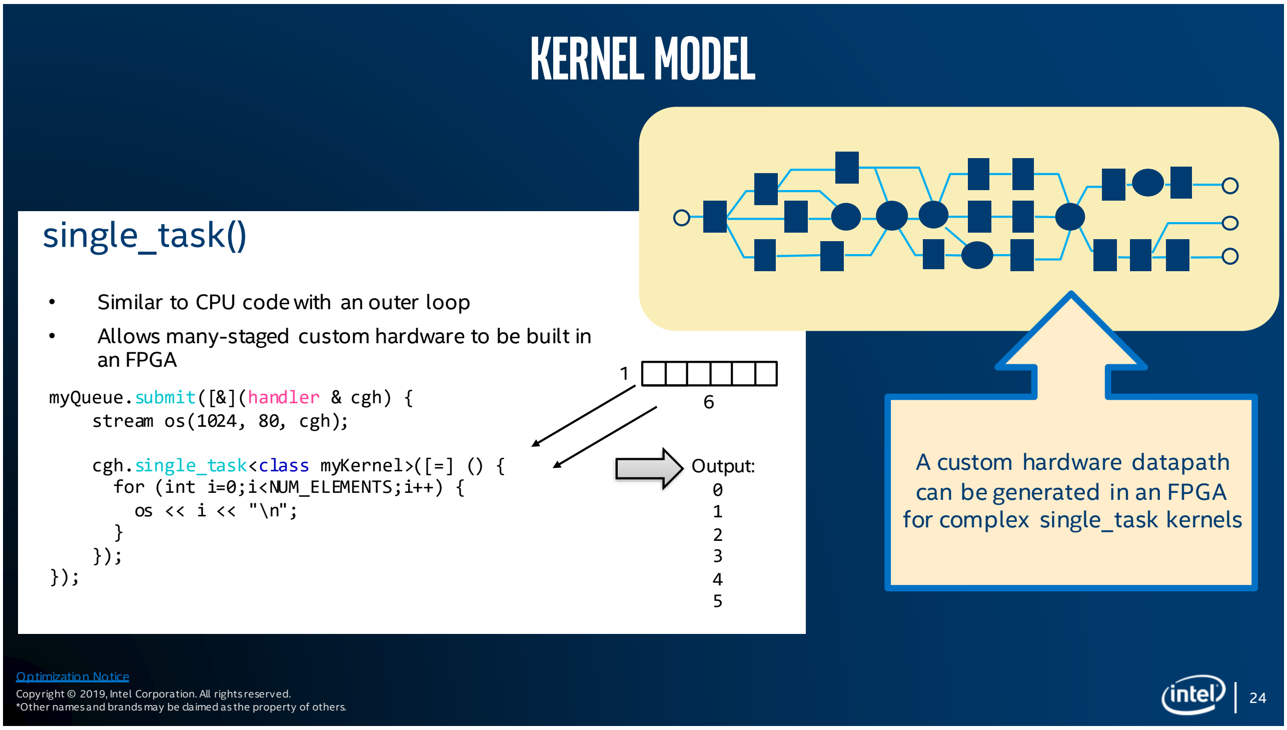
Private memory is determined by compiler or by employing private_memory class.

```
sycl::buffer<int>   a_device(a.data(), a_size);
sycl::buffer<int>   c_device(c.data(), a_size);

d_queue.submit([&](sycl::handler  &cgh) {
  sycl::accessor<int,  1, sycl::access::mode::discard_write,
   sycl::access::target::global_buffer>  c_res(c_device, cgh);
  sycl::accessor<int,  1, sycl::access::mode::read,
   sycl::access::target::constant_buffer>  a_res(a_device, cgh);
```

Access Targets

Memory Hierarchy

Optimization Notice



KERNEL MODEL

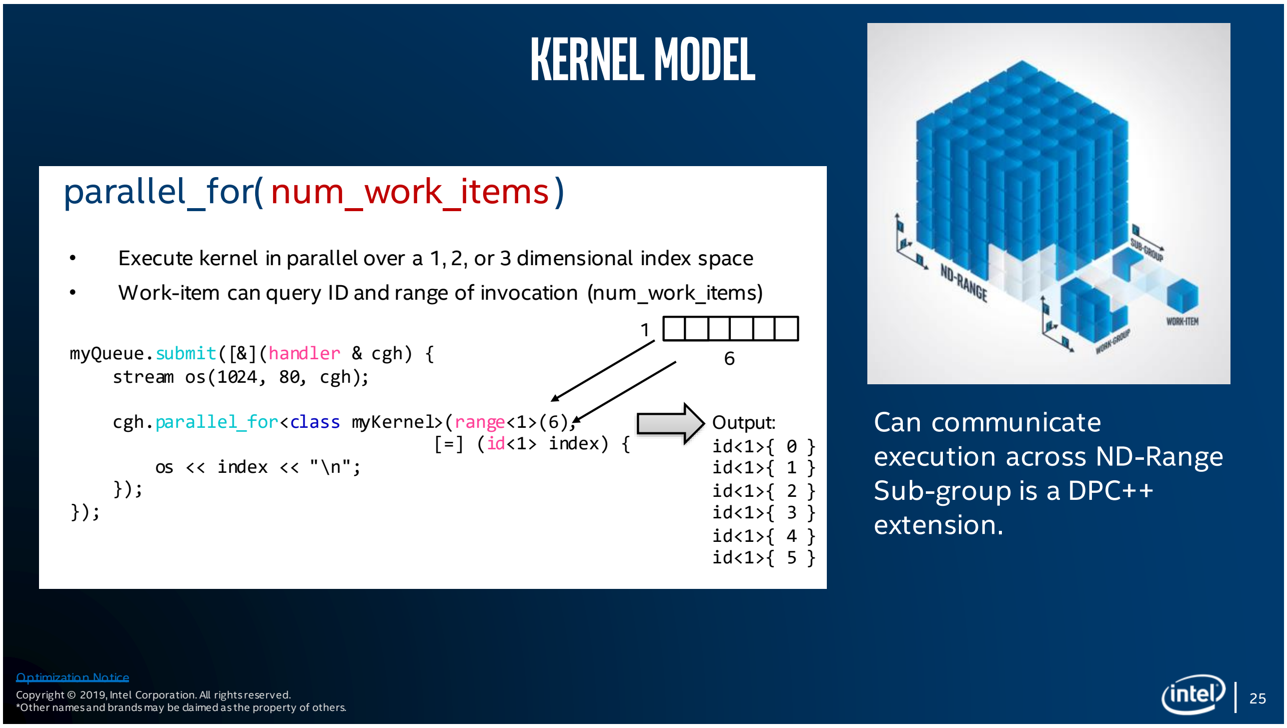
single_task()

- Similar to CPU code with an outer loop
- Allows many-staged custom hardware to be built in an FPGA

```
myQueue.submit([&](handler & cgh) {
    stream os(1024, 80, cgh);

    cgh.single_task<class myKernel>([=] () {
      for (int i=0;i<NUM_ELEMENTS;i++) {
        os << i << "\n";
      }
    });
});
```

```
Output:
 0
 1
 2
 3
 4
 5
```

A custom hardware datapath can be generated in an FPGA for complex single_task kernels



KERNEL MODEL

parallel_for(num_work_items)

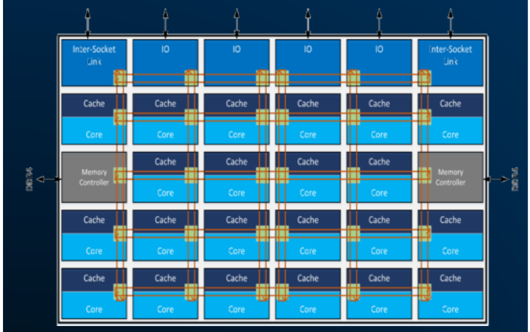
- Execute kernel in parallel over a 1, 2, or 3 dimensional index space
- Work-item can query ID and range of invocation (num_work_items)

```
myQueue.submit([&](handler & cgh) {
    stream os(1024, 80, cgh);

    cgh.parallel_for<class myKernel>(range<1>(6),
                              [=] (id<1> index) {
        os << index << "\n";
    });
});
```

```
Output:
id<1>{ 0 }
id<1>{ 1 }
id<1>{ 2 }
id<1>{ 3 }
id<1>{ 4 }
id<1>{ 5 }
```

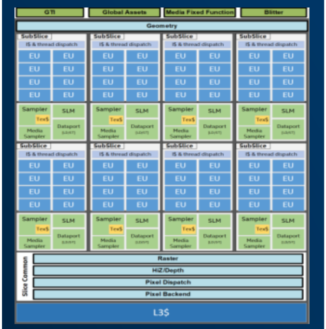
Can communicate execution across ND-Range
Sub-group is a DPC++ extension.

HOW IT CAN BE CODED FOR CPU, GPU, FPGA



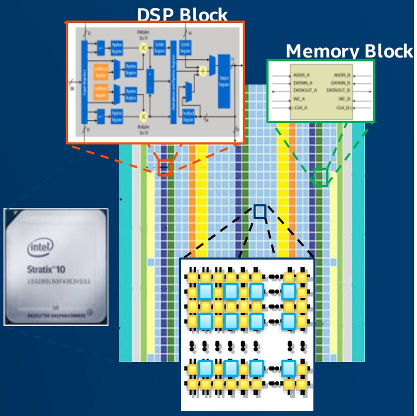
CPU

- MULTI-CORE
- MULTI-THREADED
- SIMD
- PIPELINED



GPU

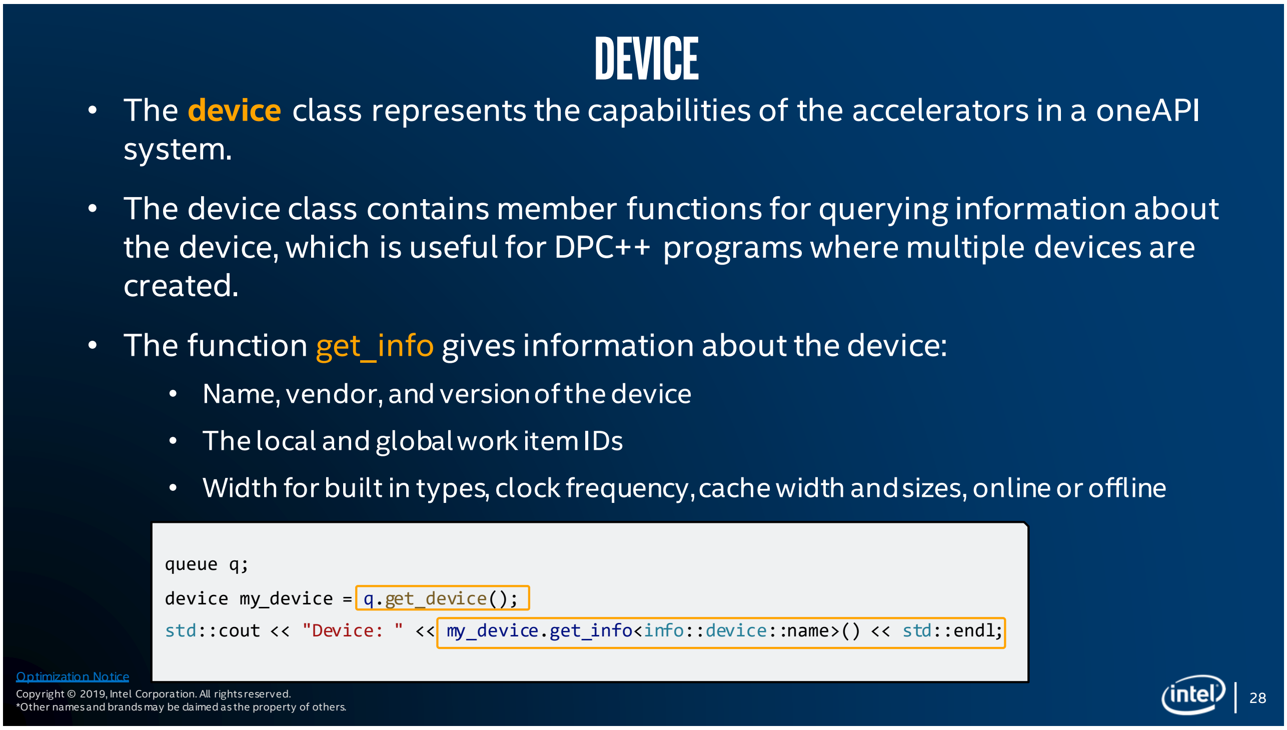
- MULTI-CORE
- MULTI-THREADED
- SIMD
- PIPELINED



FPGA

- Custom Pipeline
- MULTI-CORE (pipeline)

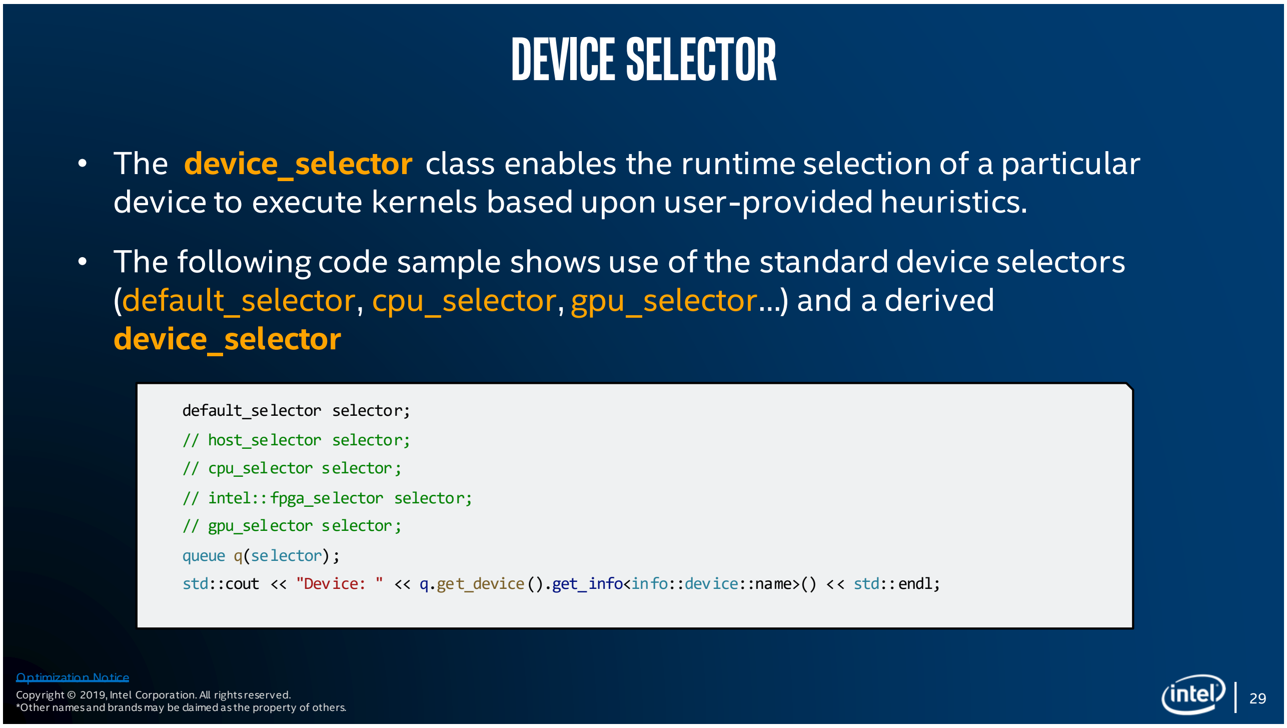
SYCL CLASSES



DEVICE

- The **device** class represents the capabilities of the accelerators in a oneAPI system.
- The device class contains member functions for querying information about the device, which is useful for DPC++ programs where multiple devices are created.
- The function get_info gives information about the device:
  - Name, vendor, and version of the device
  - The local and global work item IDs
  - Width for built in types, clock frequency, cache width and sizes, online or offline

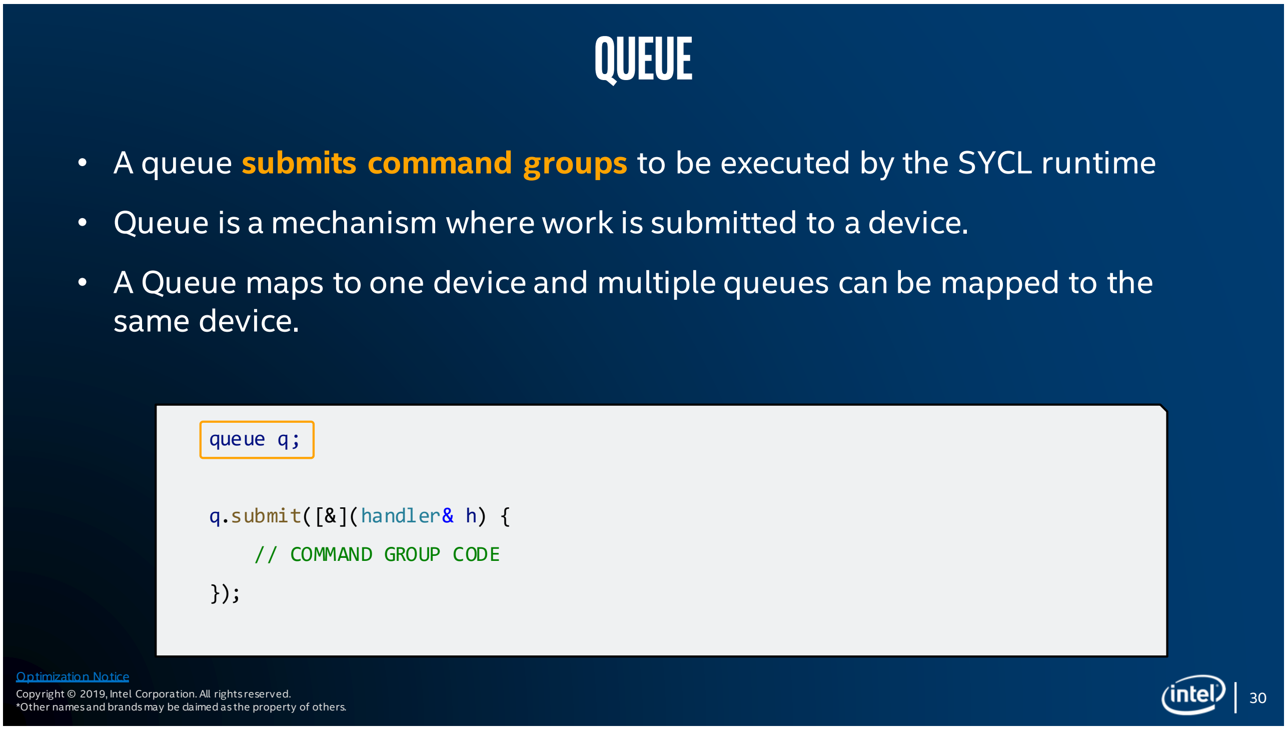
```
queue q;
device my_device = q.get_device();
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```



DEVICE SELECTOR

- The **device_selector** class enables the runtime selection of a particular device to execute kernels based upon user-provided heuristics.
- The following code sample shows use of the standard device selectors (default_selector, cpu_selector, gpu_selector…) and a derived **device_selector**

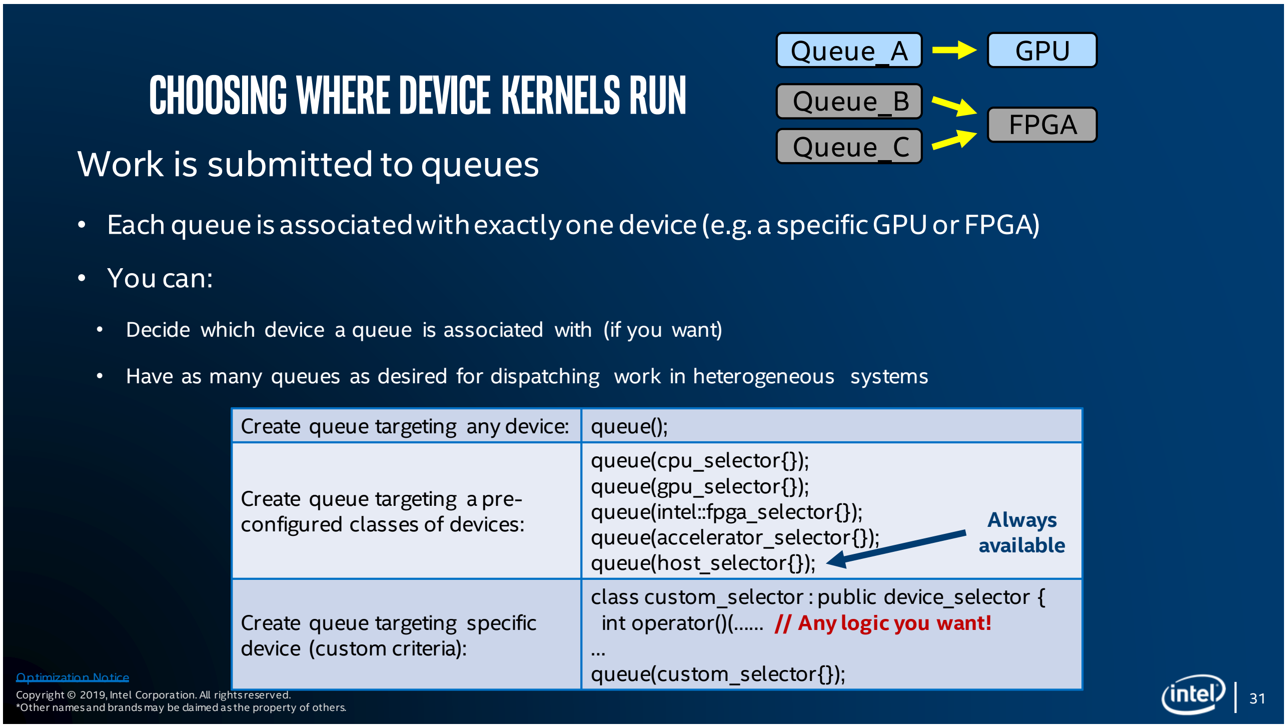
```
default_selector selector;
// host_selector selector;
// cpu_selector selector;
// intel::fpga_selector selector;
// gpu_selector selector;
queue q(selector);
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```



QUEUE

- A queue **submits command groups** to be executed by the SYCL runtime
- Queue is a mechanism where work is submitted to a device.
- A Queue maps to one device and multiple queues can be mapped to the same device.

```
queue q;

q.submit([&](handler& h) {
    // COMMAND GROUP CODE
});
```



CHOOSING WHERE DEVICE KERNELS RUN

Queue_A
GPU
Queue_B
FPGA
Queue_C

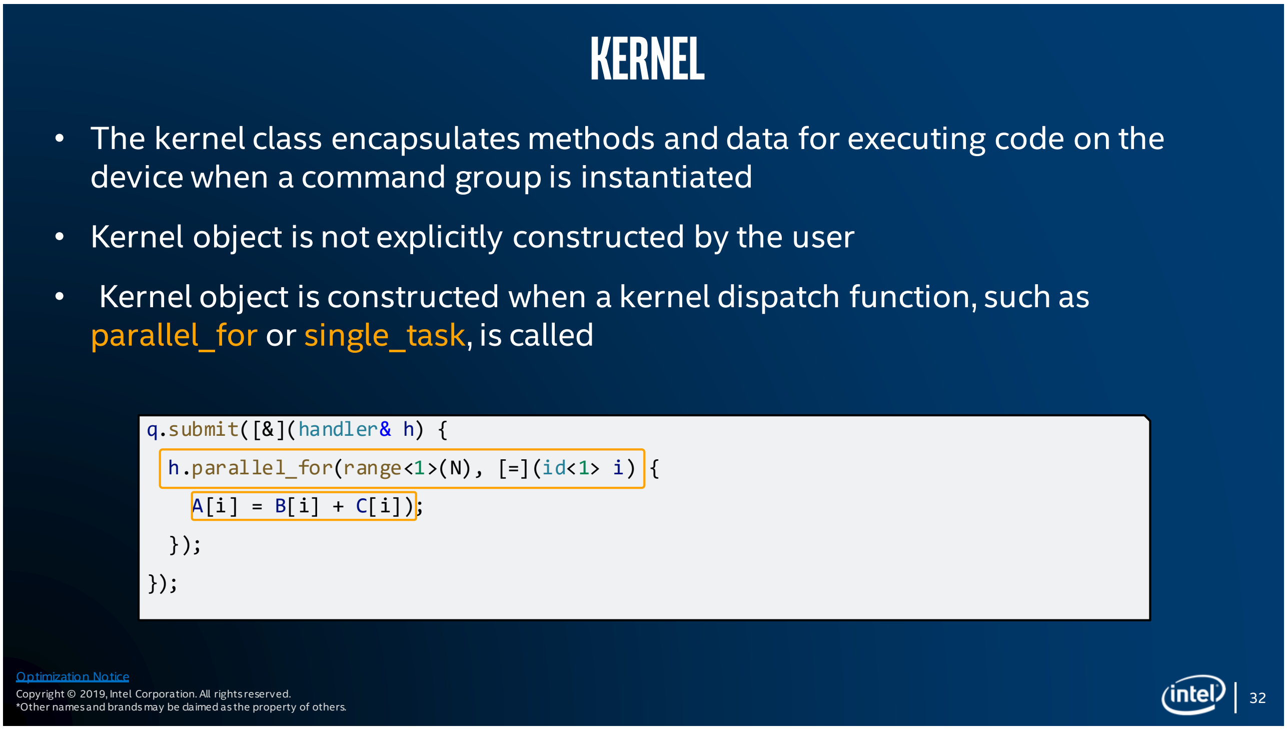
Work is submitted to queues

- Each queue is associated with exactly one device (e.g. a specific GPU or FPGA)
- You can:
  - Decide which device a queue is associated with (if you want)
  - Have as many queues as desired for dispatching work in heterogeneous systems

Create queue targeting any device:	queue();

Create queue targeting a pre-configured classes of devices:	queue(cpu_selector{});
	queue(gpu_selector{});
	queue(intel::fpga_selector{});
	queue(accelerator_selector{});
	queue(host_selector{}); **Always available**

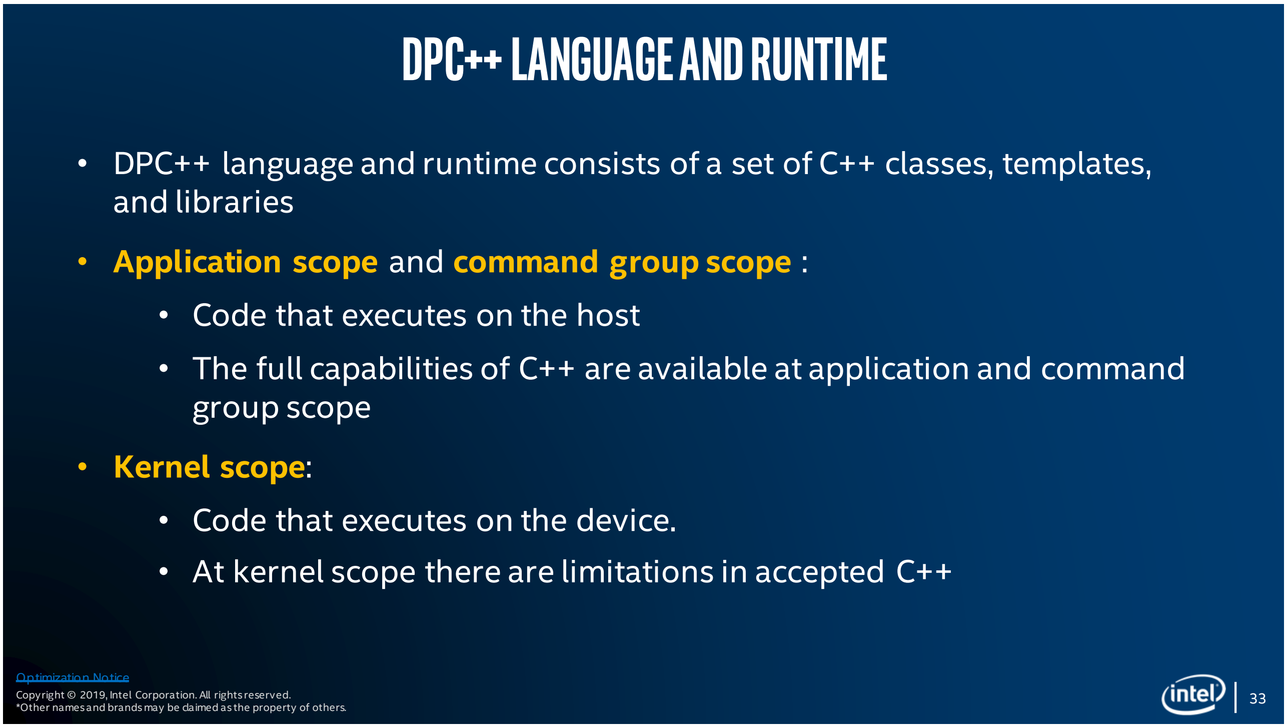
Create queue targeting specific device (custom criteria):	class custom_selector : public device_selector {
	int operator()(…… **// Any logic you want!**
	…
	queue(custom_selector{});



KERNEL

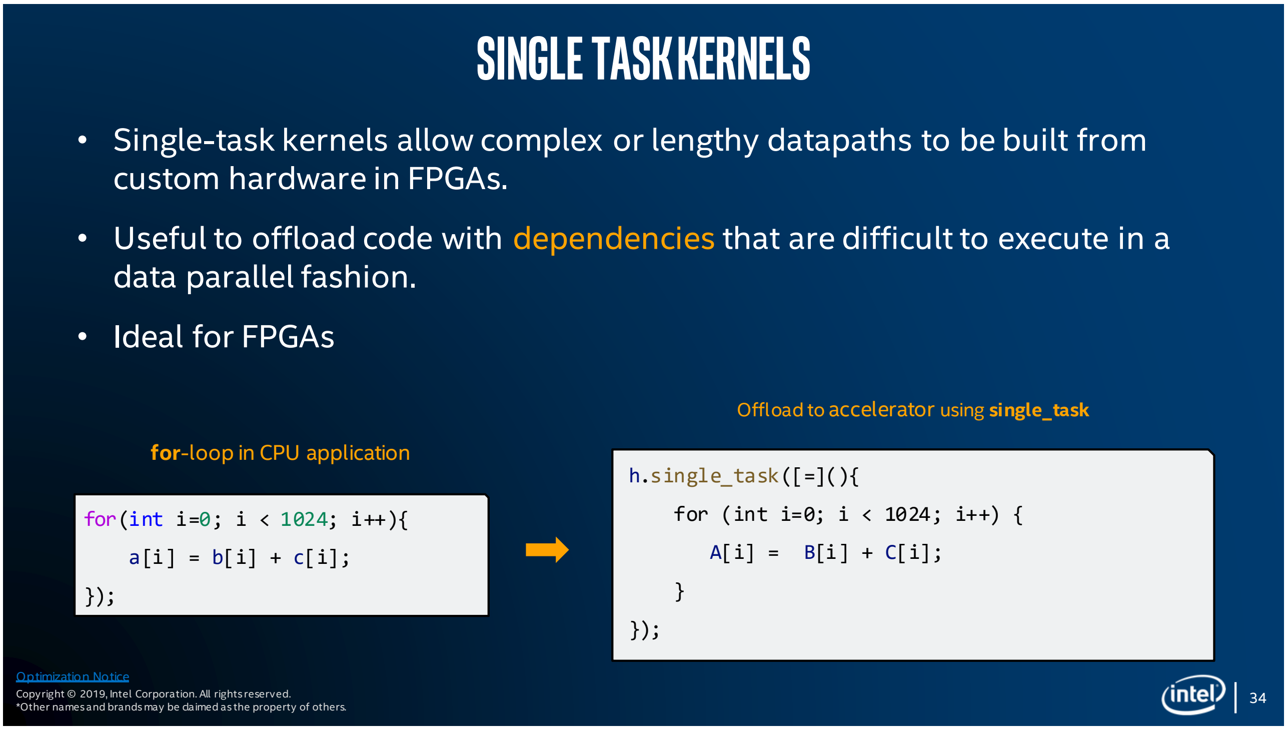
- The kernel class encapsulates methods and data for executing code on the device when a command group is instantiated
- Kernel object is not explicitly constructed by the user
- Kernel object is constructed when a kernel dispatch function, such as parallel_for or single_task, is called

```
q.submit([&](handler& h) {
  h.parallel_for(range<1>(N), [=](id<1> i) {
    A[i] = B[i] + C[i]);
  });
});
```



DPC++ LANGUAGE AND RUNTIME

- DPC++ language and runtime consists of a set of C++ classes, templates, and libraries
- **Application scope** and **command group scope** :
  - Code that executes on the host
  - The full capabilities of C++ are available at application and command group scope
- **Kernel scope**:
  - Code that executes on the device.
  - At kernel scope there are limitations in accepted C++



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.
- Ideal for FPGAs

for-loop in CPU application

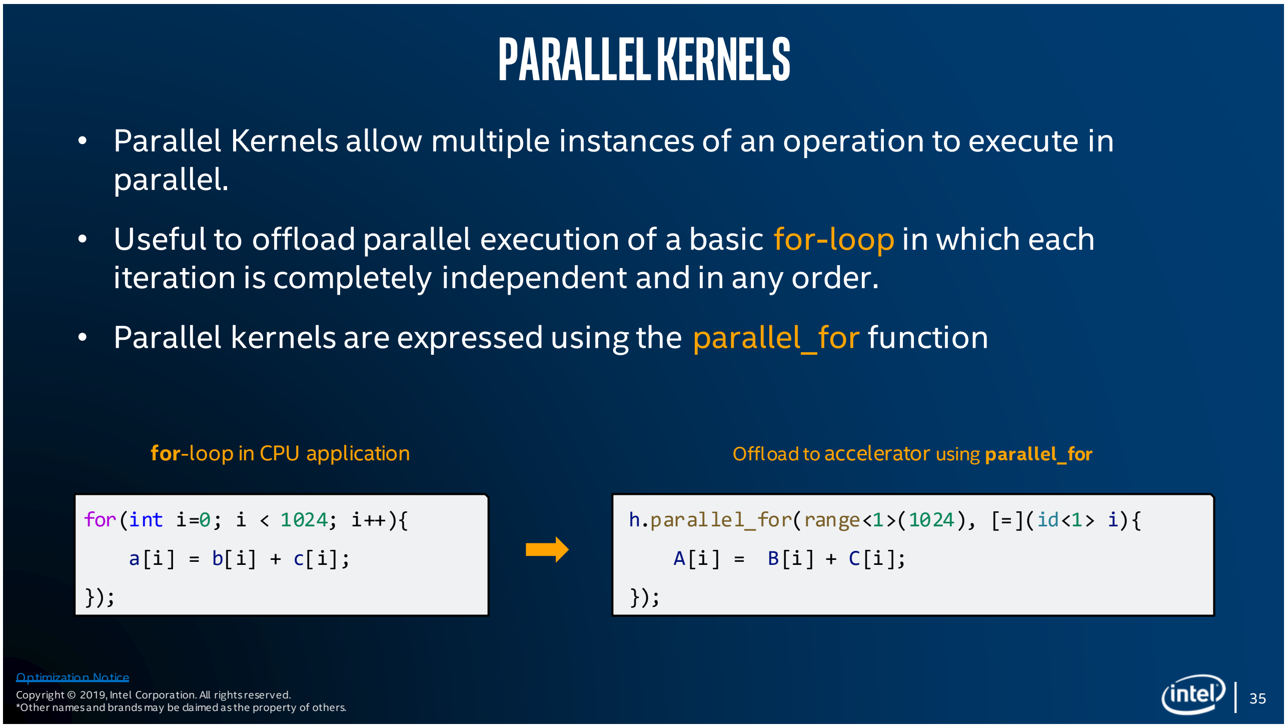
```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

➡

Offload to accelerator using **single_task**

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

Optimization Notice



PARALLEL KERNELS

- Parallel Kernels allow multiple instances of an operation to execute in parallel.
- Useful to offload parallel execution of a basic for-loop in which each iteration is completely independent and in any order.
- Parallel kernels are expressed using the parallel_for function

for-loop in CPU application

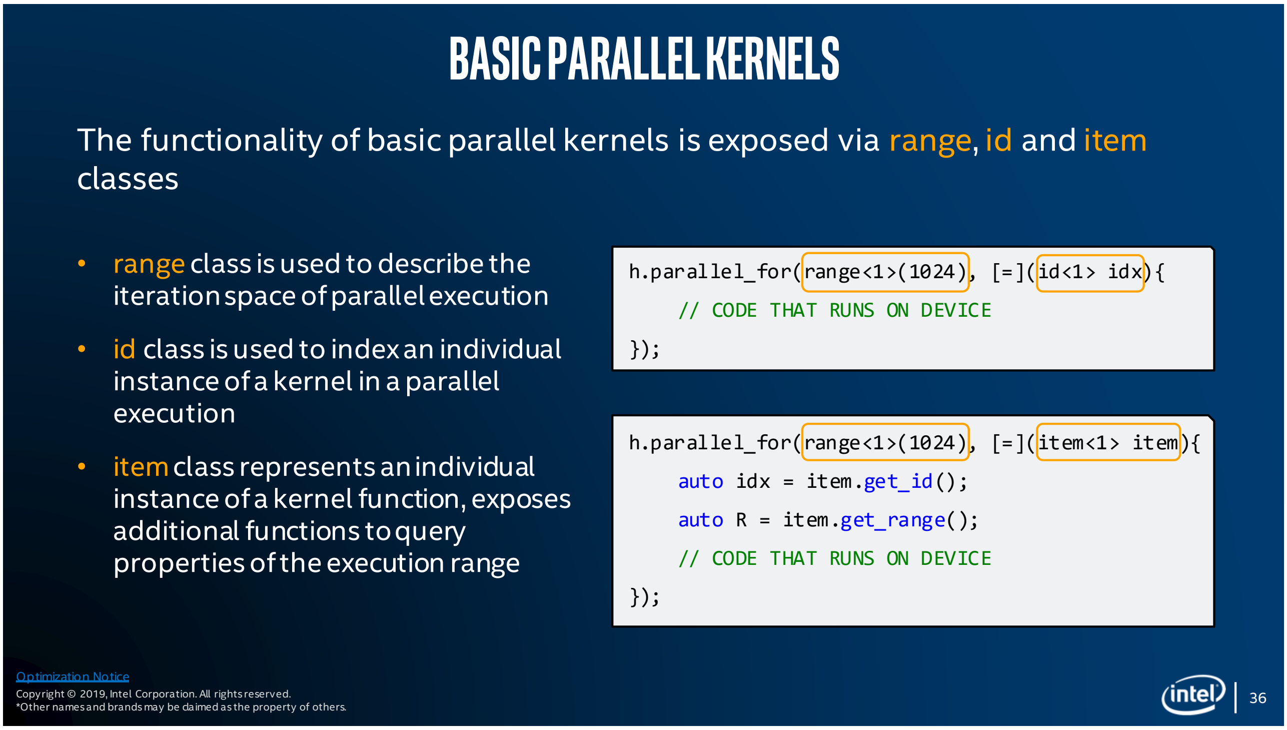
```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

➡

Offload to accelerator using **parallel_for**

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

Optimization Notice



BASIC PARALLEL KERNELS

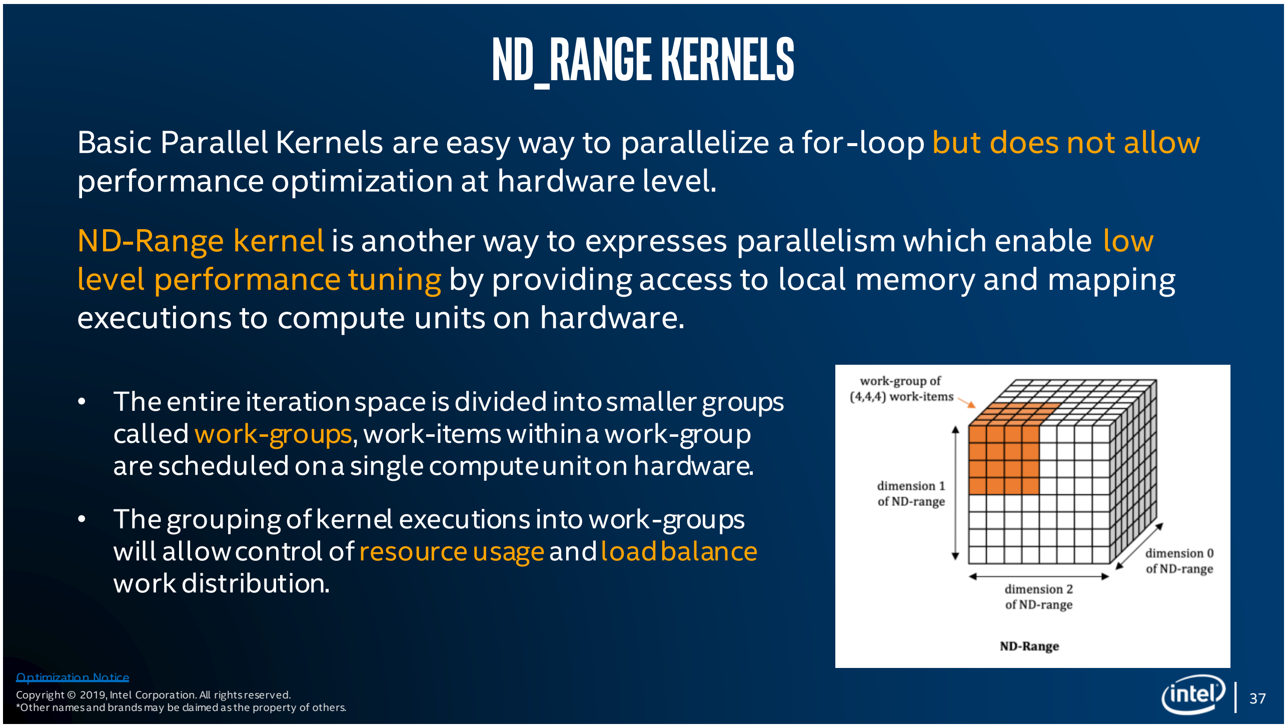
The functionality of basic parallel kernels is exposed via range, id and item classes

- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
    // CODE THAT RUNS ON DEVICE
});
```

```
h.parallel_for(range<1>(1024), [=](item<1> item){
    auto idx = item.get_id();
    auto R = item.get_range();
    // CODE THAT RUNS ON DEVICE
});
```

Optimization Notice



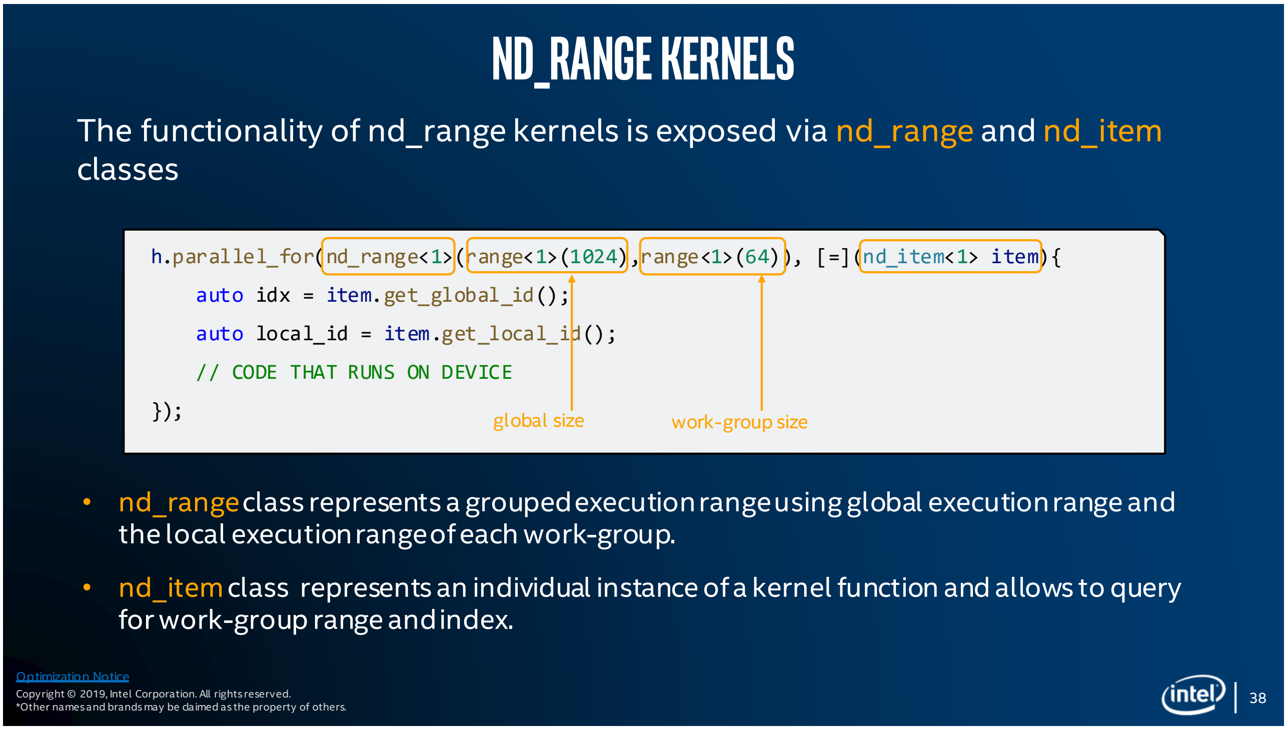
ND_RANGE KERNELS

Basic Parallel Kernels are easy way to parallelize a for-loop but does not allow performance optimization at hardware level.

ND-Range kernel is another way to expresses parallelism which enable low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.

- The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware.
- The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution.

Optimization Notice



ND_RANGE KERNELS

The functionality of nd_range kernels is exposed via nd_range and nd_item classes

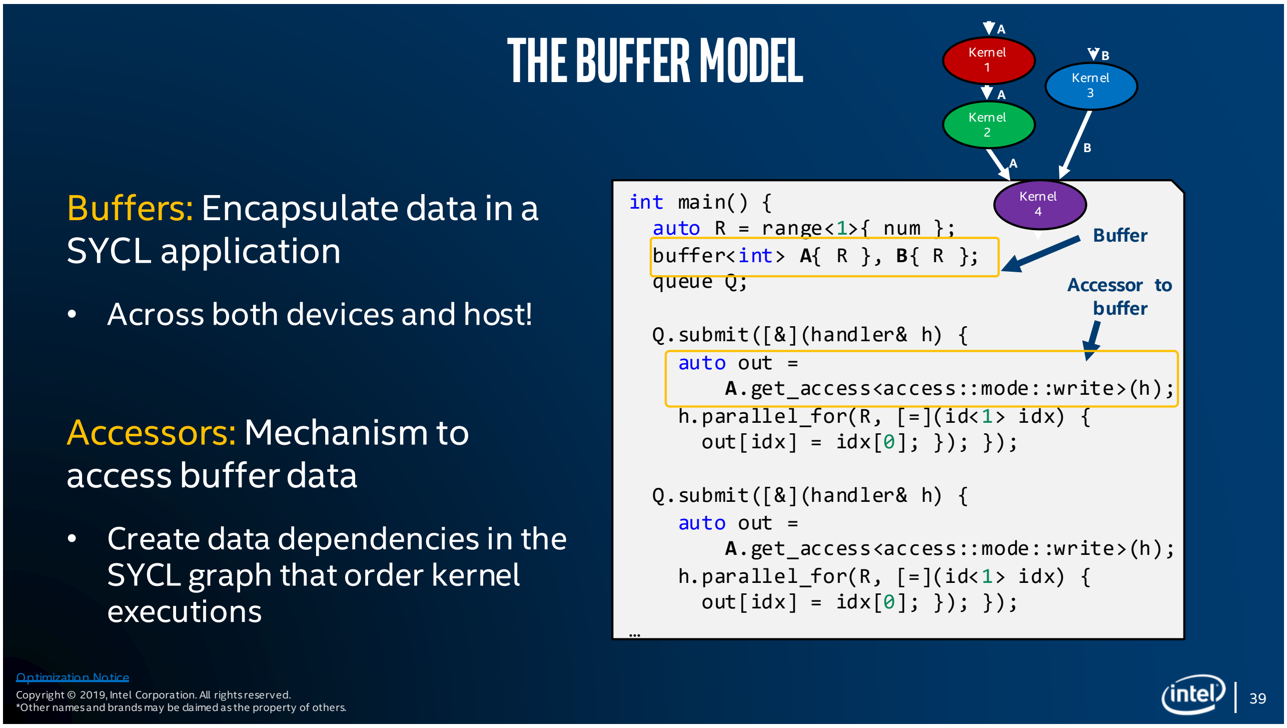
```
h.parallel_for(nd_range<1>(range<1>(1024),range<1>(64)), [=](nd_item<1> item){
    auto idx = item.get_global_id();
    auto local_id = item.get_local_id();
    // CODE THAT RUNS ON DEVICE
});
```

global size

work-group size

- nd_range class represents a grouped execution range using global execution range and the local execution range of each work-group.
- nd_item class represents an individual instance of a kernel function and allows to query for work-group range and index.

Optimization Notice



THE BUFFER MODEL

A
Kernel 1
A
Kernel 2
B
Kernel 3
B
A
Kernel 4

Buffers: Encapsulate data in a SYCL application

- Across both devices and host!

Accessors: Mechanism to access buffer data

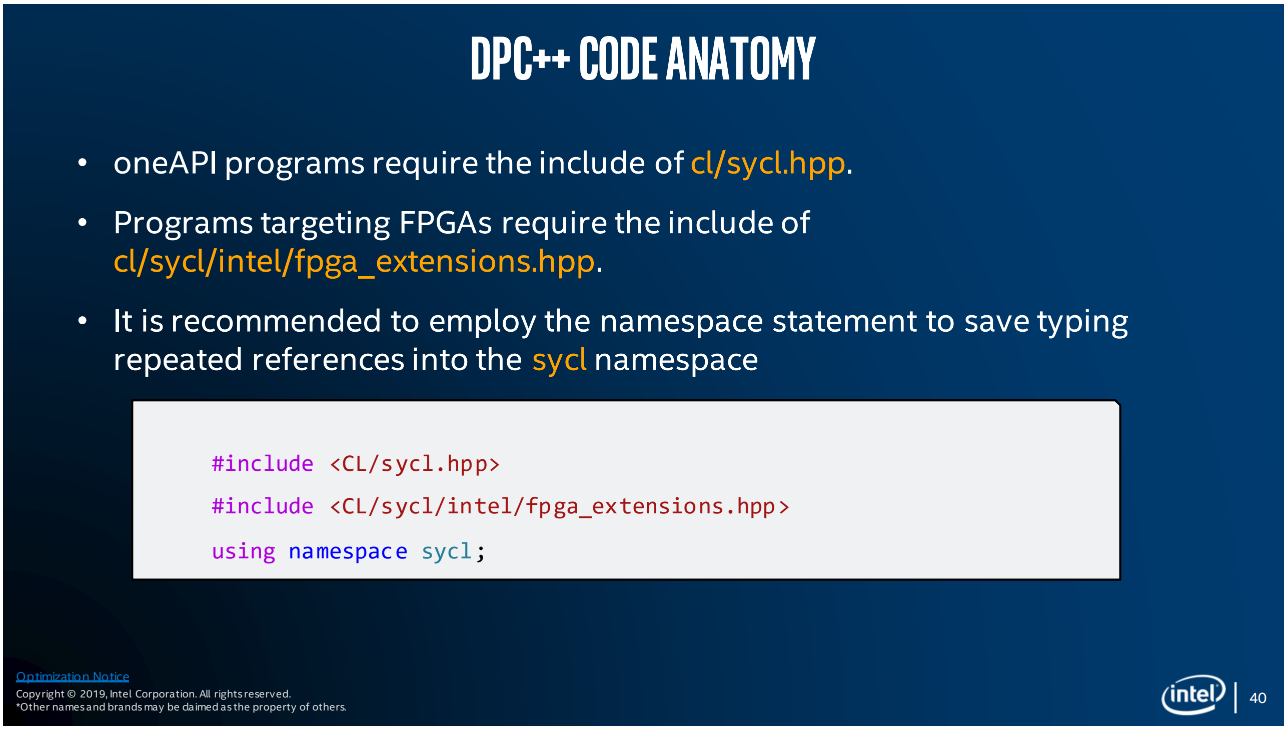
- Create data dependencies in the SYCL graph that order kernel executions

```
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::write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); });

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

Buffer

Accessor to buffer

Optimization Notice

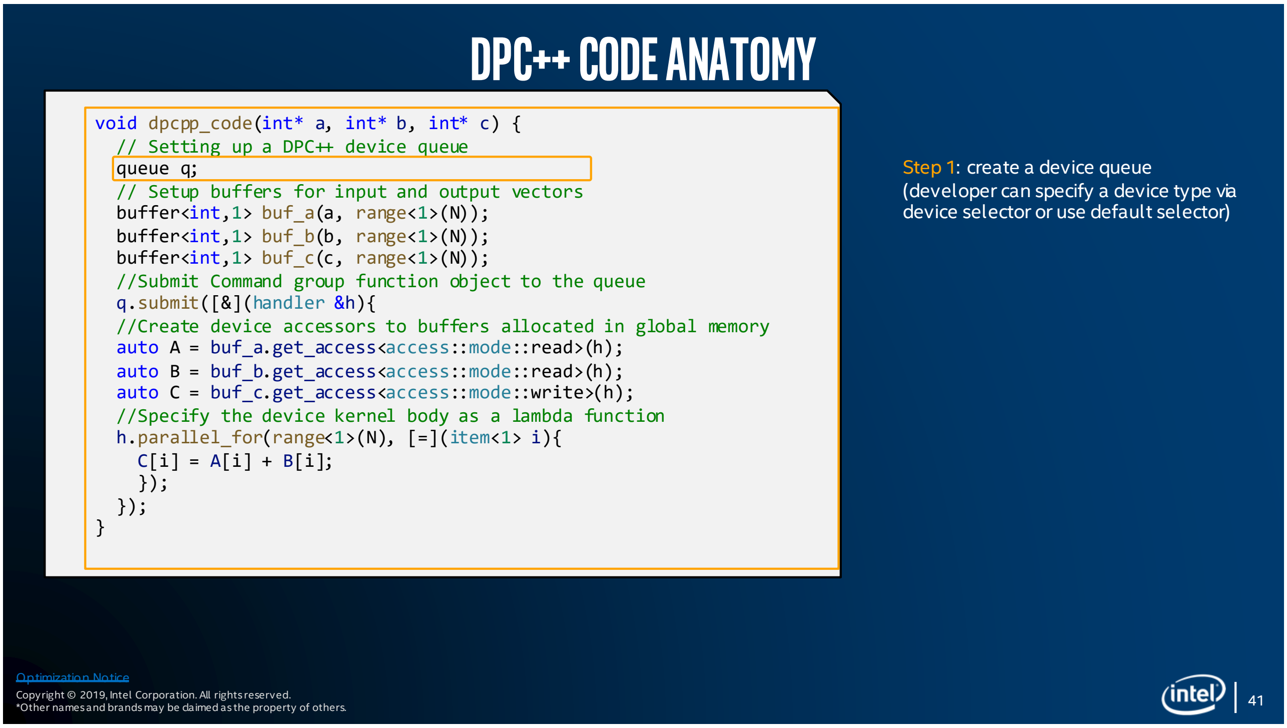


DPC++ CODE ANATOMY

- oneAPI programs require the include of cl/sycl.hpp.
- Programs targeting FPGAs require the include of cl/sycl/intel/fpga_extensions.hpp.
- It is recommended to employ the namespace statement to save typing repeated references into the sycl namespace

```
#include <CL/sycl.hpp>
#include <CL/sycl/intel/fpga_extensions.hpp>
using namespace sycl;
```

Optimization Notice

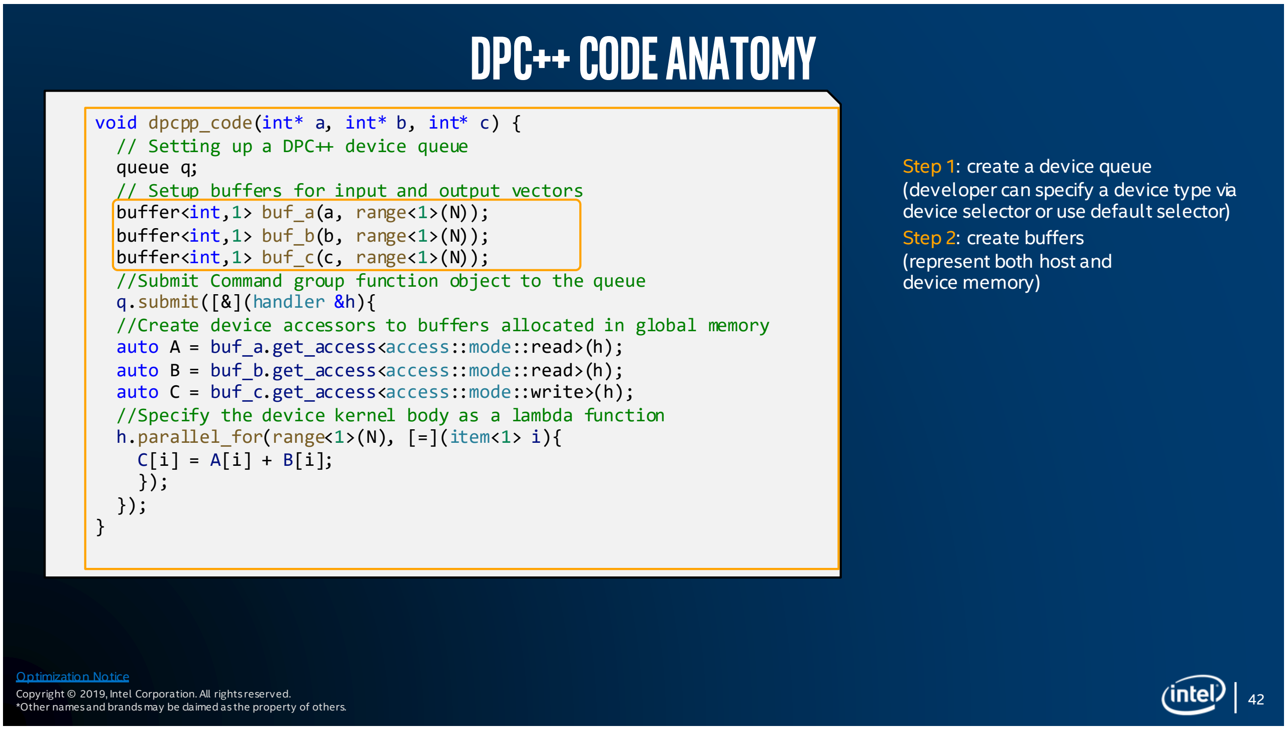


DPC++ CODE ANATOMY

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue
(developer can specify a device type via device selector or use default selector)

Optimization Notice

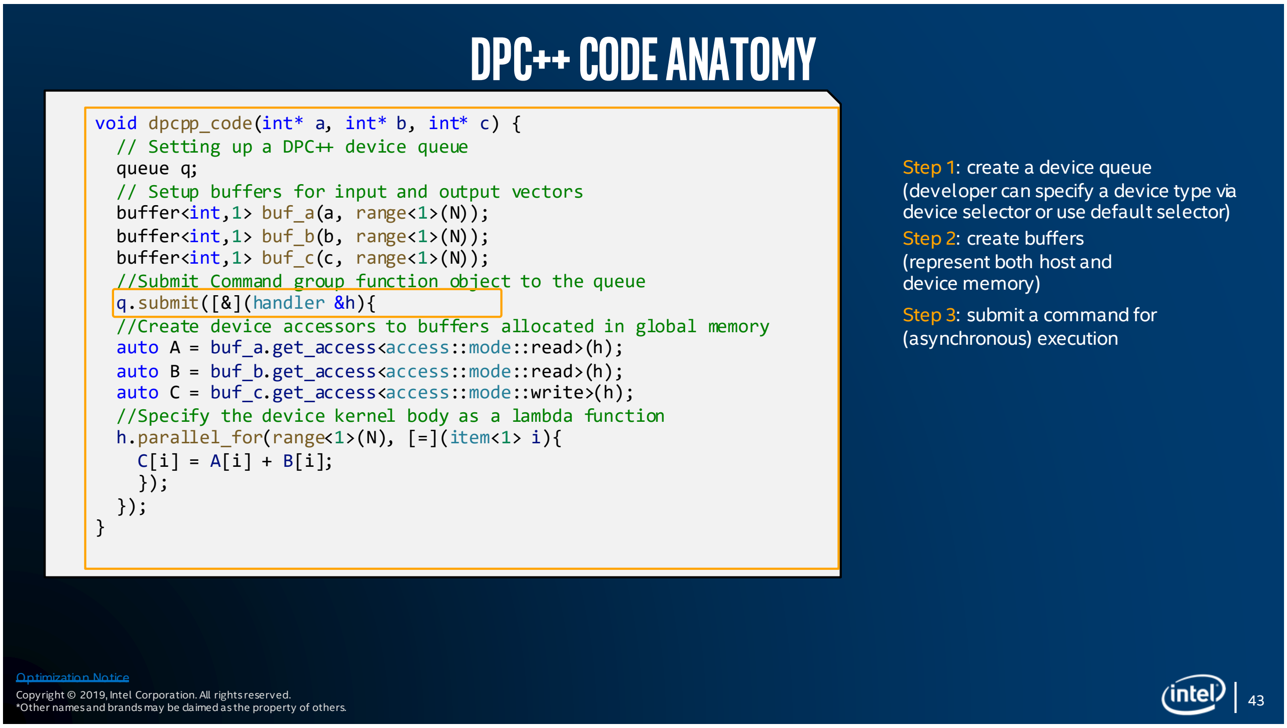


DPC++ CODE ANATOMY

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue
(developer can specify a device type via
device selector or use default selector)

Step 2: create buffers
(represent both host and
device memory)



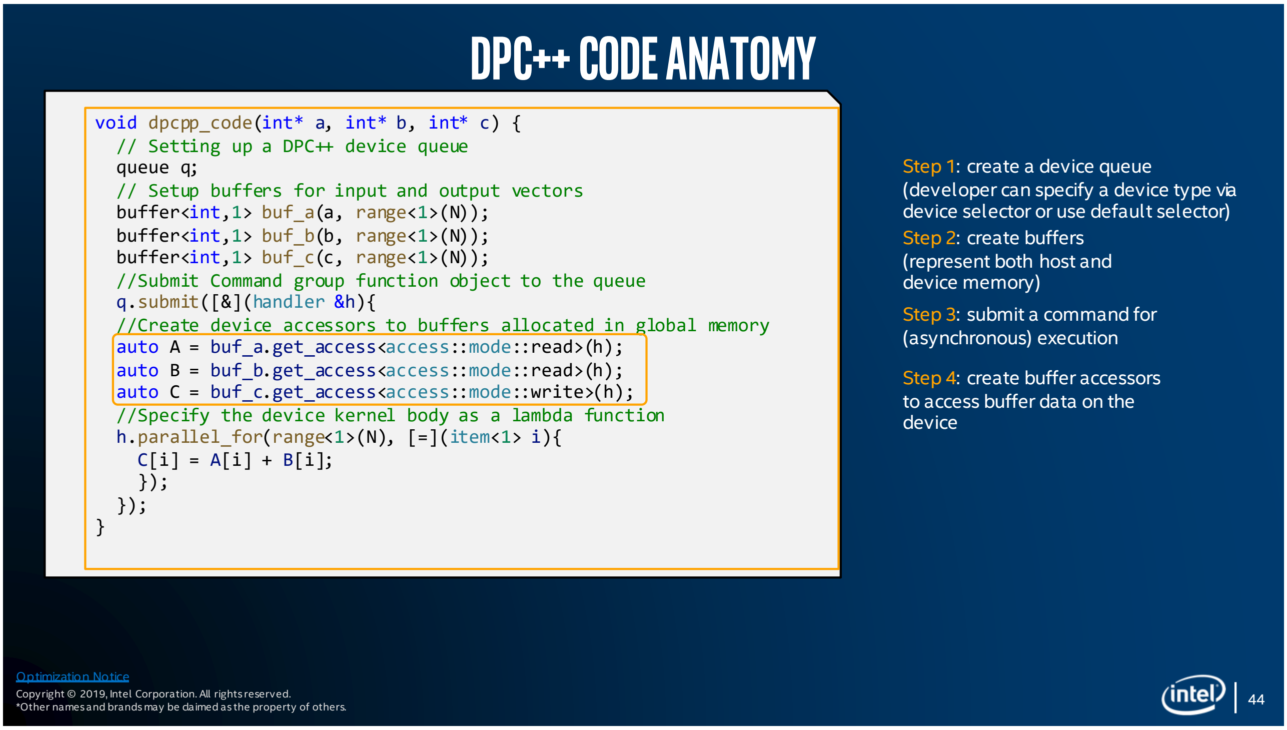
DPC++ CODE ANATOMY

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue
(developer can specify a device type via
device selector or use default selector)

Step 2: create buffers
(represent both host and
device memory)

Step 3: submit a command for
(asynchronous) execution



DPC++ CODE ANATOMY

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

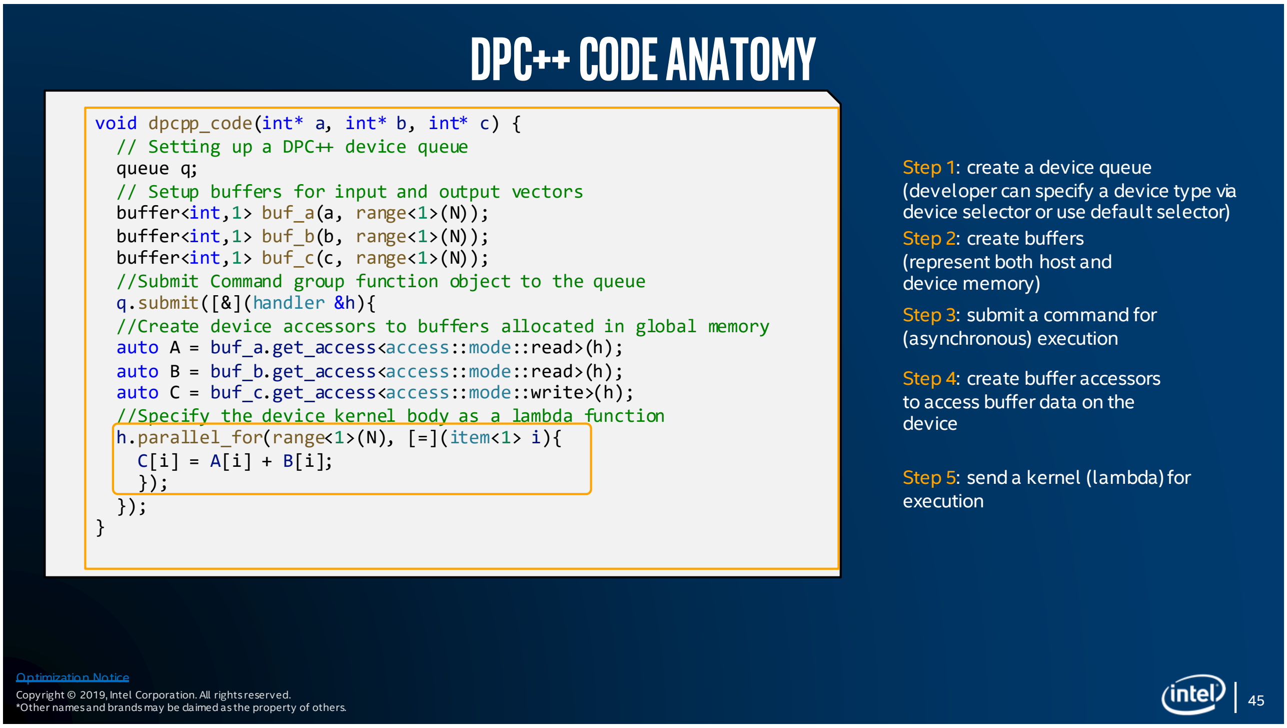
Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Optimization Notice



DPC++ CODE ANATOMY

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

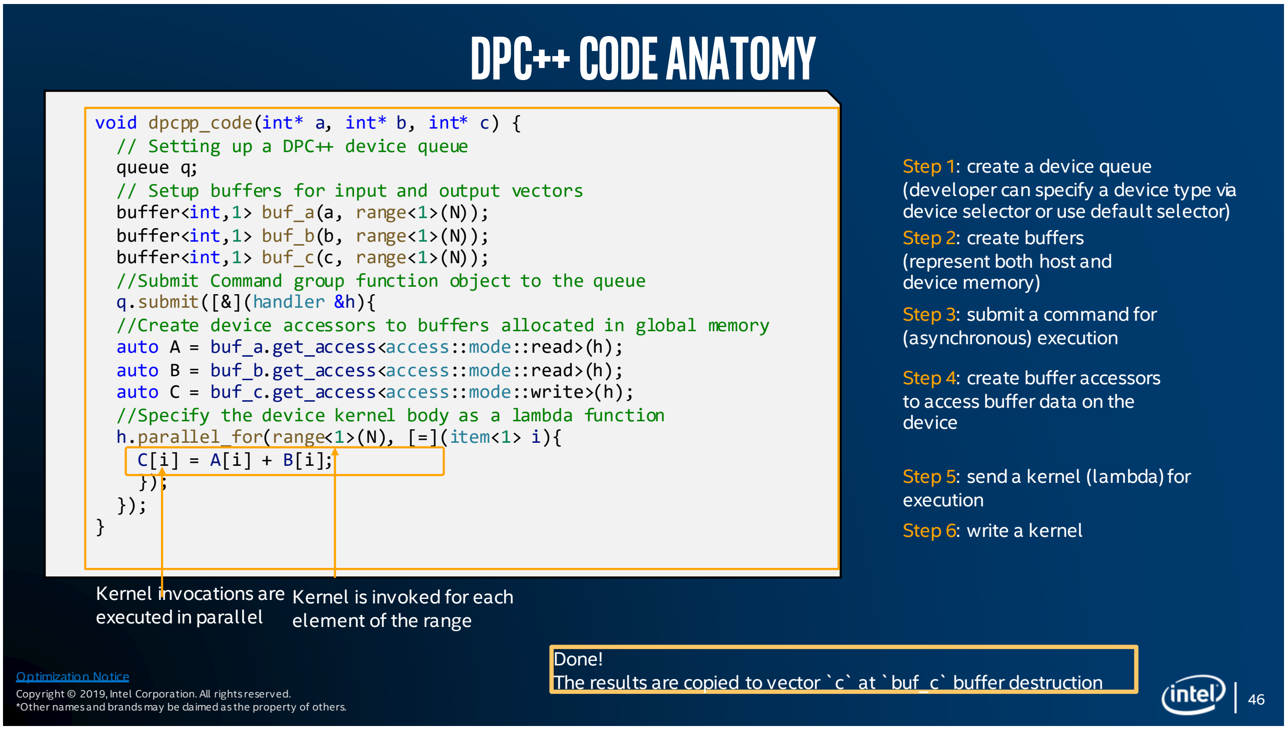
Step 2: create buffers (represent both host and device memory)

Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Step 5: send a kernel (lambda) for execution

Optimization Notice



DPC++ CODE ANATOMY

```
void dpcpp_code(int* a, int* b, int* c) {
  // Setting up a DPC++ device queue
  queue q;
  // Setup buffers for input and output vectors
  buffer<int,1> buf_a(a, range<1>(N));
  buffer<int,1> buf_b(b, range<1>(N));
  buffer<int,1> buf_c(c, range<1>(N));
  //Submit Command group function object to the queue
  q.submit([&](handler &h){
  //Create device accessors to buffers allocated in global memory
  auto A = buf_a.get_access<access::mode::read>(h);
  auto B = buf_b.get_access<access::mode::read>(h);
  auto C = buf_c.get_access<access::mode::write>(h);
  //Specify the device kernel body as a lambda function
  h.parallel_for(range<1>(N), [=](item<1> i){
    C[i] = A[i] + B[i];
    });
  });
}
```

Kernel invocations are executed in parallel

Kernel is invoked for each element of the range

Step 1: create a device queue (developer can specify a device type via device selector or use default selector)

Step 2: create buffers (represent both host and device memory)

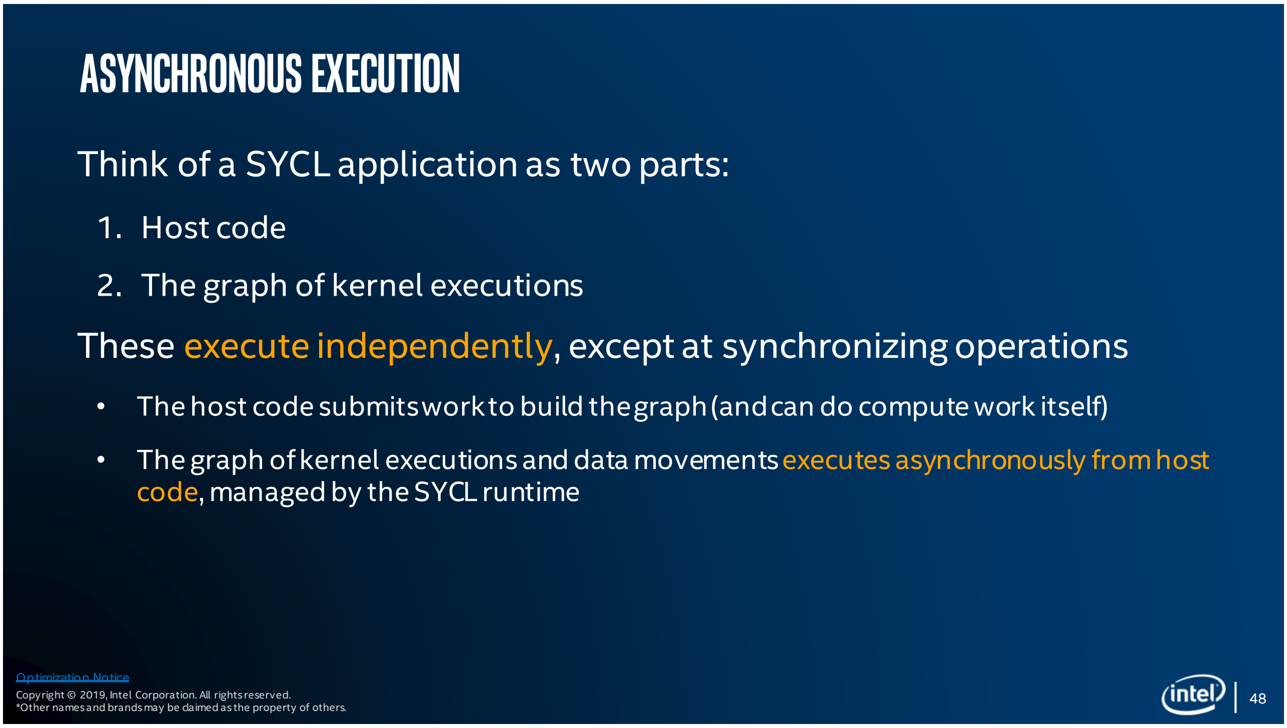
Step 3: submit a command for (asynchronous) execution

Step 4: create buffer accessors to access buffer data on the device

Step 5: send a kernel (lambda) for execution

Step 6: write a kernel

Done!
The results are copied to vector `c` at `buf_c` buffer destruction



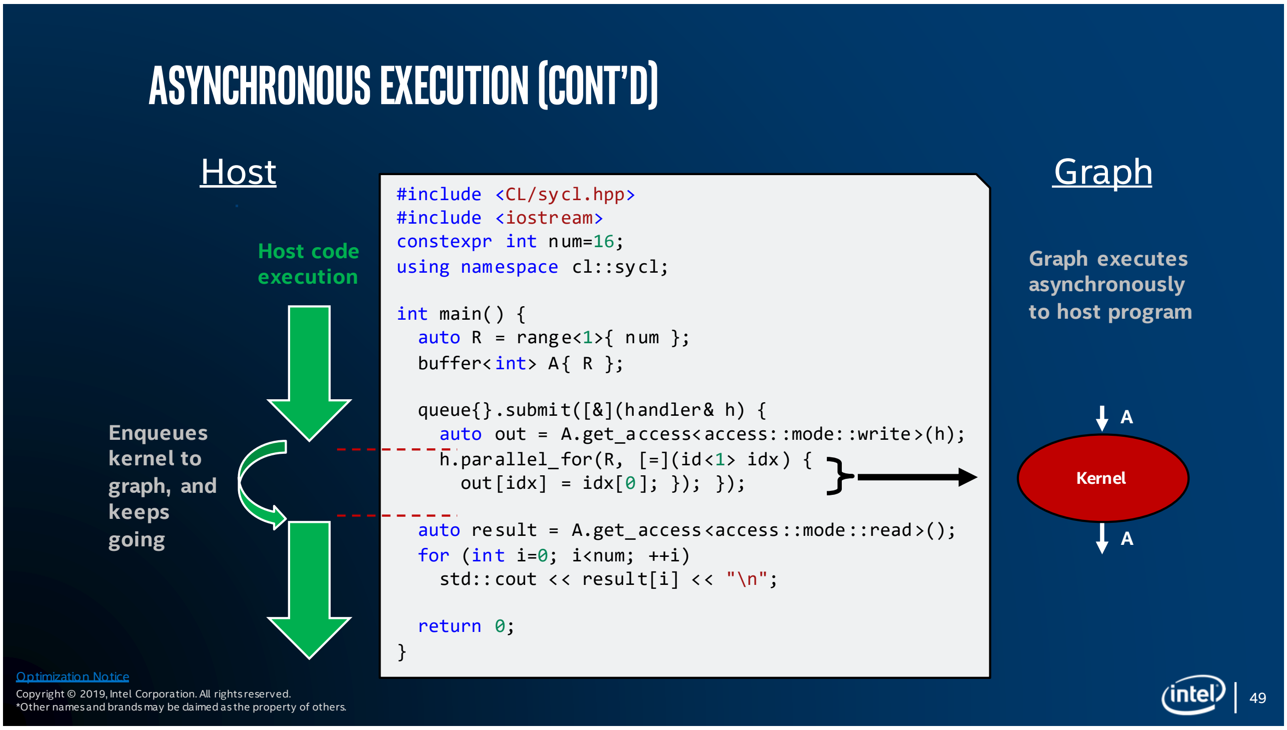
ASYNCHRONOUS EXECUTION

Think of a SYCL application as two parts:

1. Host code
2. The graph of kernel executions

These execute independently, except at synchronizing operations

- The host code submits work to build the graph (and can do compute work itself)
- The graph of kernel executions and data movements executes asynchronously from host code, managed by the SYCL runtime



ASYNCHRONOUS EXECUTION (CONT'D)

Host

Host code execution

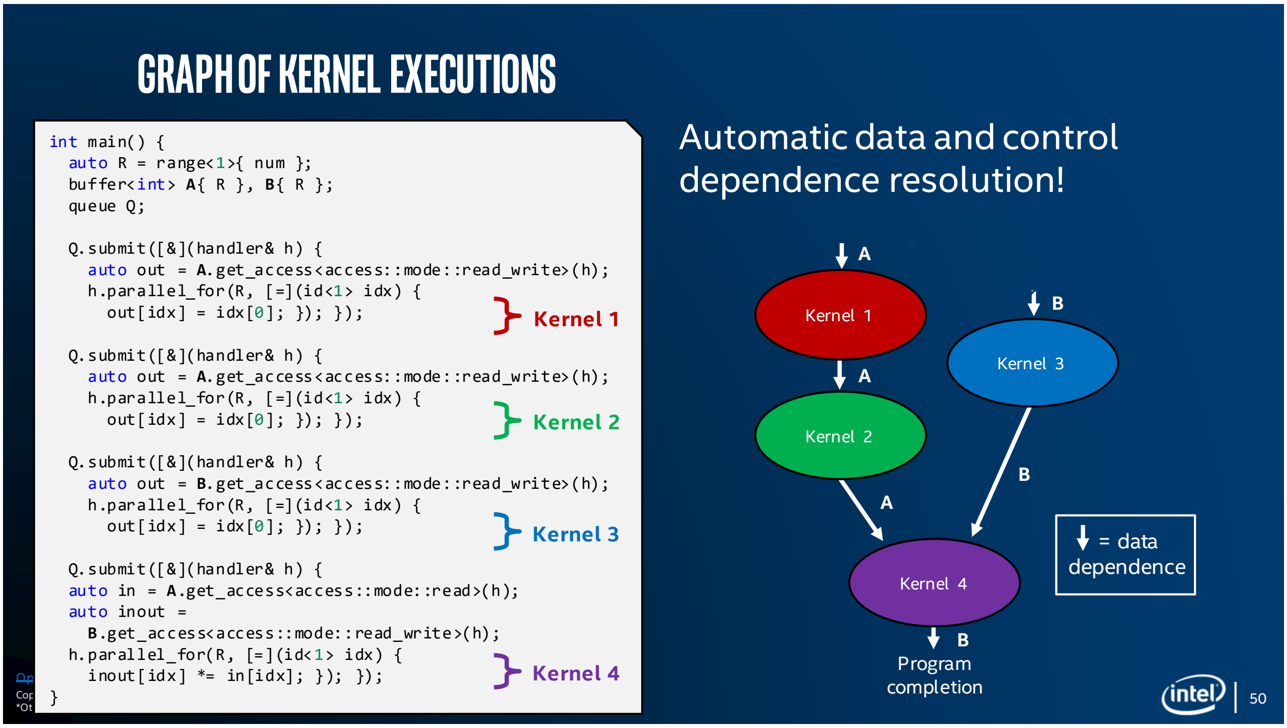
Enqueues kernel to graph, and keeps going

```
#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;
}
```

Graph

Graph executes asynchronously to host program

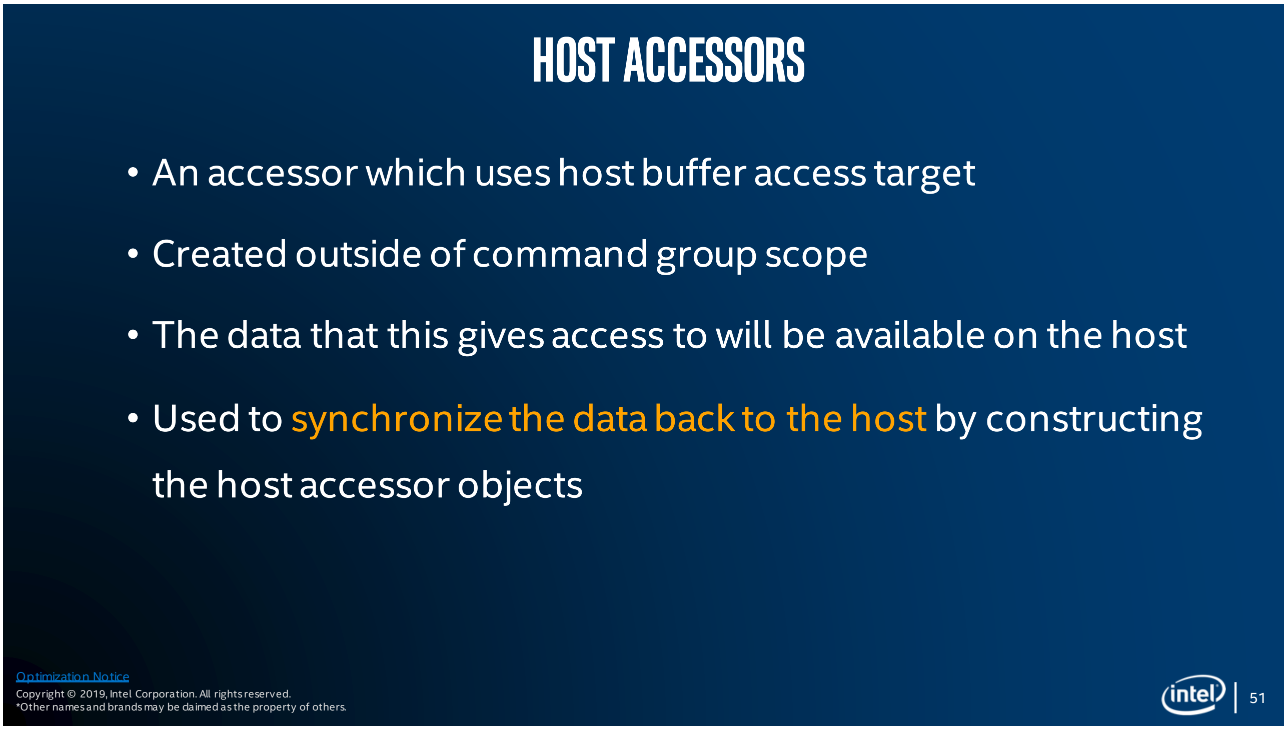
Optimization Notice



GRAPH OF KERNEL EXECUTIONS

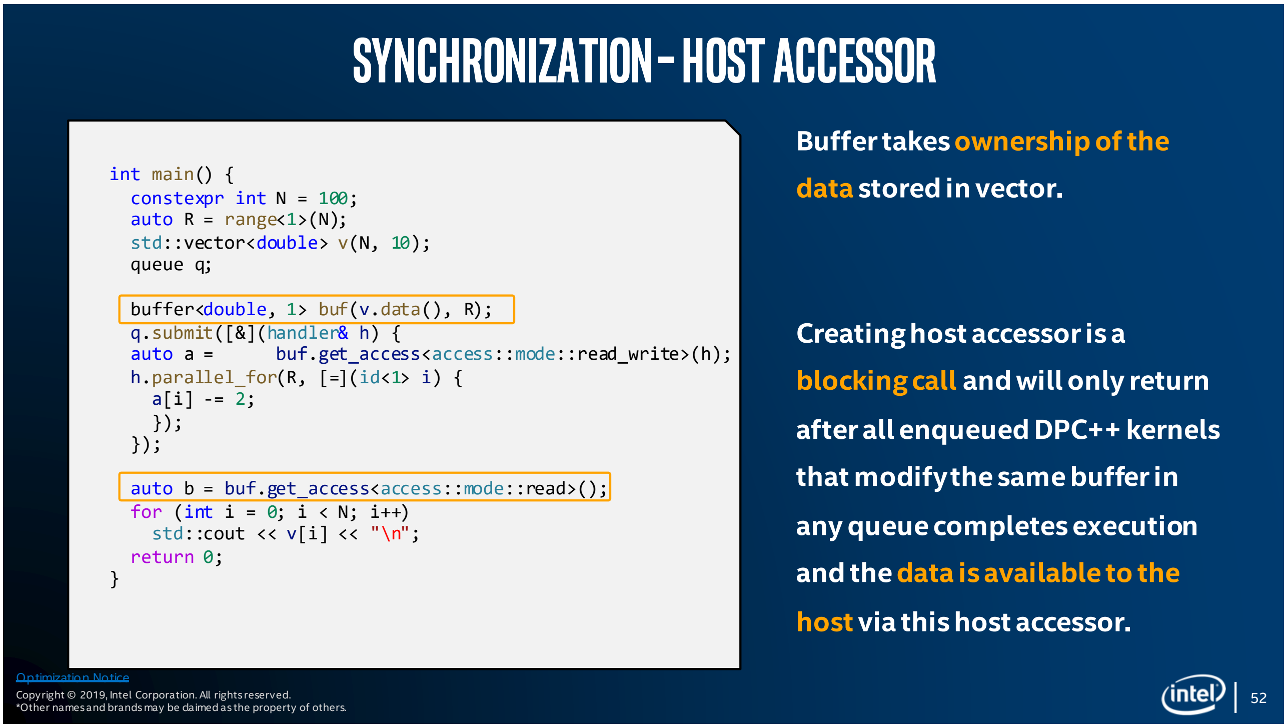
```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  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
}
```

Automatic data and control dependence resolution!



HOST ACCESSORS

- An accessor which uses host buffer access target
- Created outside of command group scope
- The data that this gives access to will be available on the host
- Used to synchronize the data back to the host by constructing the host accessor objects

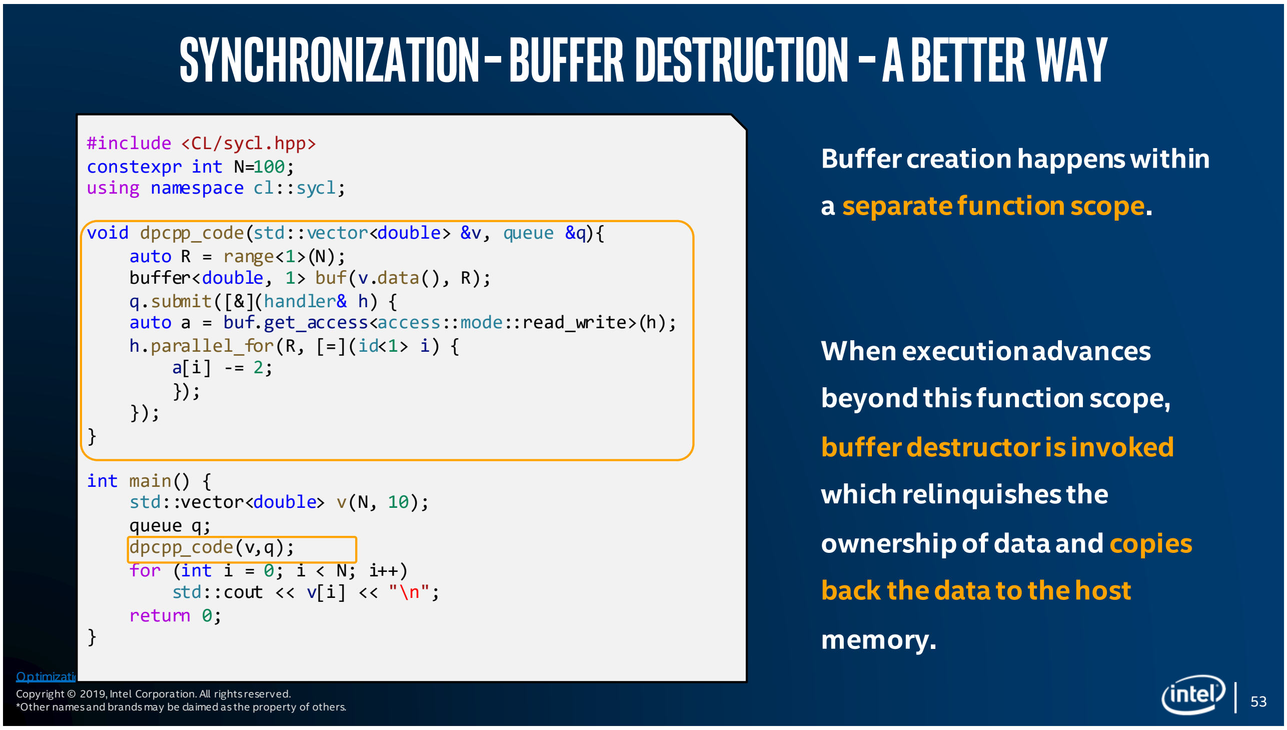


SYNCHRONIZATION – HOST ACCESSOR

```
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;
}
```

Buffer takes ownership of the data stored in vector.

Creating host accessor is a blocking call and will only return after all enqueued DPC++ kernels that modify the same buffer in any queue completes execution and the data is available to the host via this host accessor.



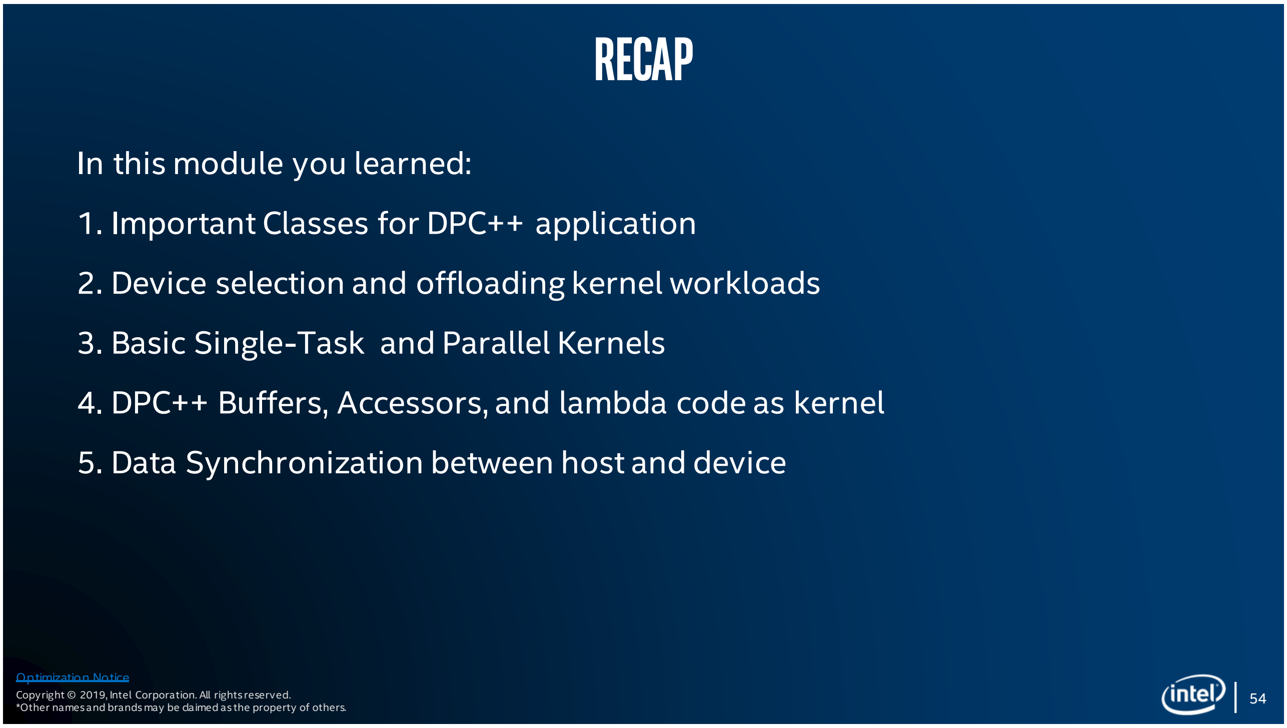
SYNCHRONIZATION – BUFFER DESTRUCTION – A BETTER WAY

```
#include <CL/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;
}
```

Buffer creation happens within a **separate function scope**.

When execution advances beyond this function scope, **buffer destructor is invoked** which relinquishes the ownership of data and **copies back the data to the host** memory.

Optimizati



RECAP

In this module you learned:

1. Important Classes for DPC++ application
2. Device selection and offloading kernel workloads
3. Basic Single-Task and Parallel Kernels
4. DPC++ Buffers, Accessors, and lambda code as kernel
5. Data Synchronization between host and device

Optimization Notice

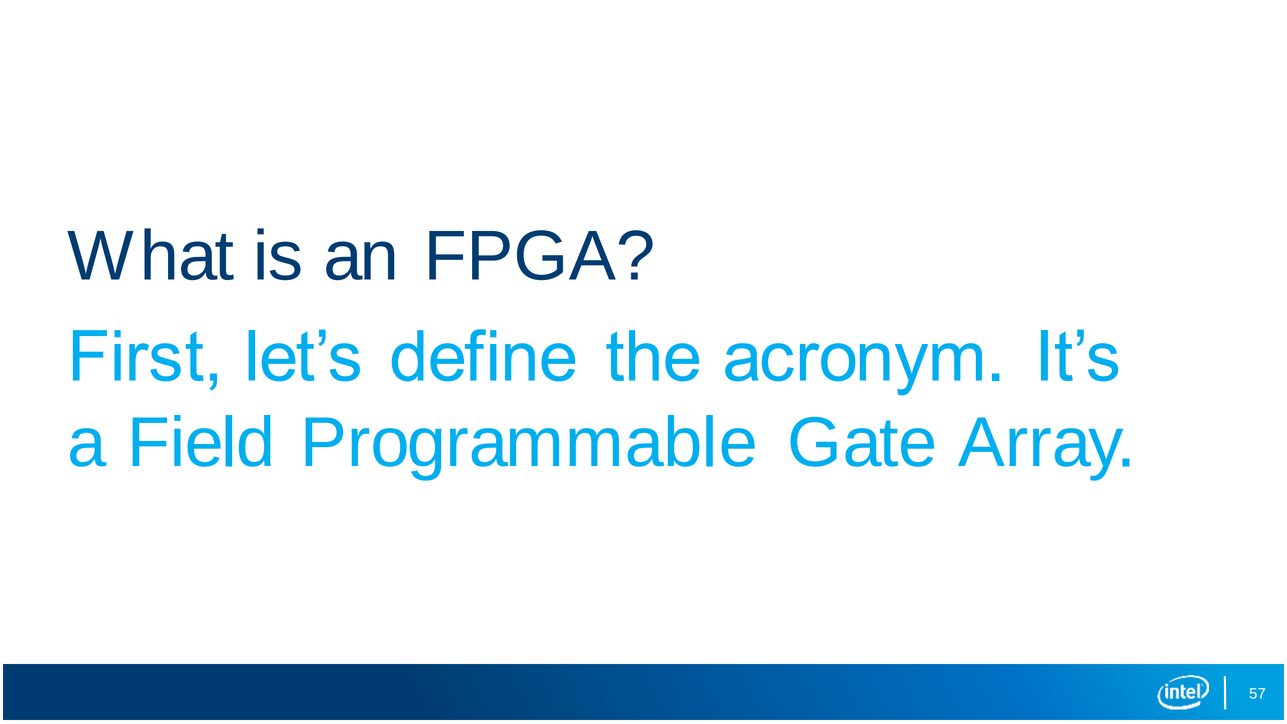
HANDS-ON LAB – COMPLEX NUMBER MULTIPLICATION

- In this lab we provide with the source code that computes multiplication of two complex numbers
- In this example the student will learn how to write basic DPC++ code

Optimization Notice

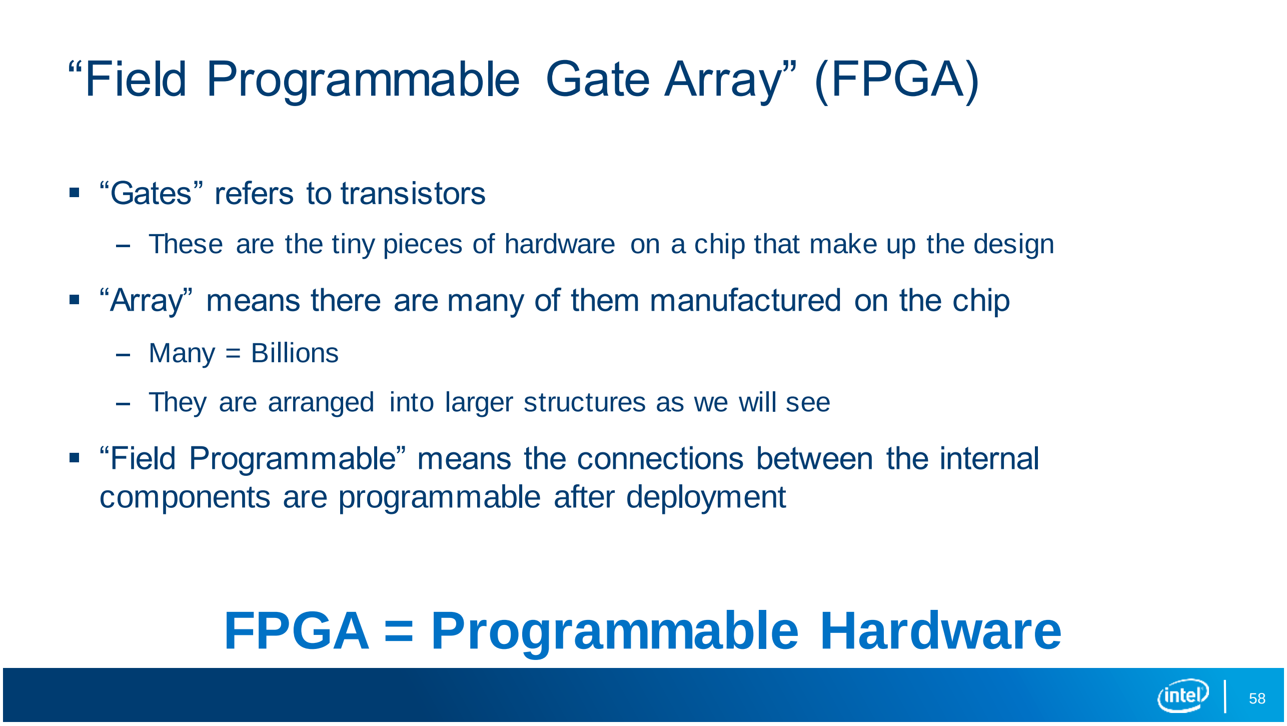
What are FPGAs and Why Should I Care About Programming Them?

A Brief Introduction



What is an FPGA?

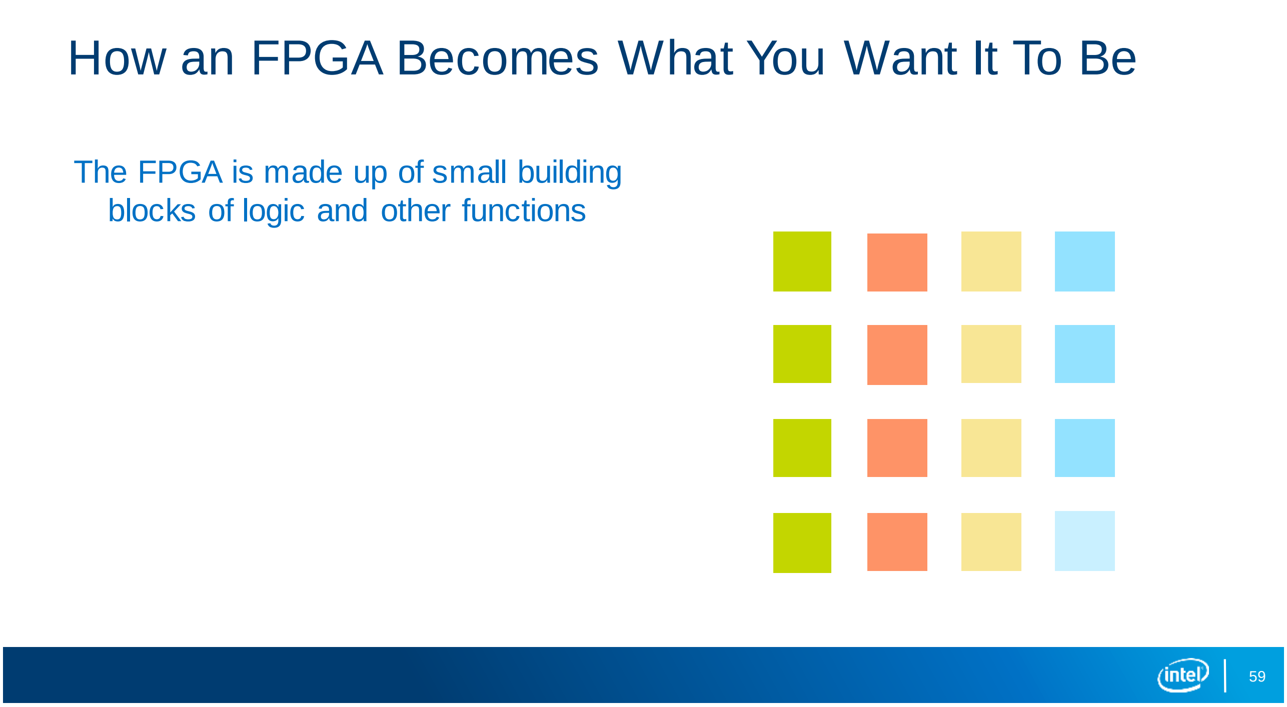
First, let’s define the acronym. It’s a Field Programmable Gate Array.



“Field Programmable Gate Array” (FPGA)

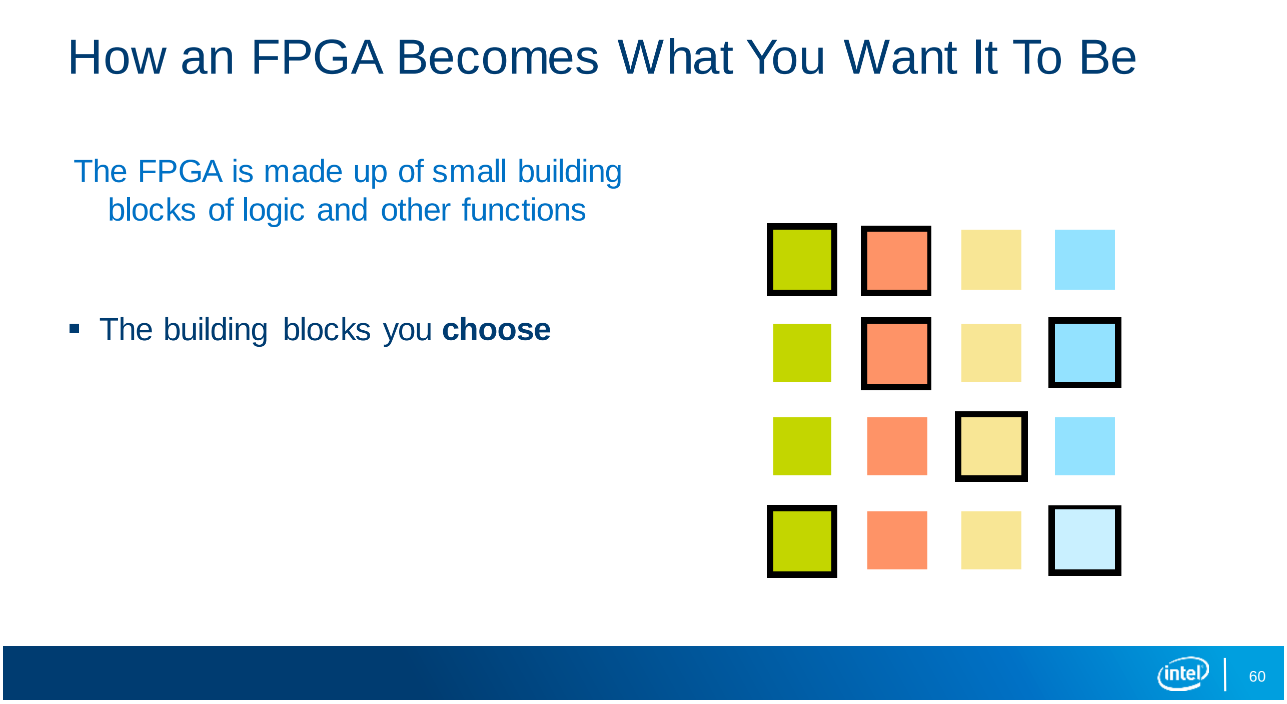
- “Gates” refers to transistors
  - These are the tiny pieces of hardware on a chip that make up the design
- “Array” means there are many of them manufactured on the chip
  - Many = Billions
  - They are arranged into larger structures as we will see
- “Field Programmable” means the connections between the internal components are programmable after deployment

FPGA = Programmable Hardware



How an FPGA Becomes What You Want It To Be

The FPGA is made up of small building blocks of logic and other functions



How an FPGA Becomes What You Want It To Be

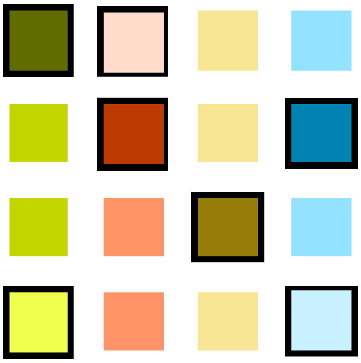
The FPGA is made up of small building blocks of logic and other functions

- The building blocks you **choose**

How an FPGA Becomes What You Want It To Be

The FPGA is made up of small building blocks of logic and other functions

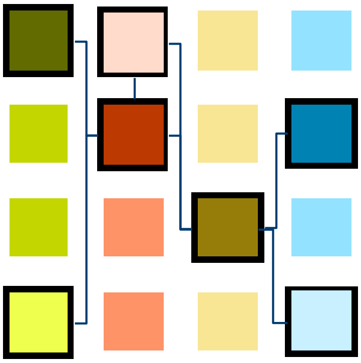
- The building blocks you **choose**
- How you **configure** them



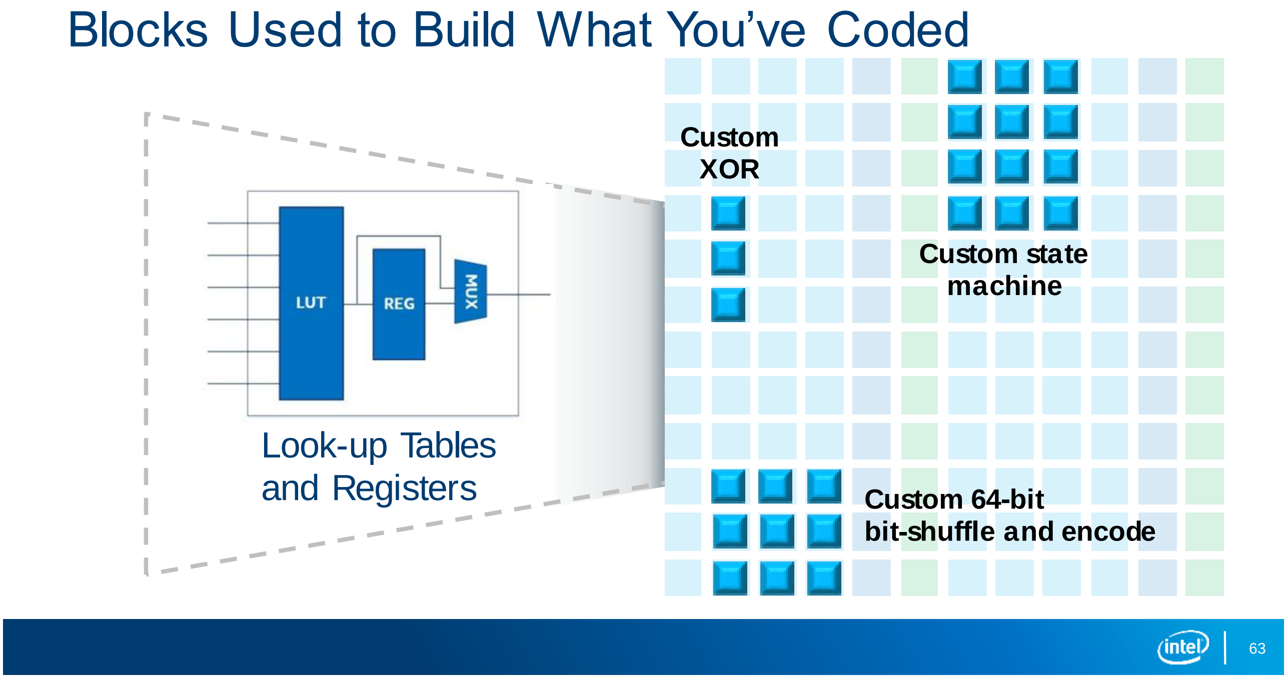
How an FPGA Becomes What You Want It To Be

The FPGA is made up of small building blocks of logic and other functions

- The building blocks you **choose**
- How you **configure** them
- And how you **connect** them



Determine what function the FPGA performs



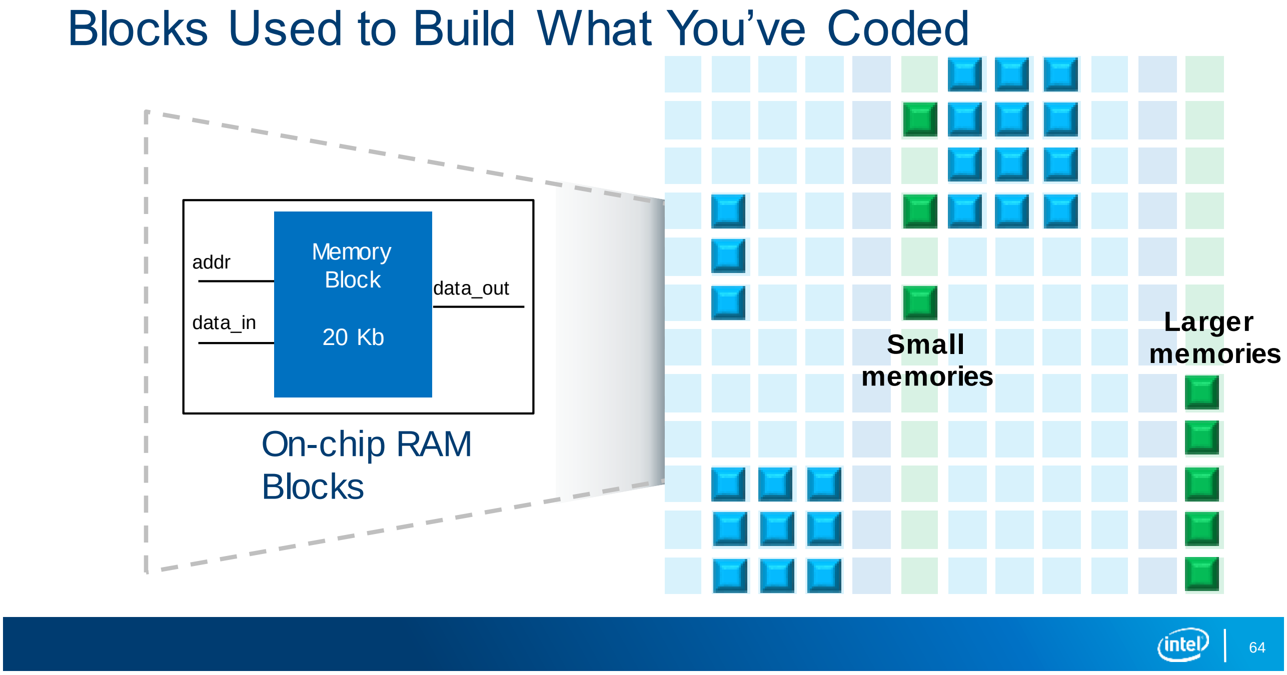
Blocks Used to Build What You’ve Coded

Look-up Tables and Registers

Custom XOR

Custom state machine

Custom 64-bit bit-shuffle and encode



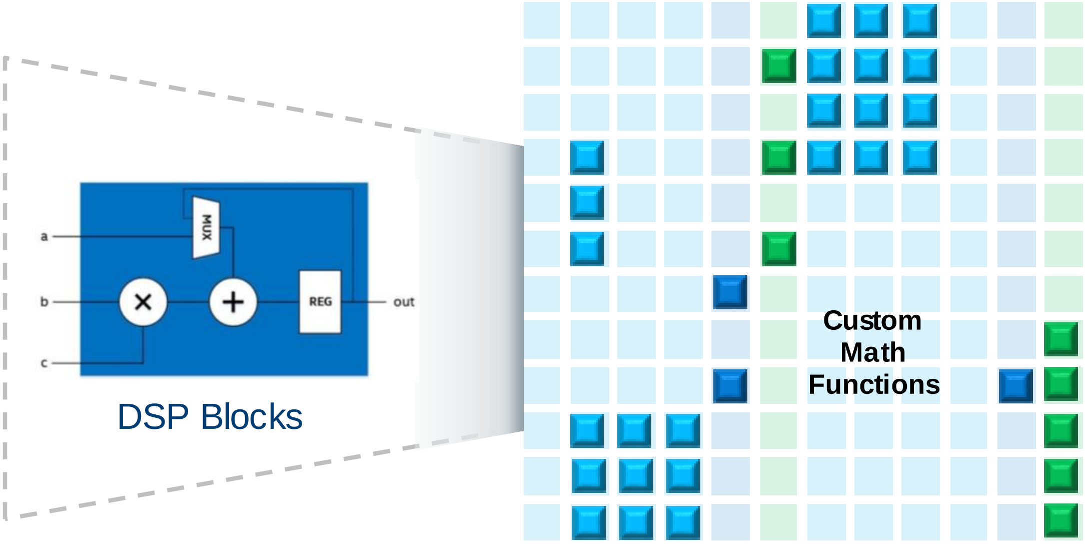
Blocks Used to Build What You’ve Coded

On-chip RAM Blocks

Small memories

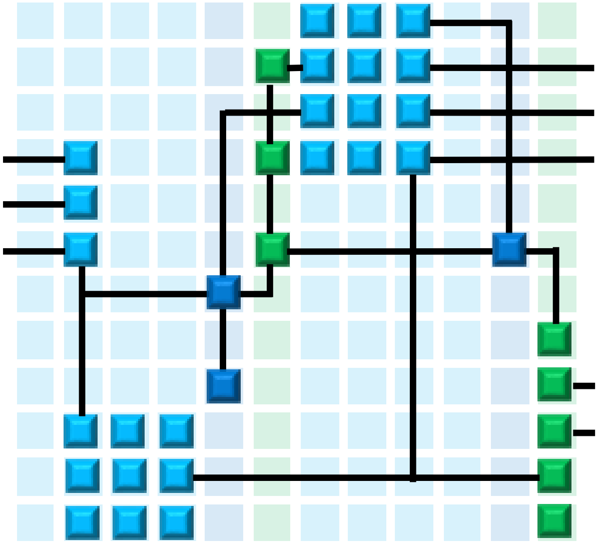
Larger memories

Blocks Used to Build What You’ve Coded

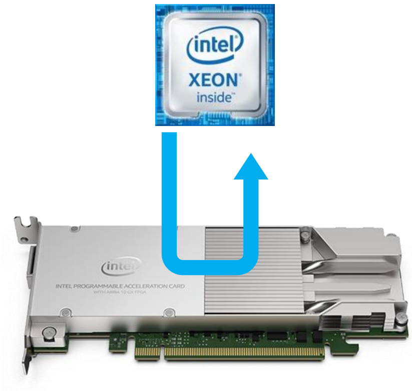


Then, It’s All Connected Together

Blocks are connected with **custom routing** determined by your code



What About Connecting to the Host?

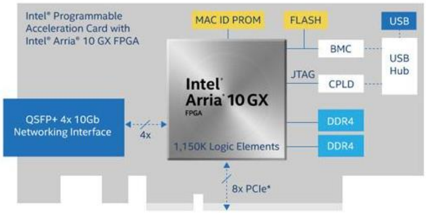


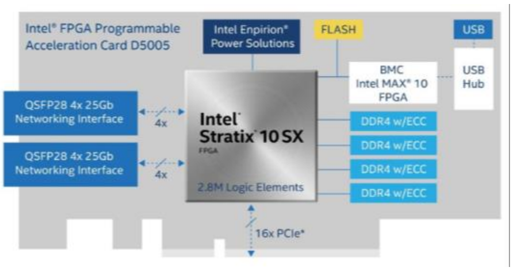
Accelerated functions run on a PCIe attached FPGA card

The host interface is also “baked in” to the FPGA design.

This portion of the design is pre-built and not dependent on your source code.

Intel® FPGAs Available


Intel® Programmable Acceleration Card with Intel® Arria® 10 GX FPGA
MAC ID PROM
FLASH
USB
BMC
USB Hub
JTAG
CPLD
Intel® Arria® 10 GX FPGA
QSFP+ 4x 10Gb Networking Interface
4x
DDR4
DDR4
1,150K Logic Elements
8x PCIe*


Intel® FPGA Programmable Acceleration Card D5005
Intel Enpirion® Power Solutions
FLASH
USB
BMC Intel MAX® 10 FPGA
USB Hub
QSFP28 4x 25Gb Networking Interface
4x
QSFP28 4x 25Gb Networking Interface
4x
Intel® Stratix® 10 SX FPGA
DDR4 w/ECC
DDR4 w/ECC
DDR4 w/ECC
DDR4 w/ECC
2.8M Logic Elements
16x PCIe*

Why should I care about programming for an FPGA?

It all comes down to the advantage of custom hardware.

First, some impressive examples…

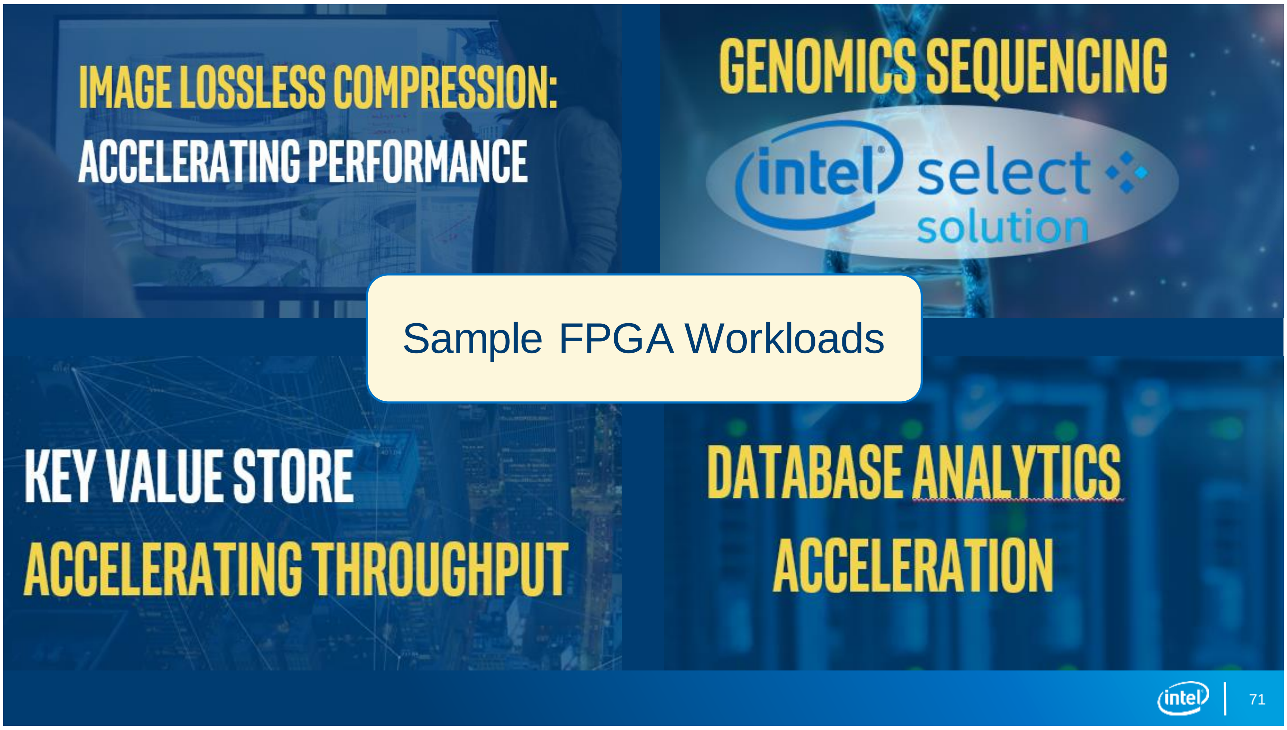


IMAGE LOSSLESS COMPRESSION: ACCELERATING PERFORMANCE

GENOMICS SEQUENCING

intel select solution

Sample FPGA Workloads

KEY VALUE STORE ACCELERATING THROUGHPUT

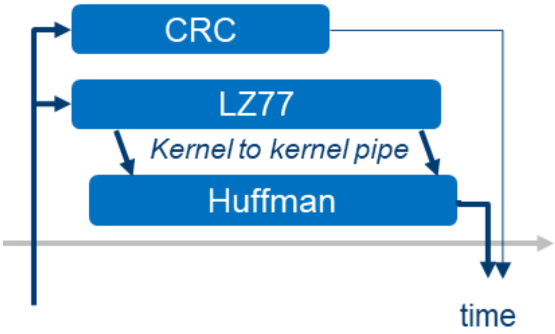
DATABASE ANALYTICS ACCELERATION

Gzip Compression

FPGA Example included with the Intel® oneAPI Base Toolkit

Concurrent kernels for LZ77, Huffman encoding and CRC

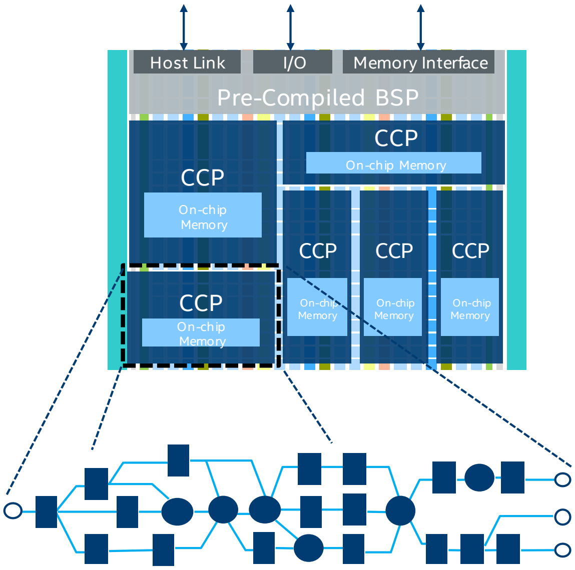
You are encouraged to try it for yourself!



Code to Hardware: An Introduction

Intel® FPGAs

Implementing Optimized **Custom Compute Pipelines (CCPs)** synthesized from compiled code

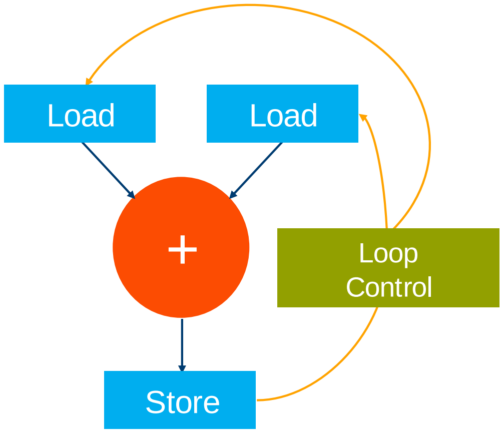


Custom Compute Pipeline

How Is a Pipeline Built?

Hardware is added for

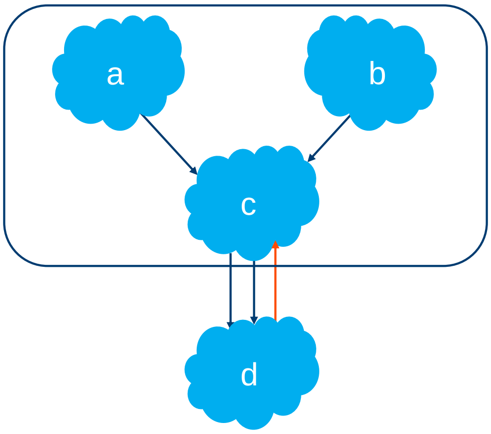
- Computation
- Memory Loads and Stores
- Control and scheduling
  - Loops & Conditionals



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

Connecting the Pipeline Together

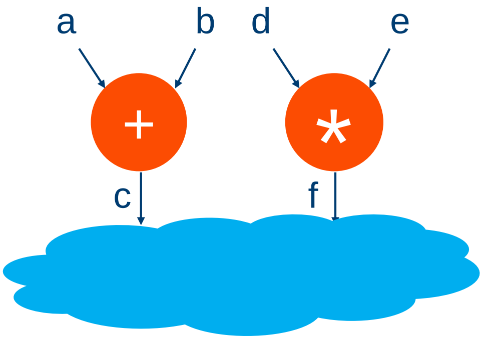
- Handshaking signals for variable latency paths
- Operations with a fixed latency are clustered together
- Fixed latency operations improve
  - Area: no handshaking signals required
  - Performance: no potential stalling due to variable latencies



Simultaneous Independent Operations

- The compiler automatically identifies independent operations
- Simultaneous hardware is built to increase performance
- This achieves data parallelism in a manner similar to a superscalar processor
- Number of independent operations only bounded by the amount of hardware

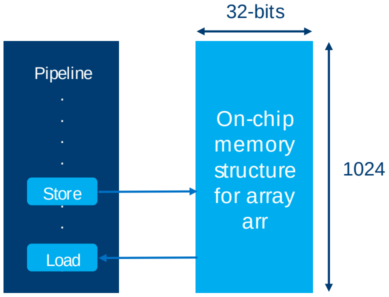
```
c = a + b;
f = d * e;
```



On-Chip Memories Built for Kernel Scope Variables

- Custom on-chip memory structures are built for the variables declared with the kernel scope
- Or, for memory accessors with a target of local
- Load and store units to the on-chip memory will be built within the pipeline

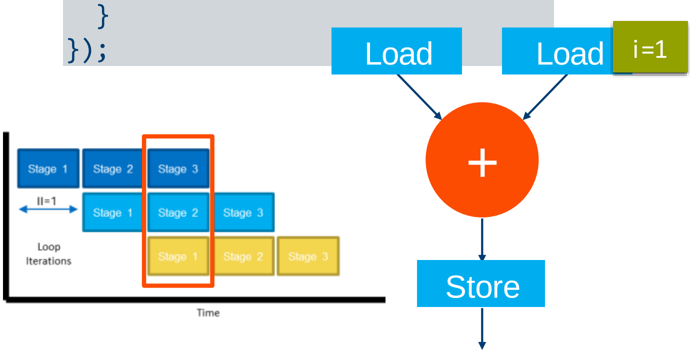
```
//kernel scope
cgh.single_task<>([=]() {
  int arr[1024];
  …
  arr[i] = …; //store to memory
  …
  … = arr[j] //load from memory
  …
} //end kernel scope
```



Pipeline Parallelism for Single Work-Item Kernels

- Single work-item kernels almost always contain an outer loop
- Work executing in multiple stages of the pipeline is called “pipeline parallelism”
- Pipelines from real-world code are normally hundreds of stages long
- **Your job is to keep the data flowing efficiently**

```
handle.single_task<>([=]() {
  … //accessor setup
  for (int i=0; i<LIMIT; i++) {
    c[i] += a[i] + b[i];
  }
});
```

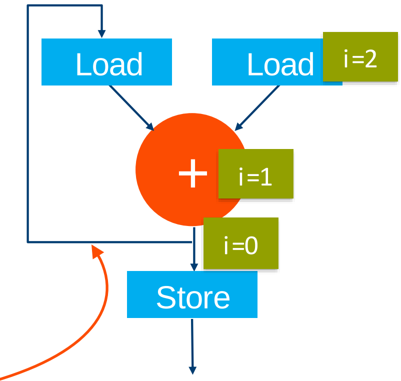


Dependencies Within the Single Work-Item Kernel

Key Concept
Custom built-in dependencies make FPGAs powerful for many algorithms

When a dependency in a single work-item kernel can be resolved by creating a path within the pipeline, the compiler will build that in.

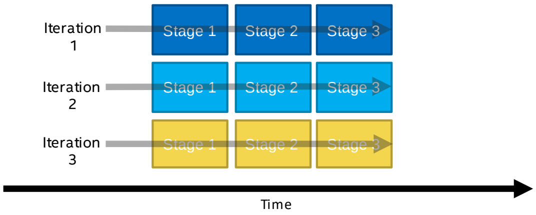
```
handle.single_task<>([=]() {
  int b = 0;
  for (int i=0; i<LIMIT; i++) {
    b += a[i];
  }
});
```



How Do I Use Tasks and Still Get Data Parallelism?

The most common technique is to unroll your loops

```
handle.single_task<>([=]() {
  … //accessor setup
  #pragma unroll
  for (int i=1; i<3; i++) {
    c[i] += a[i] + b[i];
  }
});
```

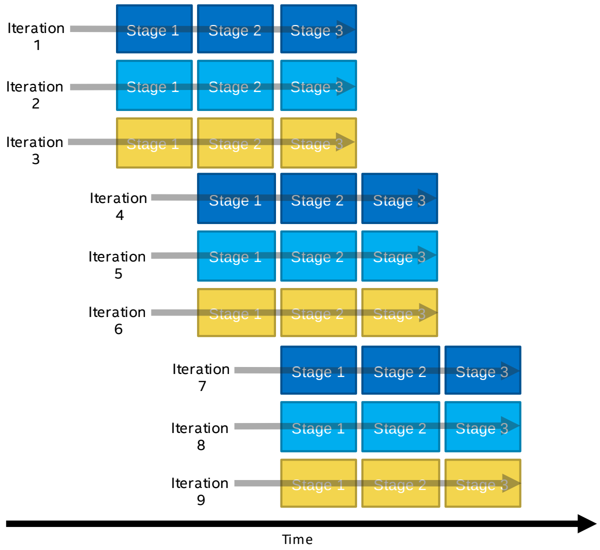


Unrolled Loops Still Get Pipelined

The compiler will still pipeline an unrolled loop, combining the two techniques

- A fully unrolled loop will not be pipelined since all iterations will kick off at once

```
handle.single_task<>([=]() {
  … //accessor setup
  #pragma unroll 3
  for (int i=1; i<9; i++) {
    c[i] += a[i] + b[i];
  }
});
```



What About Task Parallelism?

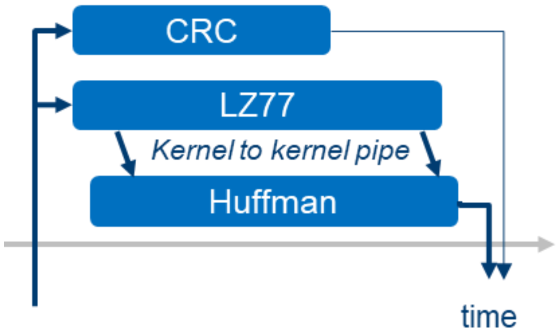
FPGAs can run more than one kernel at a time

- The limit to how many independent kernels can run is the amount of resources available to build the kernels

Data can be passed between kernels using pipes

- Another great FPGA feature explained in the Intel® oneAPI DPC++ FPGA Optimization Guide

Representation of Gzip FPGA example included with the Intel oneAPI Base Toolkit



So, Can We Build These? Parallel Kernels

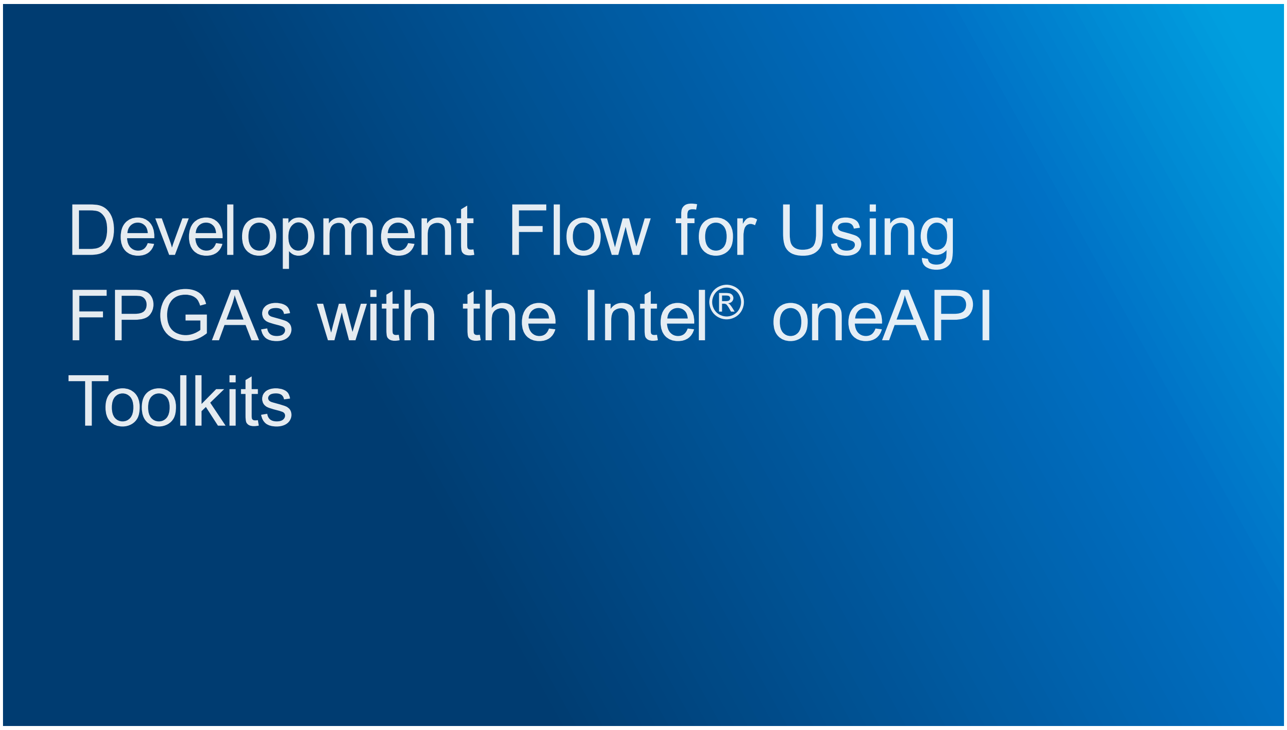
- Kernels launched parallel_for() or parallel_for_work_group()

```
…//application scope

queue.submit([&](handler &cgh) {
  auto A = A_buf.get_access<access::mode::read>(cgh);
  auto B = B_buf.get_access<access::mode::read>(cgh);
  auto C = C_buf.get_access<access::mode::write>(cgh);

  cgh.parallel_for<class VectorAdd>(num_items, [=](id<1> wiID) {
      c[wiID] = a[wiID] + b[wiID];
   });

});

…//application scope
```

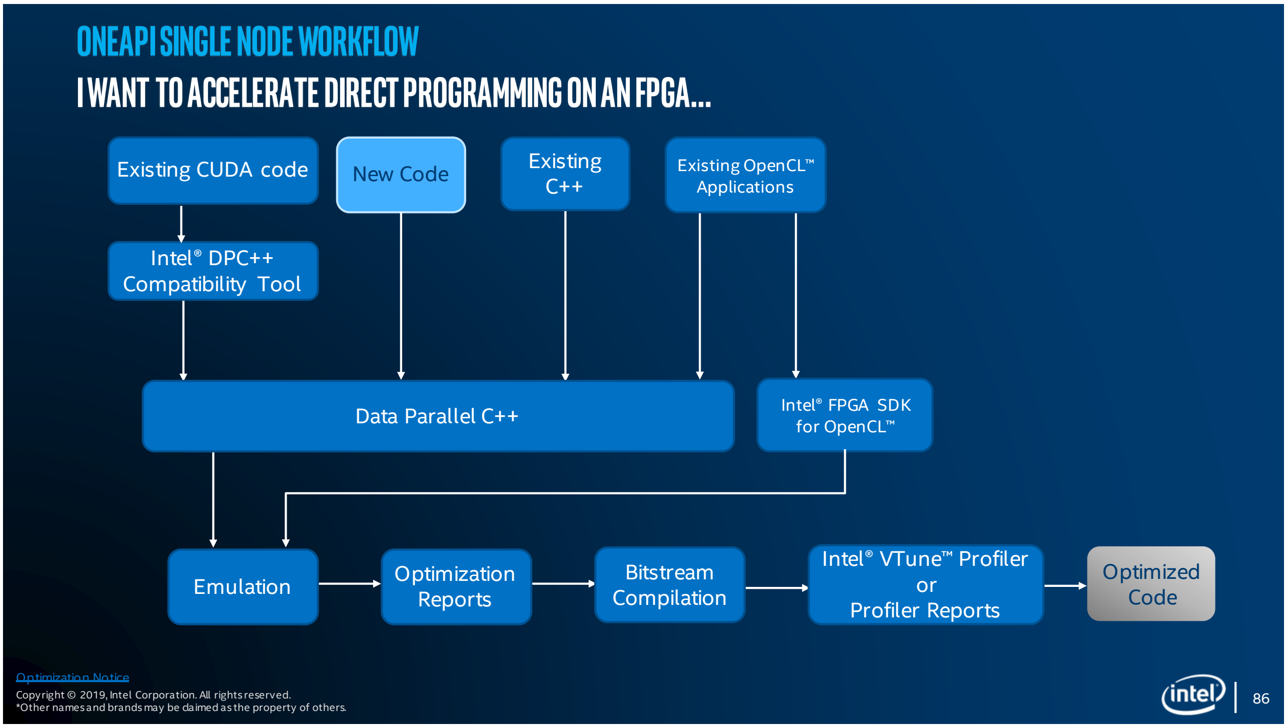
Yes, no problem, and you will learn to code them!

But, **tasks** usually imply more optimal pipeline structures.

The loop optimizations are limited for parallel kernels.



Development Flow for Using FPGAs with the Intel® oneAPI Toolkits



ONEAPI SINGLE NODE WORKFLOW

I WANT TO ACCELERATE DIRECT PROGRAMMING ON AN FPGA…

Existing CUDA code
New Code
Existing C++
Existing OpenCL™ Applications
Intel® DPC++ Compatibility Tool
Data Parallel C++
Intel® FPGA SDK for OpenCL™
Emulation
Optimization Reports
Bitstream Compilation
Intel® VTune™ Profiler or Profiler Reports
Optimized Code

Optimization Notice

Installing oneAPI

- Get started by visiting the Intel® Software Developer Zone page for the Intel® oneAPI Toolkits
  - https://software.intel.com/en-us/oneapi
- Get the Intel® oneAPI Base Toolkit for Linux*
  - Supports compiles for emulation and the optimization report
- Install the Intel FPGA Add-on for oneAPI Base Toolkit
  - Needed for compiles to FPGA hardware
  - Contains Intel® Quartus® Prime software “under the hood,” be sure to comply to required versions of operating system

Or, Skip the Setup and Use the Intel DevCloud!

Sign up here:

- https://software.intel.com/devcloud/
  - Nodes with cards installed in the group fpga_runtime
  - Nodes with extra memory for full FPGA compiles in the group fpga_compile

TRY INTEL® ONEAPI (BETA)

Development sandbox to develop, test and run your workloads across a range of Intel based CPUs, GPUs, and FPGAs using oneAPI software

What You Can Do

Learn Data Parallel C++
Learn the oneAPI(beta) Toolkits
Evaluate workloads
Build heterogeneous applications
Prototype your Project

A Fast and Easy Way to Start Coding

Learn how to use the oneAPI Toolkits, compilers, performance libraries, and tools. No software download. No configuration setups. No installations. Get your account and get started using the DevCloud environment. Learn how to use the tools and test your workload on current and emerging hardware and software to meet your needs.

- A unified & simplified language & libraries for expressing parallelism
- Uncompromised native high-level language performance
- Develop with our beta tools on the latest hardware and mix of architectures
- Based on industry standards and open specifications

About Intel® oneAPI(Beta)

What's Inside

Components of the following Intel oneAPI(Beta) Toolkits are included:

- Intel® oneAPI Toolkit
- Intel® oneAPI HPC Toolkit
- Intel® oneAPI DL Framework Developer Toolkit
- Intel® oneAPI AI Analytics Toolkit
- Intel® oneAPI Video Analytics Toolkit

Hardware

CPU:
- Intel® Xeon® Scalable 6128 Processors
- Intel® Xeon® Scalable 8256 Processors

GPU:
- Intel® Xeon® E-2176 P630 Processors (with Intel® Graphics Technology)

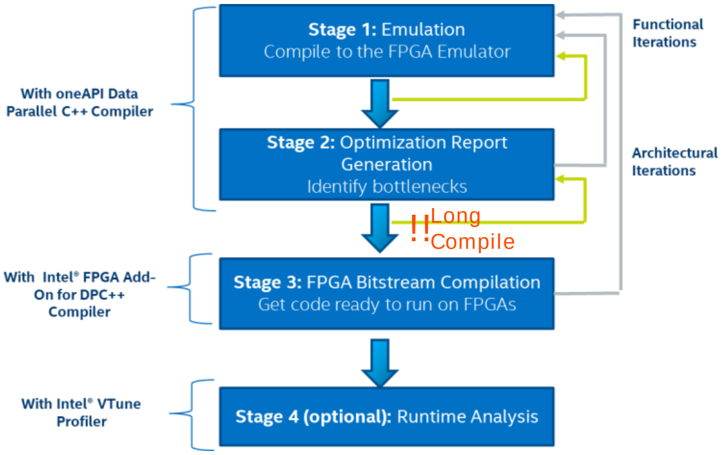
FPGA:
- Intel® Arria® 10 FPGAs

Featuring Tools, Frameworks & Libraries such as:

- oneAPI Advisor (Beta)
- oneAPI Data Analytics Library (Beta)
- oneAPI Data Parallel C++ Compiler (Beta)
- oneAPI Data Parallel C++ Library (Beta)
- oneAPI Deep Neural Network Library (Beta)
- oneAPI Math Kernel Library (Beta)
- oneAPI Threading Building Blocks (Beta)
- oneAPI Video Processing Library (Beta)
- oneAPI VTune™ Profiler (Beta)
- OpenMP* (C++)
- OpenMP* (Fortran)
- TensorFlow*
- GDB
- Intel® Cluster Checker
- Intel® Distribution of OpenVINO Toolkit
- Intel® Distribution of Python
- Intel® Inspector
- Intel® Integrated Performance Primitives
- Intel® MPI Library
- Intel® Optimization for PyTorch*
- Intel® Quartus Prime Software

FPGA Development Flow with oneAPI

- 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



With oneAPI Data Parallel C++ Compiler

Stage 1: Emulation
Compile to the FPGA Emulator

Stage 2: Optimization Report Generation
Identify bottlenecks

!!Long Compile

With Intel® FPGA Add-On for DPC++ Compiler

Stage 3: FPGA Bitstream Compilation
Get code ready to run on FPGAs

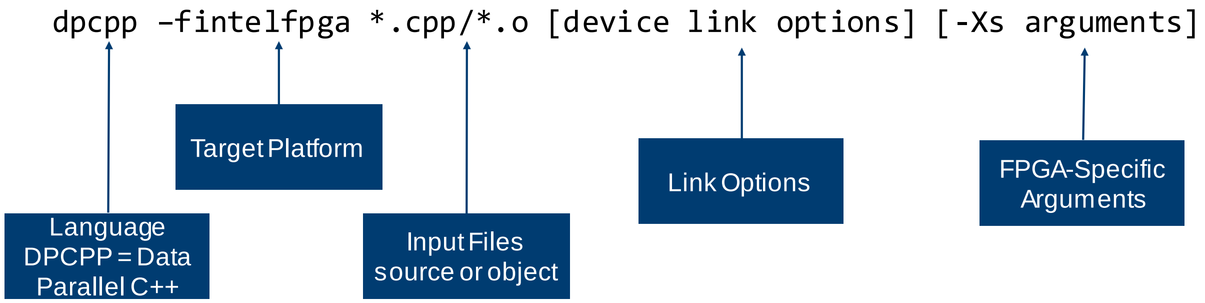
With Intel® VTune Profiler

Stage 4 (optional): Runtime Analysis

Functional Iterations

Architectural Iterations

Anatomy of a Compiler Command Targeting FPGAs



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

Language
DPCPP = Data Parallel C++

Target Platform

Input Files
source or object

Link Options

FPGA-Specific Arguments

Emulation

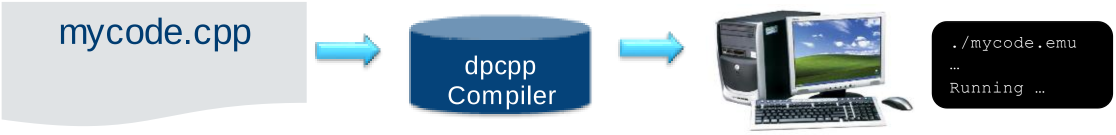
Get it Functional

Does my code give me the correct answers?

Emulation

- Quickly generate x86 executables that represent the kernel
- Debug support for
  - Standard DPC++ syntax, channels, print statements

```
dpcpp -fintelfpga <source_file>.cpp –DFPGA_EMULATOR
```



```
./mycode.emu
…
Running …
```

Explicit Selection of Emulation Device

```
dpcpp -fintelfpga <source_file>.cpp –DFPGA_EMULATOR
```

- Declare the device_selector as type cl::sycl::intel::fpga_emulator
- Include fpga_extensions.hpp
- Include –DFPGA_EMULATOR in your compilation command

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

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

queue deviceQueue(device_selector);
...
```

Using the Static Optimization Report

Get it Optimized

Where are the bottlenecks?

Compiling to Produce an Optimization Report

Two Step Method:

```
dpcpp -fintelfpga <source_file>.cpp  -c -o <file_name>.o
dpcpp -fintelfpga <file_name>.o  -fsycl-link  -Xshardware
```

One Step Method:

```
dpcpp -fintelfpga <source_file>.cpp  -fsycl-link  -Xshardware
```

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/

- We will discuss more about the report later

FPGA Bitstream Compilation

Check Runtime Behavior

Check what you can't check during static analysis

Compile to FPGA Executable with Profiler Enabled

Two Step Method:

```
dpcpp -fintelfpga <source_file>.cpp -c -o <file_name>.o
dpcpp -fintelfpga <file_name>.o –Xshardware -Xsprofile
```

One Step Method:

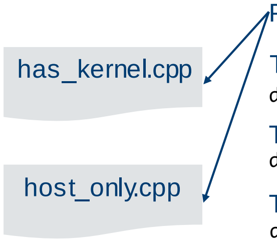
```
dpcpp -fintelfpga <source_file>.cpp –Xshardware -Xsprofile
```

The profiler will be instrumented within the image and you will be able to run the executable to return information to import into Intel® Vtune Amplifier.

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

Compiling FPGA Device Separately and Linking

- In the default case, the DPC++ Compiler handles generating the host executable, device image, and final executable
- It is sometimes desirable to compile the host and device separately so changes in the host code do not trigger a long compile



Partition code

This is the long compile

Then run this command to compile the FPGA image:

```
dpcpp -fintelfpga has_kernel.cpp –fsycl-link=image –o has_kernel.o –Xshardware
```

This command to produce an object file out of the host only code:

```
dpcpp -fintelfpga has_kernel.cpp –c –o host_only.o
```

This command to put the object files together into an executable:

```
dpcpp -fintelfpga has_kernel.o host_only.o –o a.out –Xshardware
```

References and Resources

- Website hub for using FPGAs with oneAPI
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fpga.html
- Intel® oneAPI Programming Guide
  - https://software.intel.com/content/www/us/en/develop/download/intel-oneapi-programming-guide.html
- Intel® oneAPI DPC++ FPGA Optimization Guide
  - https://software.intel.com/content/www/us/en/develop/download/oneapi-fpga-optimization-guide.html
- FPGA Tutorials GitHub
  - https://github.com/intel/BaseKit-code-samples/tree/master/FPGATutorials

Lab: Practice the FPGA Development Flow

(Followed by hour-long conference break)

Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Agenda

- Reports
- Loop Optimization
- Memory Optimization
- Other Optimization Techniques
- Lab: Optimizing the Hough Transform Kernel

Reports

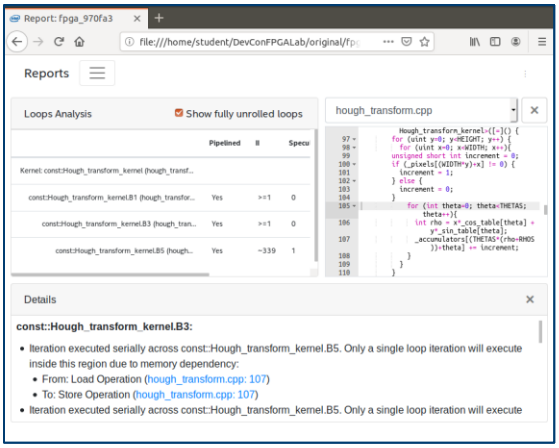
HTML Report

Static report showing optimization, area, and architectural information

- Automatically generated with the object file
  - Located in **`<file_name>.prj\reports\report.html`**
- Dynamic reference information to original source code

Optimization Report – Throughput Analysis

- Loops Analysis and Fmax II sections
- Actionable feedback on pipeline status of loops
- Show estimated Fmax of each loop


Report: fpga_970fa3
file:///home/student/DevConFPGALab/original/fpg
Reports
Loops Analysis
☑ Show fully unrolled loops
hough_transform.cpp
Details

	Pipelined	II	Specu
Kernel: const::Hough_transform_kernel (hough_transf...			
const::Hough_transform_kernel.B1 (hough_transfor...	Yes	>=1	0
const::Hough_transform_kernel.B3 (hough_tran...	Yes	>=1	0
const::Hough_transform_kernel.B5 (hough...	Yes	~339	1

```
    Hough_transform_kernel>([=]() {
  for (uint y=0; y<HEIGHT; y++) {
    for (uint x=0; x<WIDTH; x++){
  unsigned short int increment = 0;
  if (_pixels[(WIDTH*y)+x] != 0) {
    increment = 1;
  } else {
    increment = 0;
  }
      for (int theta=0; theta<THETAS;
        theta++){
      int rho = x*_cos_table[theta] +
        y*_sin_table[theta];
      _accumulators[(THETAS*(rho+RHOS
        ))+theta] += increment;
      }
    }
  }
```

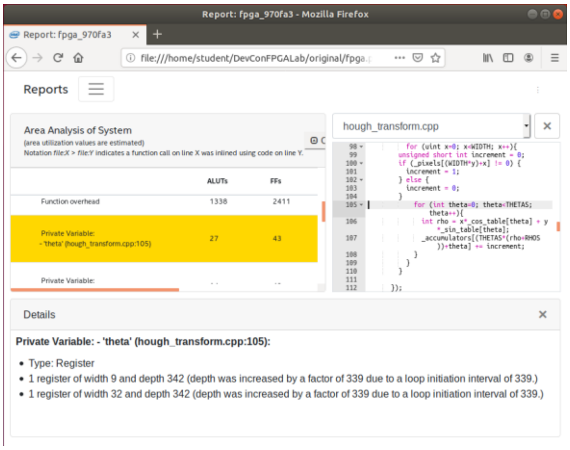
const::Hough_transform_kernel.B3:

- Iteration executed serially across const::Hough_transform_kernel.B5. Only a single loop iteration will execute inside this region due to memory dependency:
  - From: Load Operation (hough_transform.cpp: 107)
  - To: Store Operation (hough_transform.cpp: 107)
- Iteration executed serially across const::Hough_transform_kernel.B5. Only a single loop iteration will execute

Optimization Report – Area Analysis

Generate detailed estimated area utilization report of kernel scope code

- Detailed breakdown of resources by system blocks
- Provides architectural details of HW
  - Suggestions to resolve inefficiencies


Report: fpga_970fa3 - Mozilla Firefox
Report: fpga_970fa3
file:///home/student/DevConFPGALab/original/fpga.
Reports
hough_transform.cpp
Details

Area Analysis of System
(area utilization values are estimated)
Notation *file:X* > *file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs
Function overhead	1338	2411
Private Variable: - 'theta' (hough_transform.cpp:105)	27	43
Private Variable:		

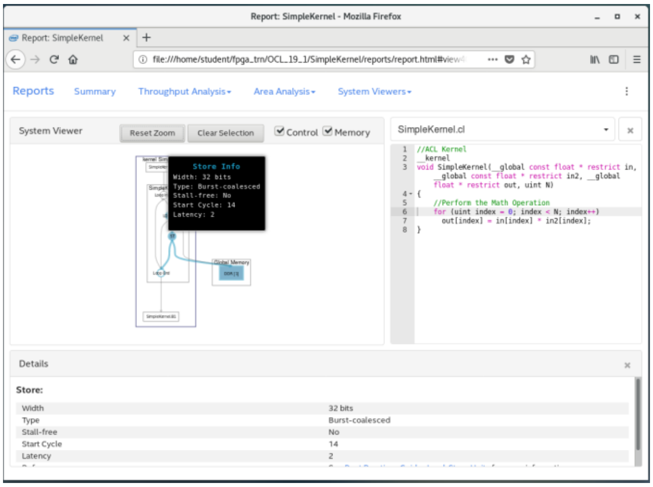
```
    for (uint x=0; x<WIDTH; x++){
  unsigned short int increment = 0;
  if (_pixels[(WIDTH*y)+x] != 0) {
    increment = 1;
  } else {
    increment = 0;
  }
      for (int theta=0; theta<THETAS;
        theta++){
      int rho = x*_cos_table[theta] + y
        *_sin_table[theta];
      _accumulators[(THETAS*(rho+RHOS
        ))+theta] += increment;
      }
    }
  }

  });
```

Private Variable: - 'theta' (hough_transform.cpp:105):

- Type: Register
- 1 register of width 9 and depth 342 (depth was increased by a factor of 339 due to a loop initiation interval of 339.)
- 1 register of width 32 and depth 342 (depth was increased by a factor of 339 due to a loop initiation interval of 339.)

Optimization Report – Graph Viewer

- The system view of the Graph Viewer shows following types of connections
  - Control
  - Memory, if your design has global or local memory
  - Pipes, if your design uses pipes



```
Store Info
Width: 32 bits
Type: Burst-coalesced
Stall-free: No
Start Cycle: 14
Latency: 2
```

```
//ACL Kernel
__kernel
void SimpleKernel(__global const float * restrict in,
    __global const float * restrict in2, __global
    float * restrict out, uint N)
{
    //Perform the Math Operation
    for (uint index = 0; index < N; index++)
      out[index] = in[index] * in2[index];
}
```

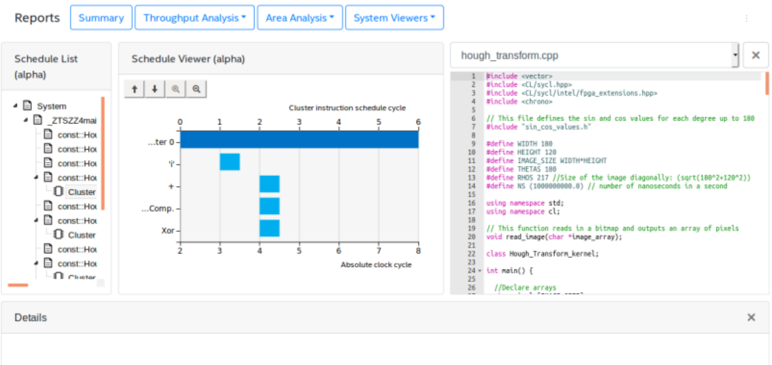
Details

Store:

Width	32 bits
Type	Burst-coalesced
Stall-free	No
Start Cycle	14
Latency	2

Optimization Report – Schedule Viewer

- Schedule in clock cycles for different blocks in your code



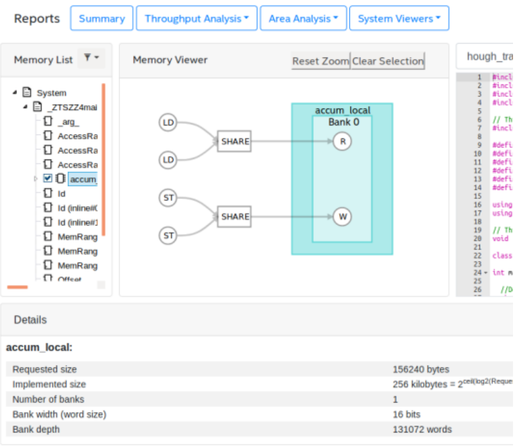
```
#include <vector>
#include <CL/sycl.hpp>
#include <CL/sycl/intel/fpga_extensions.hpp>
#include <chrono>

// This file defines the sin and cos values for each degree up to 180
#include "sin_cos_values.h"

#define WIDTH 180
#define HEIGHT 120
#define IMAGE_SIZE WIDTH*HEIGHT
#define THETAS 180
#define RHOS 217 //Size of the image diagonally: (sqrt(180^2+120^2))
#define NS (1000000000.0) // number of nanoseconds in a second

using namespace std;
using namespace cl;

// This function reads in a bitmap and outputs an array of pixels
void read_image(char *image_array);

class Hough_Transform_kernel;

int main() {

  //Declare arrays
```

Details

HTML Kernel Memory Viewer

Helps you identify data movement bottlenecks in your kernel design. Illustrates:

- Memory replication
- Banking
- Implemented arbitration
- Read/write capabilities of each memory port

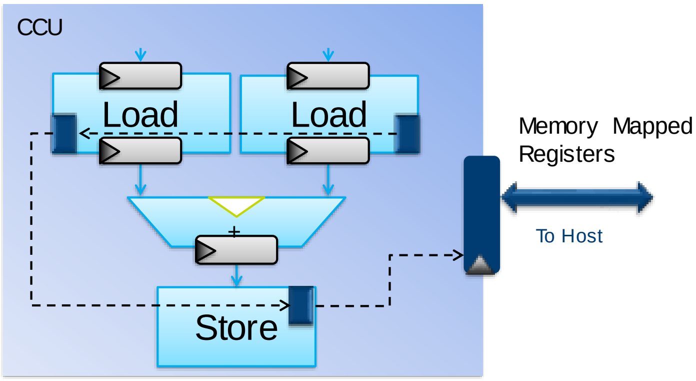


accum_local:

Requested size	156240 bytes
Implemented size	256 kilobytes = $2^{ceil(log2(Reque}$
Number of banks	1
Bank width (word size)	16 bits
Bank depth	131072 words

Profiler

- Inserts counters and profiling logic into the HW design
- Dynamically reports the performance of kernels
- Enable using the –Xsprofile option with dpcpp



Collecting Profiling Data

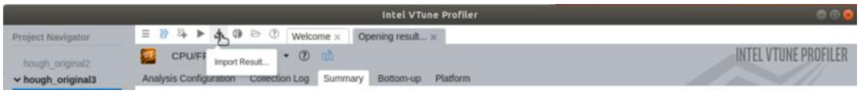
- Run the executable that integrates the kernel with the profiler using

```
aocl profile -s <path/to/source>.source /path/to/host-executable
```

- A profile.json file will be produced
- Import the profile.json file into the Intel® Vtune™ Profiler

Importing Profile Data into Intel® Vtune™ Profiler

- Place the collect profile.json file in a folder by itself
- Open the Intel Vtune Profiler using the command `vtune-gui`
- Press the Import button at the top of the GUI



- Select Import raw trace data
- Navigate to the folder in the file browser (do not click into folder), and Open
- Click the Blue Import button in the GUI

Loop Optimization

Types of Kernels (Review)

- There are two types of kernels in Data Parallel C++
  - Single work-item
  - Parallel
- For FPGAs, the compiler will automatically detect the kind of kernel input
- Loop analysis will only be done for single work-item kernels
- Most loop optimizations will only apply to single work-item kernels
- Most optimized FPGA kernels are single work-item kernels

Single Work-Item Kernels

- Single work items kernels are kernels that contain no reference to the work item ID
- Usually launched with the group handler member function single_task()
  - Or, launched with other functions without a reference to the work item ID (implying a work group size of 1)
- Almost always contain an outer loop.

```
…//application scope

queue.submit([&](handler &cgh) {
  auto A = A_buf.get_access<access::mode::read>(cgh);
  auto B = B_buf.get_access<access::mode::read>(cgh);
  auto C = C_buf.get_access<access::mode::write>(cgh);

  cgh.single_task<class swi_add>([=]() {
    for (unsigned i = 0; i < 128; i++) {
      c[i] = a[i] + b[i];
    }
   });

});

…//application scope
```

As a Reminder – Parallel Kernels

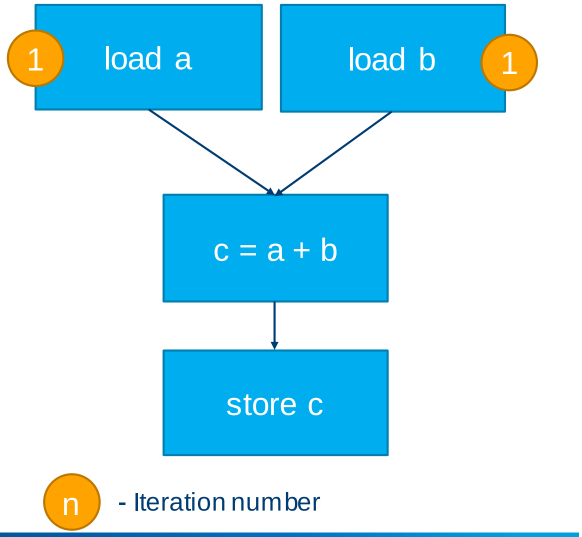
- Kernels launched with the command group handler member function parallel_for() or parallel_for_work_group() with a NDRange/work-group size of >1.
- Much of this section will not apply to parallel kernels

```
…//application scope

queue.submit([&](handler &cgh) {
  auto A = A_buf.get_access<access::mode::read>(cgh);
  auto B = B_buf.get_access<access::mode::read>(cgh);
  auto C = C_buf.get_access<access::mode::write>(cgh);

  cgh.parallel_for<class VectorAdd>(num_items, [=](id<1> wiID) {
      c[wiID] = a[wiID] + b[wiID];
   });

});

…//application scope
```

Understanding Initiation Interval

- dpcpp will infer pipelined parallel execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

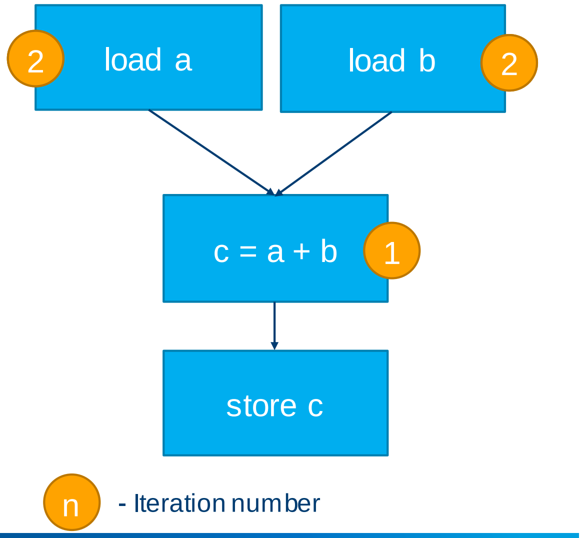
```
…
cgh.single_task<class swi_add>([=]() {
    for (unsigned i = 0; i < 128; i++) {
      c[i] = a[i] + b[i];
    }
   });
…
```



Understanding Initiation Interval

- dpcpp will infer pipelined parallel execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

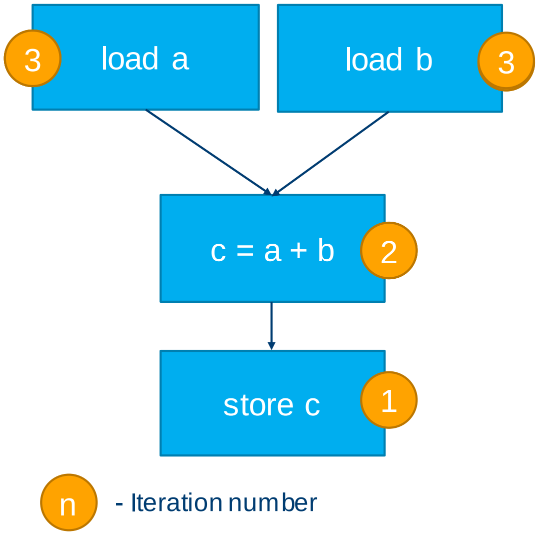
```
…
cgh.single_task<class swi_add>([=]() {
    for (unsigned i = 0; i < 128; i++) {
      c[i] = a[i] + b[i];
    }
   });
…
```



Understanding Initiation Interval

- dpcpp will infer pipelined parallel execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

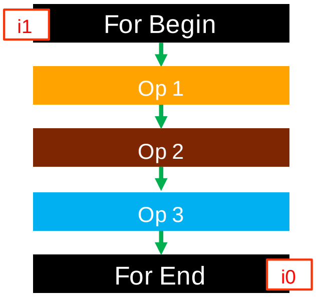
```
…
cgh.single_task<class swi_add>([=]() {
    for (unsigned i = 0; i < 128; i++) {
      c[i] = a[i] + b[i];
    }
   });
…
```



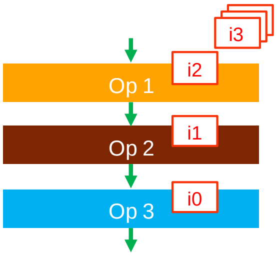
Loop Pipelining vs Serial Execution

Serial execution is the worst case. One iteration needs to complete **fully** before a new piece of data enters the pipeline.

Worst Case

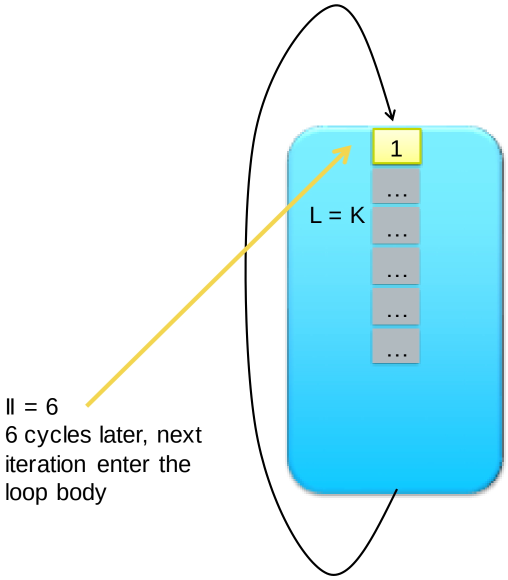


Best Case



In-Between Scenario

- Sometimes you must wait more than one clock cycle to input more data
- Because dependencies can’t resolve fast enough
- How long you have to wait is called **Initiation Interval** or **II**
- Total number of cycles to run kernel is about (loop iterations)*II
  - (neglects initial latency)
- Minimizing II is **key** to performance



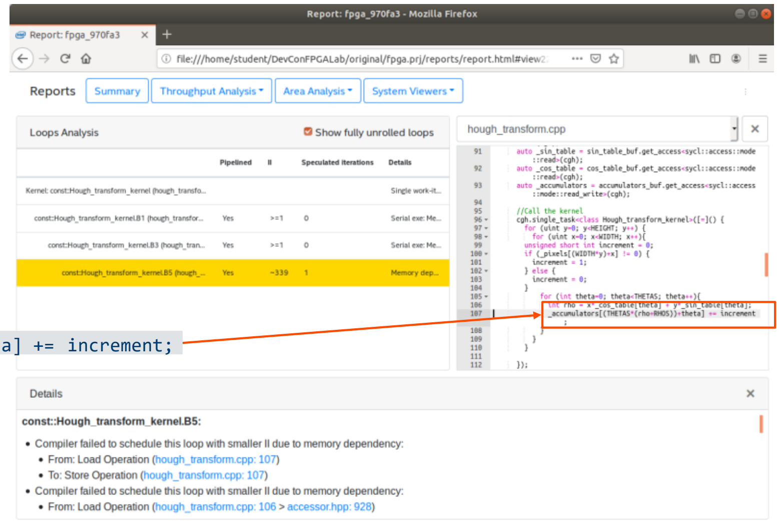
II = 6
6 cycles later, next iteration enter the loop body

Why Could This Happen?

- Memory Dependency
  - Kernel cannot retrieve data fast enough from memory

```
_accumulators[(THETAS*(rho+RHOS))+theta] += increment;
```

Value must be retrieved from global memory and incremented


Report: fpga_970fa3 - Mozilla Firefox
Report: fpga_970fa3
file:///home/student/DevConFPGALab/original/fpga.prj/reports/report.html#view2
Reports
Summary
Throughput Analysis
Area Analysis
System Viewers
Loops Analysis
☑ Show fully unrolled loops
hough_transform.cpp

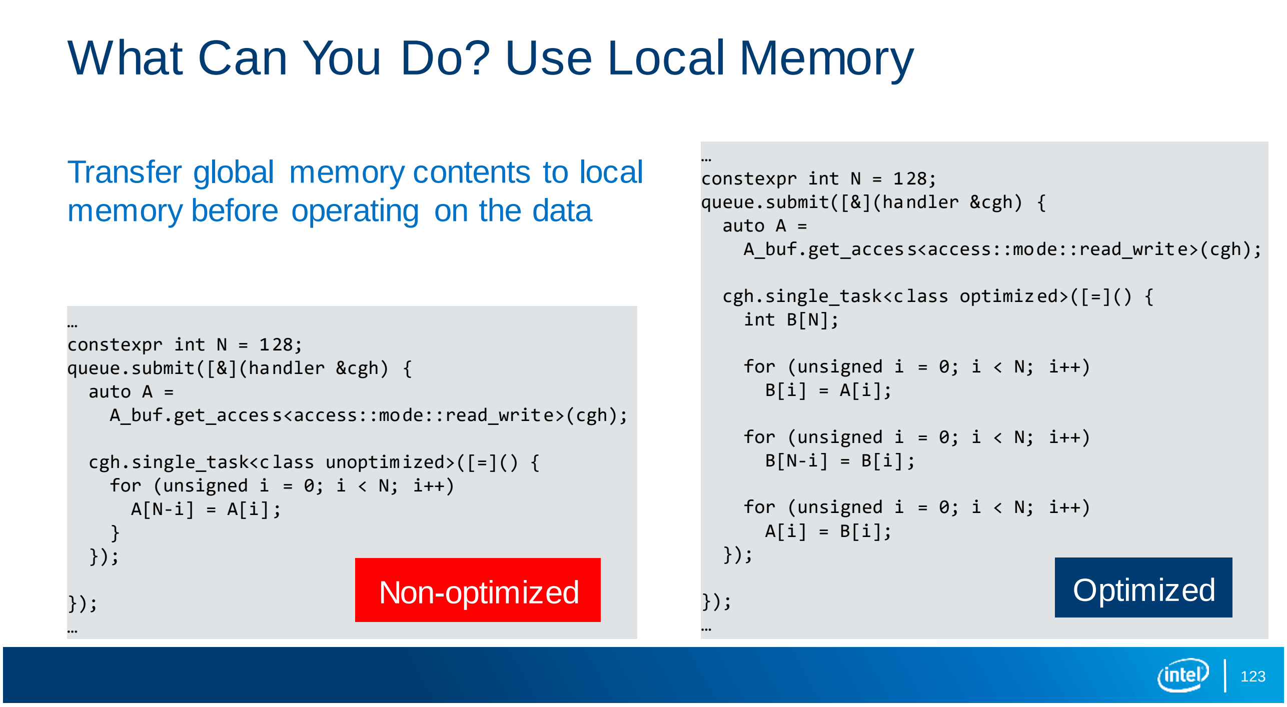
	Pipelined	II	Speculated iterations	Details
Kernel: const::Hough_transform_kernel (hough_transfo...				Single work-it...
const::Hough_transform_kernel.B1 (hough_transfor...	Yes	>=1	0	Serial exe: Me...
const::Hough_transform_kernel.B3 (hough_tran...	Yes	>=1	0	Serial exe: Me...
const::Hough_transform_kernel.B5 (hough_...	Yes	~339	1	Memory dep...

```
91      auto _sin_table = sin_table_buf.get_access<sycl::access::mode
            ::read>(cgh);
92      auto _cos_table = cos_table_buf.get_access<sycl::access::mode
            ::read>(cgh);
93      auto _accumulators = accumulators_buf.get_access<sycl::access
            ::mode::read_write>(cgh);
94
95      //Call the kernel
96      cgh.single_task<class Hough_transform_kernel>([=]() {
97        for (uint y=0; y<HEIGHT; y++) {
98          for (uint x=0; x<WIDTH; x++){
99        unsigned short int increment = 0;
100       if (_pixels[(WIDTH*y)+x] != 0) {
101         increment = 1;
102       } else {
103         increment = 0;
104       }
105           for (int theta=0; theta<THETAS; theta++){
106             int rho = x*_cos_table[theta] + y*_sin_table[theta];
107             _accumulators[(THETAS*(rho+RHOS))+theta] += increment
                  ;
108           }
109         }
110       }
111
112     });
```

Details

const::Hough_transform_kernel.B5:

- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation (hough_transform.cpp: 107)
  - To: Store Operation (hough_transform.cpp: 107)
- Compiler failed to schedule this loop with smaller II due to memory dependency:
  - From: Load Operation (hough_transform.cpp: 106 > accessor.hpp: 928)



What Can You Do? Use Local Memory

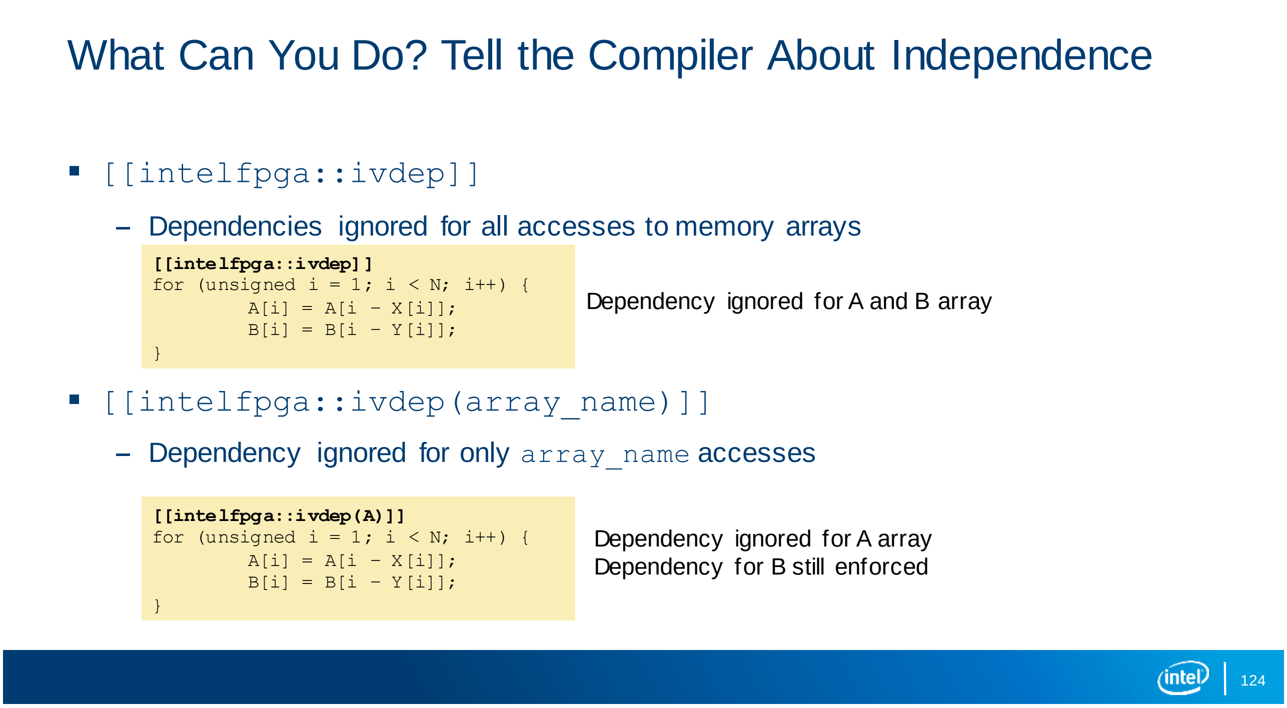
Transfer global memory contents to local memory before operating on the data

```
…
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  auto A =
    A_buf.get_access<access::mode::read_write>(cgh);

  cgh.single_task<class unoptimized>([=]() {
    for (unsigned i = 0; i < N; i++)
      A[N-i] = A[i];
    }
  });

});
…
```

Non-optimized

```
…
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  auto A =
    A_buf.get_access<access::mode::read_write>(cgh);

  cgh.single_task<class optimized>([=]() {
    int B[N];

    for (unsigned i = 0; i < N; i++)
      B[i] = A[i];

    for (unsigned i = 0; i < N; i++)
      B[N-i] = B[i];

    for (unsigned i = 0; i < N; i++)
      A[i] = B[i];
  });

});
…
```

Optimized



What Can You Do? Tell the Compiler About Independence

- `[[intelfpga::ivdep]]`
  - Dependencies ignored for all accesses to memory arrays

```
[[intelfpga::ivdep]]
for (unsigned i = 1; i < N; i++) {
        A[i] = A[i - X[i]];
        B[i] = B[i - Y[i]];
}
```

Dependency ignored for A and B array

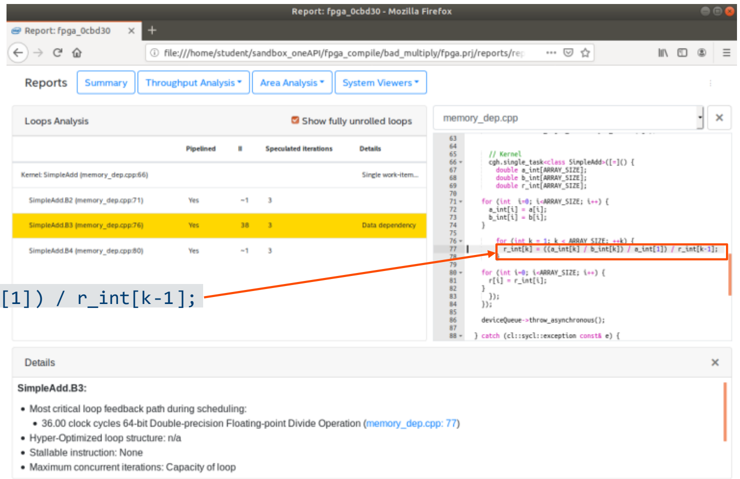
- `[[intelfpga::ivdep(array_name)]]`
  - Dependency ignored for only `array_name` accesses

```
[[intelfpga::ivdep(A)]]
for (unsigned i = 1; i < N; i++) {
        A[i] = A[i - X[i]];
        B[i] = B[i - Y[i]];
}
```

Dependency ignored for A array
Dependency for B still enforced

Why Else Could This Happen?

- Data Dependency
  - Kernel cannot complete a calculation fast enough



Report: fpga_0cbd30 - Mozilla Firefox

Report: fpga_0cbd30

file:///home/student/sandbox_oneAPI/fpga_compile/bad_multiply/fpga.prj/reports/rep

Loops Analysis ☑ Show fully unrolled loops

	Pipelined	II	Speculated iterations	Details
Kernel: SimpleAdd (memory_dep.cpp:66)				Single work-item...
SimpleAdd.B2 (memory_dep.cpp:71)	Yes	~1	3	
SimpleAdd.B3 (memory_dep.cpp:76)	Yes	38	3	Data dependency
SimpleAdd.B4 (memory_dep.cpp:80)	Yes	~1	3	

memory_dep.cpp

```
63
64
65      // Kernel
66      cgh.single_task<class SimpleAdd>([=]() {
67        double a_int[ARRAY_SIZE];
68        double b_int[ARRAY_SIZE];
69        double r_int[ARRAY_SIZE];
70
71    for (int  i=0; i<ARRAY_SIZE; i++) {
72      a_int[i] = a[i];
73      b_int[i] = b[i];
74    }
75
76        for (int k = 1; k < ARRAY_SIZE; ++k) {
77          r_int[k] = ((a_int[k] / b_int[k]) / a_int[1]) / r_int[k-1];
78        }
79
80    for (int i=0; i<ARRAY_SIZE; i++) {
81      r[i] = r_int[i];
82    }
83      });
84    });
85
86    deviceQueue->throw_asynchronous();
87
88  } catch (cl::sycl::exception const& e) {
```

Details

SimpleAdd.B3:

- Most critical loop feedback path during scheduling:
  - 36.00 clock cycles 64-bit Double-precision Floating-point Divide Operation (memory_dep.cpp: 77)
- Hyper-Optimized loop structure: n/a
- Stallable instruction: None
- Maximum concurrent iterations: Capacity of loop

```
r_int[k] = ((a_int[k] / b_int[k]) / a_int[1]) / r_int[k-1];
```

Difficult double precision floating point operation must be completed

What Can You Do?

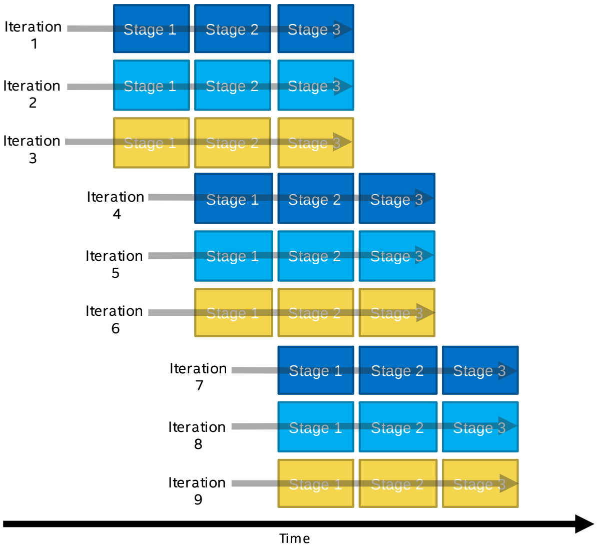
- Do a simpler calculation
- Pre-calculate some of the operations on the host
- Use a simpler type
- Use floating point optimizations (discussed later)
- Advanced technique: Increase time (pipeline stages) between start of calculation and when you use answer
  - See the “Relax Loop-Carried Dependency” in the Optimization Guide for more information

How Else to Optimize a Loop? Loop Unrolling

The compiler will still pipeline an unrolled loop, combining the two techniques

- A fully unrolled loop will not be pipelined since all iterations will kick off at once

```
handle.single_task<>([=]() {
  … //accessor setup
  #pragma unroll 3
  for (int i=1; i<9; i++) {
    c[i] += a[i] + b[i];
  }
});
```



Fmax

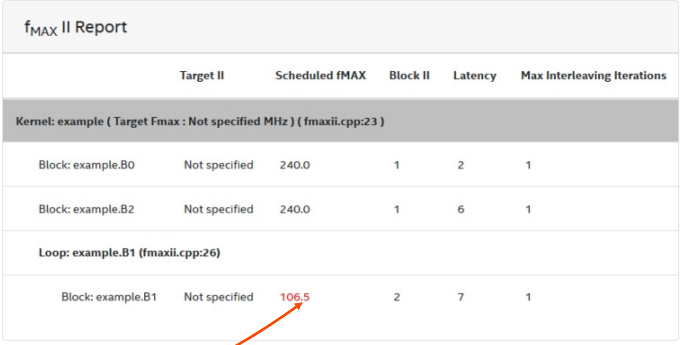
- The clock frequency the FPGA will be clocked at depends on what hardware your kernel compiles into
- More complicated hardware cannot run as fast
- The whole kernel will have one clock
- The compiler's heuristic is to sacrifice clock frequency to get a higher II

A slow operation can slow down your entire kernel by lowering the clock frequency

How Can You Tell This Is a Problem?

Fmax II report tells you the target frequency for each loop in your code.

```
cgh.single_task<example>([=]() {
  int res = N;
  #pragma unroll 8
  for (int i = 0; i < N; i++) {
    res += 1;
    res ^= i;
  }
  acc_data[0] = res;
});
```



f_{MAX} II Report

	Target II	Scheduled fMAX	Block II	Latency	Max Interleaving Iterations
Kernel: example (Target Fmax : Not specified MHz) (fmaxii.cpp:23)					
Block: example.B0	Not specified	240.0	1	2	1
Block: example.B2	Not specified	240.0	1	6	1
Loop: example.B1 (fmaxii.cpp:26)					
Block: example.B1	Not specified	106.5	2	7	1

What Can You Do?

- Make the calculation simpler
- Tell the compiler you'd like to change the trade off between II and Fmax
  - Attribute placed on the line before the loop
  - Set to a higher II than what the loop currently has

```
[[intelfpga::ii(n)]]
```

Area

The compiler sacrifices area in order to improve loop performance. What if you would like to save on the area in some parts of your design?

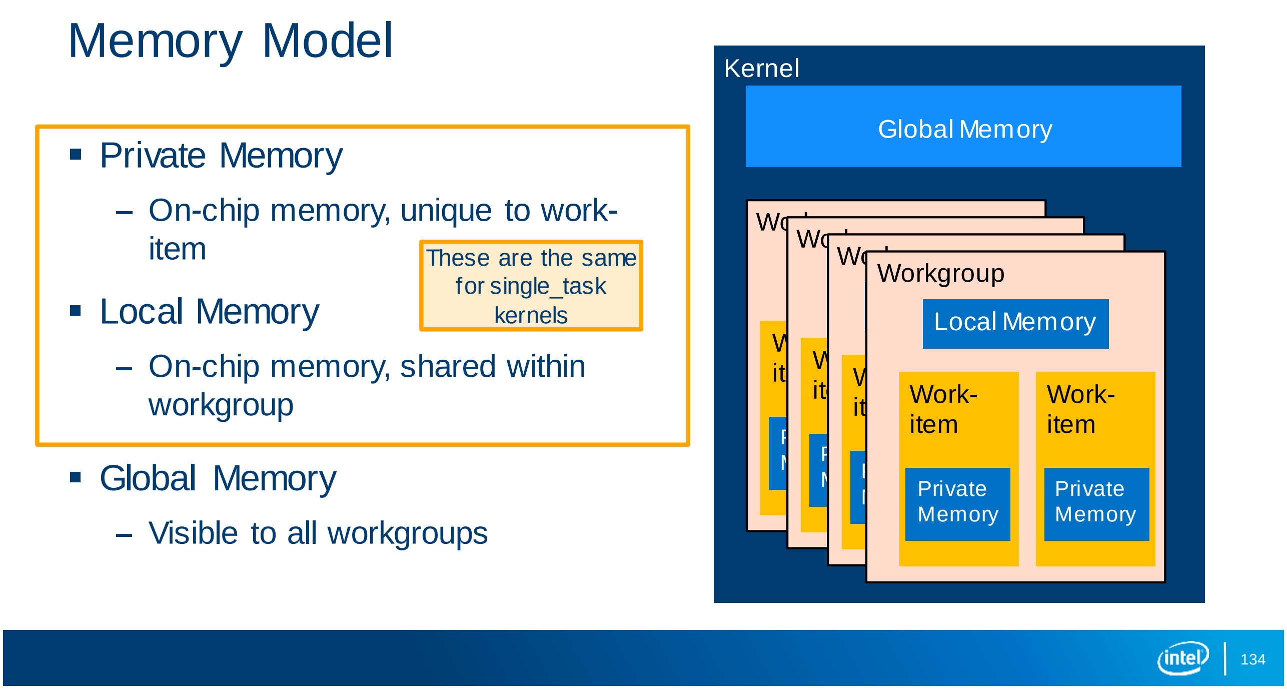
- Give up II for less area
  - Set the II higher than what compiler result is

```
[[intelfpga::ii(n)]]
```

- Give up loop throughput for area
  - Compiler increases loop concurrency to achieve greater throughput
  - Set the max_concurrency value lower than what the compiler result is

```
[[intelfpga::max_concurrency(n)]]
```

Memory Optimization



Memory Model

- Private Memory
  - On-chip memory, unique to work-item
- Local Memory
  - On-chip memory, shared within workgroup

These are the same for single_task kernels

- Global Memory
  - Visible to all workgroups

Kernel

Global Memory

Workgroup

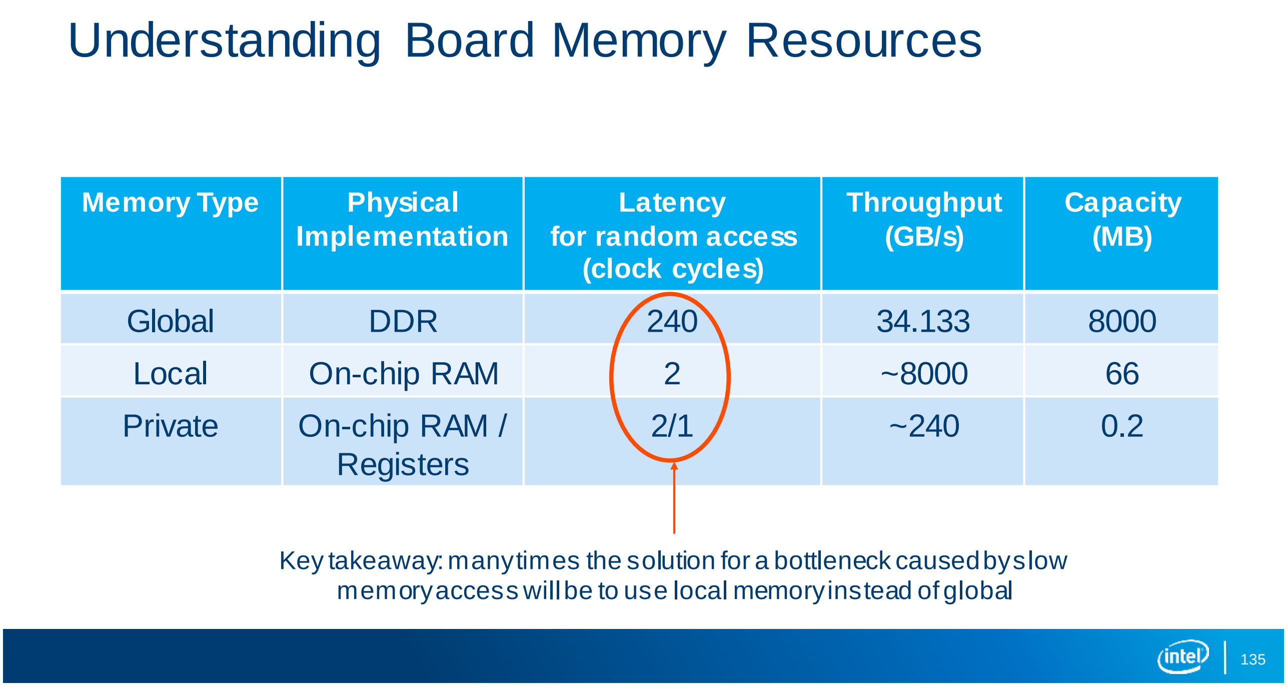
Local Memory

Work-item

Private Memory

Work-item

Private Memory



Understanding Board Memory Resources

Memory Type	Physical Implementation	Latency for random access (clock cycles)	Throughput (GB/s)	Capacity (MB)
Global	DDR	240	34.133	8000
Local	On-chip RAM	2	~8000	66
Private	On-chip RAM / Registers	2/1	~240	0.2

Key takeaway: many times the solution for a bottleneck caused by slow memory access will be to use local memory instead of global

Global Memory Access is Slow – What to Do? (4)

We've seen this before… This will appear as a *memory dependency* problem

Transfer global memory contents to local memory before operating on the data

```
…
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  auto A =
    A_buf.get_access<access::mode::read_write>(cgh);

  cgh.single_task<class unoptimized>([=]() {
    for (unsigned i = 0; i < N; i++)
      A[N-i] = A[i];
    }
  });

});
…
```

Non-optimized

```
…
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  auto A =
    A_buf.get_access<access::mode::read_write>(cgh);

  cgh.single_task<class optimized>([=]() {
    int B[N];

    for (unsigned i = 0; i < N; i++)
      B[i] = A[i];

    for (unsigned i = 0; i < N; i++)
      B[N-i] = B[i];

    for (unsigned i = 0; i < N; i++)
      A[i] = B[i];
  });

});
…
```

Optimized

Local Memory Considerations for Parallel Kernels

- Declaring a variable in the kernel scope creates a private memory space
  - Always works with single work-item kernels because there's just 1 work item
- Visibility between work-items is a concern in parallel kernels
- 2 ways to create local memory that work-items in a work-group can share

Creating Local Memory for Parallel Kernels

By creating an accessor with the access::target::local target

No buffer argument because host cannot access it

No way to size the memory, it will be the default size of 16kB always

```
accessor <int, 1,
  sycl::access::mode::read_write,
  sycl::access::target::local>
  local_mem(sycl::range<1>(wgroup_size), cgh);
```

Use the parallel_for_work_group function to launch the kernel (this is still somewhat experimental)

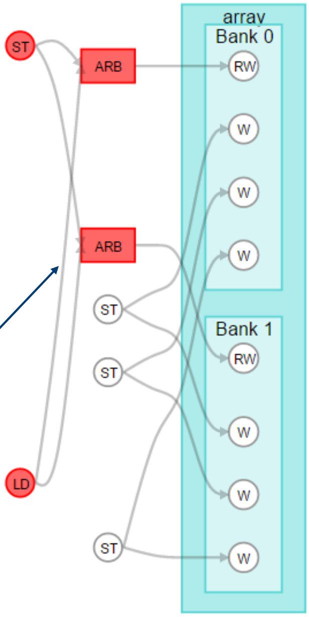
```
cgh.parallel_for_work_group<class myKernel>(
range<1>(8), range<1>(8), [=](group<1> myGroup)
{
  //Code for work-group
  //This variable is shared between work-items
  //This variable can be sized
  int myLocal[8];
…
```

Local Memory Bottlenecks

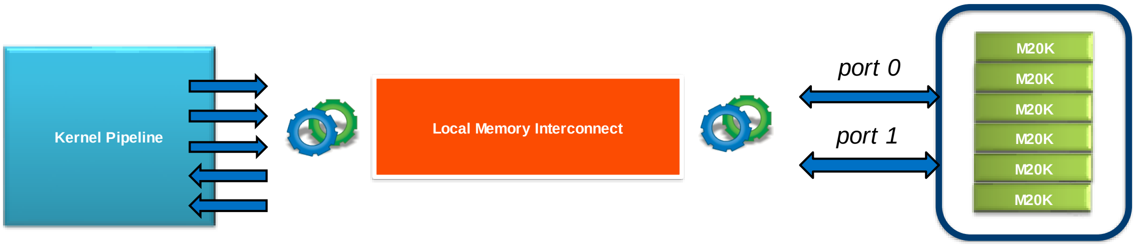
If more load and store points want to access the local memory than there are ports available, arbiters will be added

These can stall, so are a potential bottleneck

Show up in red in the Memory Viewer section of the optimization report



Local Memory Bottlenecks



Natively, the memory architecture has 2 ports

The compiler optimizes memory accesses to map to these without arbitration

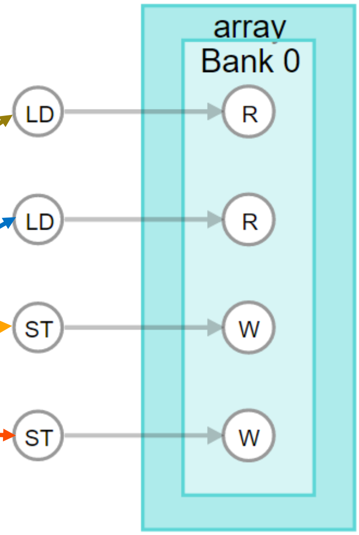
Your job is to write code the compiler can optimize

Double-Pumped Memory Example

Increase the clock rate to 2x

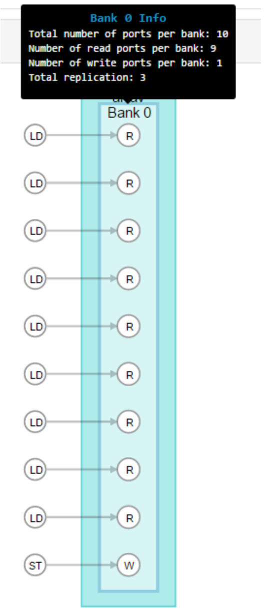
Compiler can automatically implement double-pumped memory – turning 2 ports to 4

```
//kernel scope
...
  int array[1024];

  array[ind1] = val;

  array[ind1+1] = val;

  calc = array[ind2] + array[ind2+1];
…
```



Local Memory Replication Example

```
//kernel scope
…
    int array[1024];
    int res = 0;

ST  array[ind1] = val;
    #pragma unroll
    for (int i = 0; i < 9; i++)
LD       res += array[ind2+i];

    calc = res;
…
```



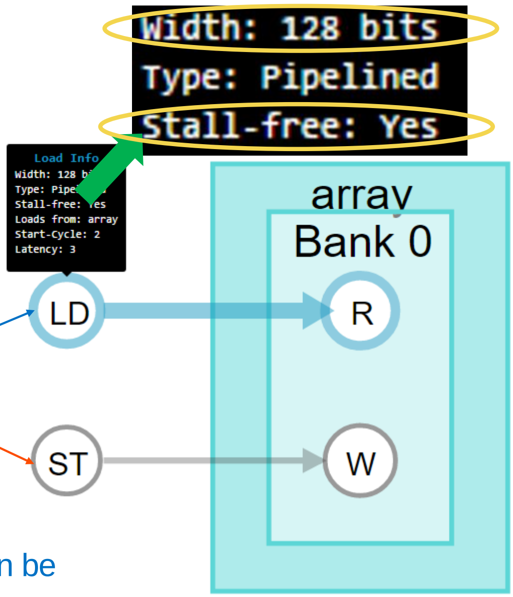
Bank 0 Info
Total number of ports per bank: 10
Number of read ports per bank: 9
Number of write ports per bank: 1
Total replication: 3

Turn 4 ports of double-pumped memory to unlimited ports

Drawbacks: logic resources, stores must go to each replication

Coalescing

```
//kernel scope
…
local int array[1024];
int res = 0;

#pragma unroll
for (int i = 0; i < 4; i++)
    array[ind1*4 + i] = val;

#pragma unroll
for (int i = 0; i < 4; i++)
    res += array[ind2*4 + i];

calc = res;
…
```



Width: 128 bits
Type: Pipelined
Stall-free: Yes

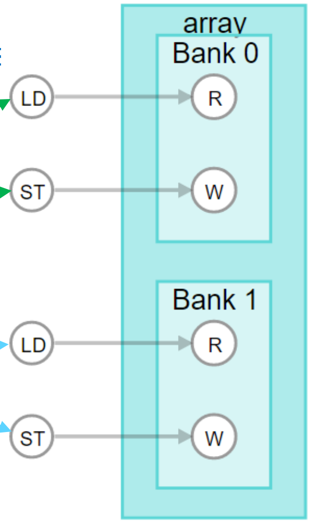
Load Info
Width: 128 bi
Type: Pipe
Stall-free: Yes
Loads from: array
Start-Cycle: 2
Latency: 3

Continuous addresses can be coalesced into wider accesses

Banking

Divide the memory into independent fractional pieces (banks

```
//kernel scope
…
int array[1024][2];

array[ind1][0] = val1;
array[ind2][1] = val2;

calc =  (array[ind2][0] +
         array[ind1][1]);
…
```



Compiler looks at lower indices by default

Indices for banking must be a power of 2 size

Attributes for Local Memory Optimization

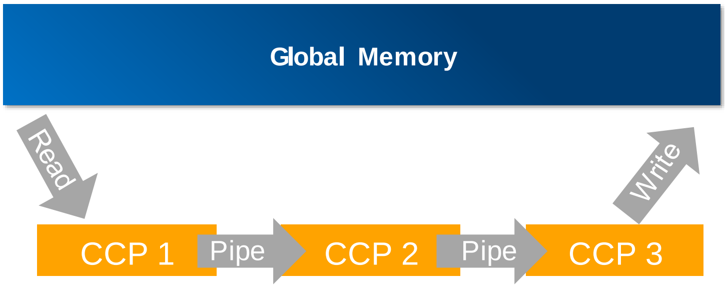
Note: Let the compiler try on it's own first. It's very good at inferring an optimal structure!

Attribute	Usage
numbanks	[[intelfpga::numbanks(N)]]
bankwidth	[[intelfpga::bankwidth(N)]]
singlepump	[[intelfpga::singlepump]]
doublepump	[[intelfpga::doublepump]]
max_replicates	[[intelfpga::max_replicates(N)]]
simple_dual_port	[[intelfpga::simple_dual_port]]

Note: This is not a comprehensive list. Consult the Optimization Guide for more.

Pipes – Element the Need for Some Memory

Create custom direct point-to-point communication between CCPs with Pipes

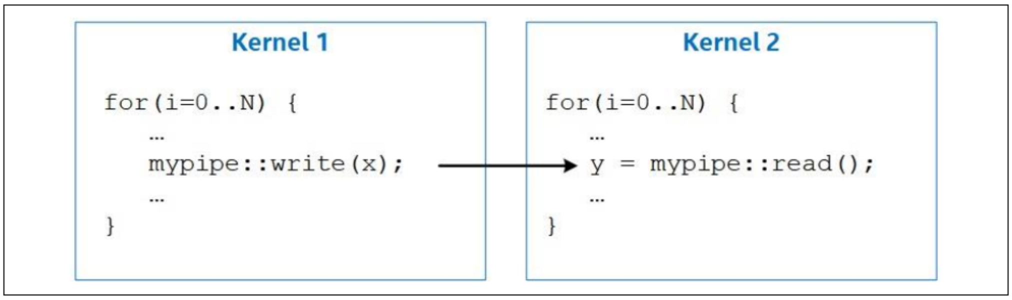


Task Parallelism By Using Pipes

Launch separate kernels simultaneously

Achieve synchronization and data sharing using pipes

Make better use of your hardware



```
for(i=0..N) {
    …
    mypipe::write(x);
    …
}
```

```
for(i=0..N) {
    …
    y = mypipe::read();
    …
}
```

Other Optimization Techniques

Avoid Expensive Functions

- Expensive functions take a lot of hardware and run slow
- Examples
  - Integer division and modulo (remainder) operators
  - Most floating-point operations except addition, multiplication, absolution, and comparison
  - Atomic functions

Inexpensive Functions

- Use instead of expensive functions whenever possible
  - Minimal effects on kernel performance
  - Consumes minimal hardware
- Examples
  - Binary logic operations such as AND, NAND, OR, NOR, XOR, and XNOR
  - Logical operations with one constant argument
  - Shift by constant
  - Integer multiplication and division by a constant that is to the power of 2
  - Bit swapping (Endian adjustment)

Use Least-“Expensive” Data Type

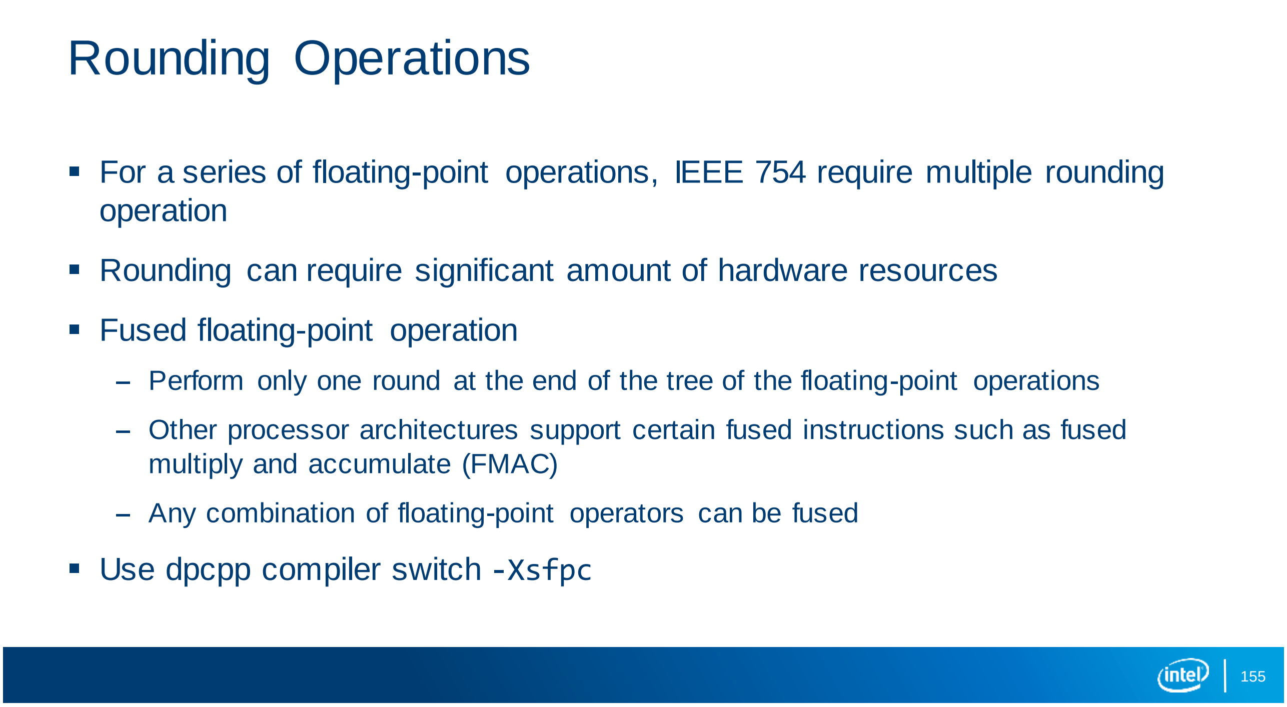
- Understand cost of each data type in latency and logic usage
  - Logic usage may be > 4x for double vs. float operations
  - Latency may be much larger for float and double operations compared to fixed point types
- Measure or restrict the range and precision (if possible)
  - Be familiar with the width, range and precision of data types
  - Use half or single precision instead of double (default)
  - Use fixed point instead of floating point
  - Don’t use float if short is sufficient

Floating-Point Optimizations

- Apply to `half`, `float` and `double` data types
- Optimizations will cause small differences in floating-point results
  - **Not** IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) compliant
- Floating-point optimizations:
  - Tree Balancing
  - Reducing Rounding Operations

Tree-Balancing

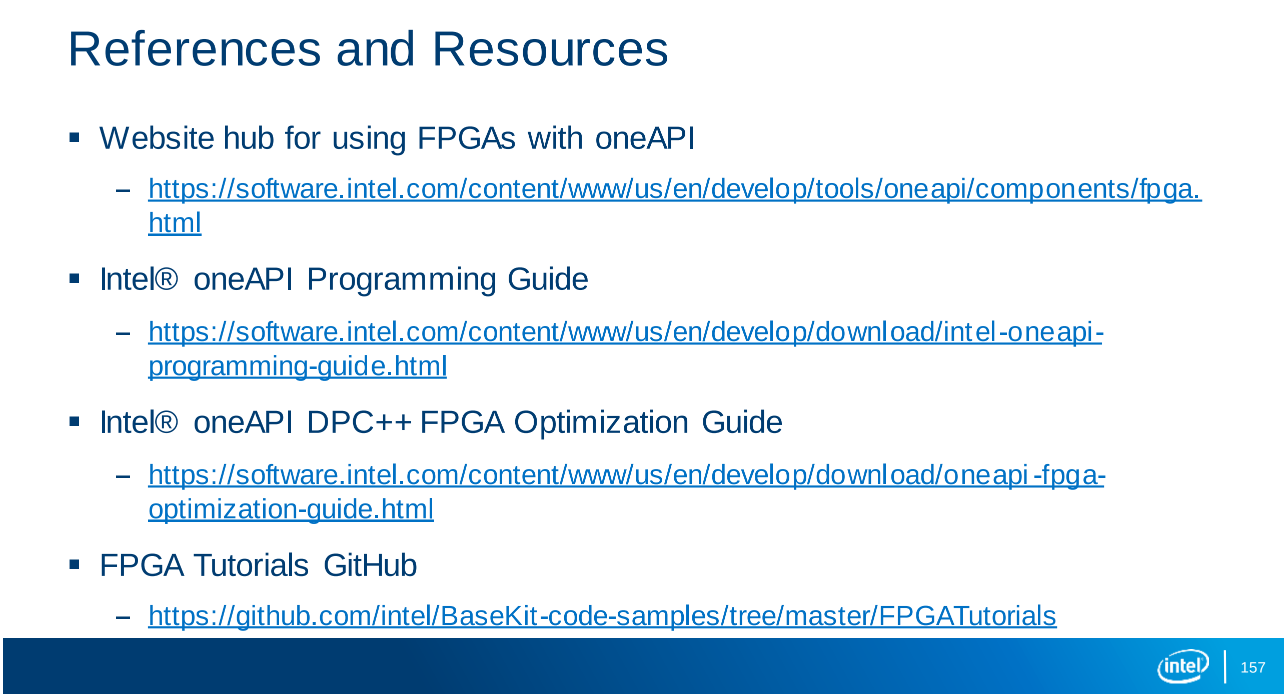
- Floating-point operations are not associative
  - Rounding after each operation affects the outcome
  - ie. ((a+b) + c) != (a+(b+c))
- By default the compiler doesn’t reorder floating-point operations
  - May creates an imbalance in a pipeline, costs latency and possibly area
- Manually enable compiler to balance operations
  - For example, create a tree of floating-point additions in SGEMM, rather than a chain
  - Use **-Xsfp-relaxed=true** flag when calling dpcpp



Rounding Operations

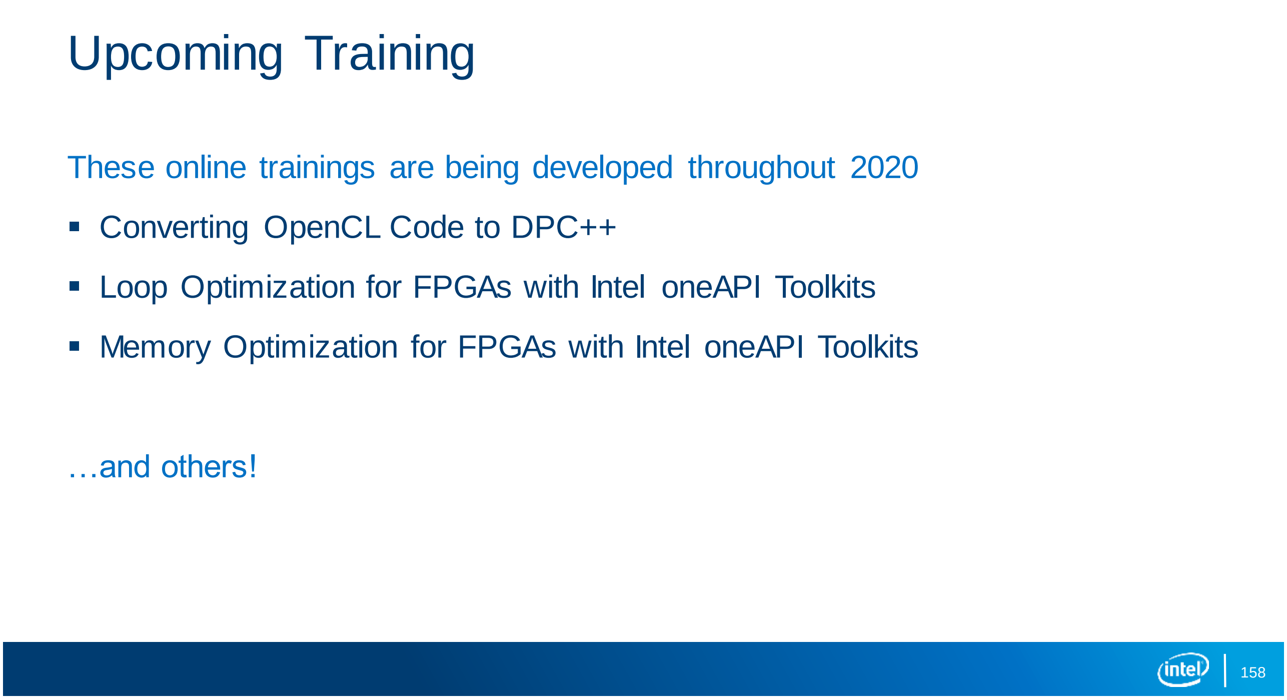
- For a series of floating-point operations, IEEE 754 require multiple rounding operation
- Rounding can require significant amount of hardware resources
- Fused floating-point operation
  - Perform only one round at the end of the tree of the floating-point operations
  - Other processor architectures support certain fused instructions such as fused multiply and accumulate (FMAC)
  - Any combination of floating-point operators can be fused
- Use dpcpp compiler switch `-Xsfpc`

References and Resources



References and Resources

- Website hub for using FPGAs with oneAPI
  - https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/fpga.html
- Intel® oneAPI Programming Guide
  - https://software.intel.com/content/www/us/en/develop/download/intel-oneapi-programming-guide.html
- Intel® oneAPI DPC++ FPGA Optimization Guide
  - https://software.intel.com/content/www/us/en/develop/download/oneapi-fpga-optimization-guide.html
- FPGA Tutorials GitHub
  - https://github.com/intel/BaseKit-code-samples/tree/master/FPGATutorials



Upcoming Training

These online trainings are being developed throughout 2020

- Converting OpenCL Code to DPC++
- Loop Optimization for FPGAs with Intel oneAPI Toolkits
- Memory Optimization for FPGAs with Intel oneAPI Toolkits

…and others!

Lab: Optimizing the Hough Transform Kernel

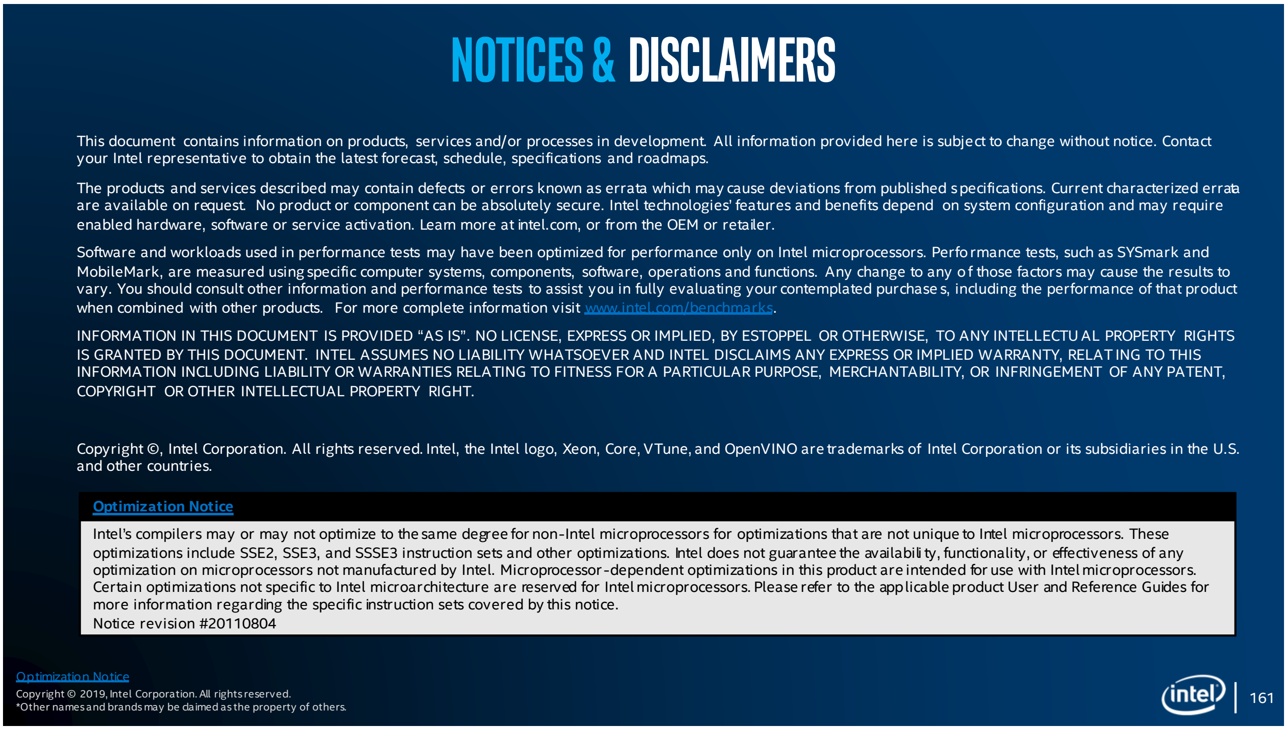
Legal Disclaimers/Acknowledgements

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at www.intel.com.

Intel, the Intel logo, Intel Inside, the Intel Inside logo, MAX, Stratix, Cyclone, Arria, Quartus, HyperFlex, Intel Atom, Intel Xeon and Enpirion are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

OpenCL is the trademark of Apple Inc. used by permission by Khronos

*Other names and brands may be claimed as the property of others



NOTICES & DISCLAIMERS

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804