Using Intel® oneAPI Toolkits with FPGAs

Copyright © 2021 Intel Corporation.

This document is intended for personal use only.

Unauthorized distribution, modification, public performance,
public display, or copying of this material via any medium is strictly prohibited.

Course Objectives

- Understand the development flow for FPGAs with the Intel® oneAPI toolkits
- Gain an understanding of common optimization methods for FPGAs

Copyright © 2021 Intel Corporation

Course Agenda

- Using FPGAs with the Intel® oneAPI Toolkits
  - Recap: Introduction to DPC++
  - 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

Copyright © 2021 Intel Corporation

Timeline

Section	Time
Slides: Using FPGAs with the Intel® oneAPI Toolkits	14:00 -14:30
Lab: Practice the FPGA Development Flow	14:30 -15:30
Break	15:30 - 16:00
Slides: Optimizing Your Code for FPGAs	16:00 -16:30
Lab: Optimizing the Hough Transform Kernel	16:30 - 17:30

Copyright © 2021 Intel Corporation

Section:
Using FPGAs with the Intel® oneAPI Toolkits

Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- What are FPGAs and Why Should I Care About Programming Them?
- Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

Copyright © 2021 Intel Corporation

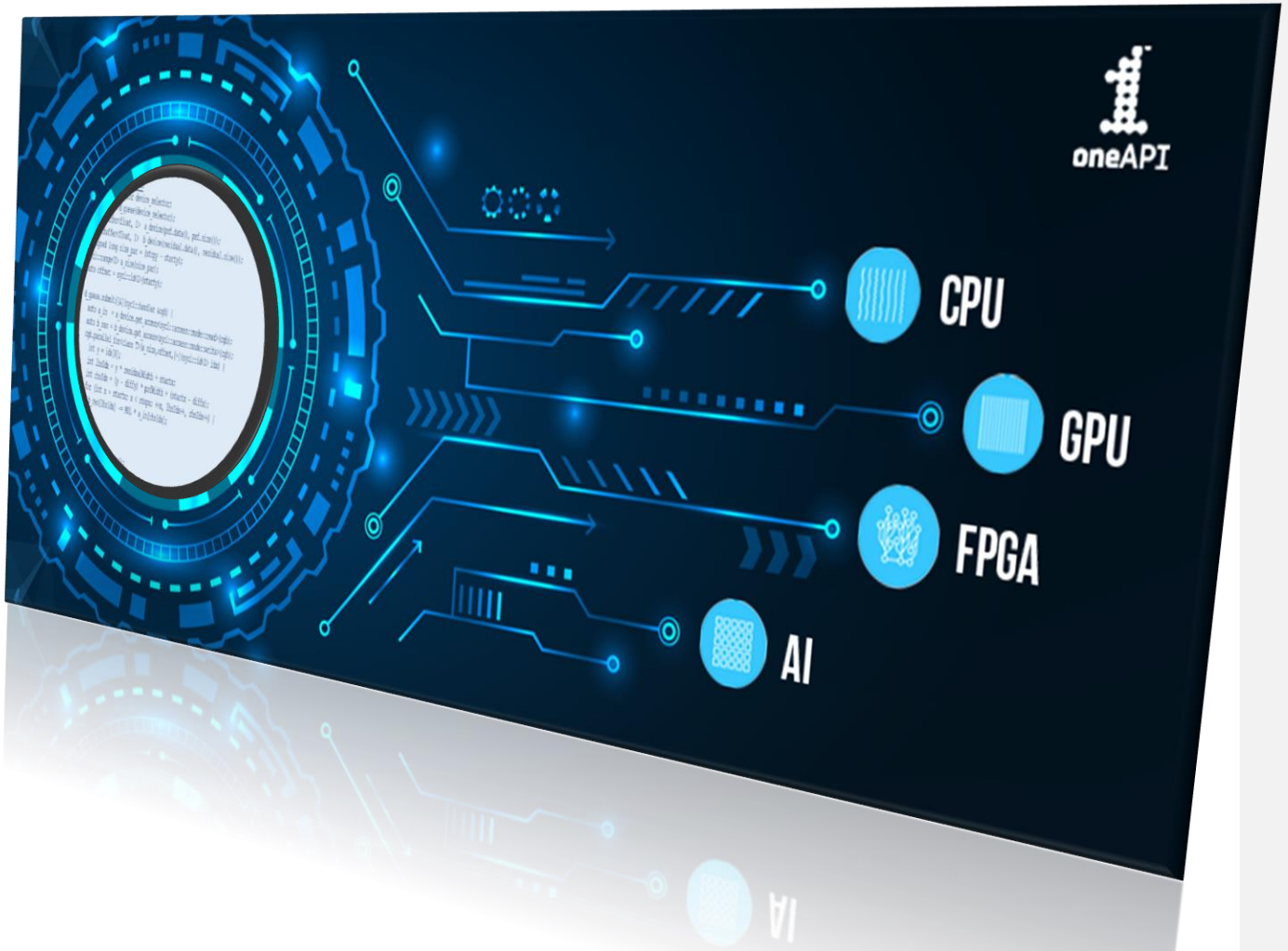
A Unified Programming Model

Multiple Architectures

The **oneAPI** product delivers a unified programming model to simplify development across diverse architectures.

It guarantees:

- **Common developer experience** across Scalar, Vector, Matrix and Spatial architectures (CPU, GPU, AI and FPGA)
- Uncompromised native high-level language **performance**
- Industry standardization and open specifications



Copyright © 2021 Intel Corporation

Intel® oneAPI Product

Faster Development

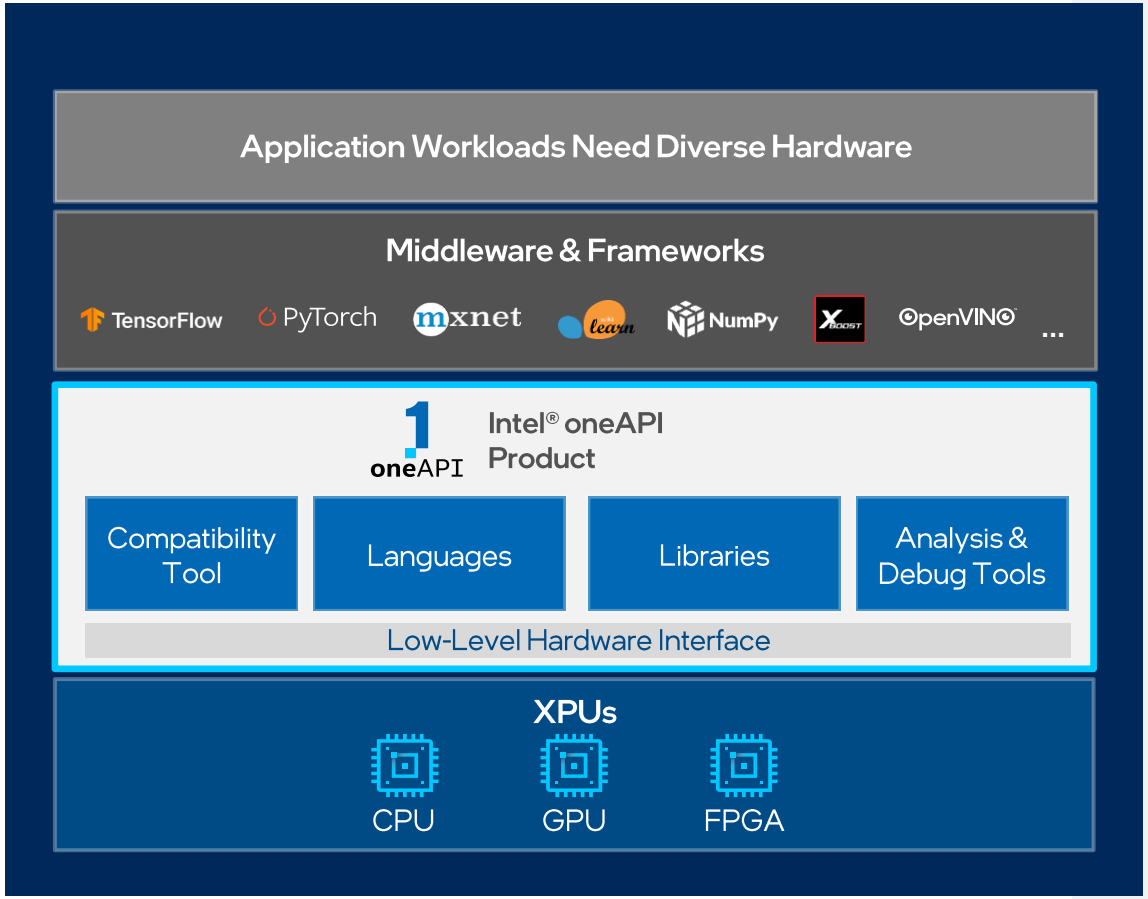
- Performance tuning and timing closure through **emulation** and **reports**.
- Runtime analysis via **VTune™ Profiler**
- Complex hardware patterns implemented through built-in language features: **macros**, **pragmas**, **headers**

Extensible Code

- **Code re-use** across architectures and vendors**.**
- Compatible with existing high-performance languages.

Reduced Barrier of Entry

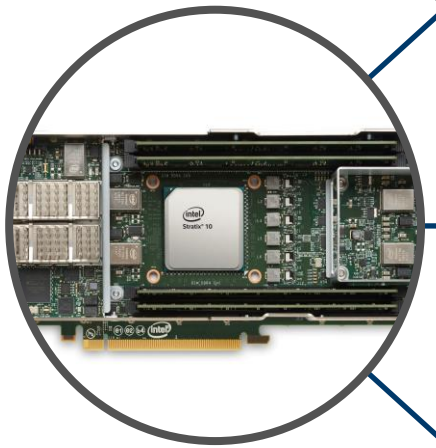
- Leverage familiar sequential programming languages: improved **ramp-up** and **debug** time.
- IDE Integration: **Eclipse, VS, VS Code**



Available Now

Copyright © 2021 Intel Corporation

Intel® FPGAs + Intel® oneAPI Toolkits



Spatial Architecture

- Data-dependent parallelism
- Streaming and graph processing patterns

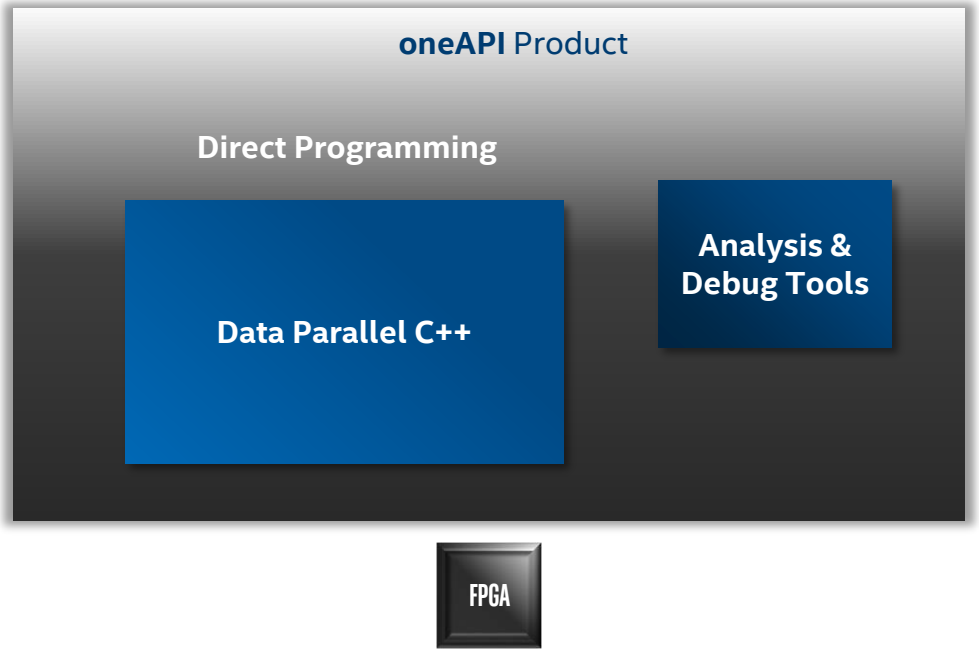
Rich I/O

- Low and deterministic latency
- Customizable network interfaces and protocols

Memory

- Customizable memory architecture
- Distributed, high bandwidth, on-chip memory topology

+



Copyright © 2021 Intel Corporation

Section:
Using FPGAs with the Intel® oneAPI Toolkits

Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- What are FPGAs and Why Should I Care About Programming Them?
- Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

Copyright © 2021 Intel Corporation

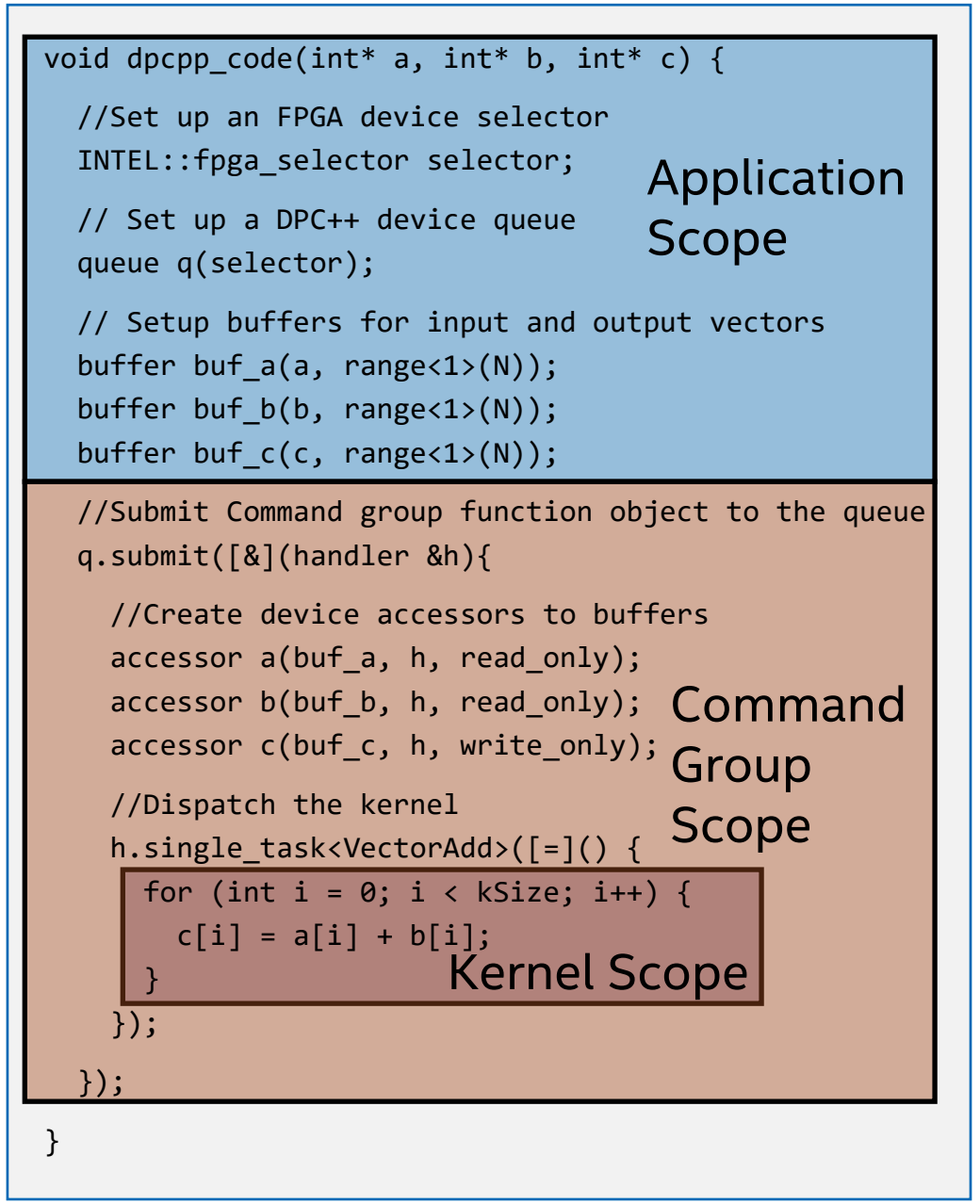
Data Parallel C++ (DPC++)

- Based on C++ and SYCL
  - SYCL is based on OpenCL
  - Think of it as SYCL + extensions
- Allows for single-source targeting of accelerators
  - (Doesn't require multiple files)
- Open specification
- Common language meant to target all XPUs
  - You do still need to "tune"
- Goal is for the language to incorporate everything needed to get the best performance out of every architecture

Copyright © 2021 Intel Corporation

DPC++: Three Scopes

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



Copyright © 2021 Intel Corporation

The “Runtime”

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

Copyright © 2021 Intel Corporation

DPC++ Simple Program Walk-Through

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

Step 1: Create a device selector targeting the FPGA

Copyright © 2021 Intel Corporation

DPC++ Simple Program Walk-Through

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

Step 1: Create a device selector targeting the FPGA

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

Copyright © 2021 Intel Corporation

DPC++ Simple Program Walk-Through

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

Step 1: Create a device selector targeting the FPGA

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

Step 3: Create buffers

Copyright © 2021 Intel Corporation

DPC++ Simple Program Walk-Through

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

Step 1: Create a device selector targeting the FPGA

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

Step 3: Create buffers

Step 4: Submit a command group for execution

Copyright © 2021 Intel Corporation

DPC++ Simple Program Walk-Through

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

Step 1: Create a device selector targeting the FPGA

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

Step 3: Create buffers

Step 4: Submit a command for execution

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

Copyright © 2021 Intel Corporation

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

DPC++ Simple Program Walk-Through

Step 1: Create a device selector targeting the FPGA

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

Step 3: Create buffers

Step 4: Submit a command for execution

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

Step 6: Send a kernel for execution

Copyright © 2021 Intel Corporation

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

DPC++ Simple Program Walk-Through

Step 1: Create a device selector targeting the FPGA

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

Step 3: Create buffers

Step 4: Submit a command for execution

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

Step 6: Send a kernel for execution

Done!

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

(because of the buffer destruction of buf_c)

Copyright © 2021 Intel Corporation

Section:
Using FPGAs with the Intel® oneAPI Toolkits

Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- What are FPGAs and Why Should I Care About Programming Them?
- Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

Copyright © 2021 Intel Corporation

What is an FPGA?

FPGA stands for Field Programmable Gate Array

Gate refers to logic gates

- The basic building blocks for all the hardware on the chip

Array means there are many of them manufactured on the chip

- Many = billions
- Arranged into larger structures (more on this later)

Field Programmable means the internal components of the device and the connections between them are programmable after deployment

- Programmable = configurable

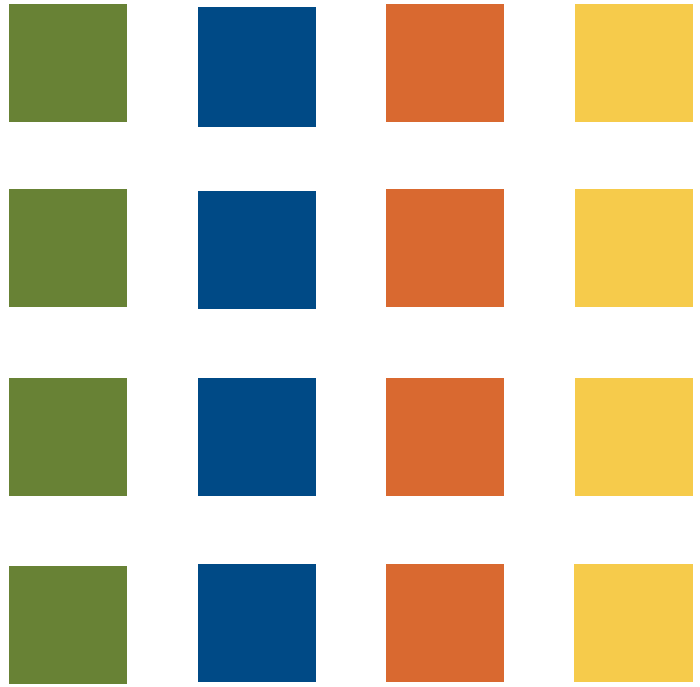
FPGA = Configurable Hardware

Copyright © 2021 Intel Corporation

Programming an FPGA

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

Programming it means choosing:



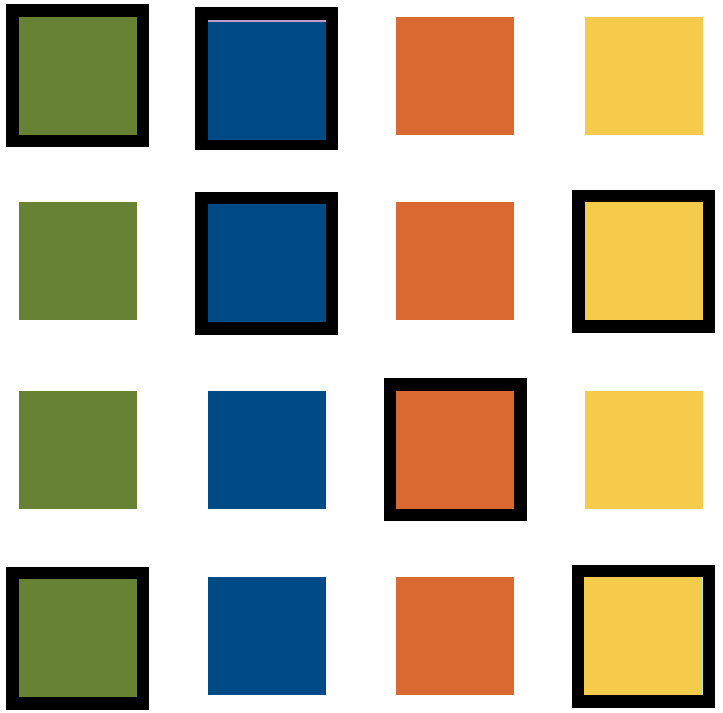
Copyright © 2021 Intel Corporation

Programming an FPGA

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

Programming it means choosing:

- The building blocks to use



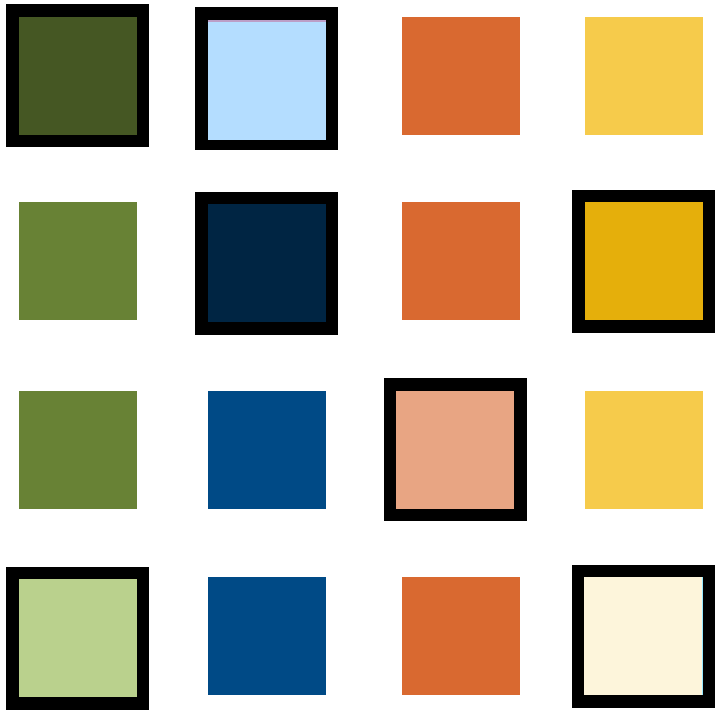
Copyright © 2021 Intel Corporation

Programming an FPGA

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

Programming it means choosing:

- The building blocks to use
- How to configure them



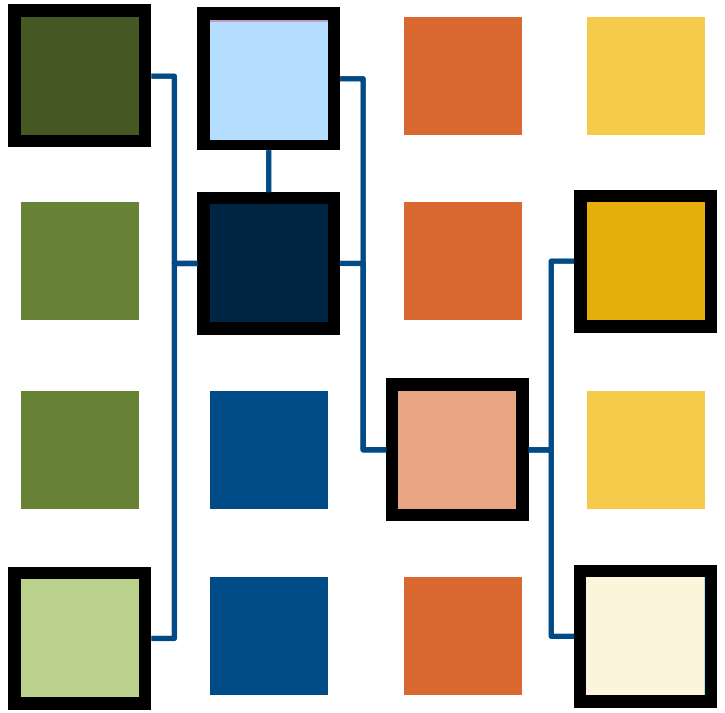
Copyright © 2021 Intel Corporation

Programming an FPGA

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

Programming it means choosing:

- The building blocks to use
- How to configure them
- And how to connect them



Copyright © 2021 Intel Corporation

Programming an FPGA

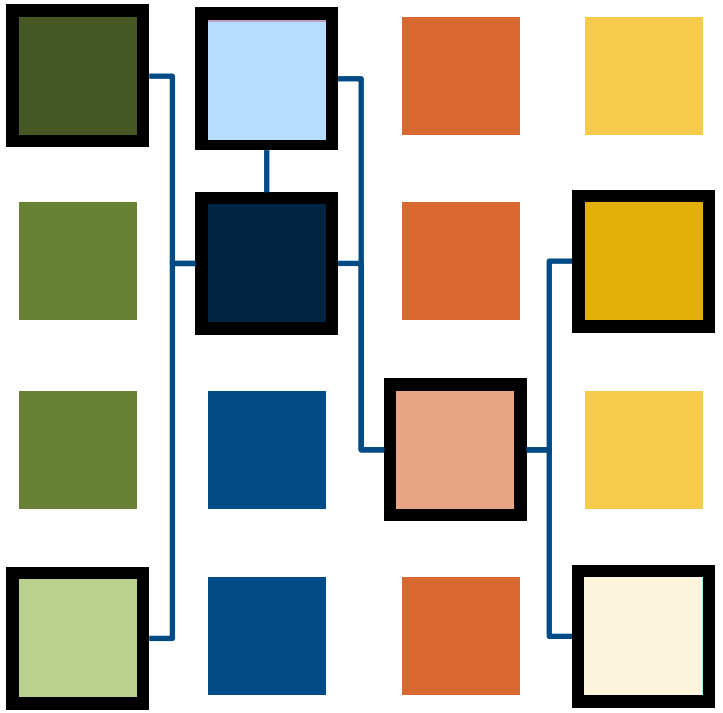
The FPGA is made up of small building blocks of logic and other functions

Programming it means choosing:

- The building blocks to use
- How to configure them
- And how to connect them

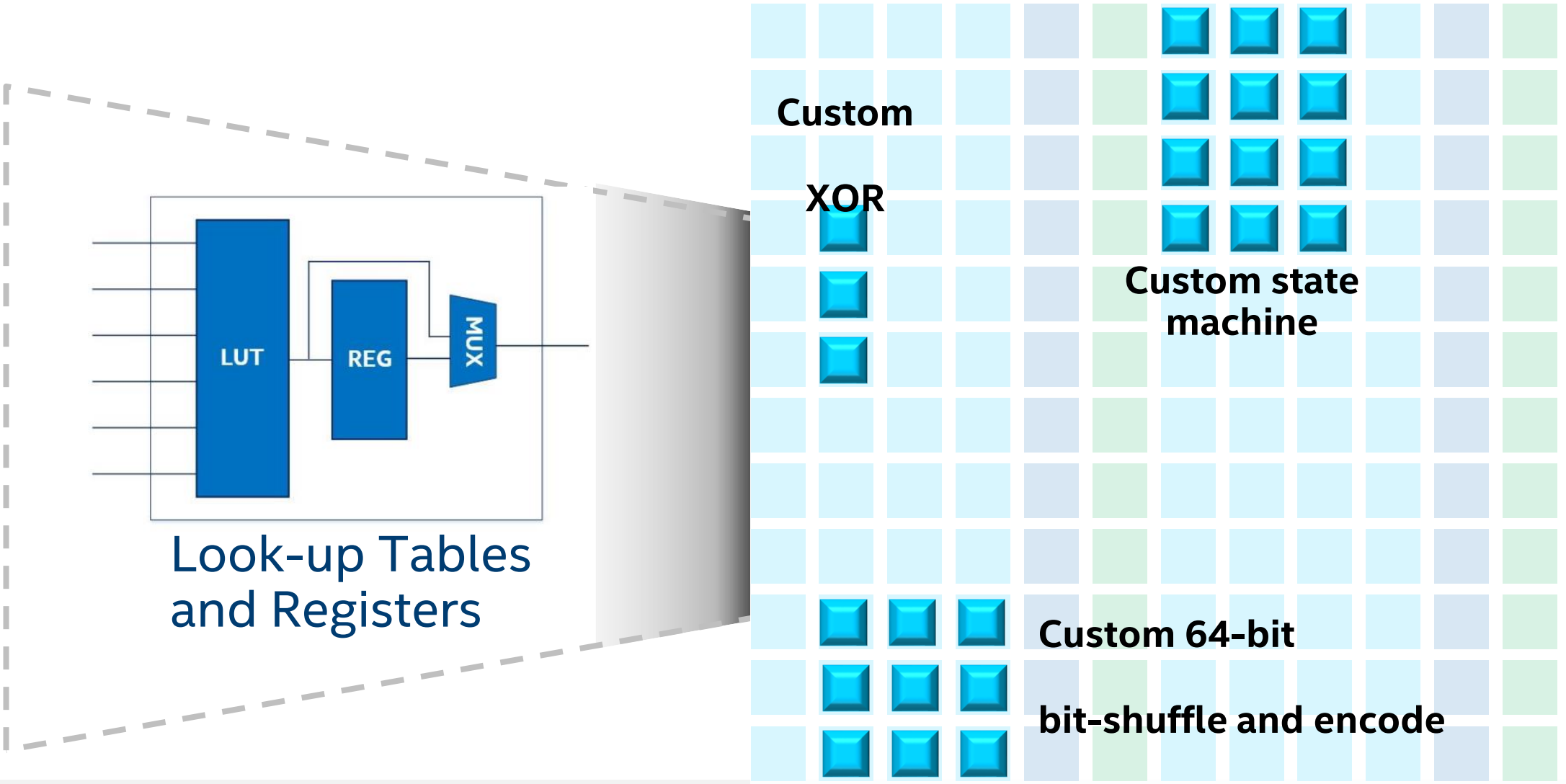
Programming determines the processing architecture implemented in the FPGA

=> what function the FPGA performs



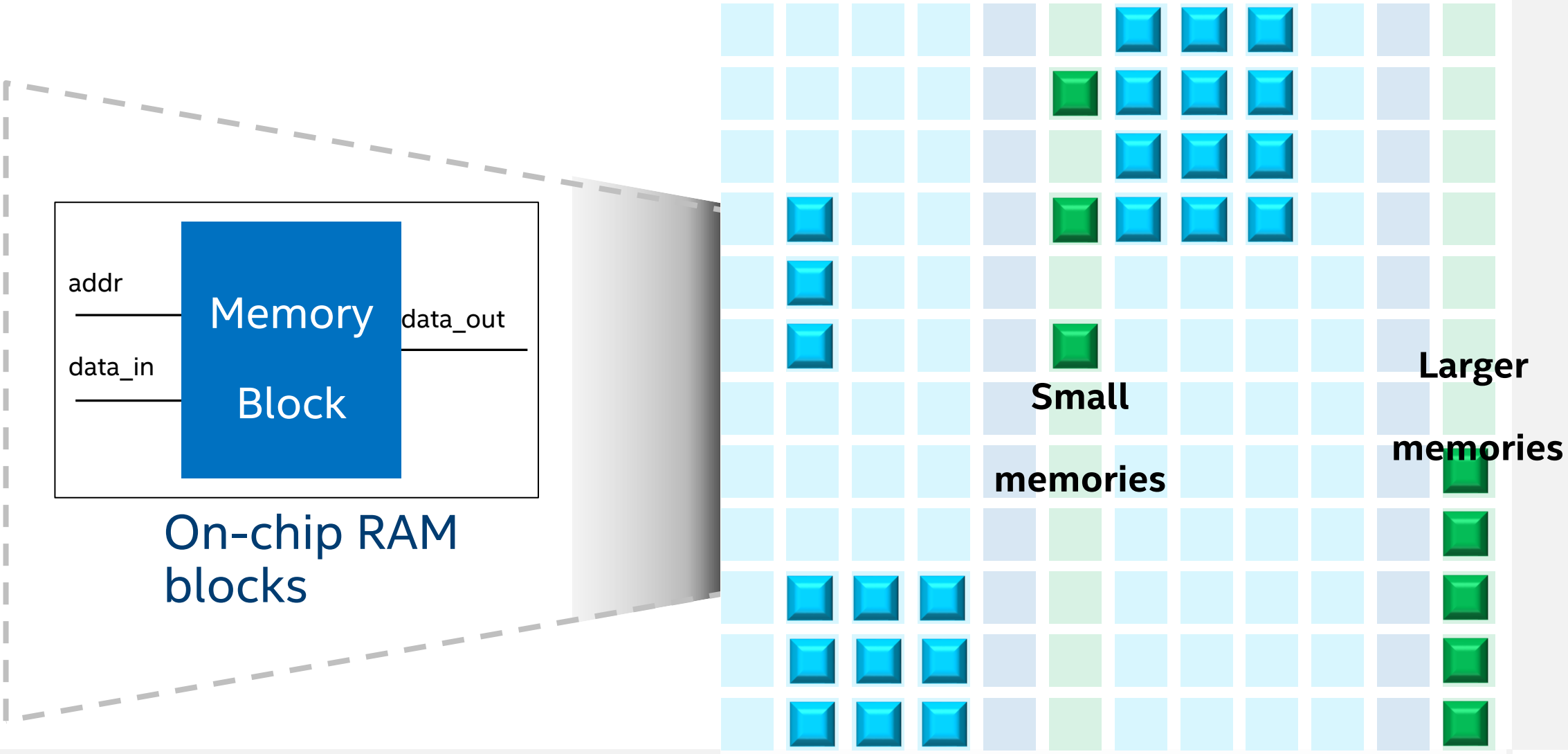
Copyright © 2021 Intel Corporation

FPGA basic building blocks -ALMs



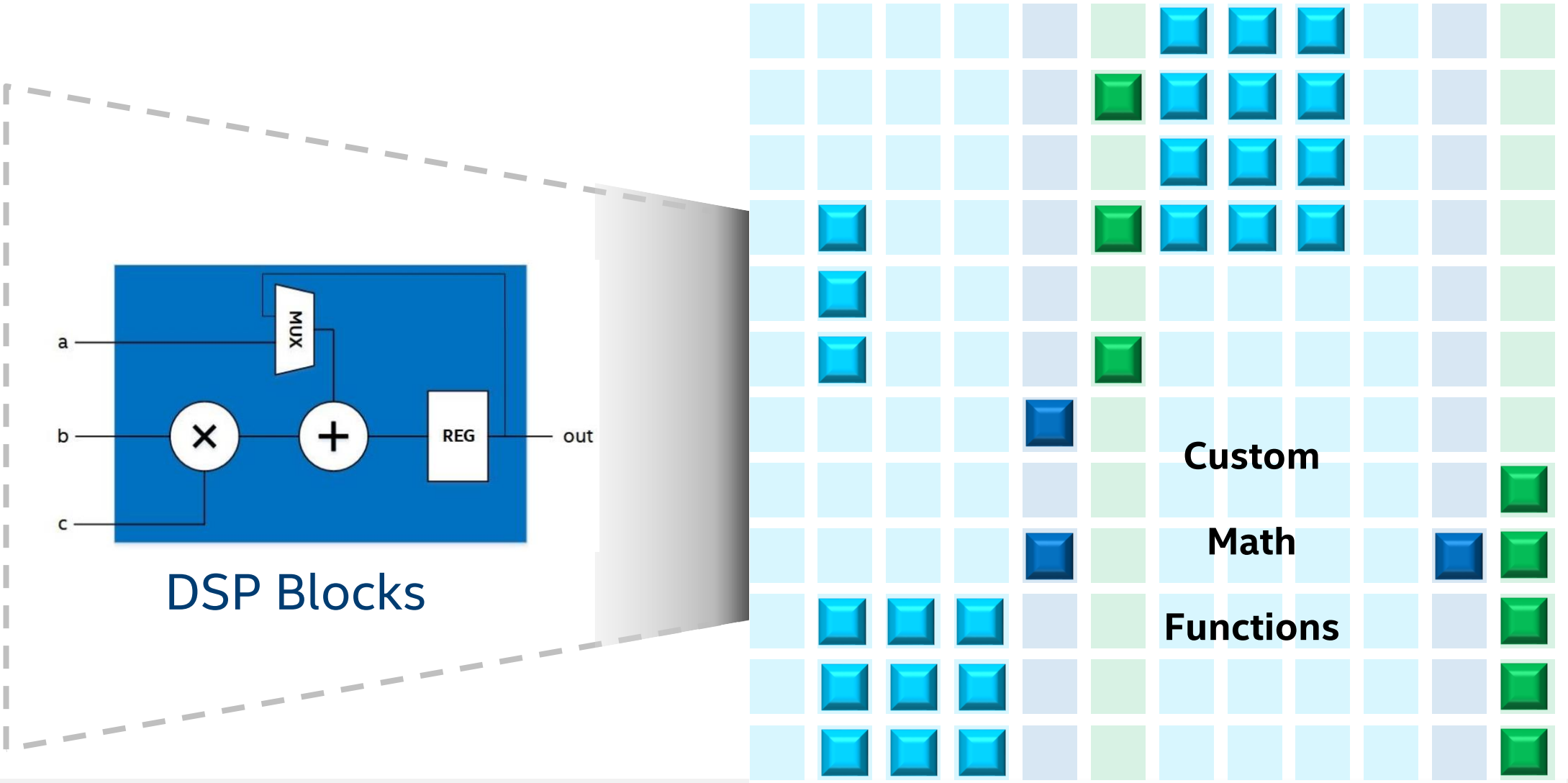
Copyright © 2021 Intel Corporation

FPGA basic building blocks - RAM



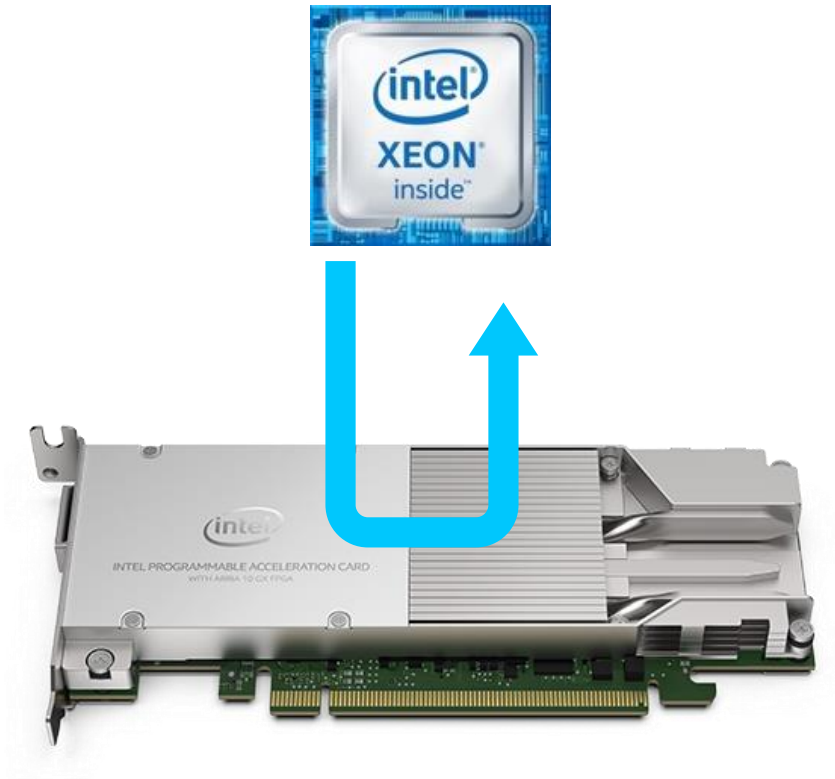
Copyright © 2021 Intel Corporation

FPGA basic building blocks - DSP blocks



Copyright © 2021 Intel Corporation

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.

Copyright © 2021 Intel Corporation

Program Implementation in FPGA

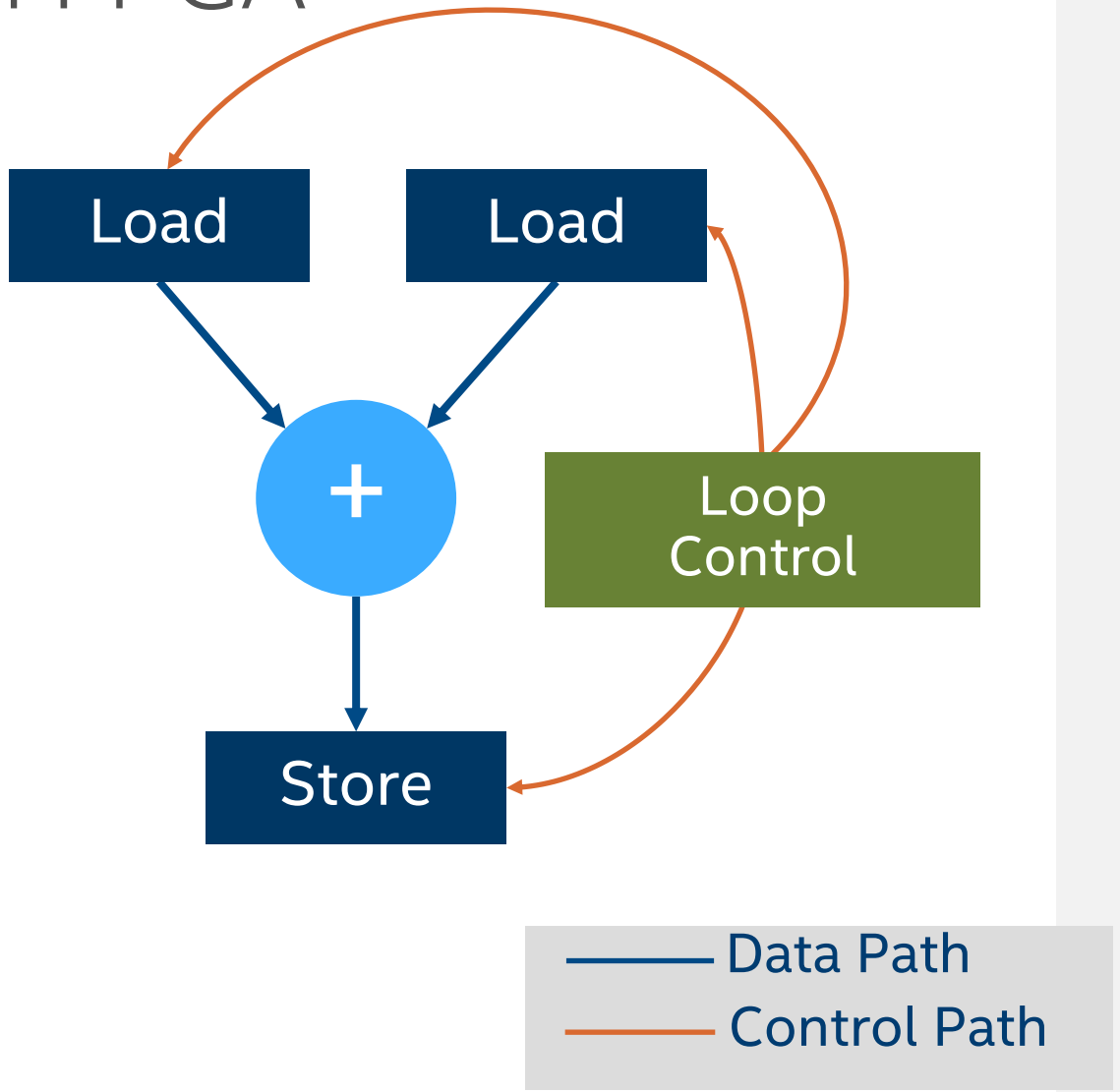
Pipelined hardware is implemented for:

- Computation (operators, …)
- Memory loads and stores
- Control and scheduling (loops, conditionals, …)

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

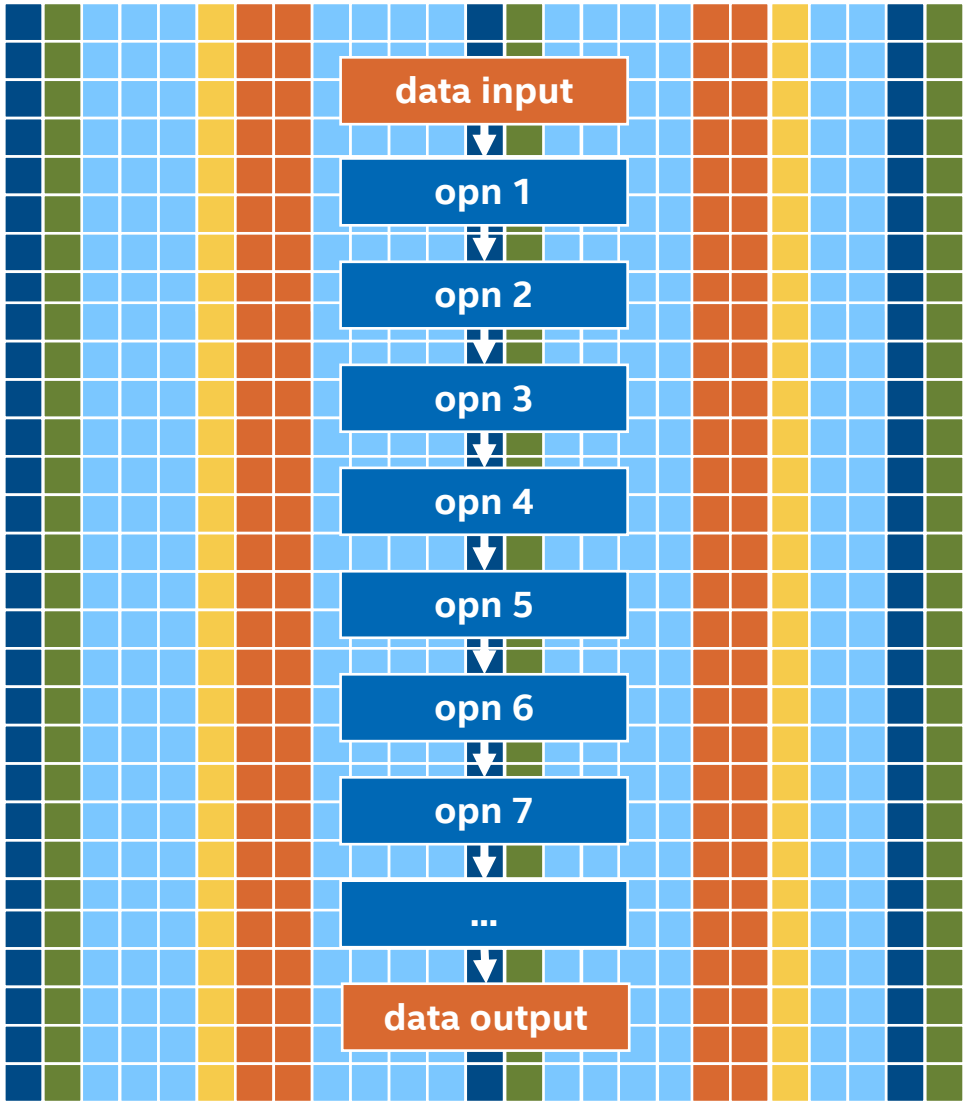
Custom on-chip memory structures are implemented for:

- Array variables declared within kernel scope
- Memory accessors with local target



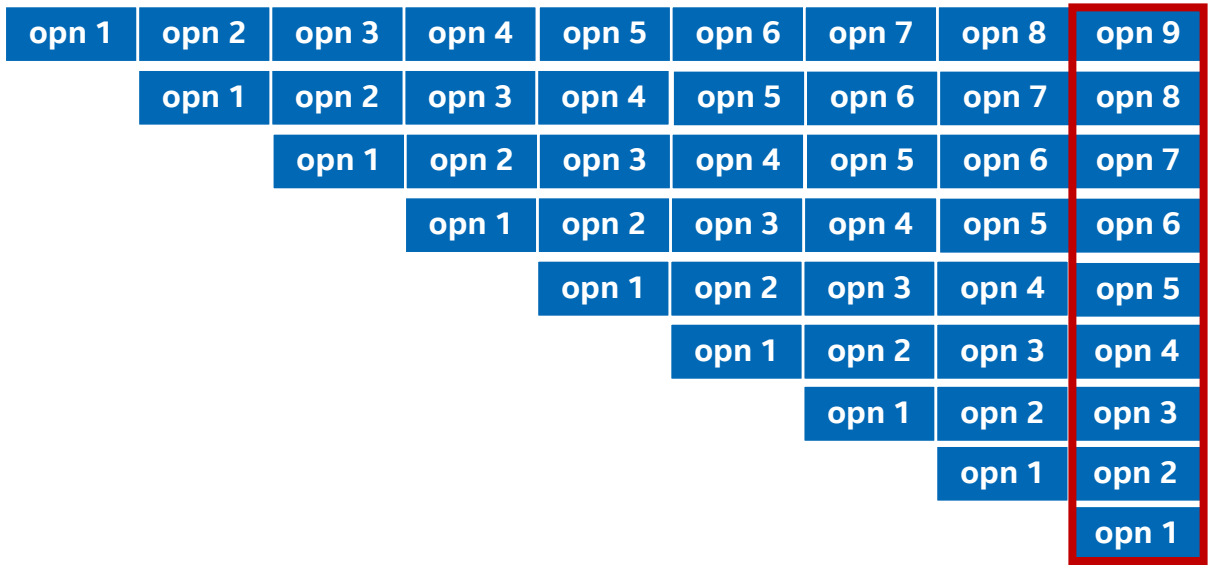
Copyright © 2021 Intel Corporation

Program execution on FPGA



Different from CPUs and GPUs

- No instruction fetched, decoded or executed
- **Data flow through hardware pipelines** matching the operations in the source code
- **No control overhead** (the dataflow hardware matches the software)
- In optimal implementations, a **new instruction stream** operating on new data starts executing **every clock cycle**
- **Pipeline parallelism** - the deeper the pipeline, the higher the parallelism



Copyright © 2021 Intel Corporation

Orthogonal Implementation Approaches

CPUs/GPUs (ISA-based architectures)

- Program => sequence of instructions
- Every Execution Unit executes one instruction at a time (some if superscalar)
- Fixed architecture
- Shared hardware

FPGA (spatial architecture)

- Program => pipelined datapath
- All program instructions can execute in parallel on different data
- Flexible architecture
- Dedicated hardware

Copyright © 2021 Intel Corporation

FPGA parallelism

Pipeline parallelism

- All hardware components execute in parallel on different data sets

Data parallelism

- Each pipeline stage can operate on multiple data on the same clock cycle

Task parallelism

- Multiple pipelines implementing different tasks can operate in parallel in the same FPGA image

Superscalar execution

- Multiple independent instructions in pipelines execute on the same clock cycle

Copyright © 2021 Intel Corporation

Section:
Using FPGAs with the Intel® oneAPI Toolkits

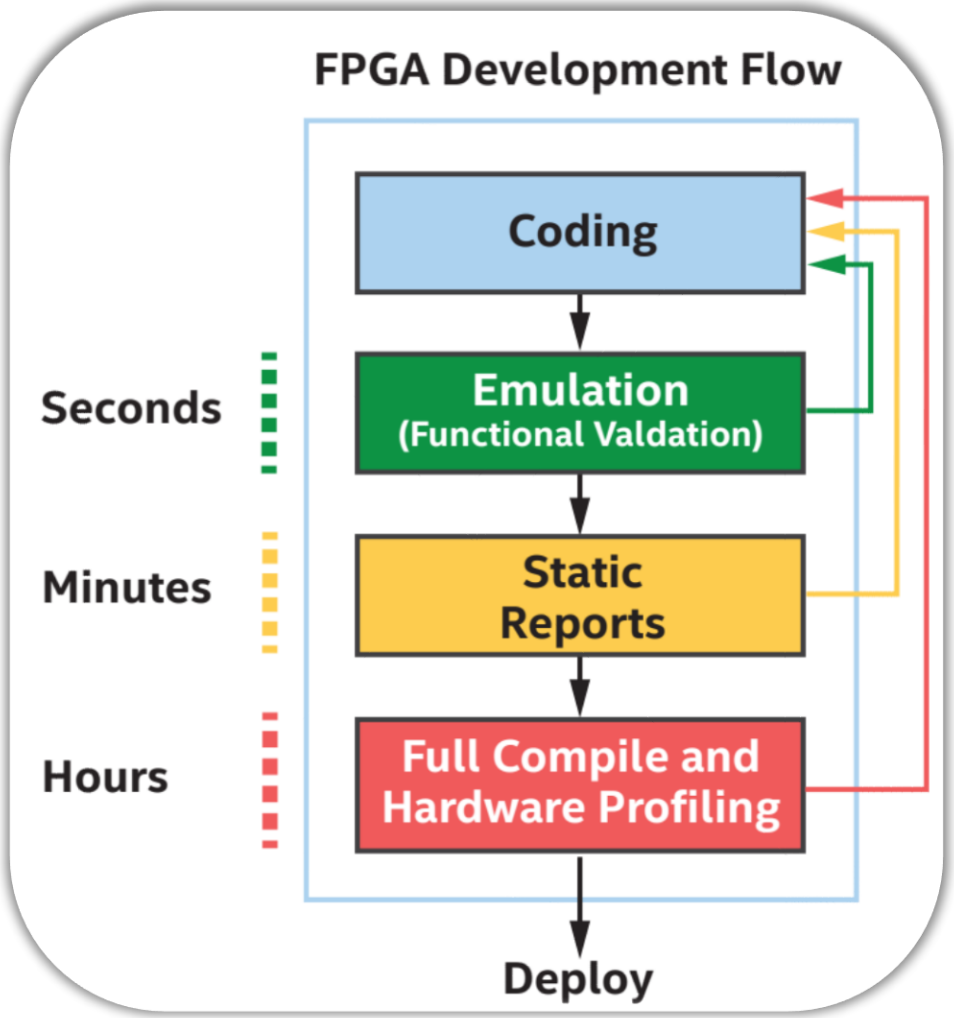
Sub-Topics:

- Introduction to oneAPI
- Introduction to DPC++
- What are FPGAs and Why Should I Care About Programming Them?
- Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

Copyright © 2021 Intel Corporation

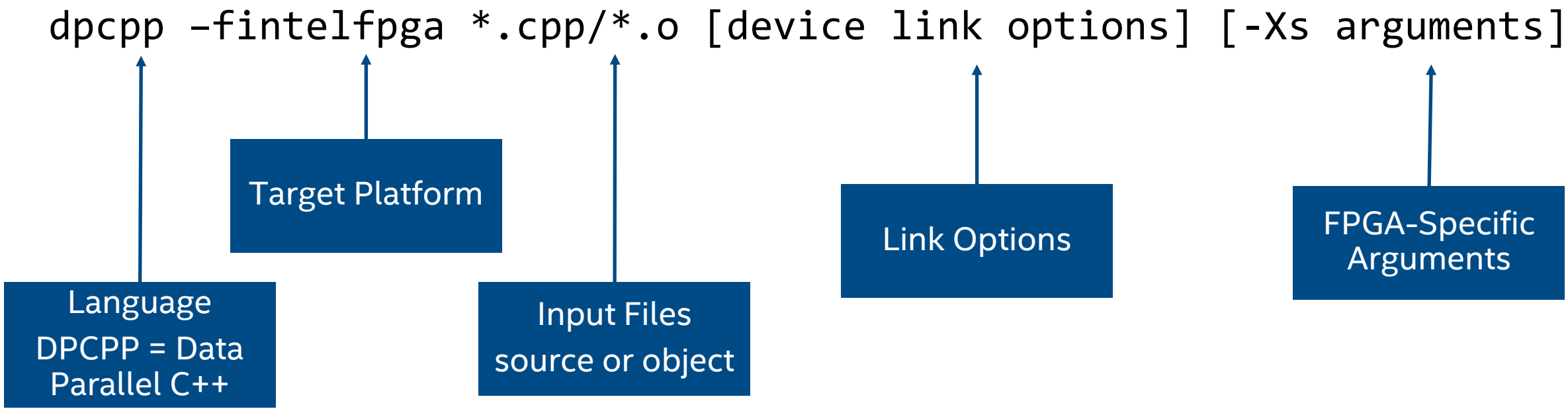
FPGA Development Flow for oneAPI Projects

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



Copyright © 2021 Intel Corporation

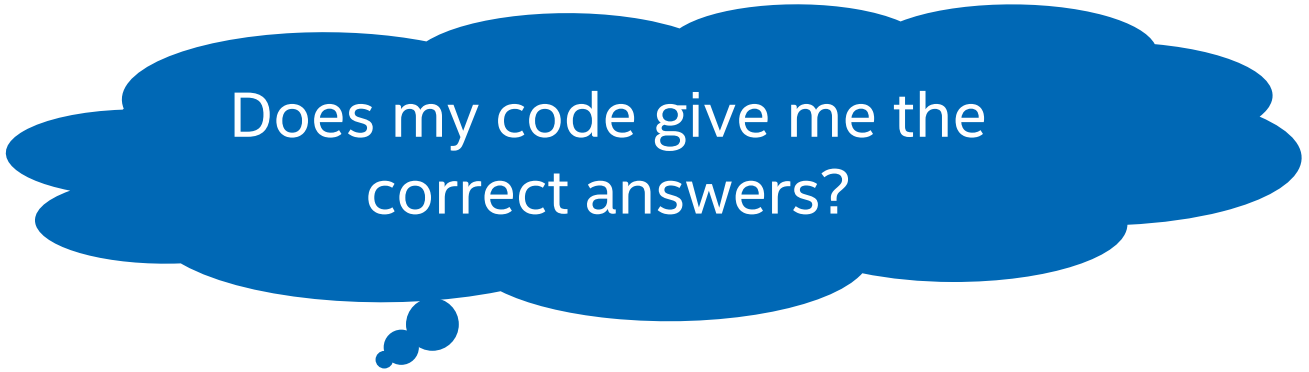
Anatomy of a dpcpp Command Targeting FPGAs



Copyright © 2021 Intel Corporation

Emulation

Seconds of Compilation



Quickly generate code that runs on the x86 host to emulate the FPGA

Developers can:

- Verify functionality of design through CPU compile and emulation.
- Identify quickly syntax and pointer implementation errors for iterative design/algorithm development.
- Enable deep, system-wide debug with Intel® Distribution for GDB.
- Functional debug of SYCL code with FPGA extensions.

Copyright © 2021 Intel Corporation

Emulation Command

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

Include this construct in your code

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

Copyright © 2021 Intel Corporation

Report Generation

Minutes of Compilation

Quickly generate a report to guide optimization efforts

Developers can:

- Identify any memory, performance, data-flow bottlenecks in their design.
- Receive suggestions for optimization techniques to resolve said bottlenecks.
- Get area and timing estimates of their designs for the desired FPGA.

Copyright © 2021 Intel Corporation

Command 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

Copyright © 2021 Intel Corporation

Bitstream Compilation

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

Developers can:

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

Copyright © 2021 Intel Corporation

Compile to FPGA Executable with Profiler

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
```

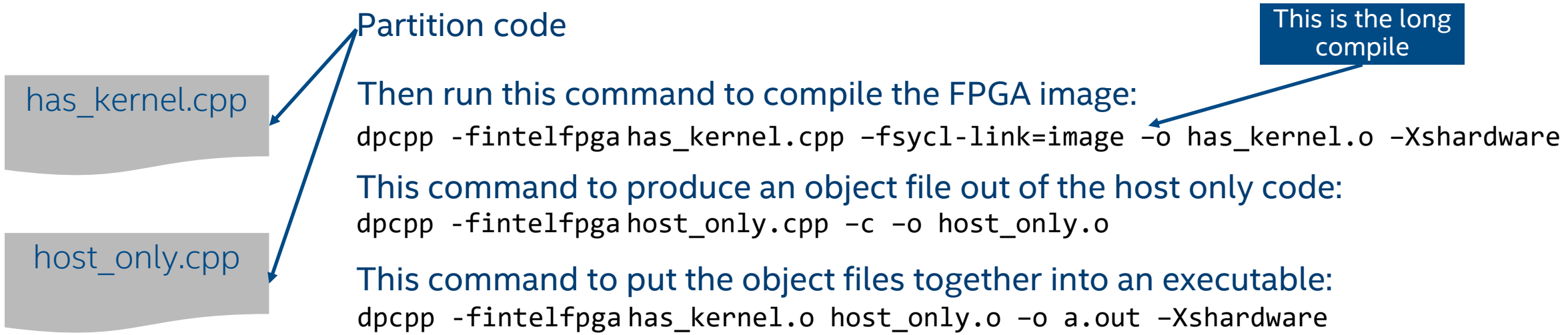
The profiler will be instrumented within the image and you will be able to run the executable to return information to import into Intel® Vtune Amplifier.

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

Copyright © 2021 Intel Corporation

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



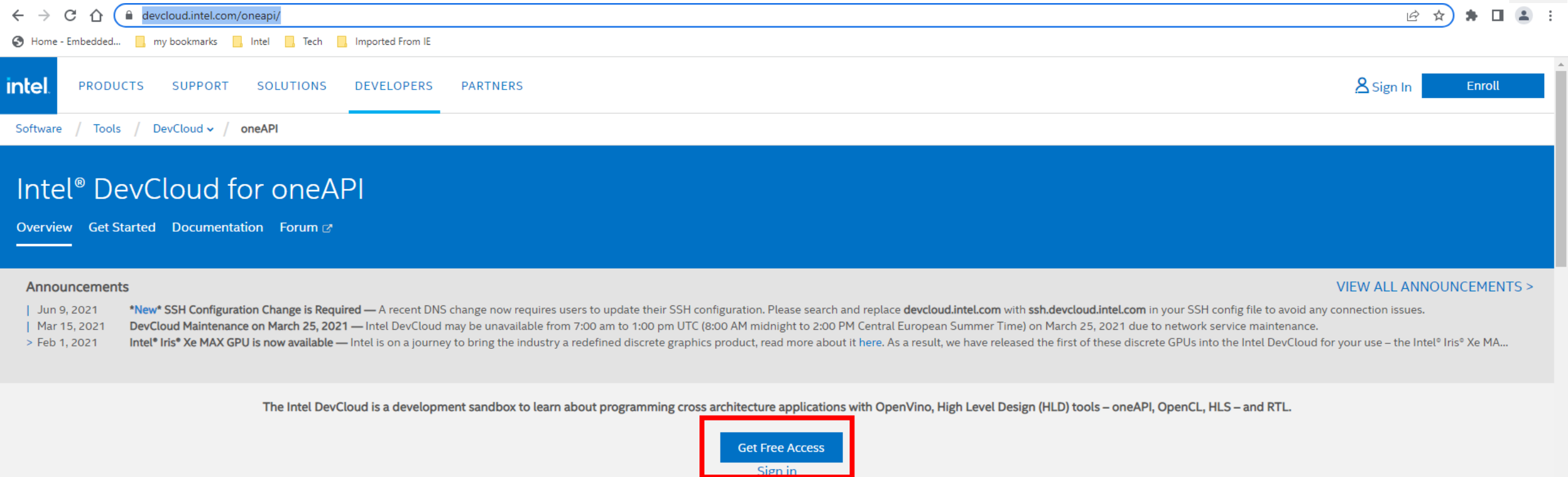
Copyright © 2021 Intel Corporation

Lab: Practice the FPGA Development Flow

Copyright © 2021 Intel Corporation

Lab instructions

- 1. Create a DevCloud account
  - Open this link: https://devcloud.intel.com/oneapi/
  - Click on the “Get Free Access” button



Lab instructions

- 1. Create a DevCloud account
  - Enter required information
  - Read and accept terms of use
  - Check your email for the verification link and click on it
  - Sign in
  - Click on “Working with oneAPI”
  - Provision your account, read and accept T&C for oneAPI

Copyright © 2021 Intel Corporation

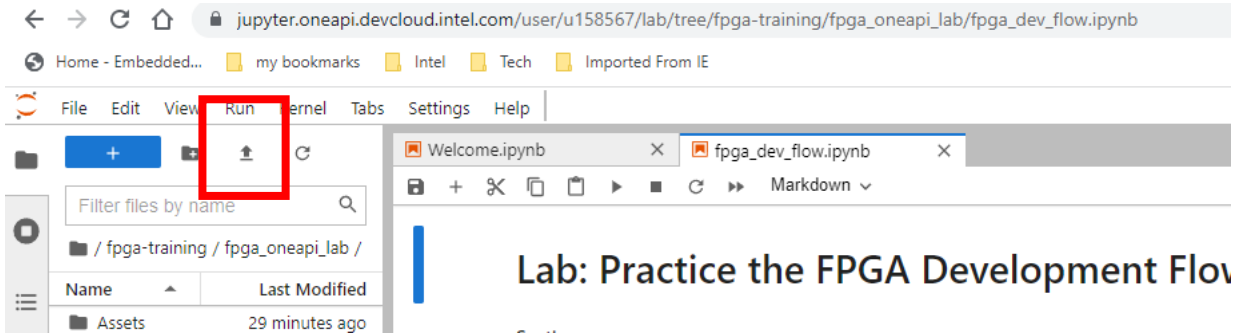
Lab instructions

- In a different browser page navigate to https://github.com/intel/fpga-training/tree/main/fpga_oneapi_lab
- Follow the instructions at the bottom of the page

Copyright © 2021 Intel Corporation

Lab instructions

- If the Jupyter notebook errors out:
  "dpcpp: command not found"
- Download the two provided files "bashrc" and "bash_profile" to your DevCloud home directory



Copyright © 2021 Intel Corporation

Lab instructions

- Rename the two files to .bashrc and .bash_profile (can be done in a terminal)
- Log out from the Jupyter server
- Log in again

Copyright © 2021 Intel Corporation

Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

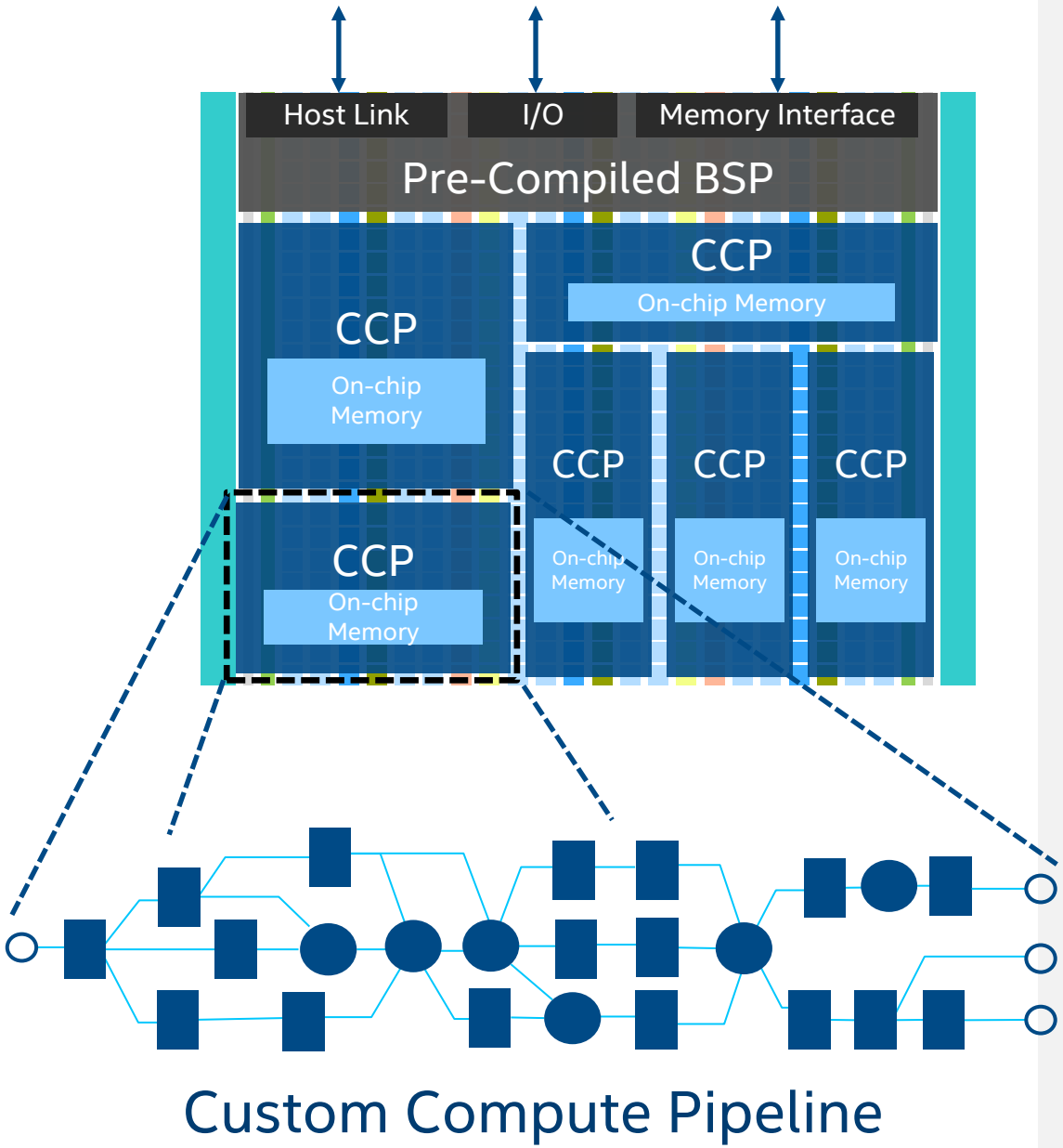
Sub-Topics:

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

Copyright © 2021 Intel Corporation

Intel® FPGAs

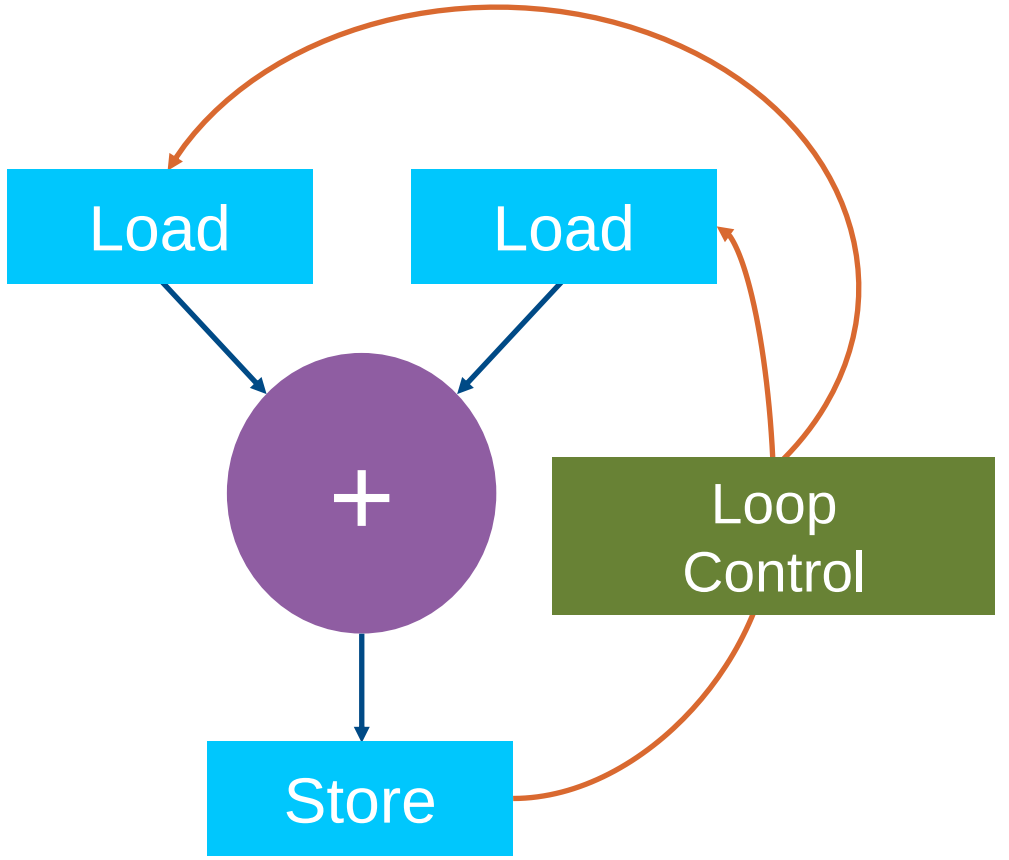
- Implementing Optimized **Custom Compute Pipelines (CCPs)** synthesized from compiled code



Copyright © 2021 Intel Corporation

How Is a Pipeline Built?

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



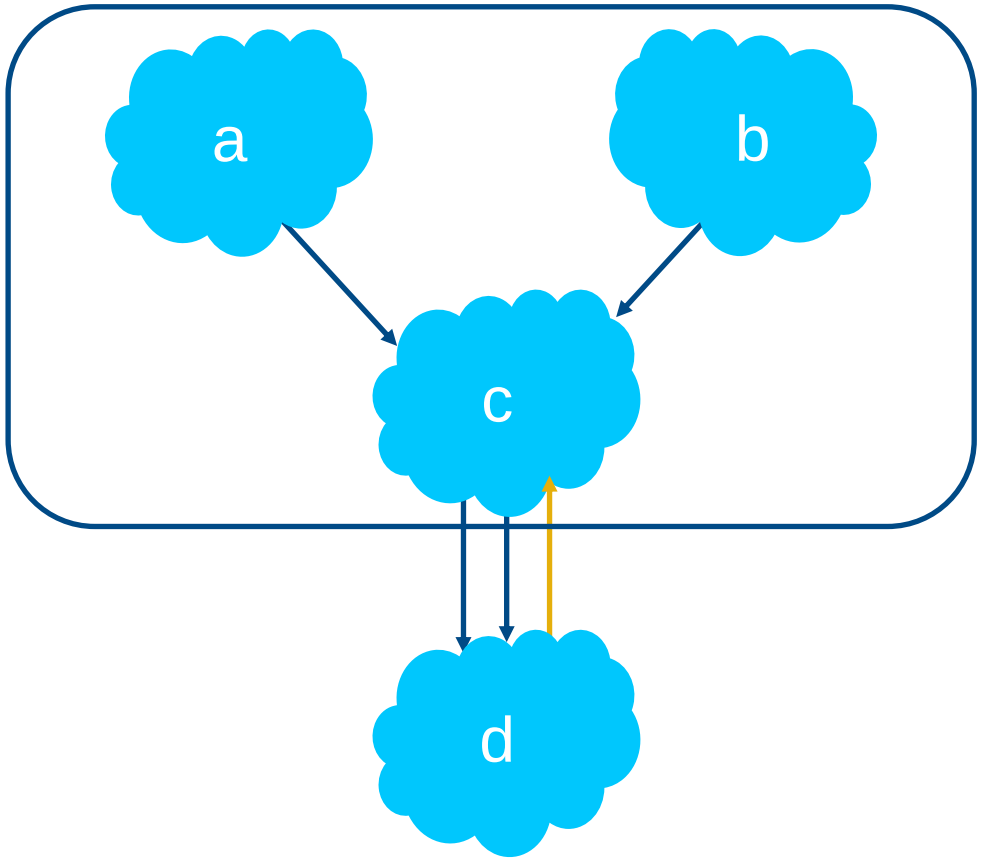

```
for (int i=0; i<LIMIT; i++) {
  c[i] = a[i] + b[i];
}
```

Data Path
Control Path

Copyright © 2021 Intel Corporation

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

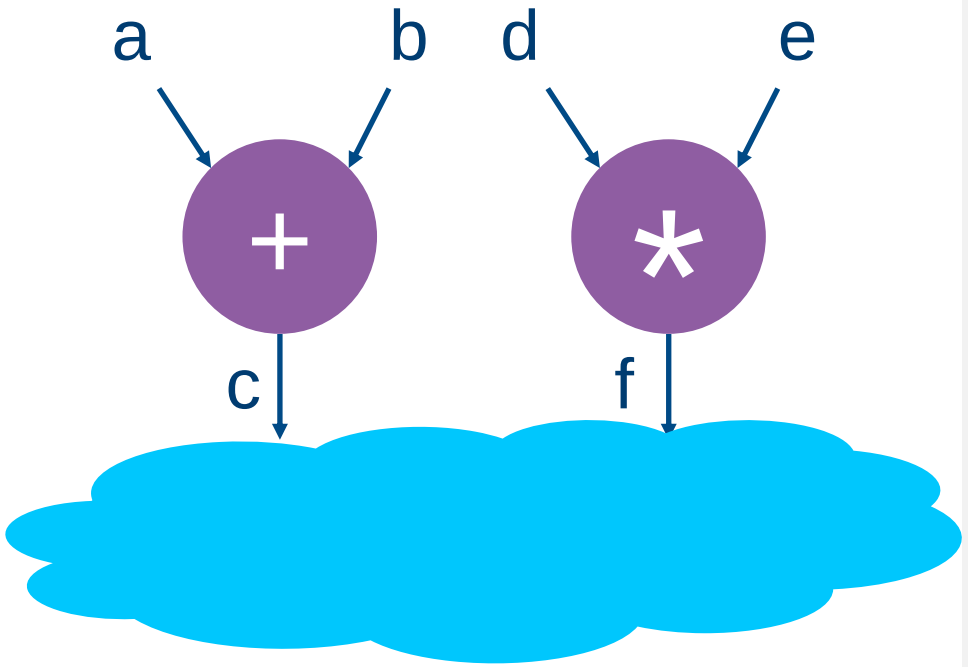


Copyright © 2021 Intel Corporation

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;
```

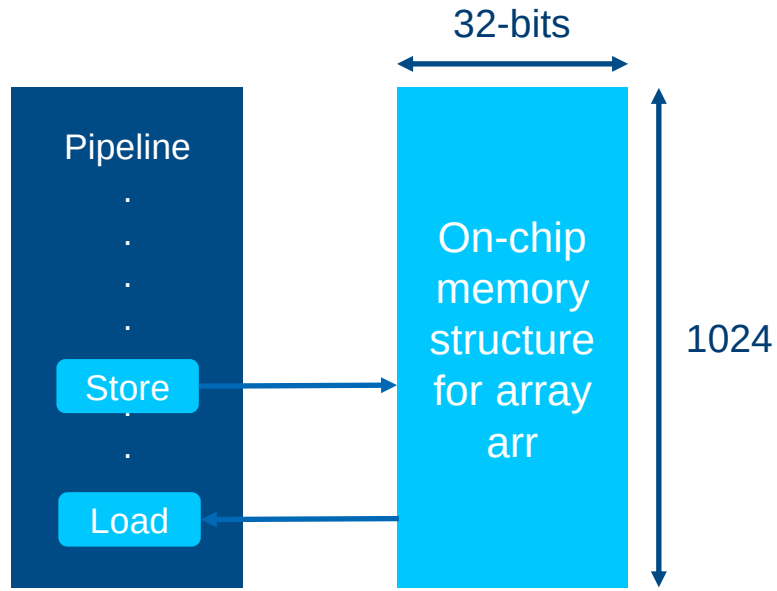


Copyright © 2021 Intel Corporation

On-Chip Memories Built for Kernel 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
```

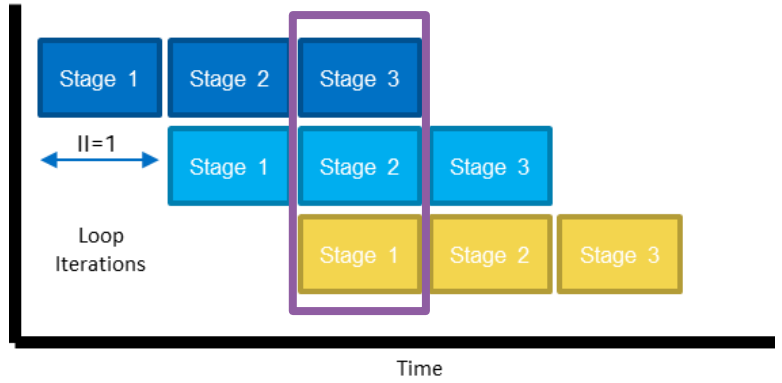


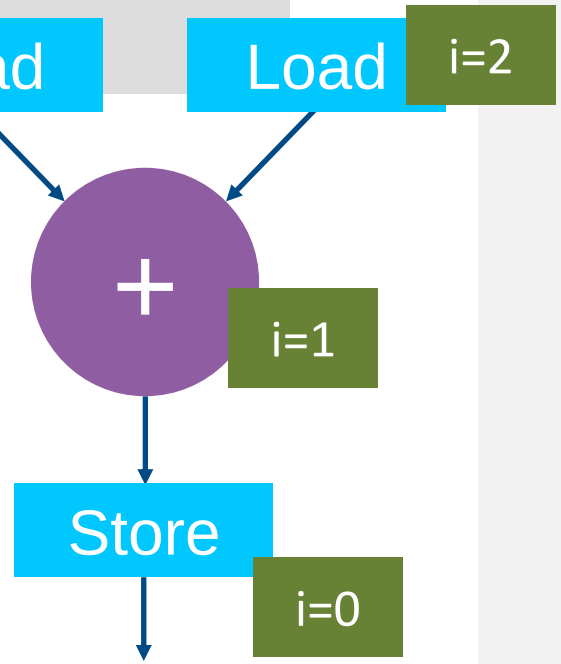
Copyright © 2021 Intel Corporation

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];
  }
});
```





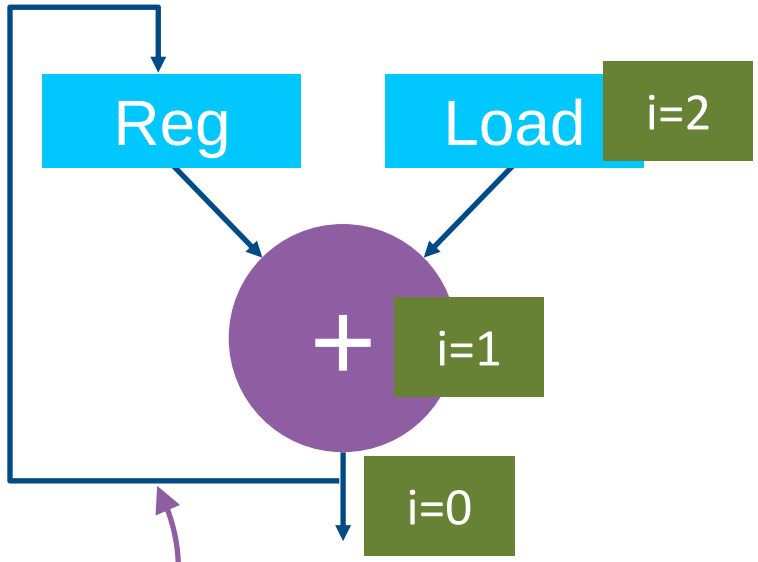
Copyright © 2021 Intel Corporation

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];
  }
});
```

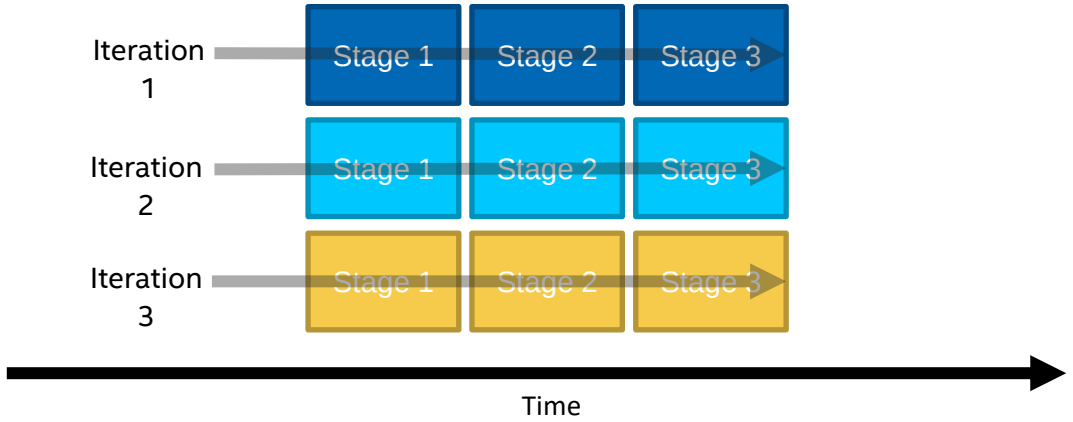


Copyright © 2021 Intel Corporation

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];
  }
});
```



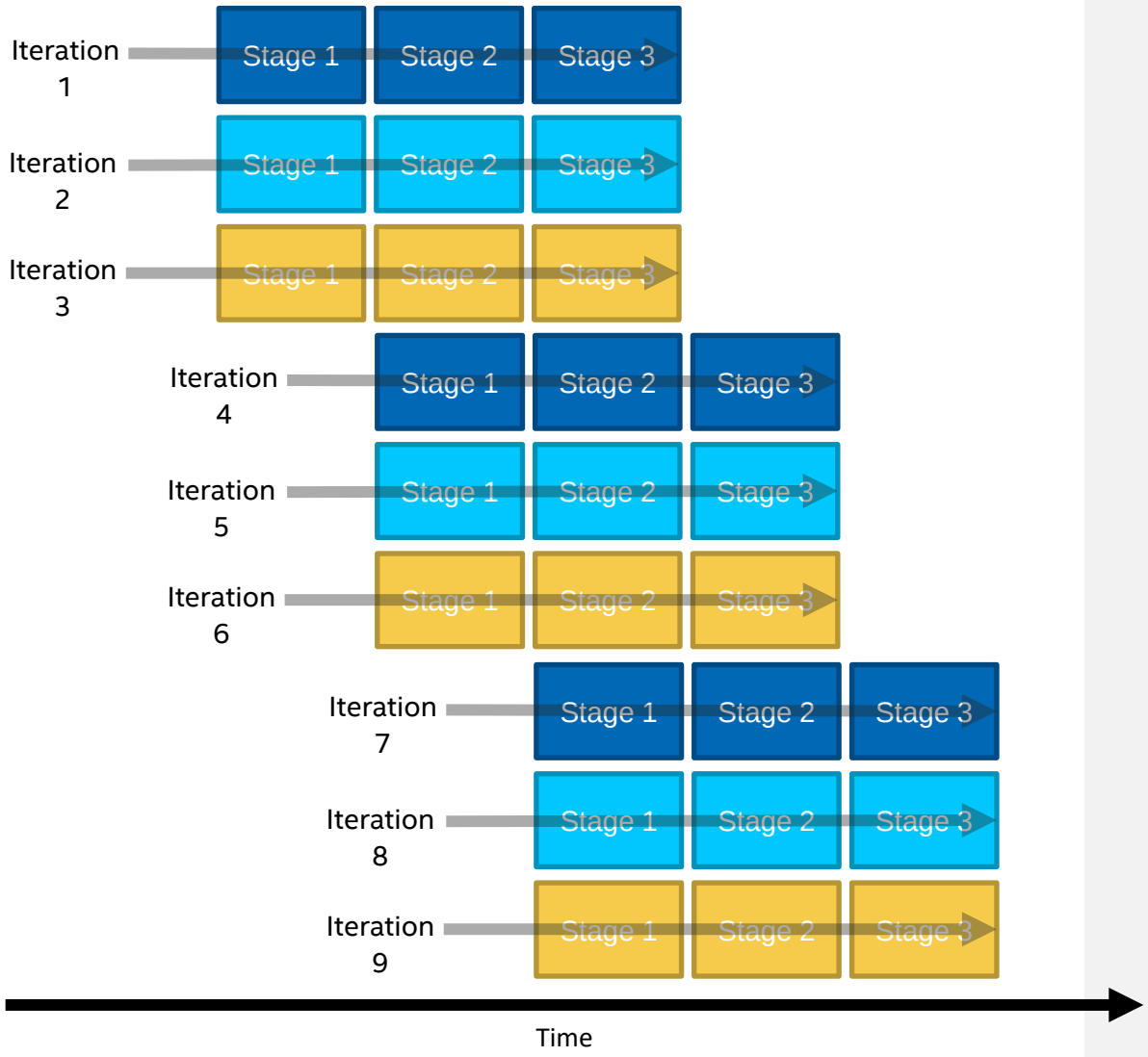
Copyright © 2021 Intel Corporation

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];
  }
});
```

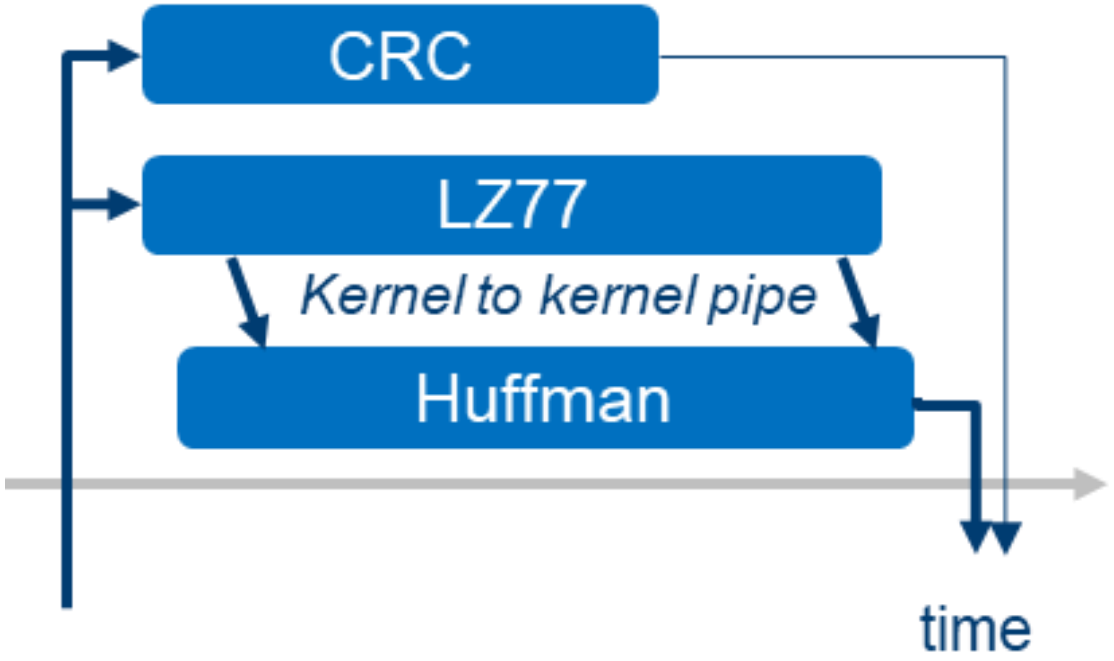


Copyright © 2021 Intel Corporation

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



Copyright © 2021 Intel Corporation

So, Can We Build These? Parallel Kernels

- Kernels launched using 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,
but, single_tasks
are recommended
for FPGAs.

Also, warning: the loop optimizations we talk about do not all apply for parallel kernels

Copyright © 2021 Intel Corporation

Section:
Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Sub-Topics:

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

Copyright © 2021 Intel Corporation

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)
- Contain an outer loop

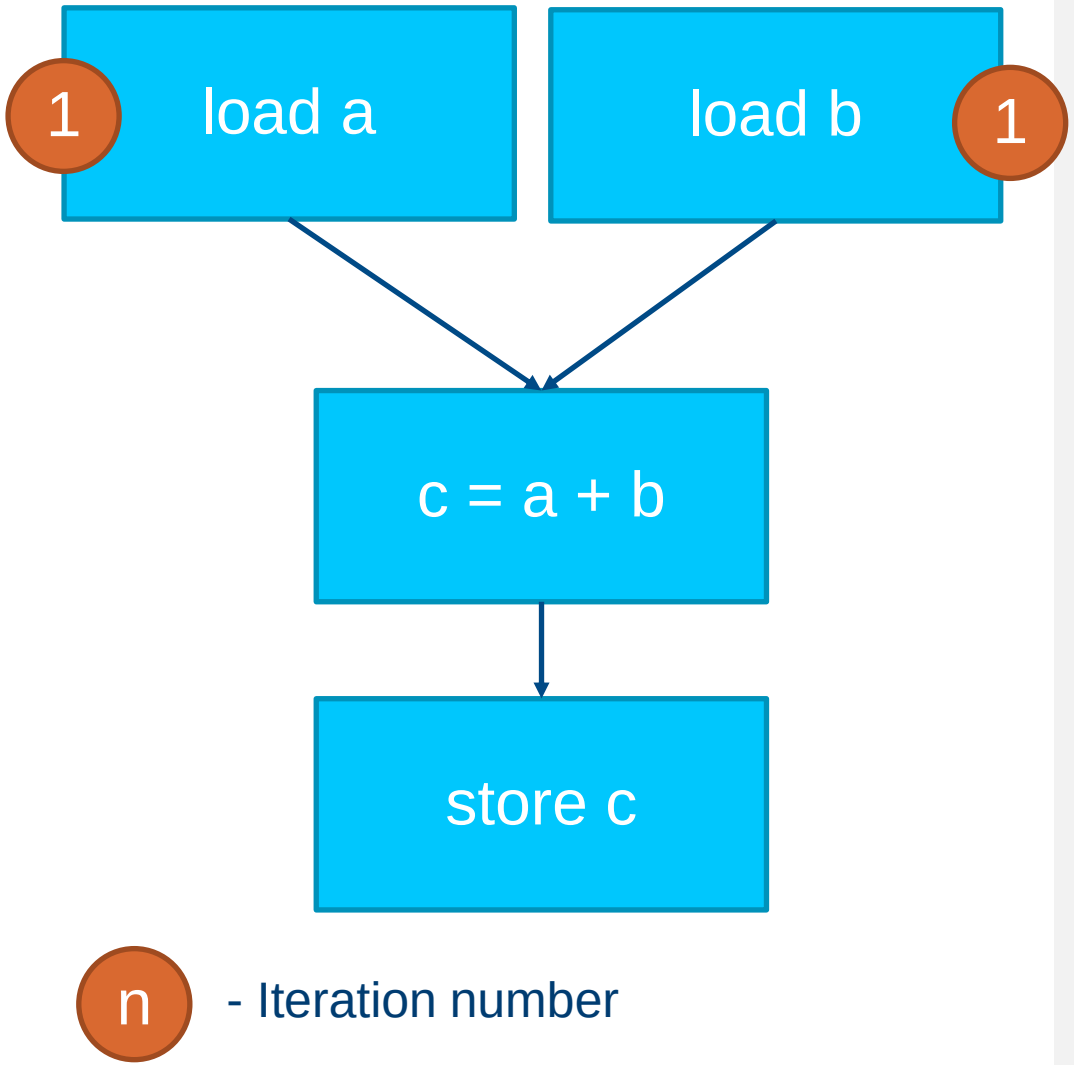
```
…//application scope

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
```

Copyright © 2021 Intel Corporation

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];
    }
  });
…
```

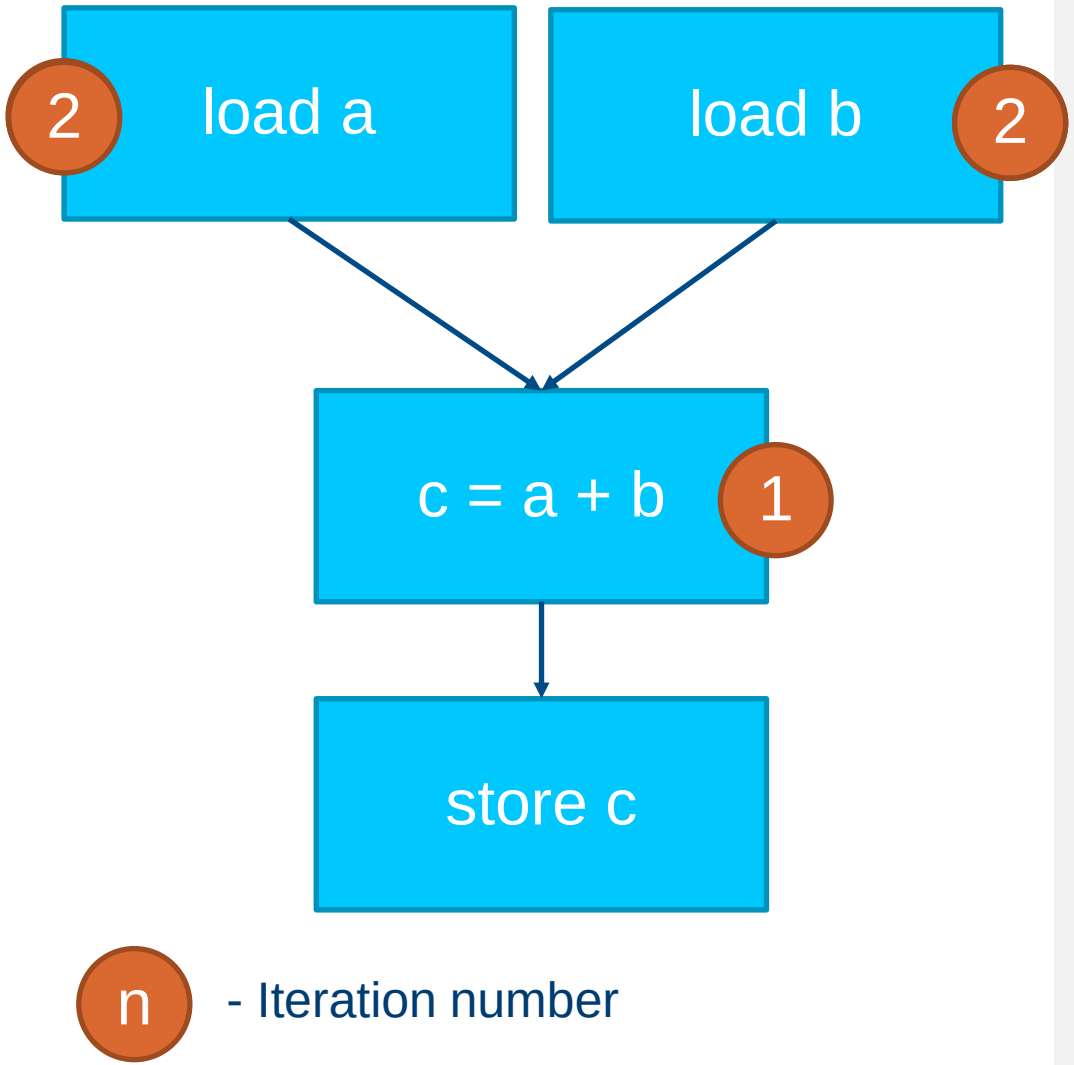


Copyright © 2021 Intel Corporation

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];
    }
  });
…
```

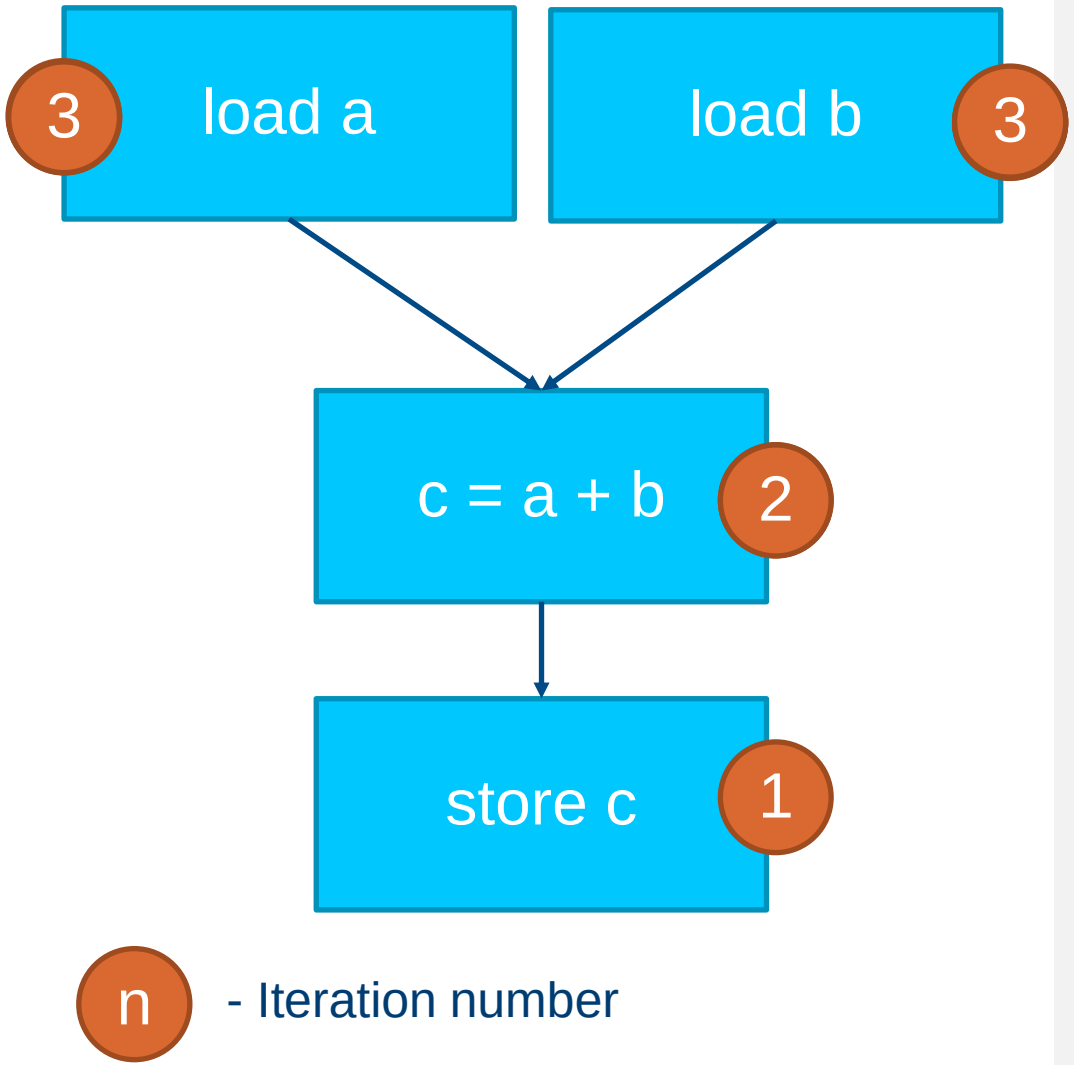


Copyright © 2021 Intel Corporation

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];
    }
  });
…
```

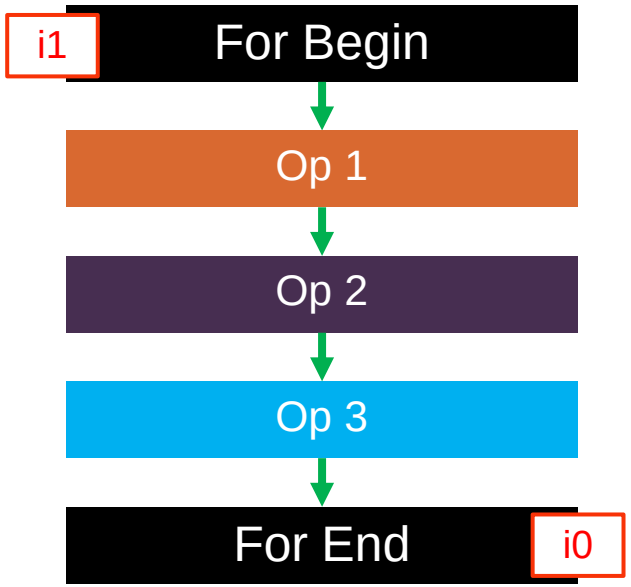


Copyright © 2021 Intel Corporation

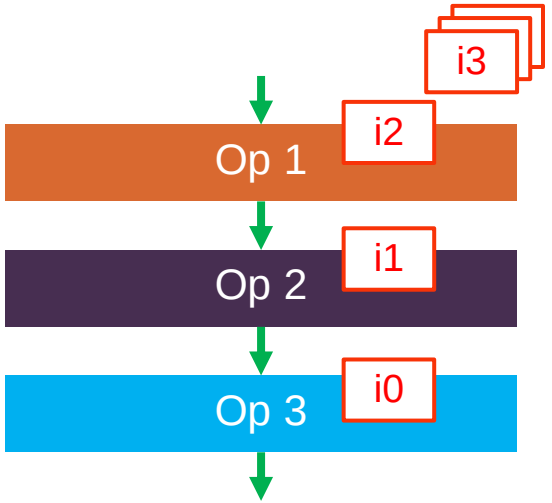
Loop Pipelining vs Serial Execution

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

Worst Case



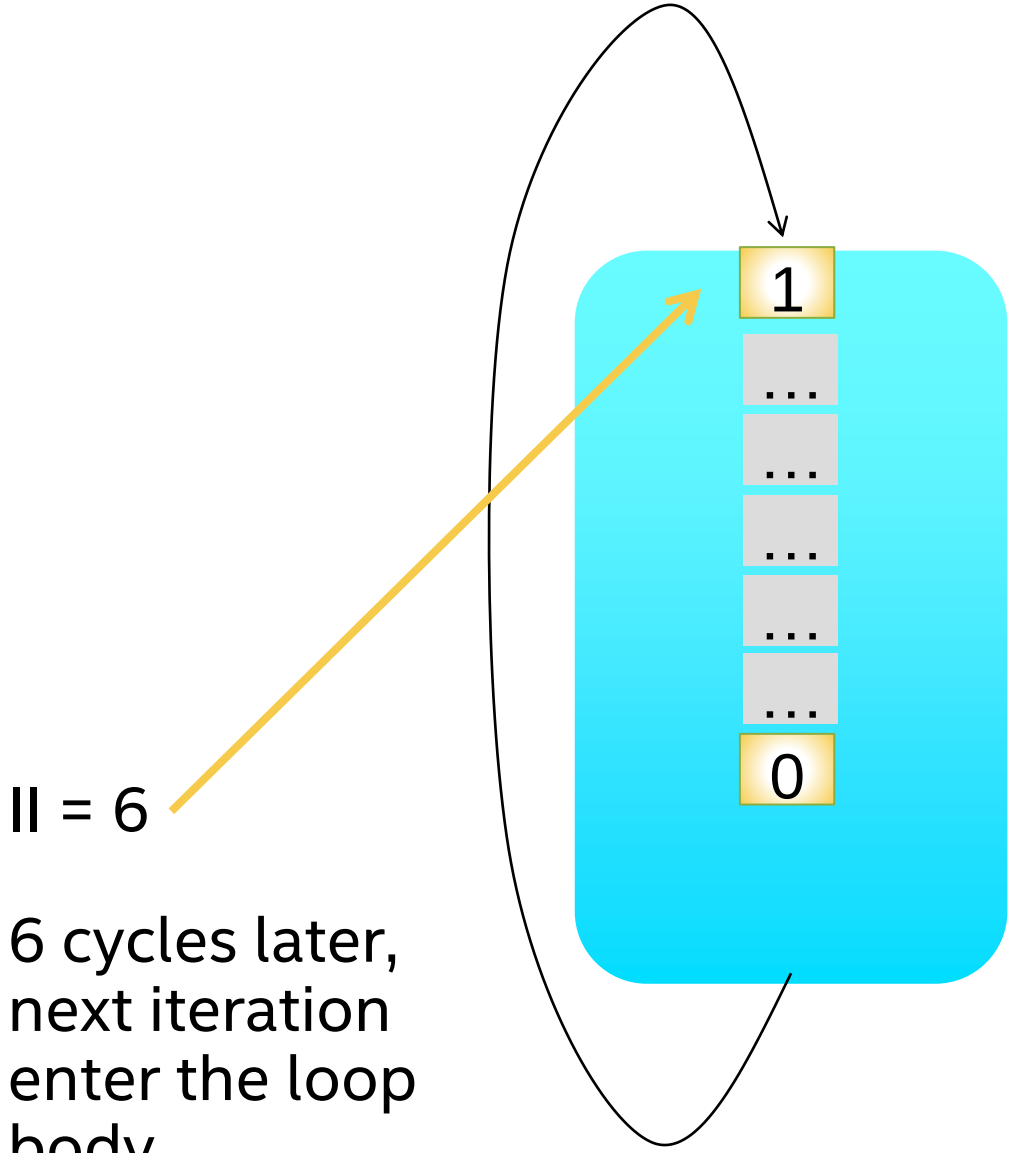
Best Case



Copyright © 2021 Intel Corporation

In-Between Scenario

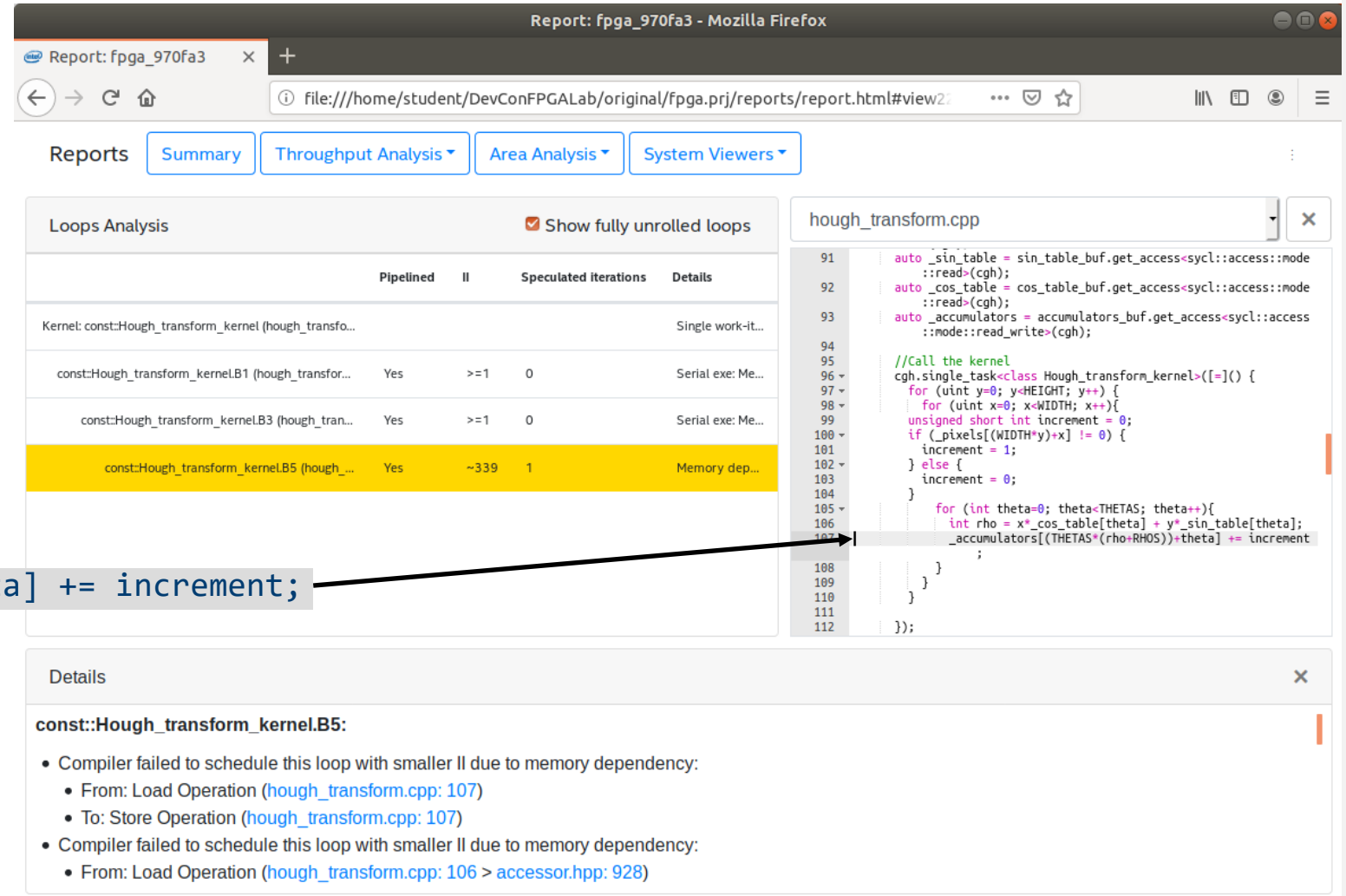
- Sometimes you must wait more than one clock cycle to input more data
- Because dependencies can't resolve fast enough
- How long you have to wait is called **Initiation Interval** or **II**
- Total number of cycles to run kernel is about (loop iterations)*II
  - (neglects initial latency)
- Minimizing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



Copyright © 2021 Intel Corporation

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

Copyright © 2021 Intel Corporation

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

Copyright © 2021 Intel Corporation

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

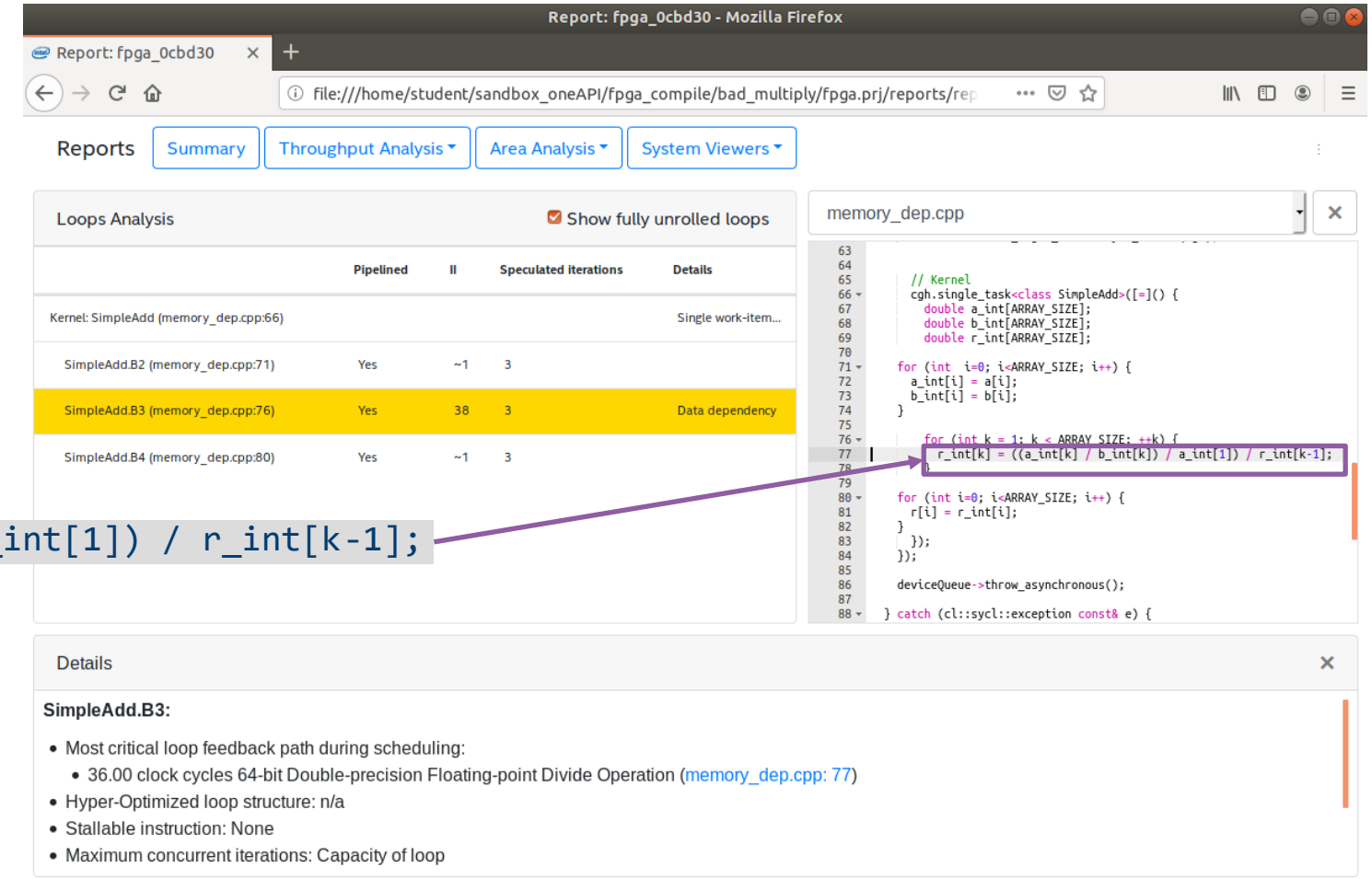
Copyright © 2021 Intel Corporation

Why Else Could This Happen?

- Data Dependency
  - Kernel cannot complete a calculation fast enough

```
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



Copyright © 2021 Intel Corporation

What Can You Do?

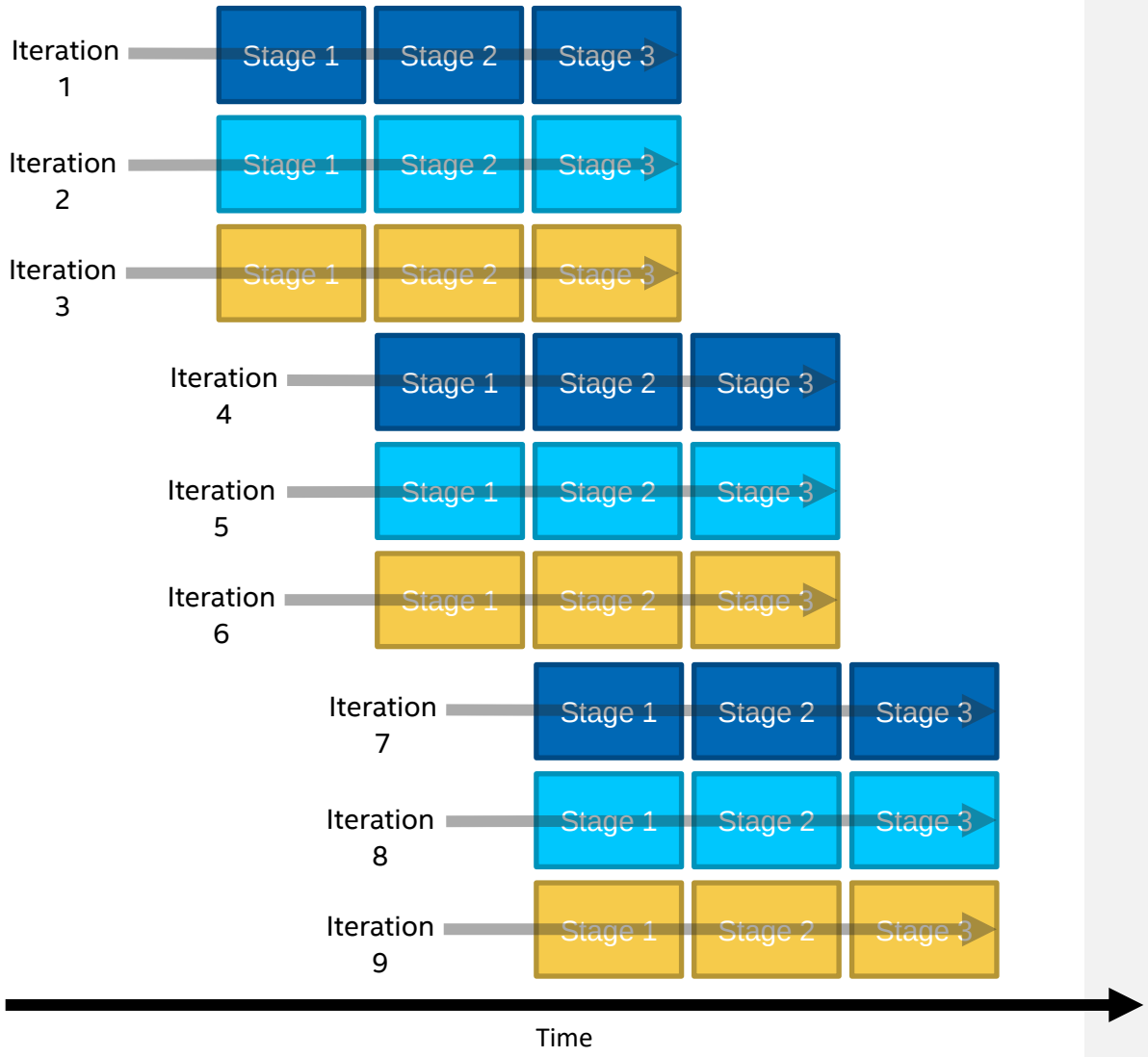
- Do a simpler calculation
- Pre-calculate some of the operations on the host
- Use a simpler type
- Use floating point optimizations (discussed later)
- Advanced technique: Increase time (pipeline stages) between start of calculation and when you use answer
  - See the “Relax Loop-Carried Dependency”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

Copyright © 2021 Intel Corporation

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];
  }
});
```



Copyright © 2021 Intel Corporation

Maximum Clock Frequency (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 get a lower II, sacrificing a higher Fmax

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

Copyright © 2021 Intel Corporation

How Can You Tell This Is a Problem?

- Optimization 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;
});
```

	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

Copyright © 2021 Intel Corporation

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)]]
```

Copyright © 2021 Intel Corporation

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)]]
```

Copyright © 2021 Intel Corporation

Section:
Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

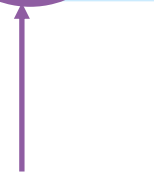
Sub-Topics:

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

Copyright © 2021 Intel Corporation

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
	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

Copyright © 2021 Intel Corporation

Global Memory Access is Slow – What to Do?

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

```
…
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

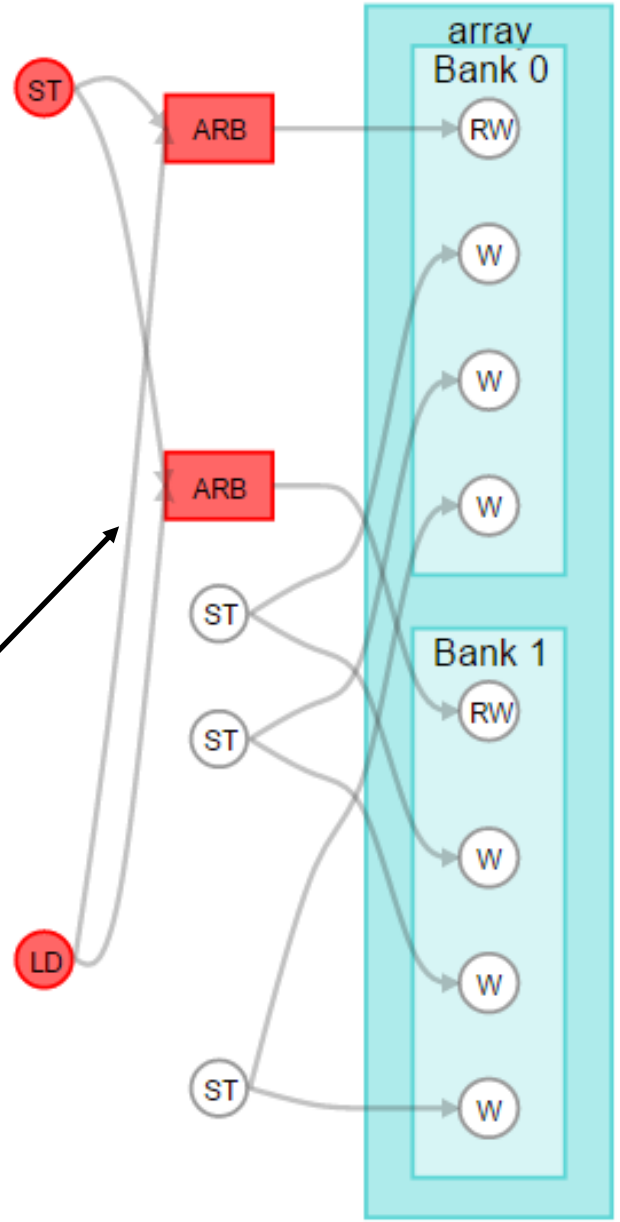
```
…
constexpr int N = 128;
queue.submit([&](handler &cgh) {
  auto A =
    A_buf.get_access<access::mode::read_write>(cgh);

  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

Copyright © 2021 Intel Corporation

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



Copyright © 2021 Intel Corporation

Local Memory Bottlenecks

Natively, the memory architecture has 2 ports

The compiler uses optimizations to minimize arbitration

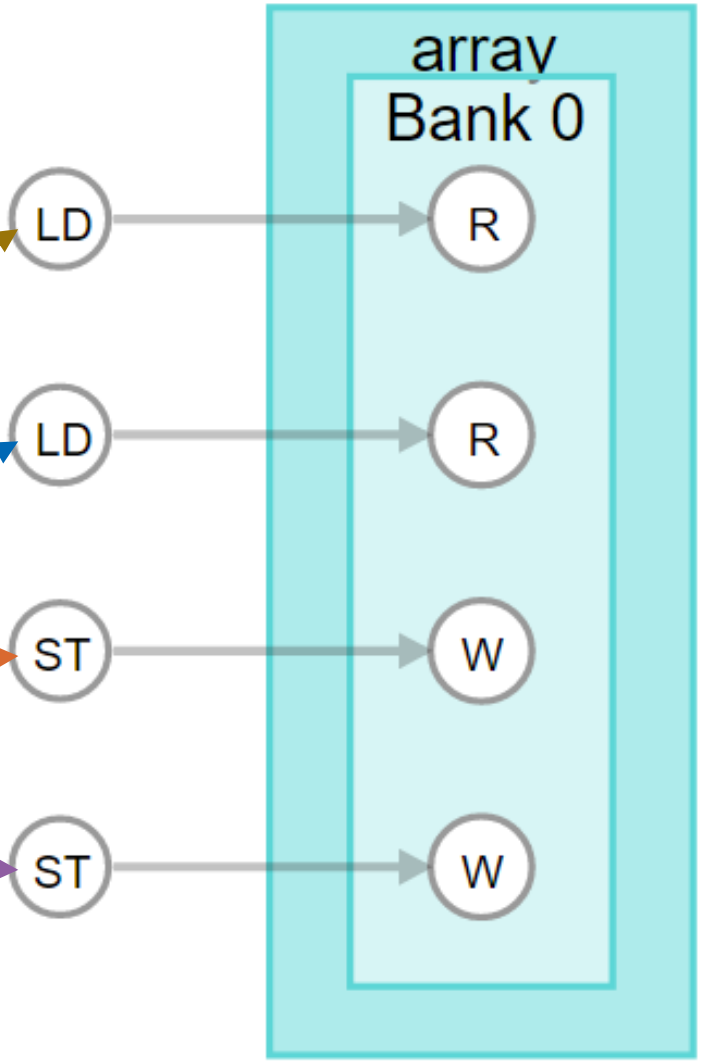
Your job is to write code the compiler can optimize

Copyright © 2021 Intel Corporation

Double-Pumped Memory Example

- Increase the clock rate to 2x
- Compiler can automatically implement double-pumped memory

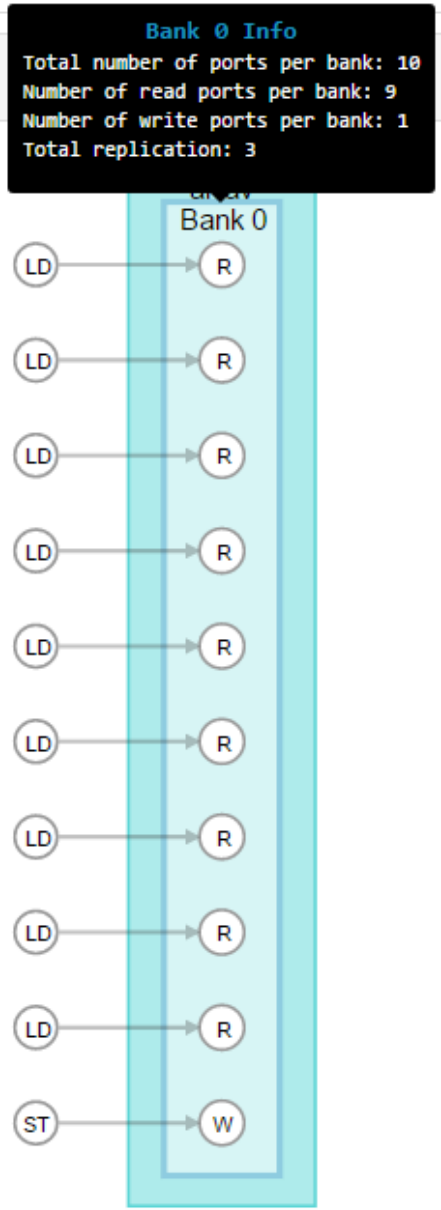
```
//kernel scope
...
  int array[1024];

  array[ind1] = val;

  array[ind1+1] = val;

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



Copyright © 2021 Intel Corporation

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;
…
```



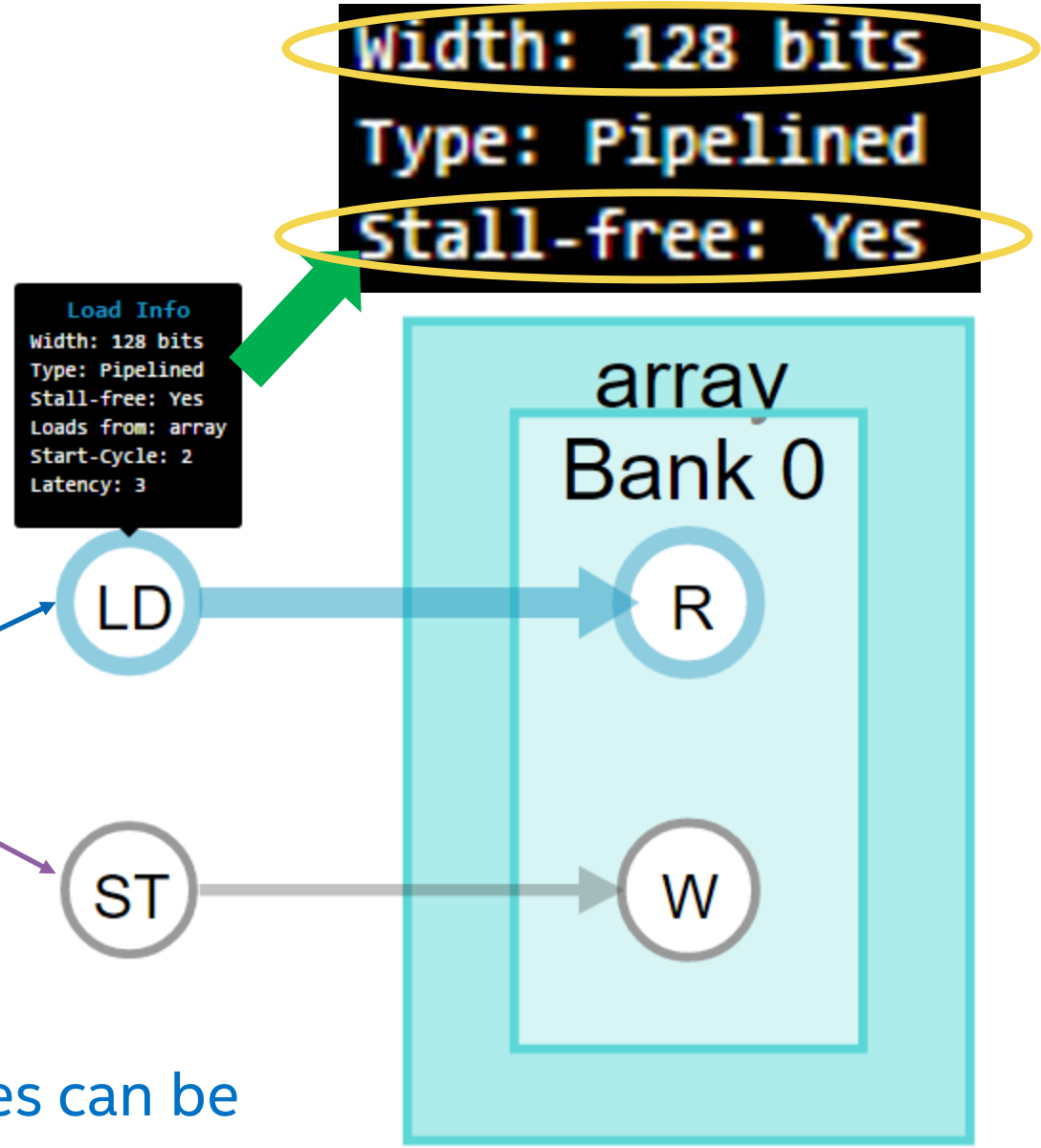
Turn 4 ports of double-pumped memory to unlimited ports

Drawbacks: logic resources, stores must go to each replication

Copyright © 2021 Intel Corporation

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;
…
```



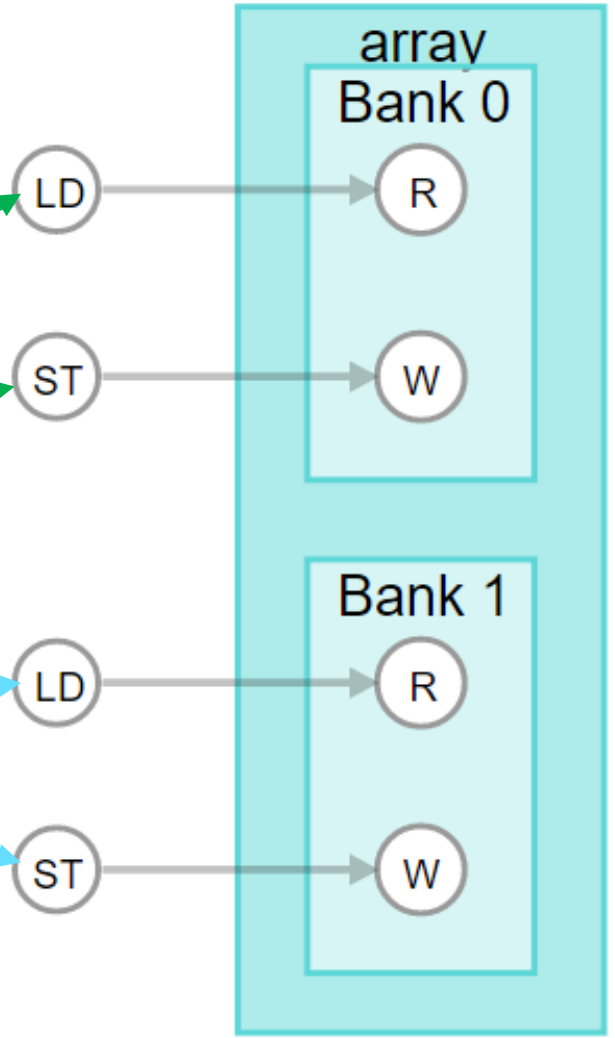
Continuous addresses can be coalesced into wider accesses

Copyright © 2021 Intel Corporation

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]);
…
```



Copyright © 2021 Intel Corporation

Attributes for Local Memory Optimization

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

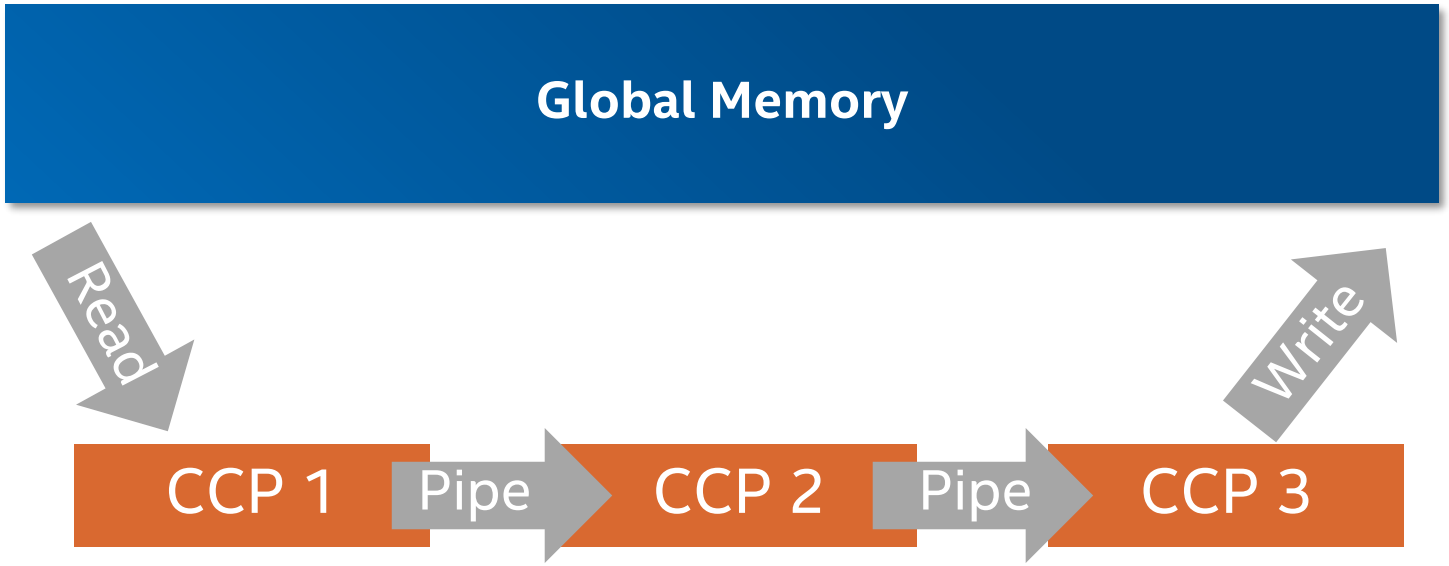
Attribute	Usage
numbanks	[[intelfpga::numbanks(N)]]
bankwidth	[[intelfpga::bankwidth(N)]]
singlepump	[[intelfpga::singlepump]]
doublepump	[[intelfpga::doublepump]]
max_replicates	[[intelfpga::max_replicates(N)]]
simple_dual_port	[[intelfpga::simple_dual_port]]

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

Copyright © 2021 Intel Corporation

Pipes – Element the Need for Some Memory

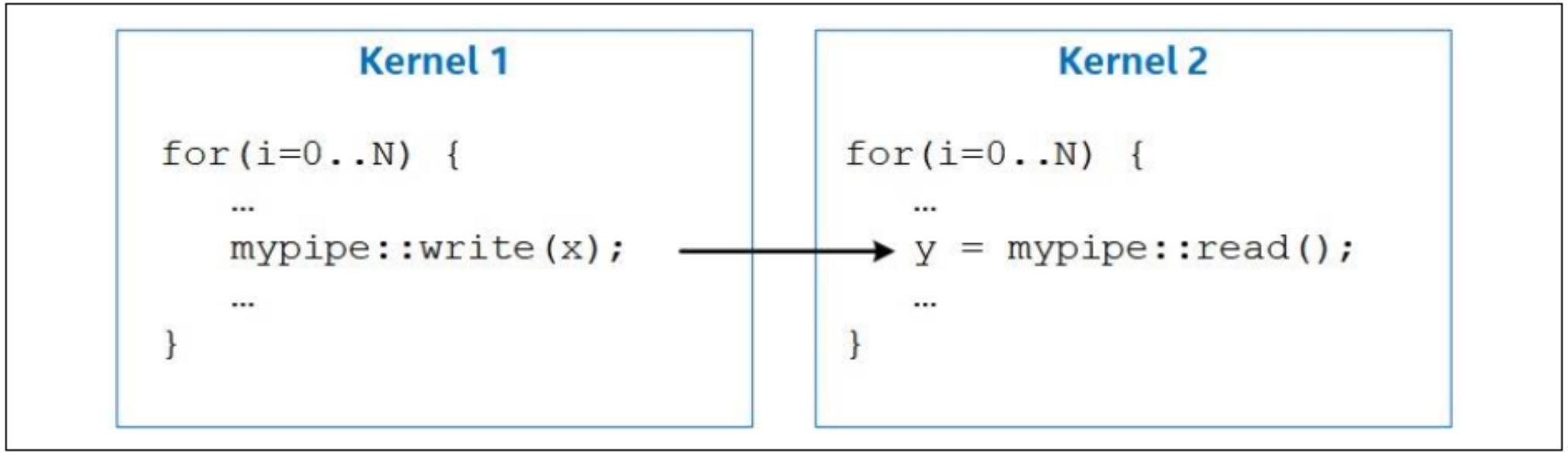
Create custom direct point-to-point communication between CCPs with Pipes



Copyright © 2021 Intel Corporation

Task Parallelism By Using Pipes

- Launch separate kernels simultaneously
- Achieve synchronization and data sharing using pipes
- Make better use of your hardware



Copyright © 2021 Intel Corporation

Section:
Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Sub-Topics:

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

Copyright © 2021 Intel Corporation

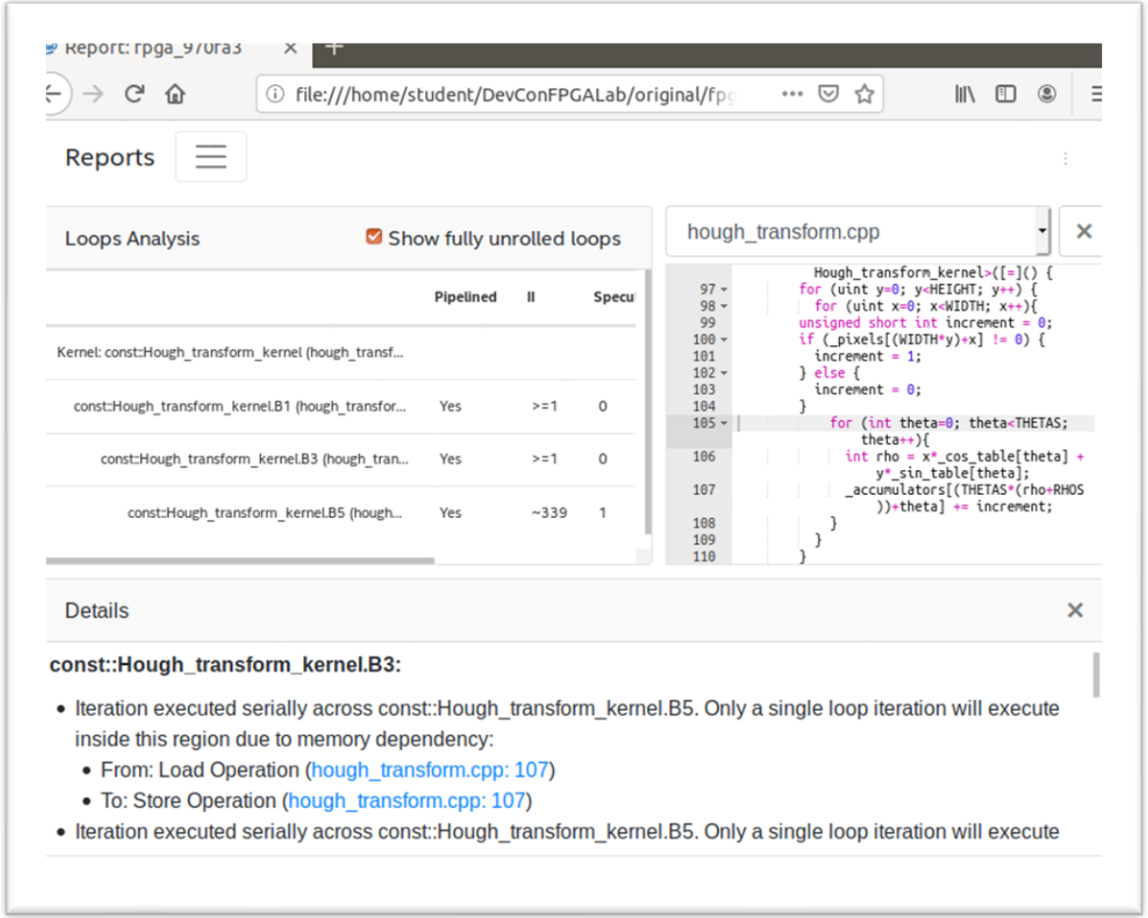
HTML Optimization Report

- 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

Copyright © 2021 Intel Corporation

Optimization Report – Throughput Analysis

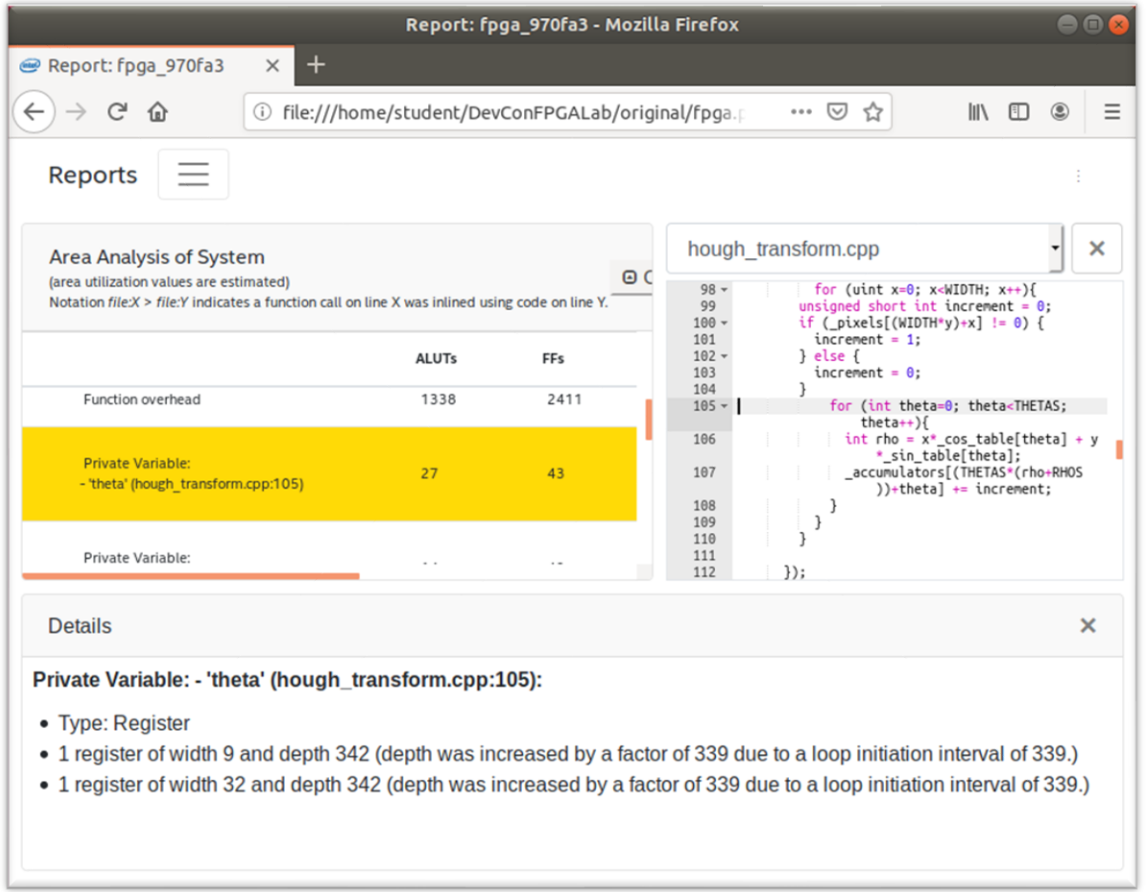
- Loops Analysis and Fmax II sections
- Actionable feedback on pipeline status of loops
- Show estimated Fmax of each loop



Copyright © 2021 Intel Corporation

Optimization Report – Area Analysis

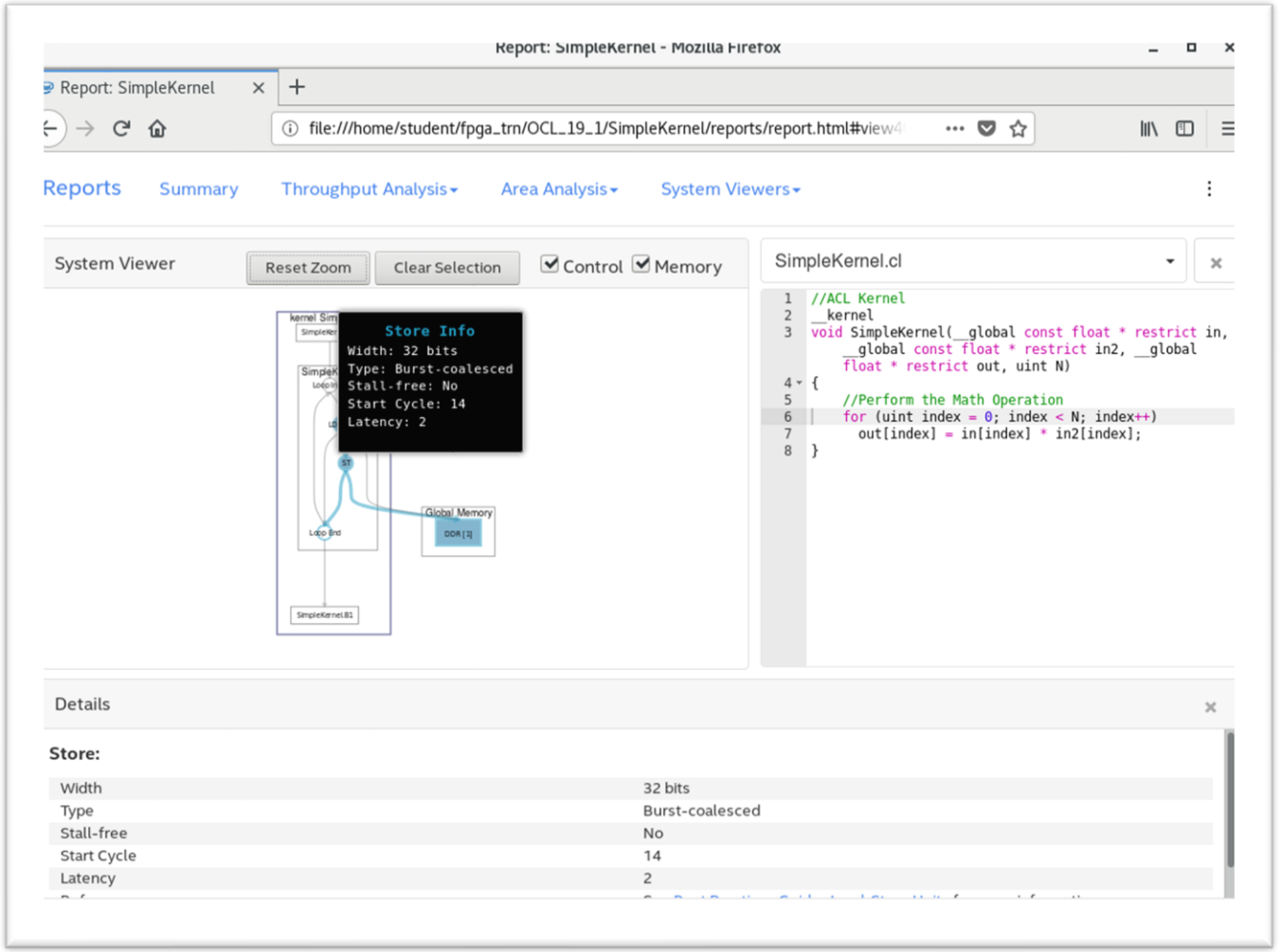
- Generate detailed estimated area utilization report of kernel scope code
  - Detailed breakdown of resources by system blocks
  - Provides architectural details of HW
    - Suggestions to resolve inefficiencies



Copyright © 2021 Intel Corporation

Optimization Report – Graph Viewer

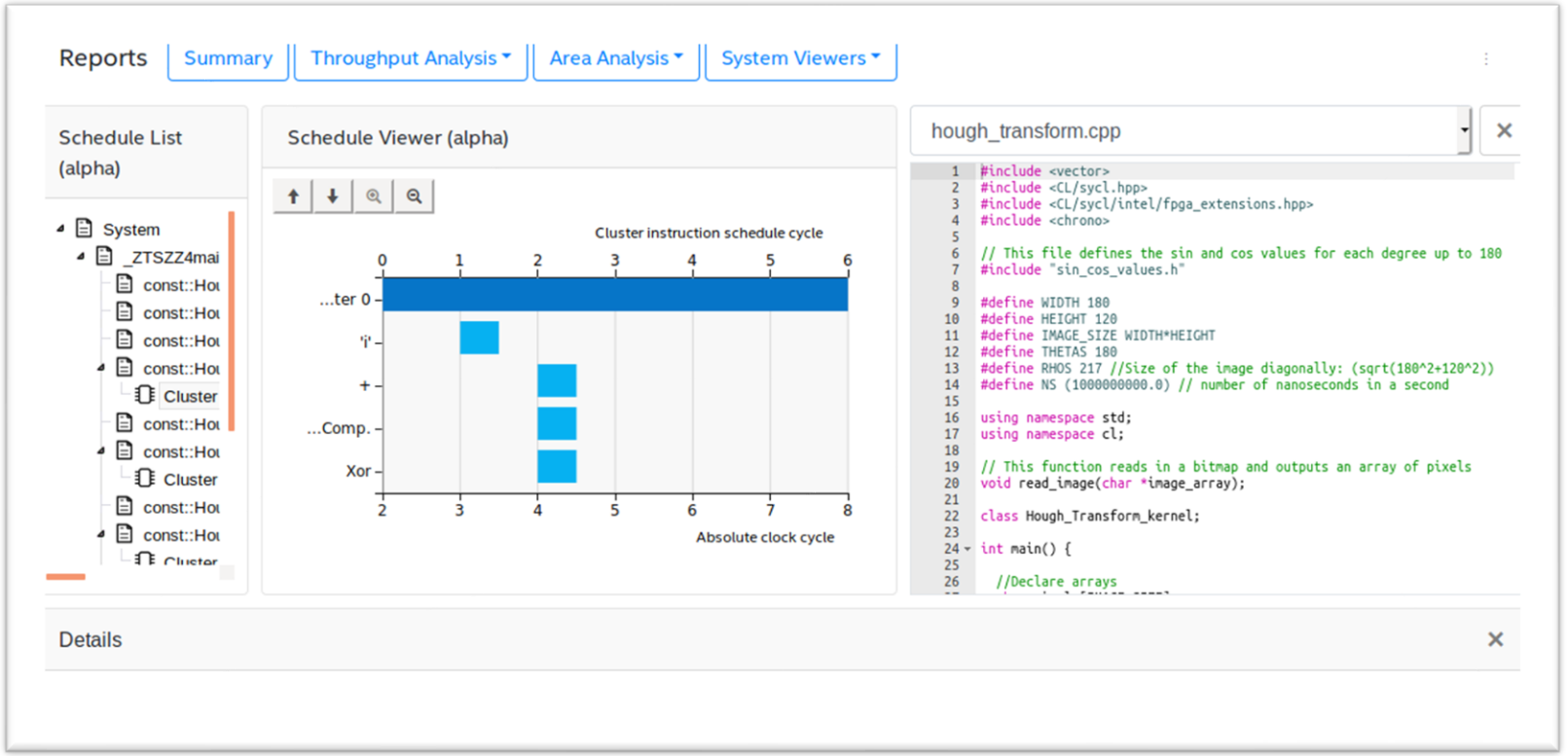
- The system view of the Graph Viewer shows following types of connections
  - Control
  - Memory, if your design has global or local memory
  - Pipes, if your design uses pipes



Copyright © 2021 Intel Corporation

Optimization Report – Schedule Viewer

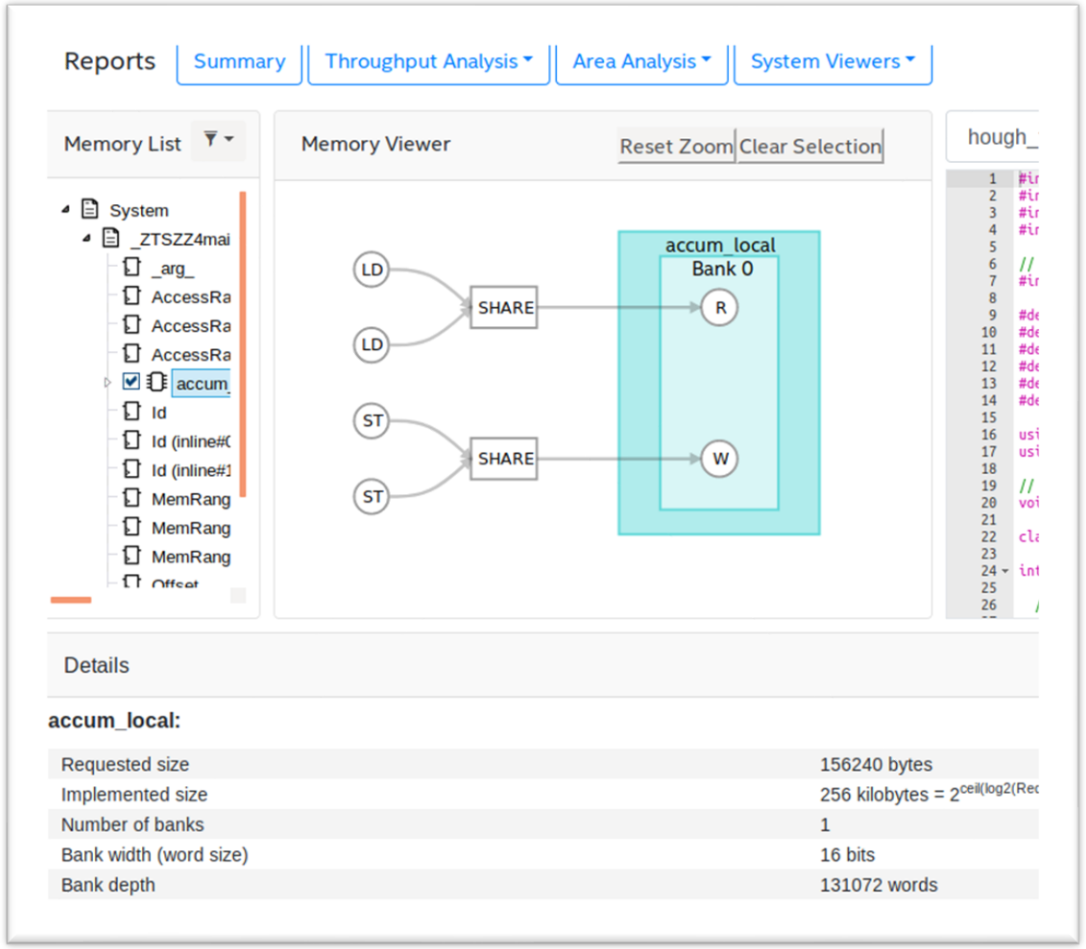
Schedule in clock cycles for different blocks in your code



Copyright © 2021 Intel Corporation

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



Copyright © 2021 Intel Corporation

Section:
Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Sub-Topics:

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

Copyright © 2021 Intel Corporation

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

Copyright © 2021 Intel Corporation

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)

Copyright © 2021 Intel Corporation

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

Copyright © 2021 Intel Corporation

Floating-Point Optimizations

- Applies 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

Copyright © 2021 Intel Corporation

Tree-Balancing

- Floating-point operations are not associative
  - Rounding after each operation affects the outcome
  - i.e. ((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**

Copyright © 2021 Intel Corporation

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`

Copyright © 2021 Intel Corporation

References and Resources

Copyright © 2021 Intel Corporation

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

Copyright © 2021 Intel Corporation

Lab: Optimizing the Hough Transform Kernel

Copyright © 2021 Intel Corporation

Lab instructions

- Download to DevCloud the provided event_labs.zip file
- Open a terminal in your Jupyter server
- Unzip the file
- In the Jupyter server, navigate to labs/lab3
- Open Hough_transform_lab.pdf and follow the instructions

Copyright © 2021 Intel Corporation

Legal Disclaimers/Acknowledgements

- 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.
- Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.
- Your costs and results may vary.
- Intel technologies may require enabled hardware, software or service activation
- No product or component can be absolutely secure
- Your costs and results may vary
- Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others
- OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos
- *Other names and brands may be claimed as the property of others

Copyright © 2021 Intel Corporation

Copyright © 2021 Intel Corporation.

This document is intended for personal use only.

Unauthorized distribution, modification, public performance, public display, or copying of this material via any medium is strictly prohibited.