

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

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

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



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

A Unified Programming Model

Multiple Architectures

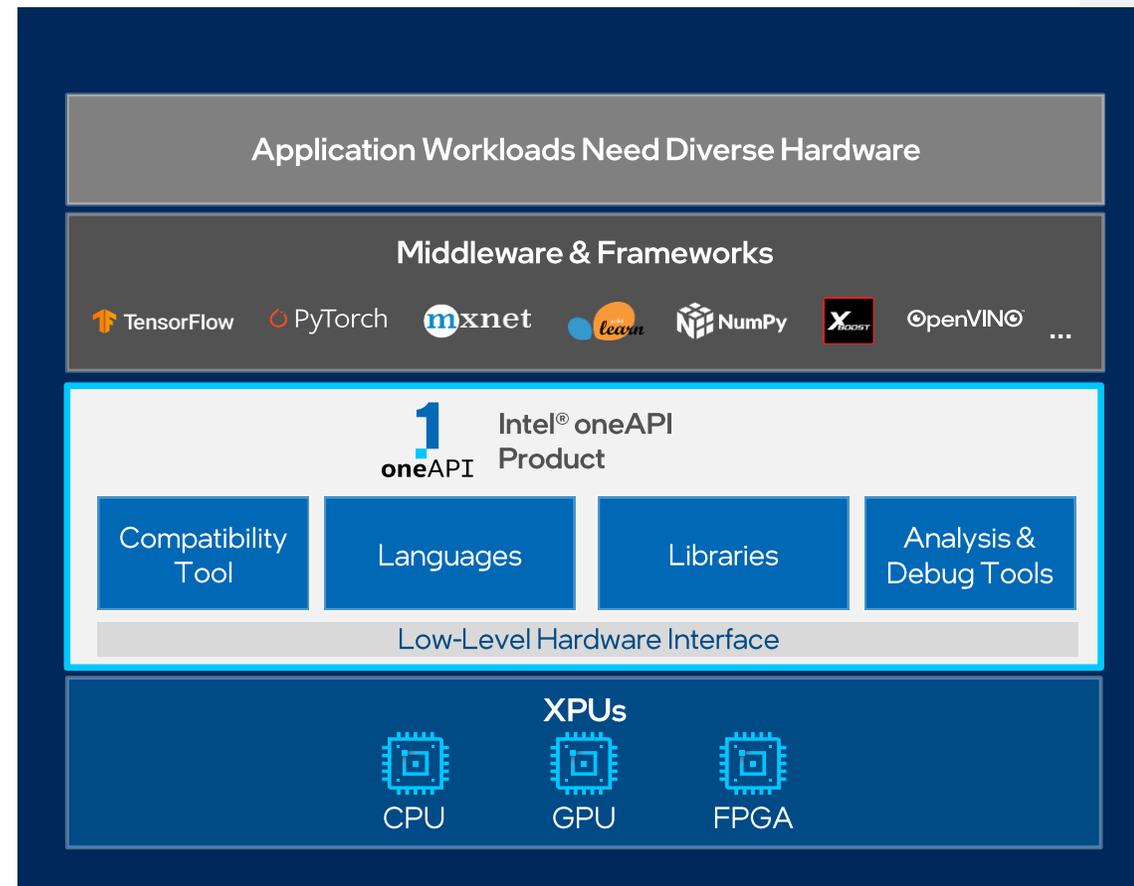
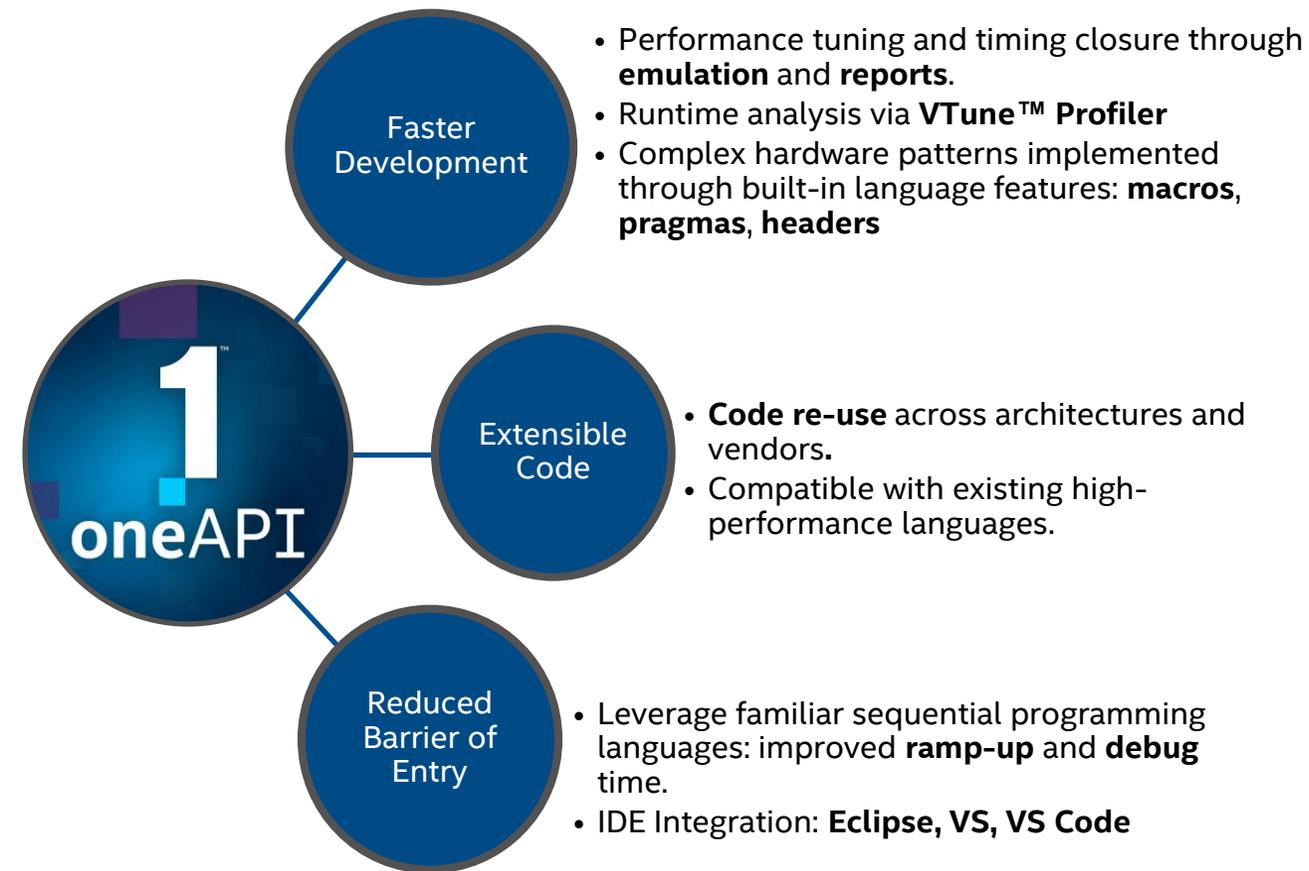
The **oneAPI** product delivers a unified programming model to simplify development across diverse architectures.

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

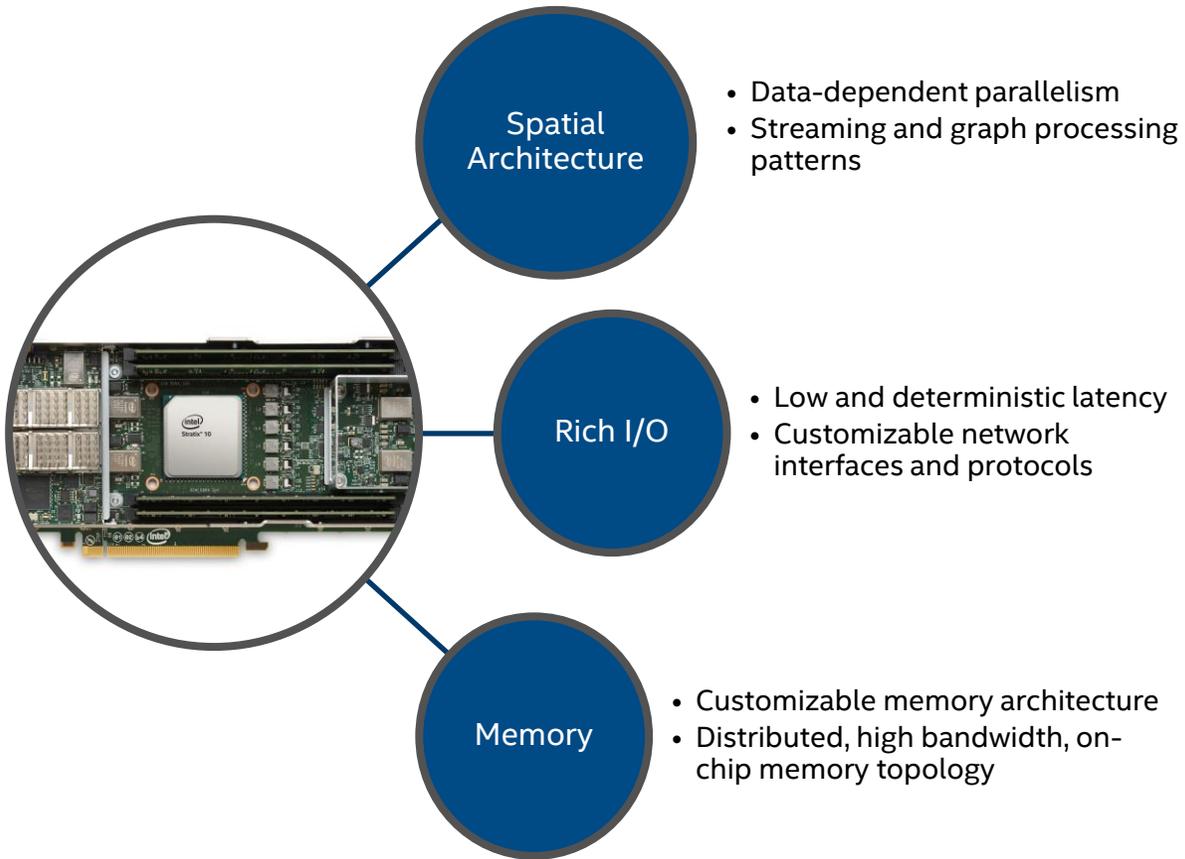


Intel® oneAPI Product

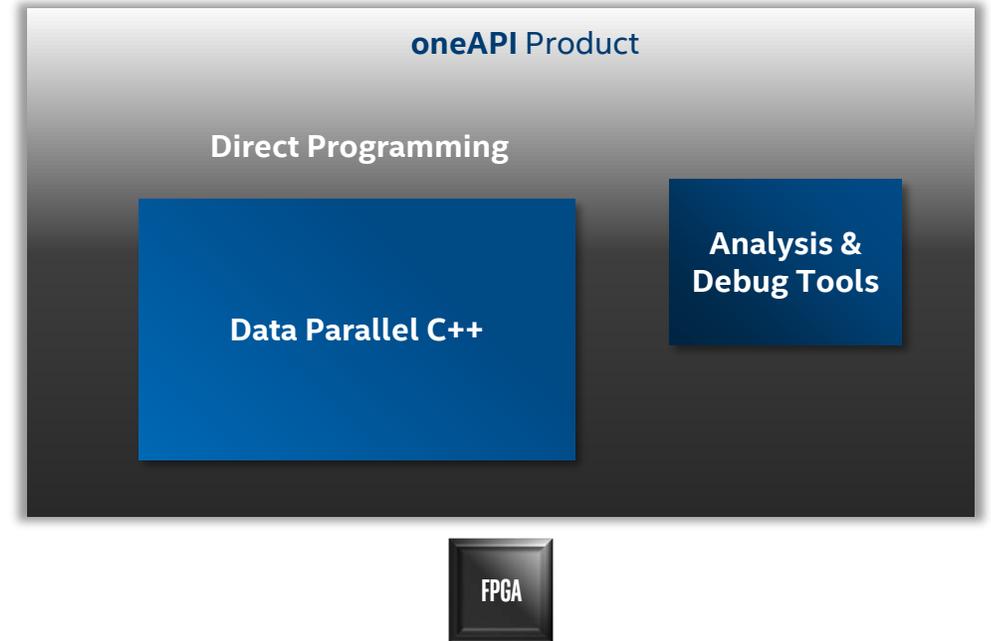


[Available Now](#)

Intel® FPGAs + Intel® oneAPI Toolkits



+





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

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

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

```
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];  
            }  
        });  
    });  
}
```

Application Scope

Command Group Scope

Kernel Scope

The “Runtime”

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

DPC++ 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

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

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

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

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

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

Step 6: Send a kernel for execution

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

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)



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

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

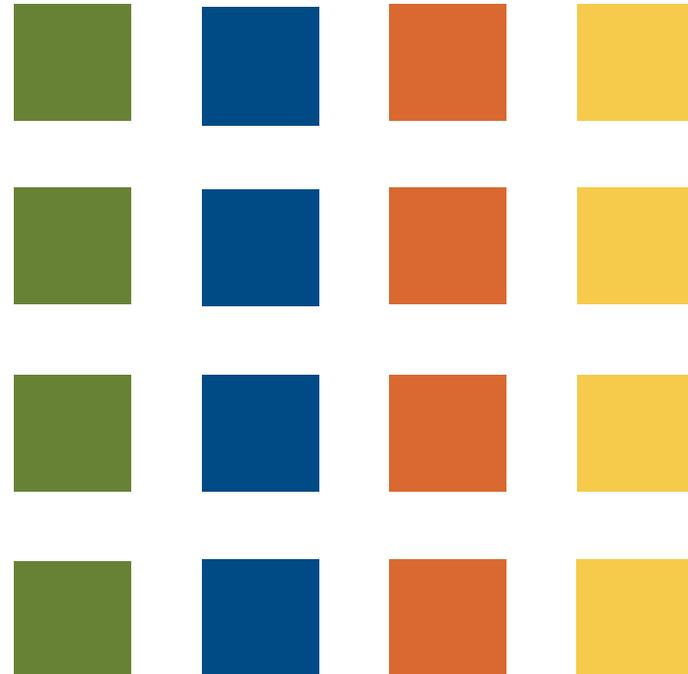
- Programmable = configurable

FPGA = Configurable Hardware

Programming an FPGA

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

Programming it means choosing:

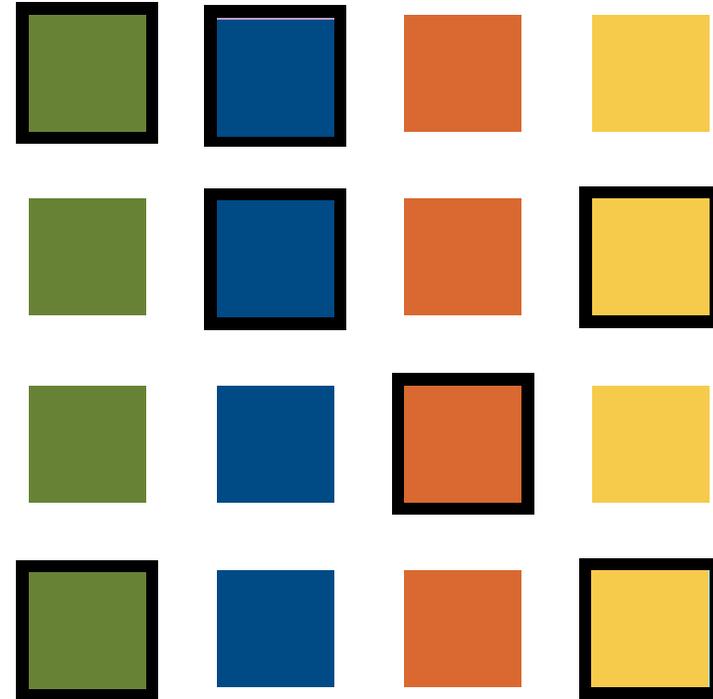


Programming an FPGA

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

Programming it means choosing:

- The building blocks to use

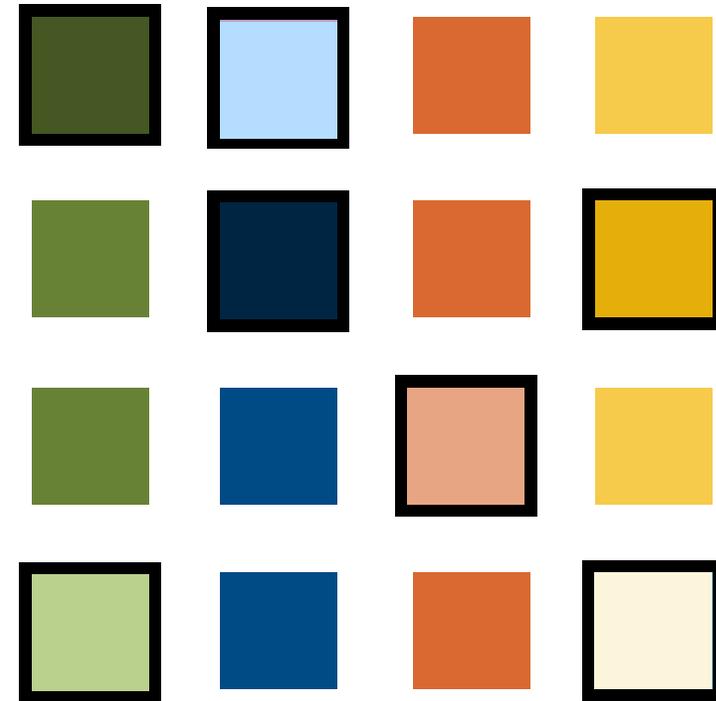


Programming an FPGA

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

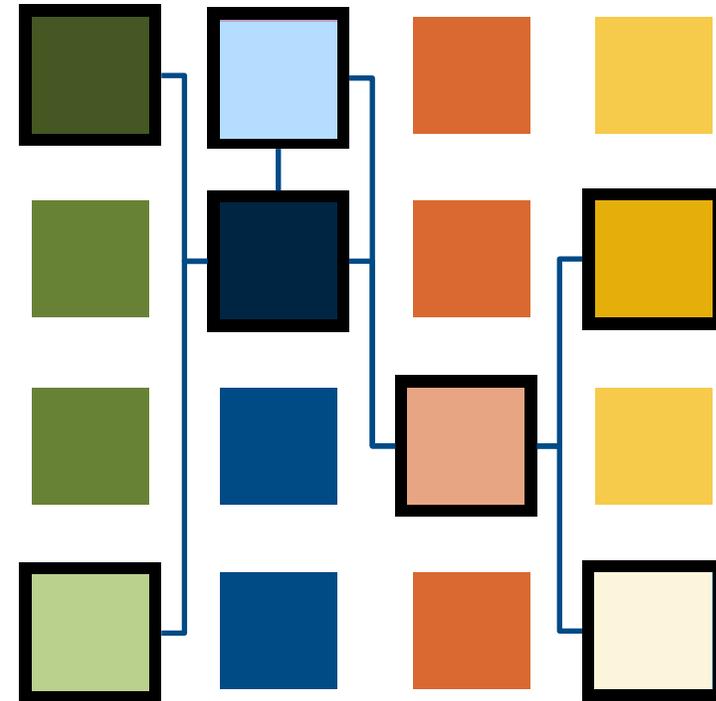


Programming an FPGA

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 an FPGA

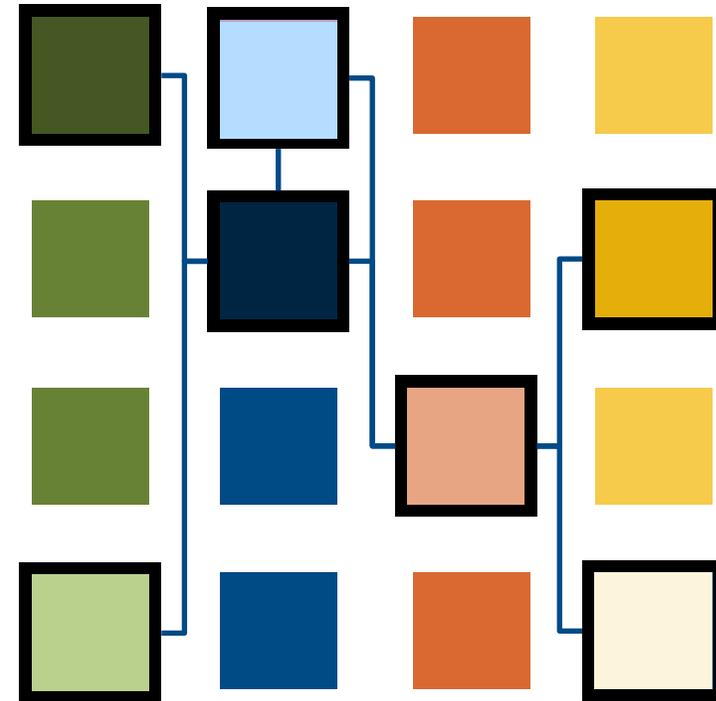
The FPGA is made up of small building blocks of logic and other functions

Programming it means choosing:

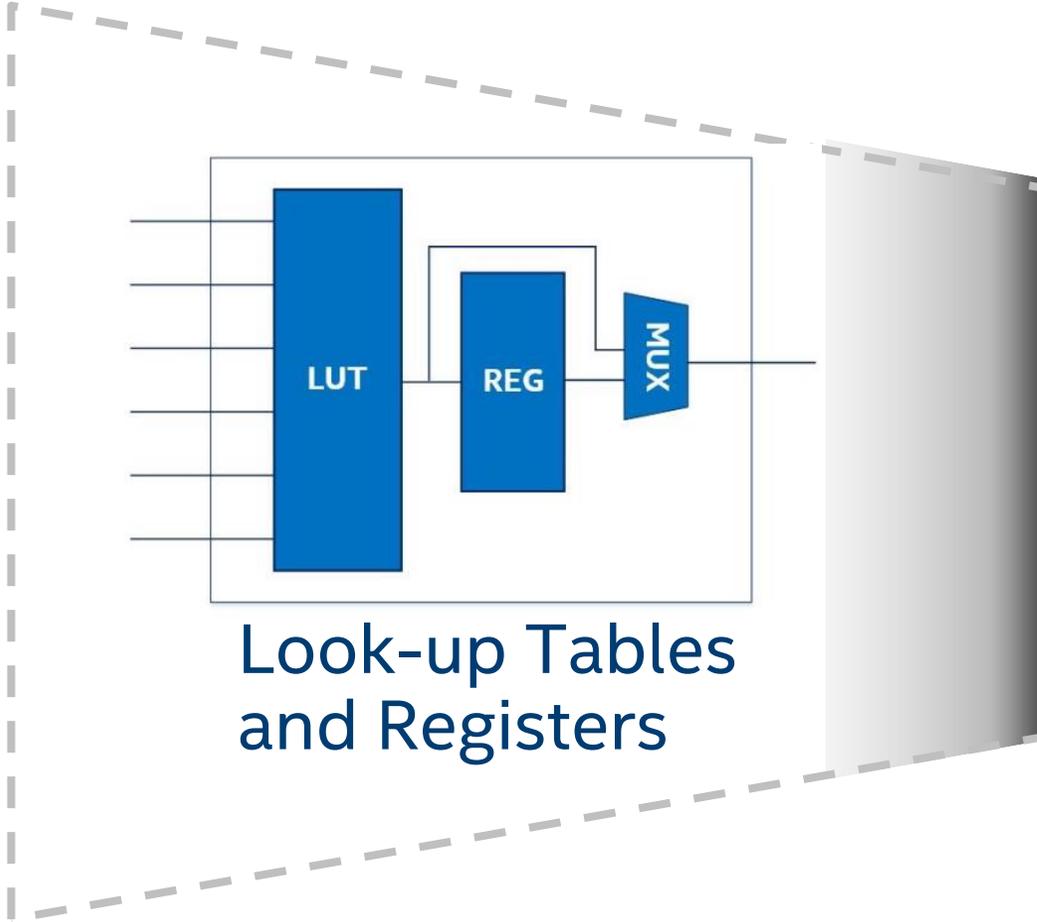
- The building blocks to use
- How to configure them
- And how to connect them

Programming determines the processing architecture implemented in the FPGA

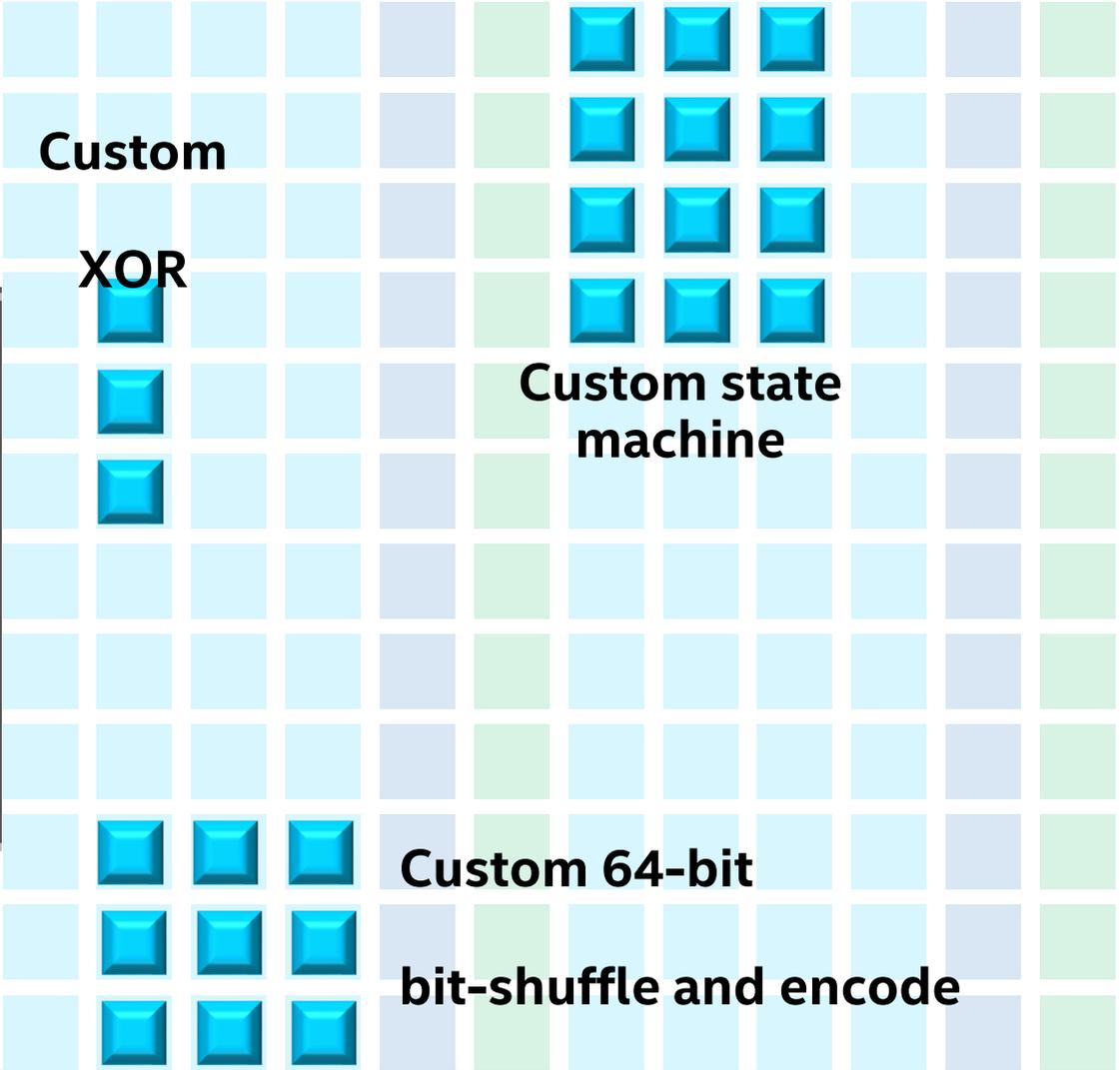
=> what function the FPGA performs



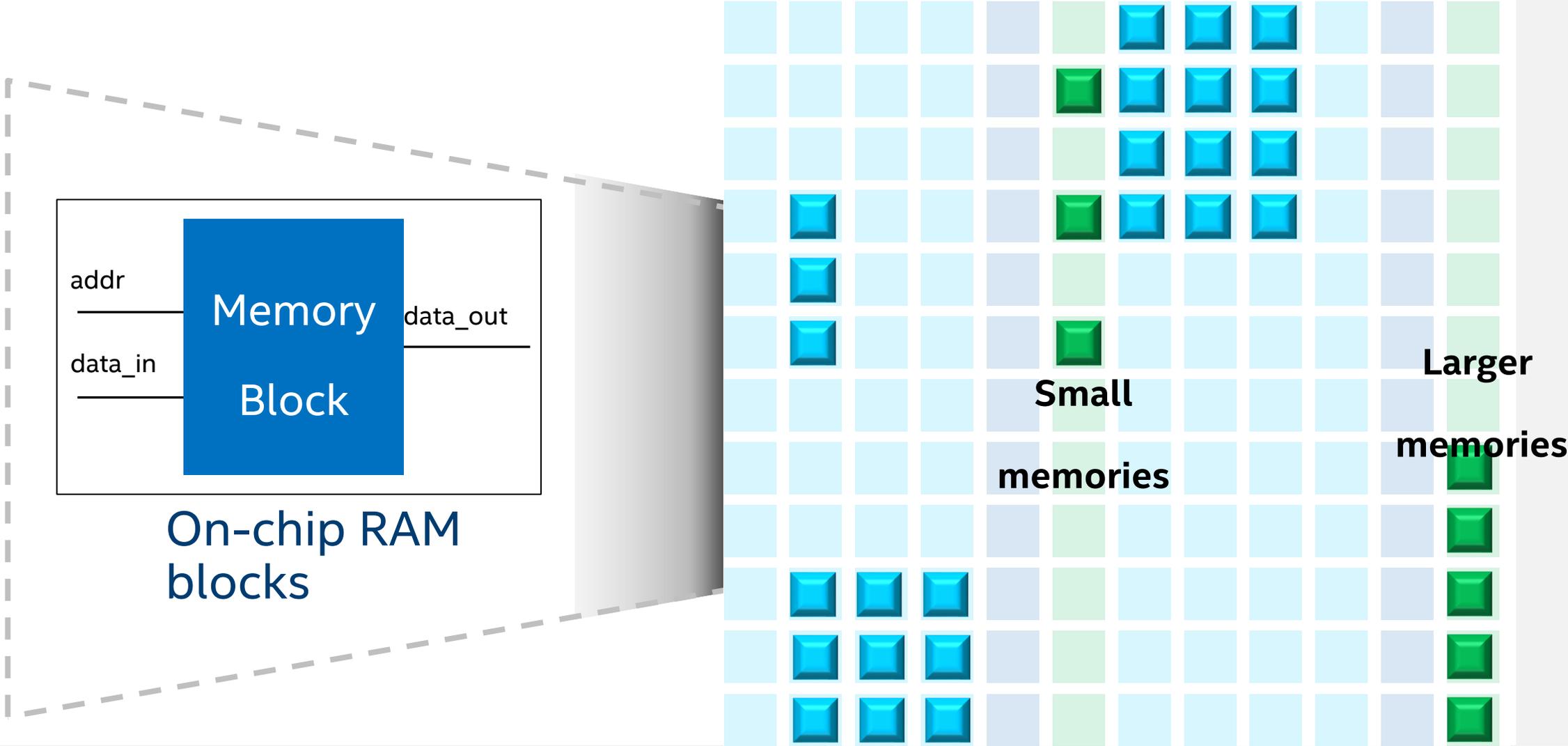
FPGA basic building blocks -ALMs



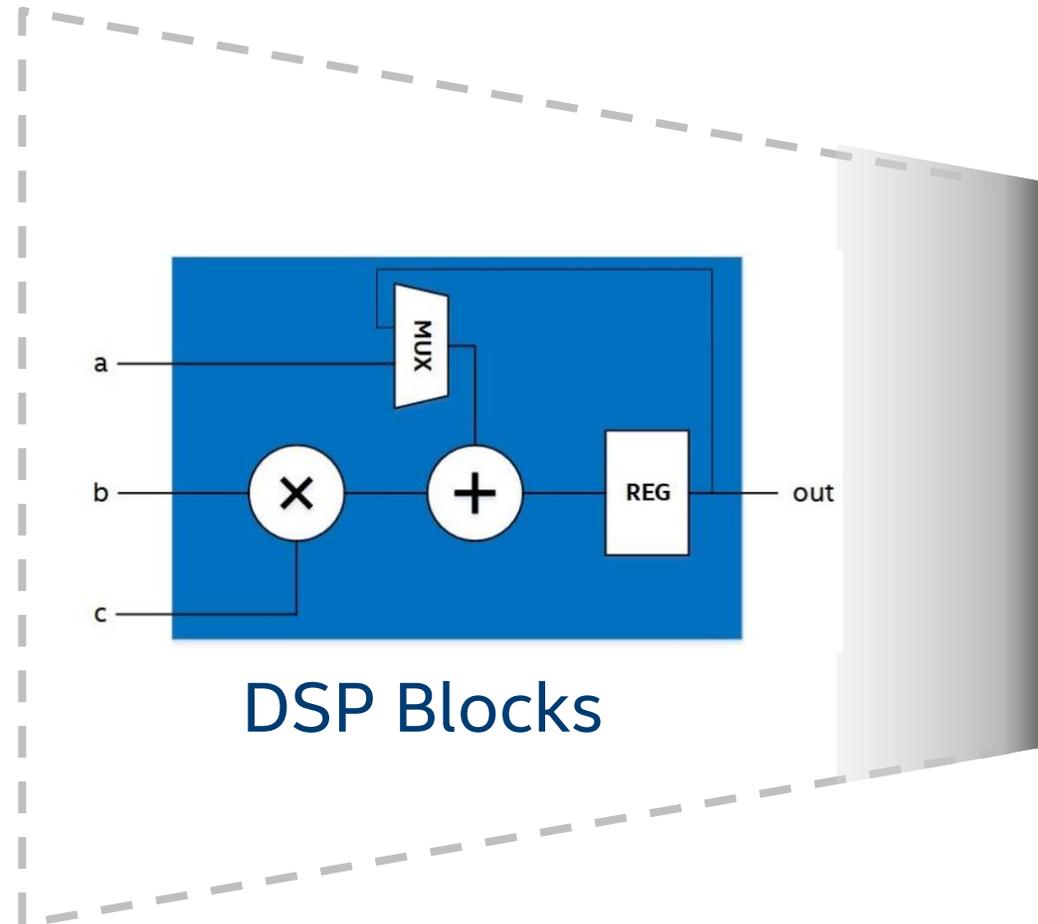
Look-up Tables and Registers



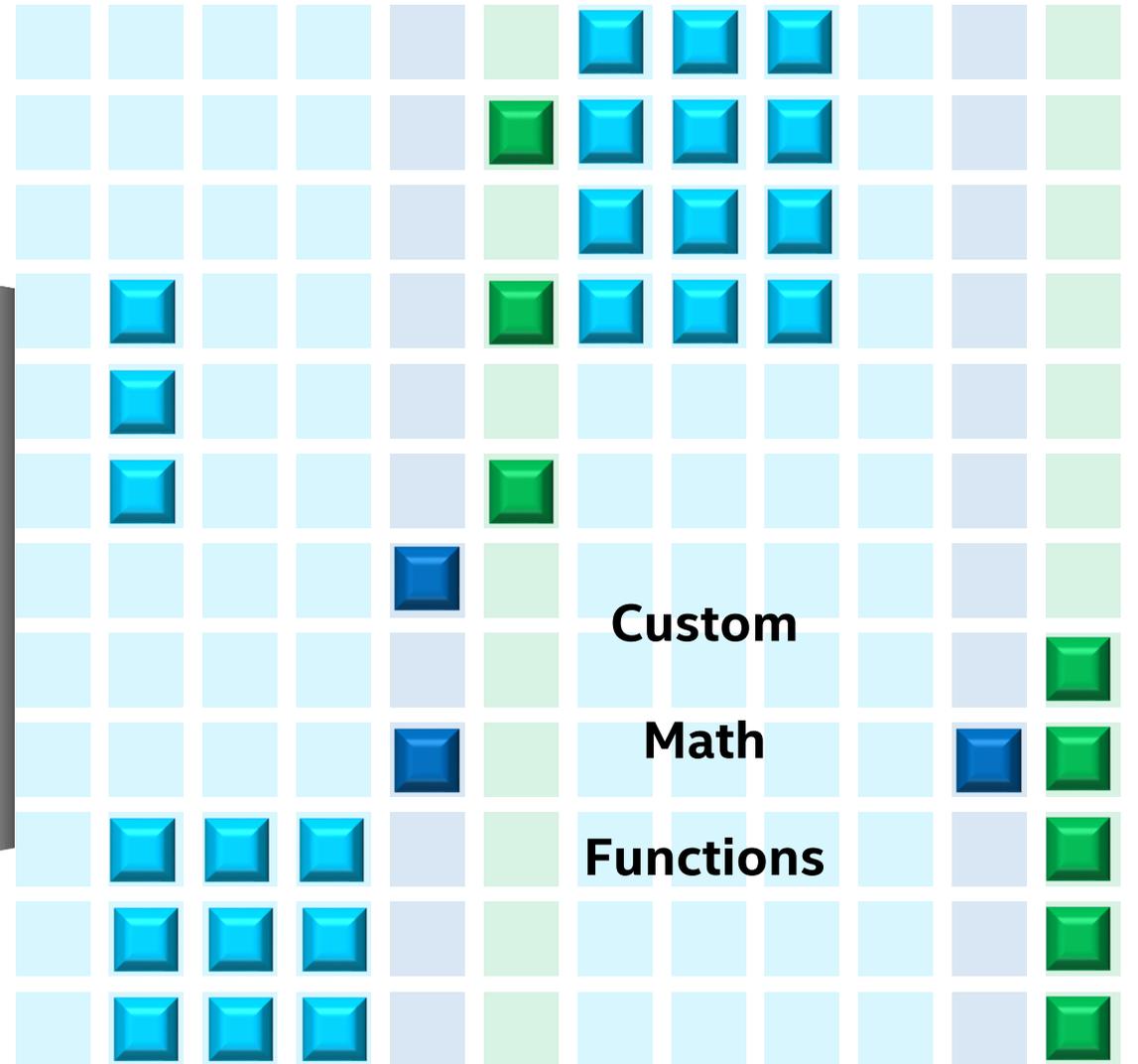
FPGA basic building blocks - RAM



FPGA basic building blocks - DSP blocks



DSP Blocks

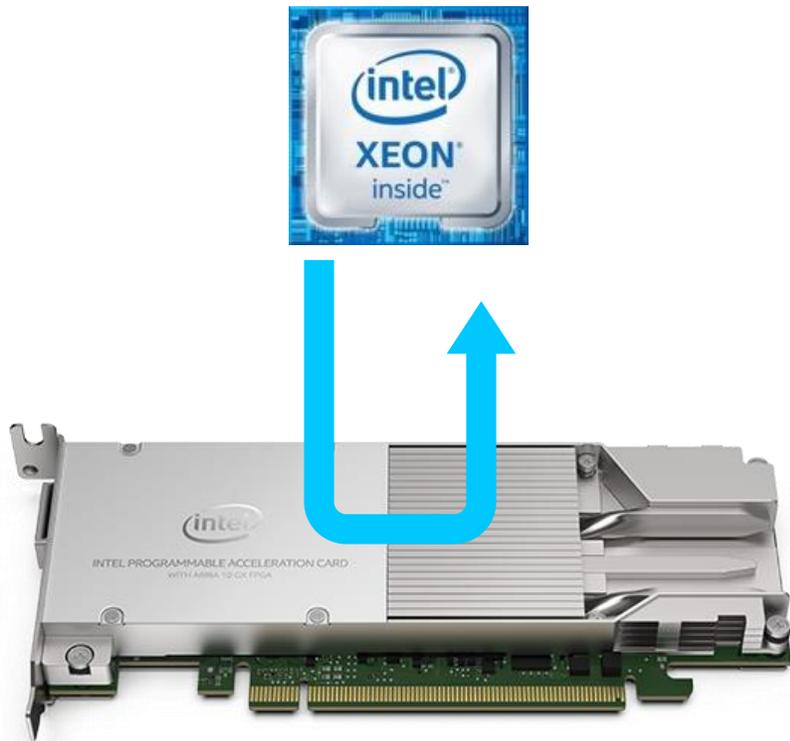


What About Connecting to the Host?

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.



Program Implementation in FPGA

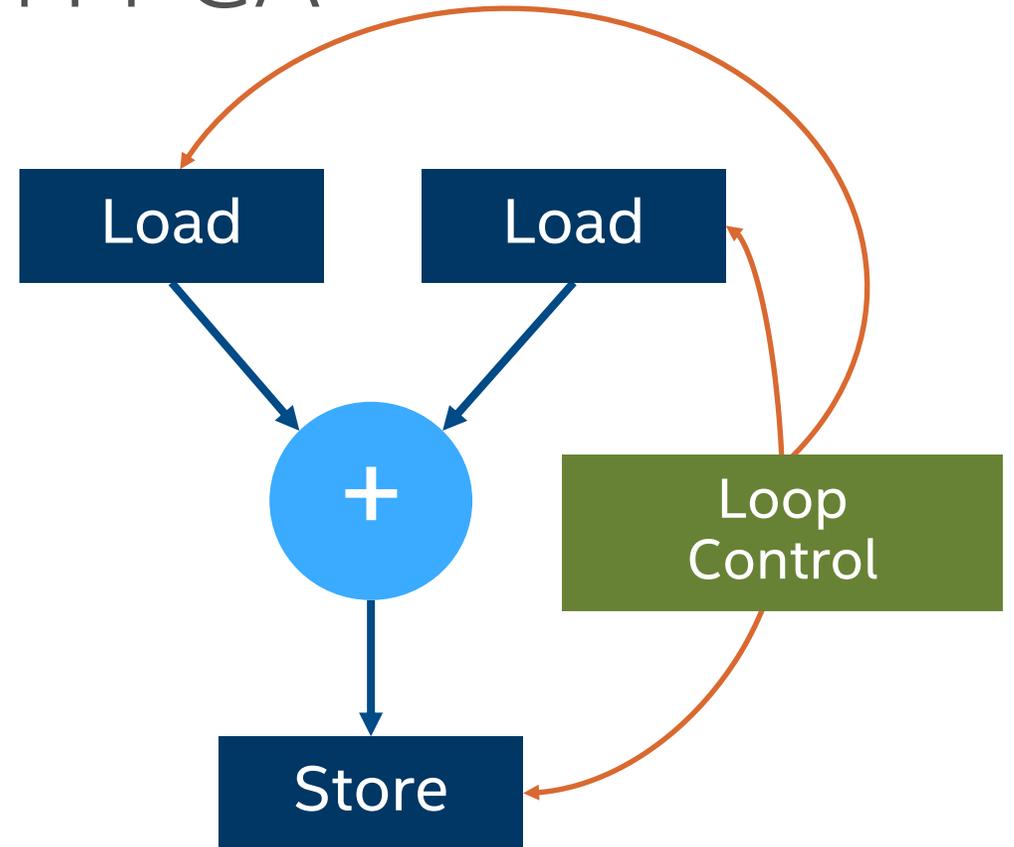
Pipelined hardware is implemented for:

- 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



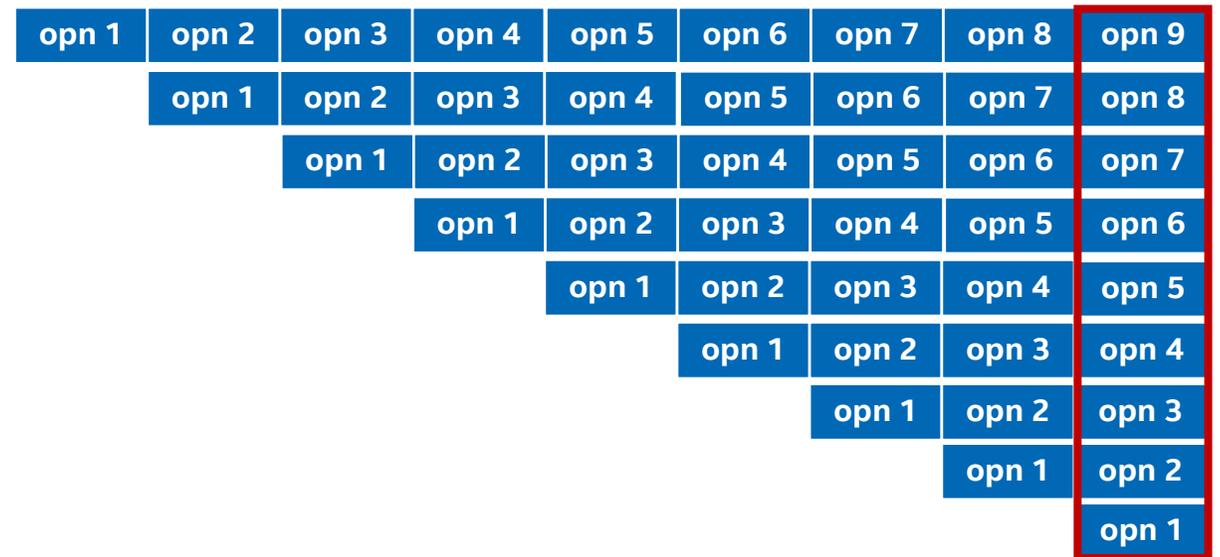
— Data Path
— Control Path

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



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

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



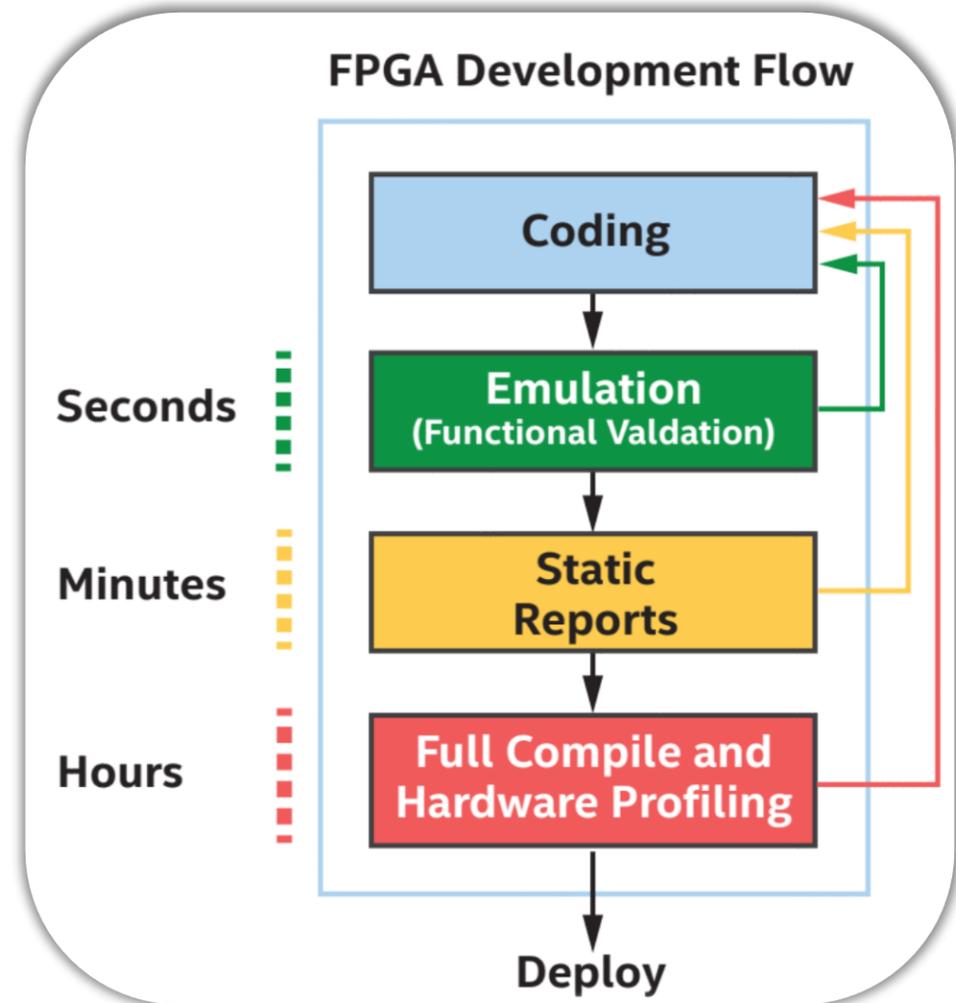
Section: Using FPGAs with the Intel[®] oneAPI Toolkits

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**

FPGA Development Flow for oneAPI Projects

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



Anatomy of a dpcpp Command Targeting FPGAs

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

Target Platform

Link Options

FPGA-Specific Arguments

Language
DPCPP = Data
Parallel C++

Input Files
source or object

Emulation

Seconds of Compilation

Does my code give me the correct answers?

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.

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 -fintel fpga <source_file>.cpp -DFPGA_EMULATOR
```



Report Generation

Minutes of Compilation

Where are the bottlenecks?

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.

Command to Produce an Optimization Report

Two Step Method:

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

One Step Method:

```
dpcpp -fintel FPGA <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

Bitstream Compilation



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

Developers can:

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

Compile to FPGA Executable with Profiler

Two Step Method:

```
dpcpp -fintel fpga <source_file>.cpp -c -o <file_name>.o  
dpcpp -fintel fpga <file_name>.o -Xshardware -Xsprofile
```

One Step Method:

```
dpcpp -fintel fpga <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

has_kernel.cpp

host_only.cpp

Then run this command to compile the FPGA image:

```
dpcpp -fintel-fpga has_kernel.cpp -fsycl-link=image -o has_kernel.o -Xhardware
```

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

```
dpcpp -fintel-fpga host_only.cpp -c -o host_only.o
```

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

```
dpcpp -fintel-fpga has_kernel.o host_only.o -o a.out -Xhardware
```

This is the long compile

Lab: Practice the FPGA Development Flow

Lab instructions

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

The screenshot shows a web browser window with the URL <https://devcloud.intel.com/oneapi/> in the address bar. The browser's bookmark bar shows 'Home - Embedded...', 'my bookmarks', 'Intel', 'Tech', and 'Imported From IE'. The website's navigation bar includes the Intel logo, 'PRODUCTS', 'SUPPORT', 'SOLUTIONS', 'DEVELOPERS', and 'PARTNERS'. On the right side of the navigation bar, there are 'Sign In' and 'Enroll' buttons. Below the navigation bar, the breadcrumb trail reads 'Software / Tools / DevCloud / oneAPI'. The main heading is 'Intel® DevCloud for oneAPI', with sub-links for 'Overview', 'Get Started', 'Documentation', and 'Forum'. A section titled 'Announcements' is visible, with a link to 'VIEW ALL ANNOUNCEMENTS >'. At the bottom of the page, a descriptive sentence reads: 'The Intel DevCloud is a development sandbox to learn about programming cross architecture applications with OpenVino, High Level Design (HLD) tools – oneAPI, OpenCL, HLS – and RTL.' A red box highlights the 'Get Free Access' button, which is positioned above a partially visible 'Sign in' link.

Lab instructions

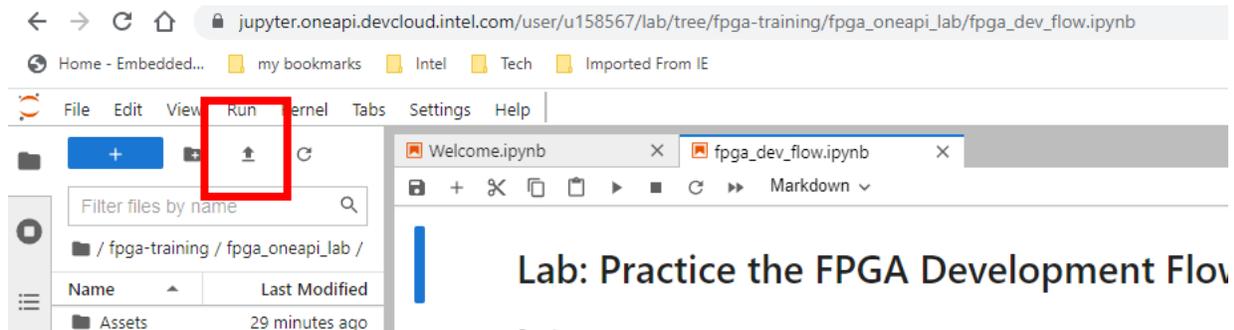
- 1. Create a DevCloud account
 - Enter required information
 - Read and accept terms of use
 - Check your email for the verification link and click on it
 - Sign in
 - Click on “Working with oneAPI”
 - Provision your account, read and accept T&C for oneAPI

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

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



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



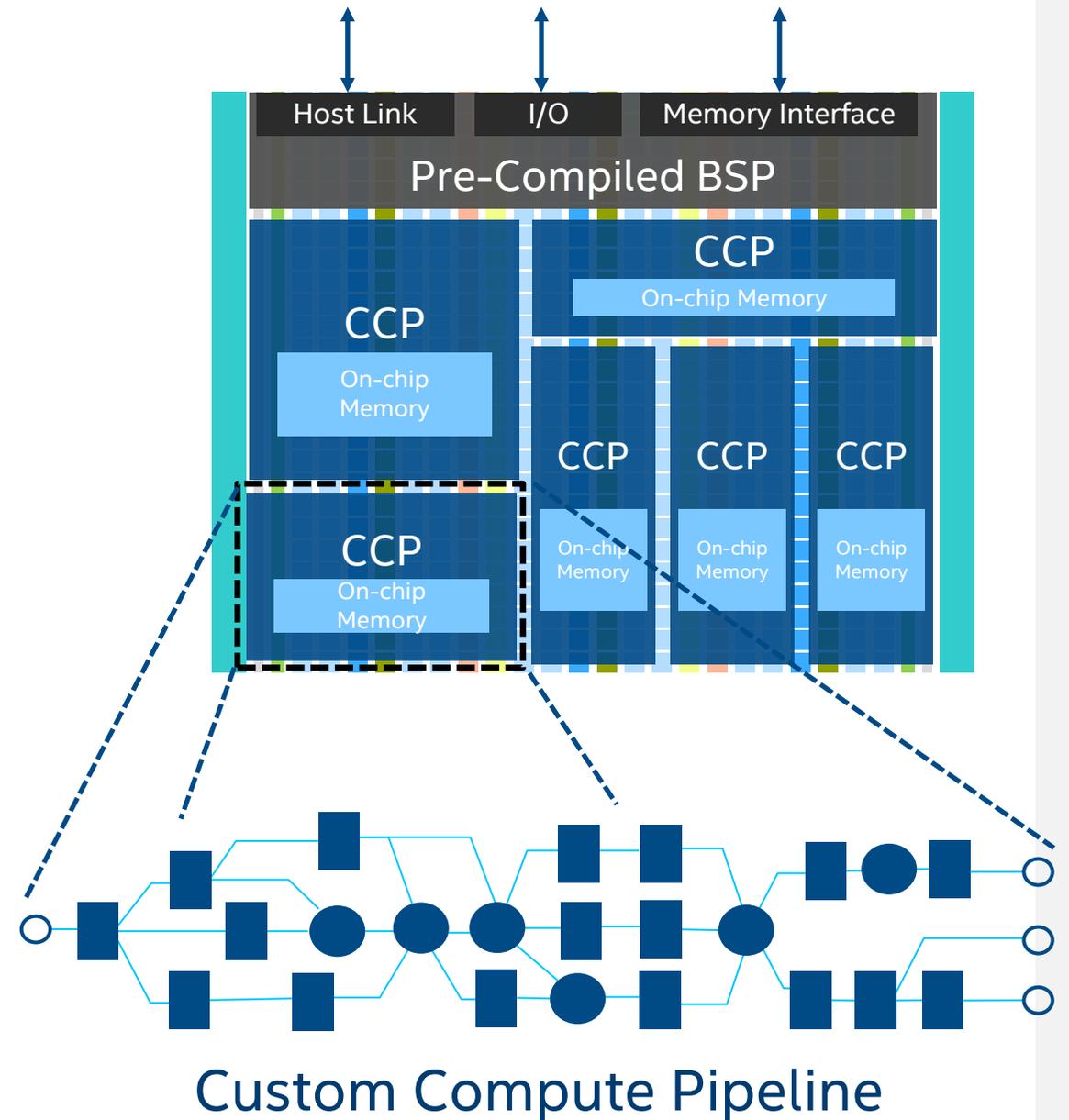
Section: Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Sub-Topics:

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

Intel® FPGAs

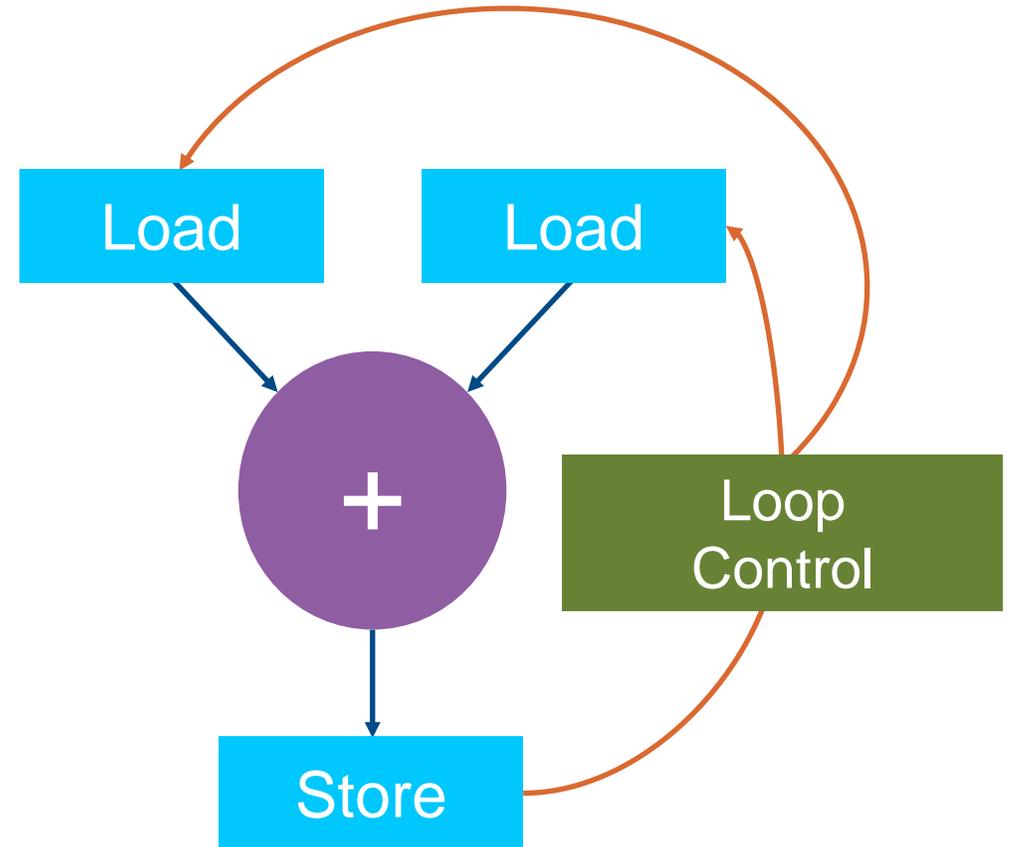
- Implementing Optimized Custom Compute Pipelines (CCPs) synthesized from compiled code



How Is a Pipeline Built?

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

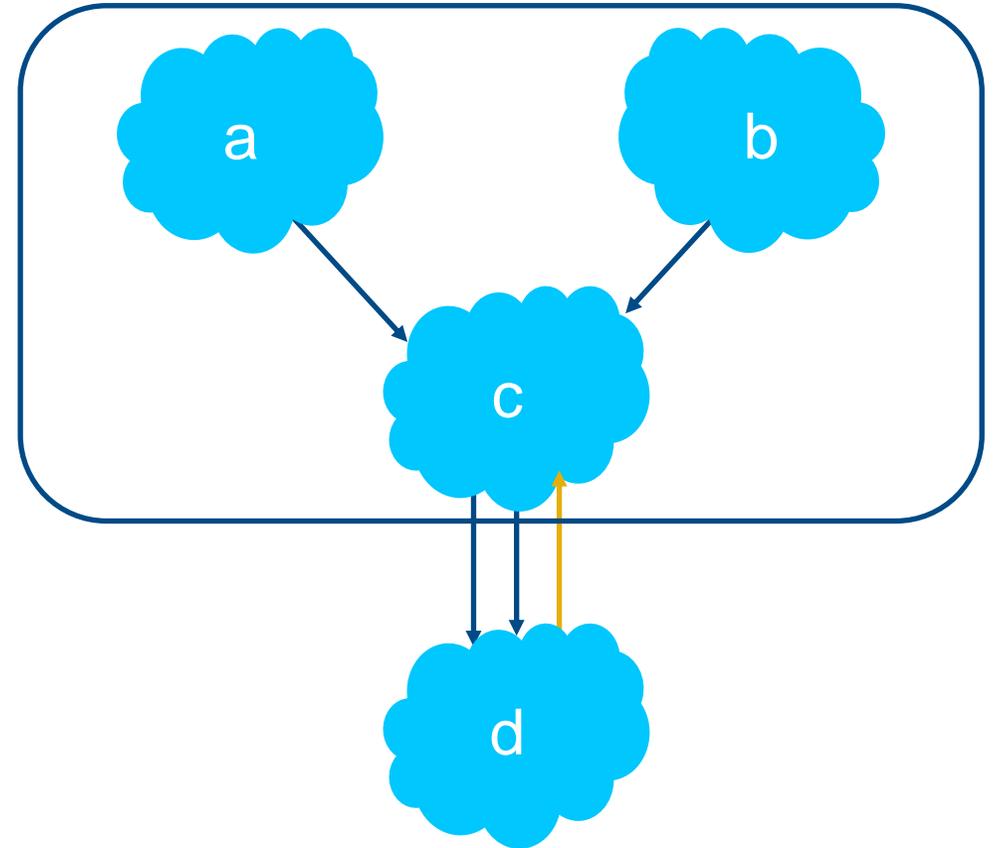
```
for (int i=0; i<LIMIT; i++) {  
    c[i] = a[i] + b[i];  
}
```



— Data Path
— Control Path

Connecting the Pipeline Together

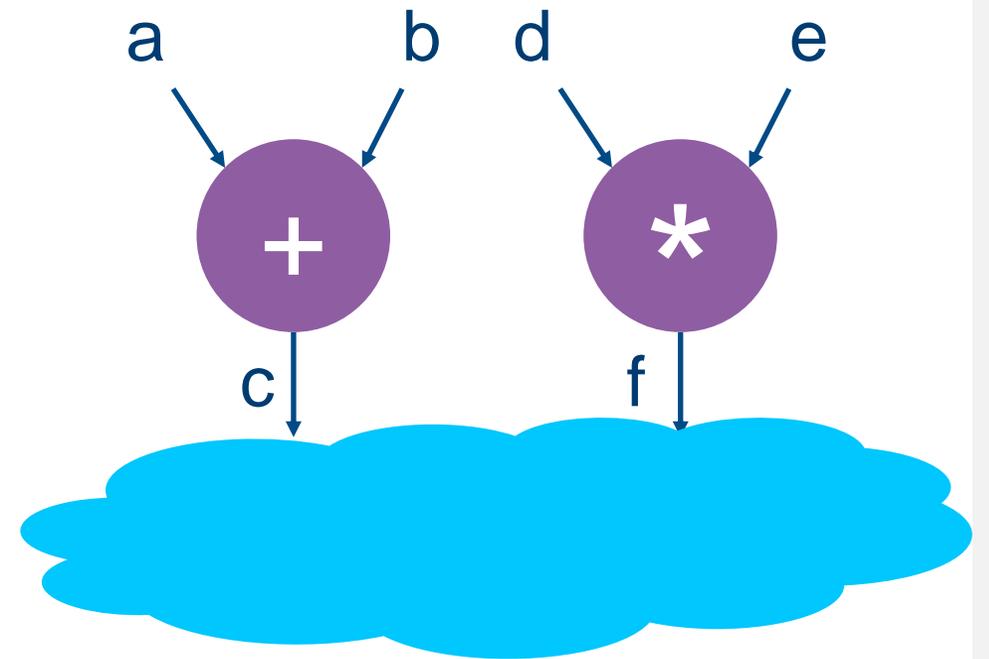
- Handshaking signals for variable latency paths
- Operations with a fixed latency are clustered together
- Fixed latency operations improve
 - Area: no handshaking signals required
 - Performance: no potential stalling due to variable latencies



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

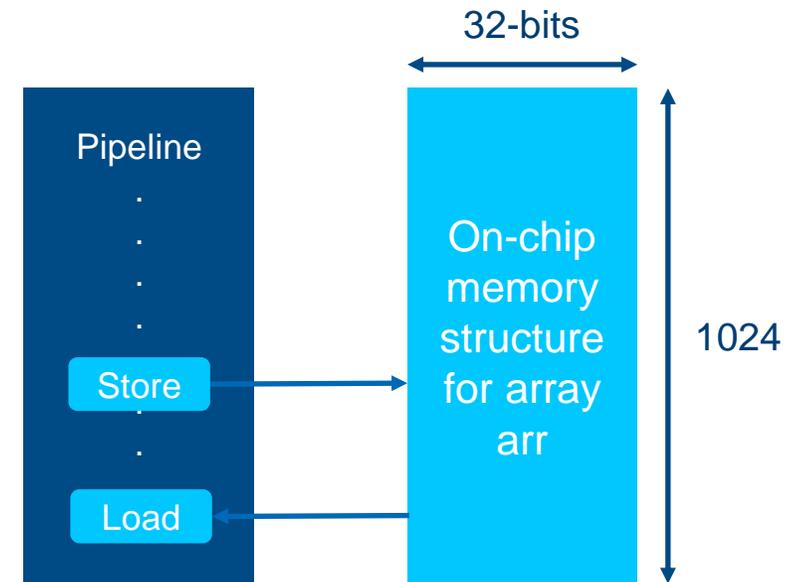
```
c = a + b;  
f = d * e;
```



On-Chip Memories Built for Kernel 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

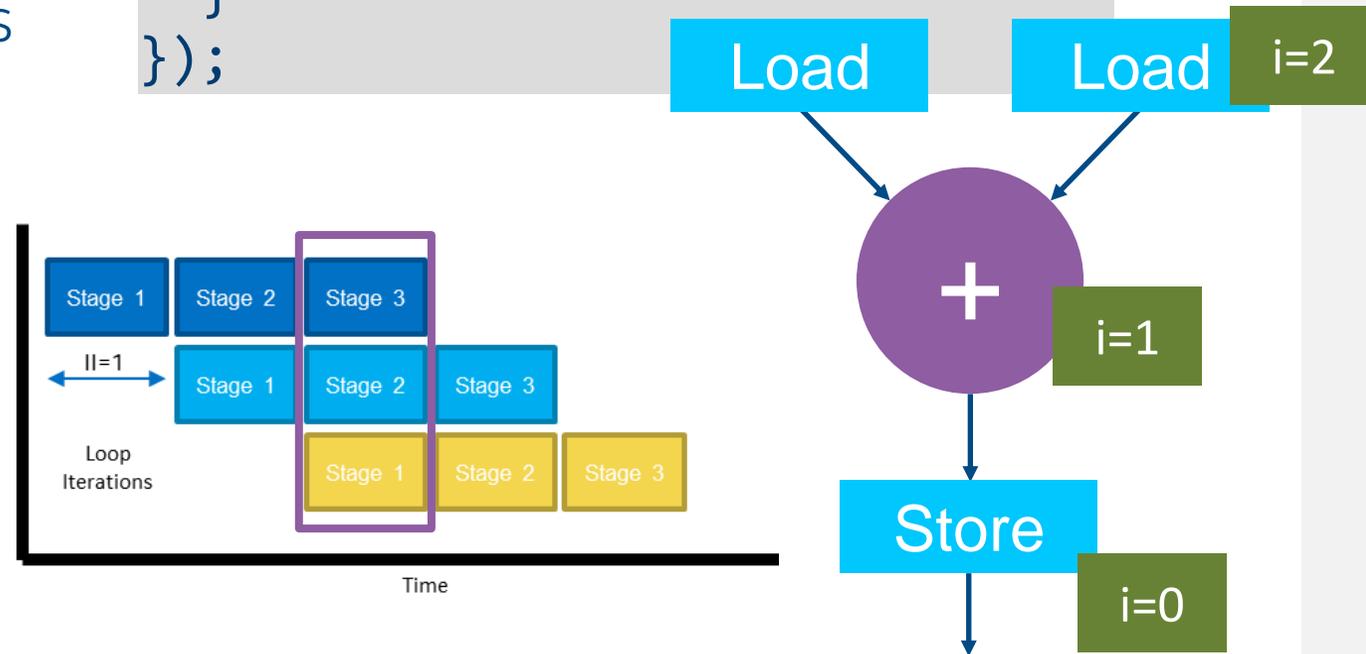
```
//kernel scope
cgh.single_task<>([=]()) {
  int arr[1024];
  ...
  arr[i] = ...; //store to memory
  ...
  ... = arr[j] //load from memory
  ...
} //end kernel scope
```



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];  
    }  
});
```

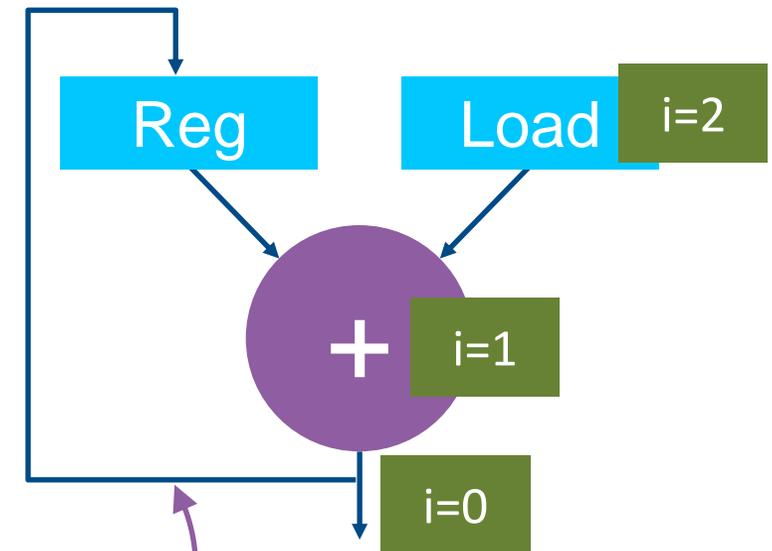


Dependencies Within the Single Work-Item Kernel

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];  
  }  
});
```

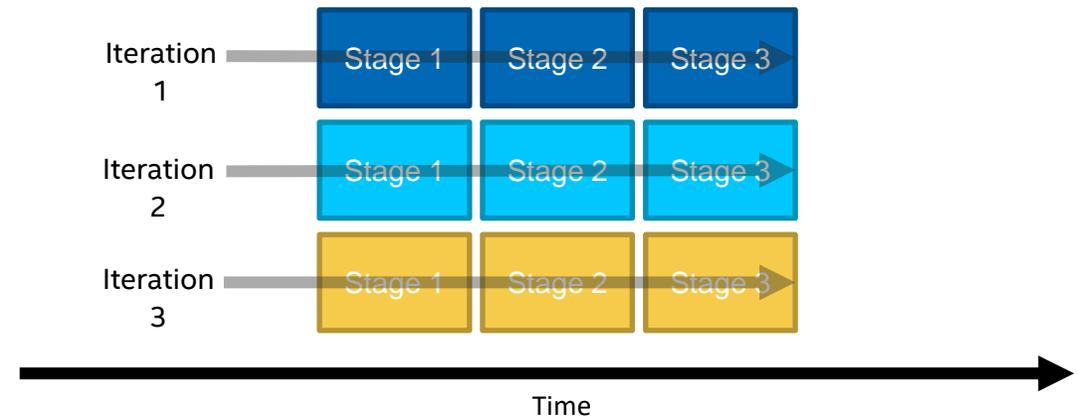
Key Concept
Custom built-in dependencies make FPGAs powerful for many algorithms



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];  
    }  
});
```

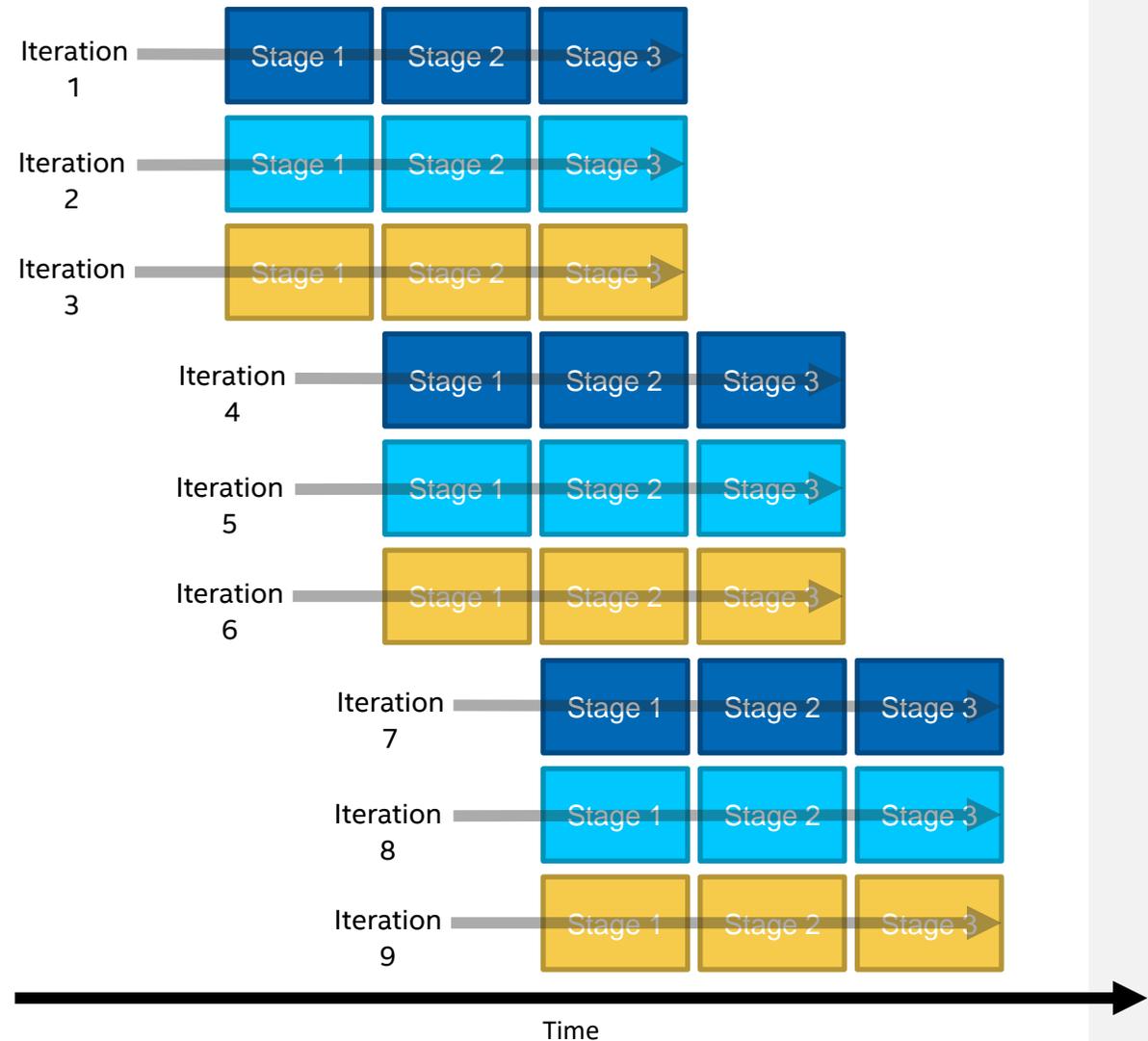


Unrolled Loops Still Get Pipelined

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

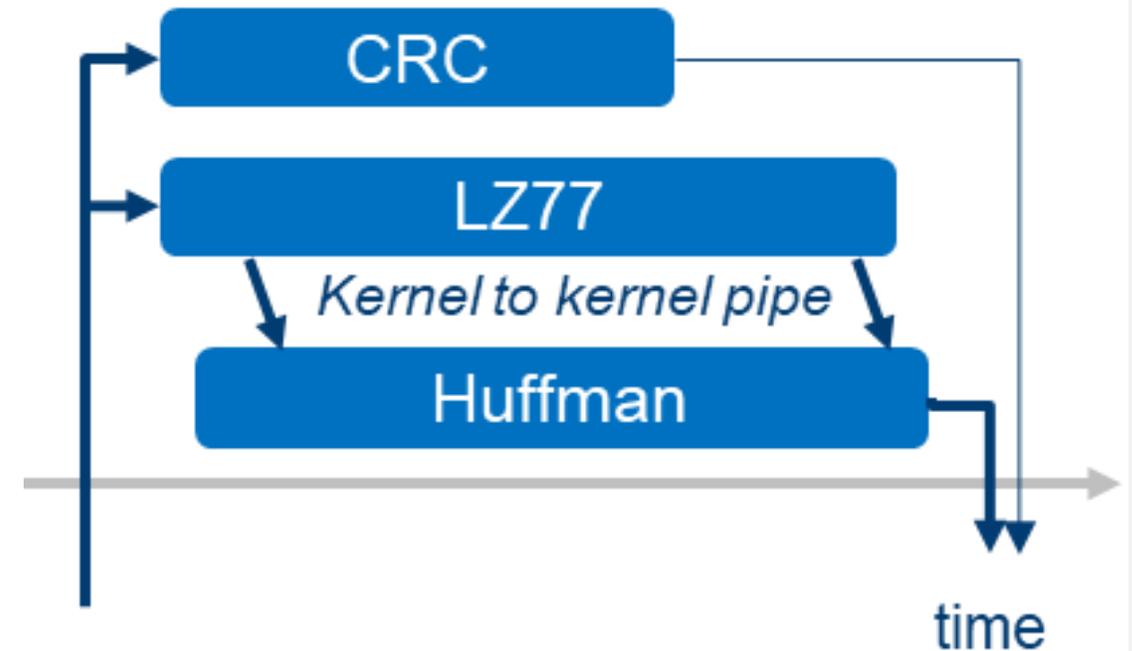
```
handle.single_task<>([=]()) {  
    ... //accessor setup  
    #pragma unroll 3  
    for (int i=1; i<=9; i++) {  
        c[i] += a[i] + b[i];  
    }  
};
```



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 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



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

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];
        }
    });

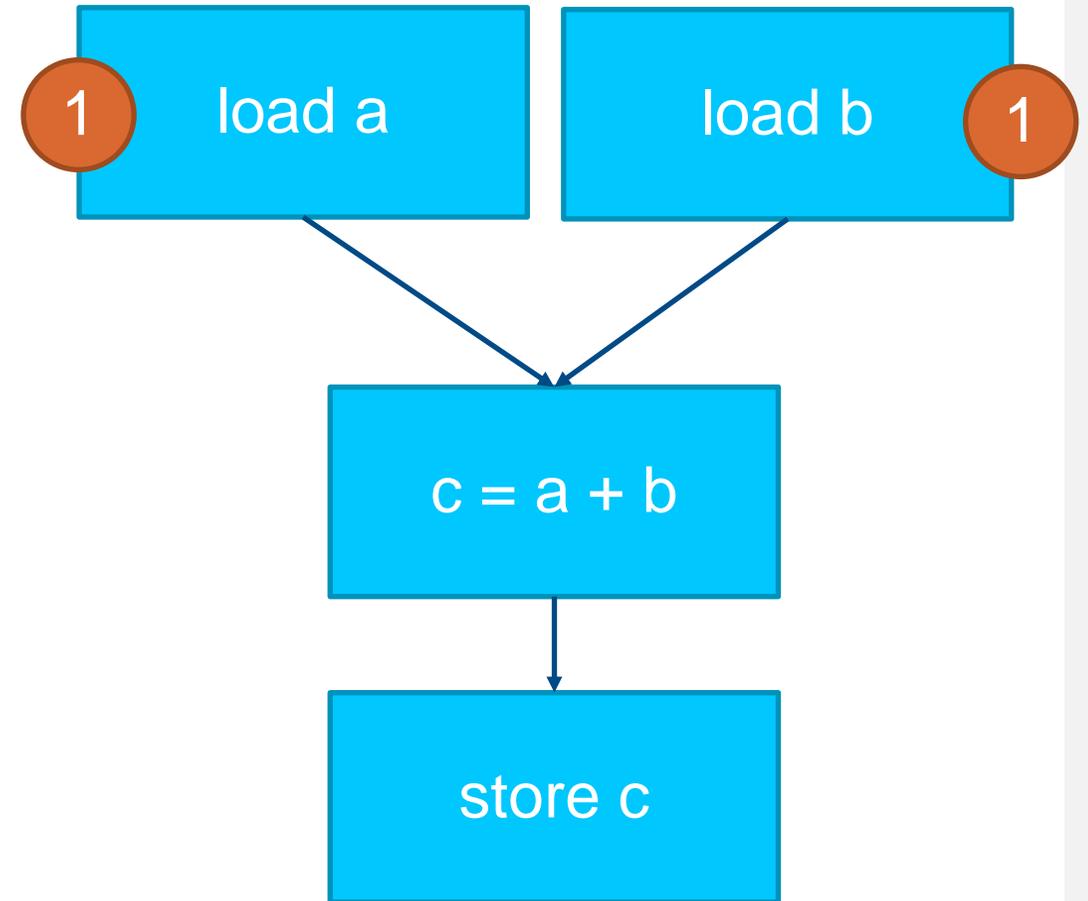
});

...//application scope
```

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];  
    }  
};  
...
```

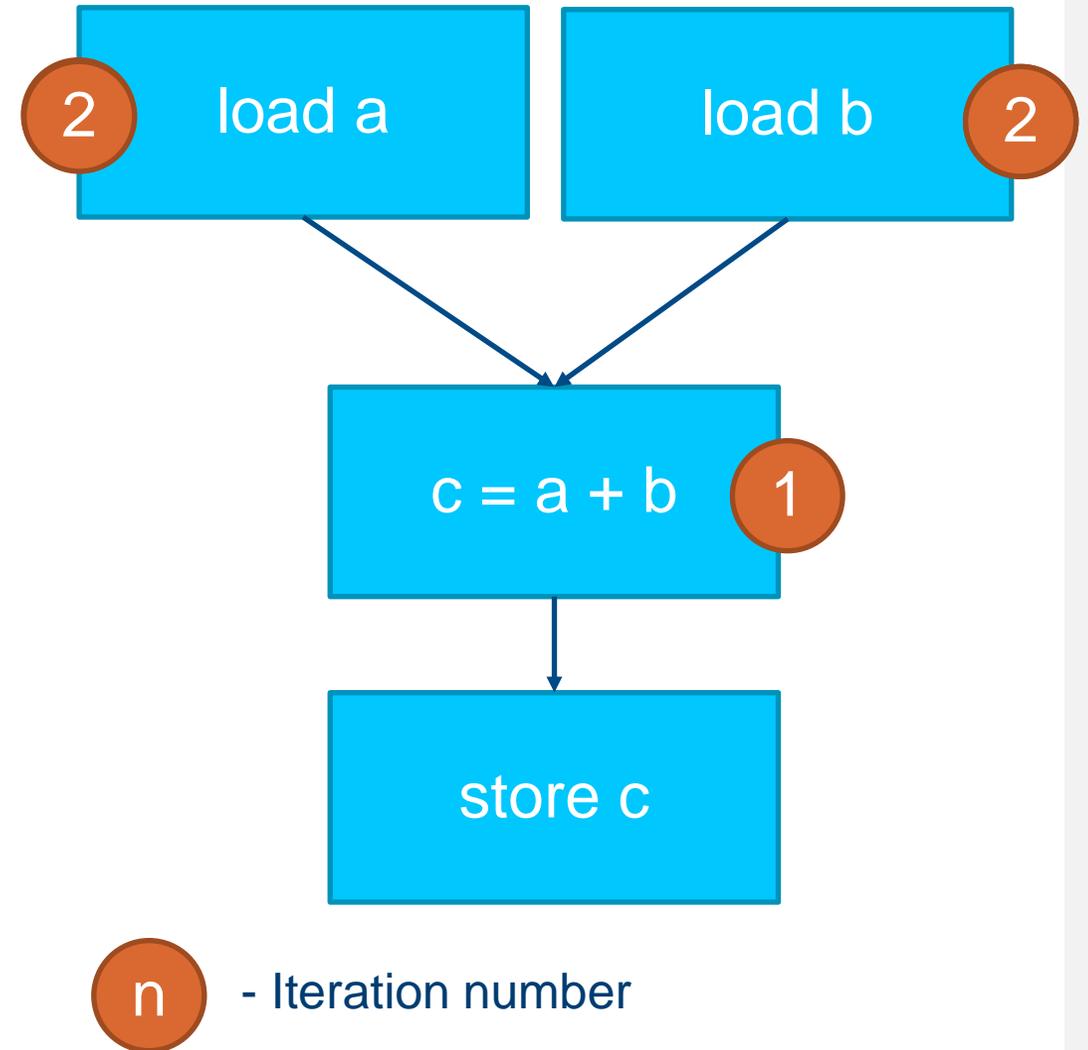


n - Iteration number

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

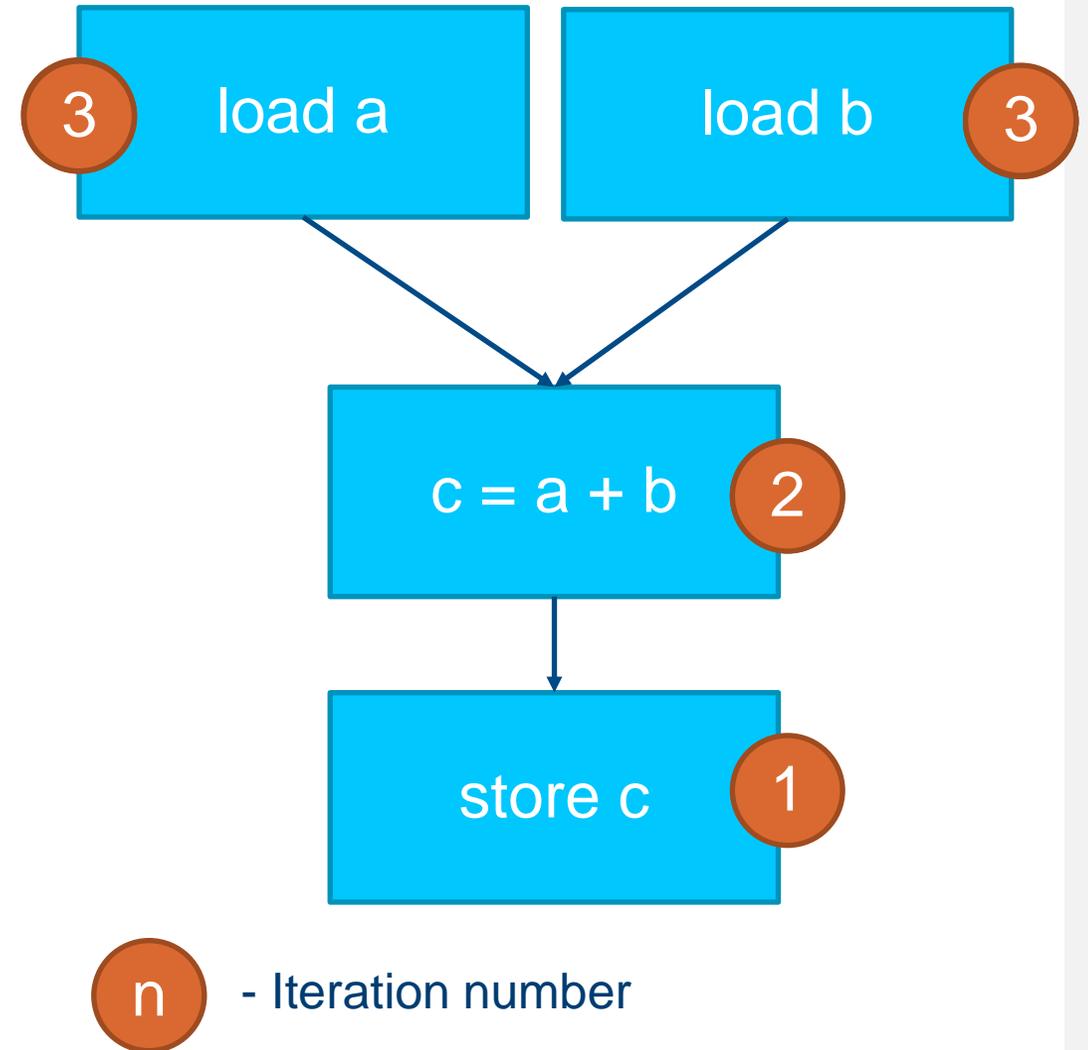
```
...
cgh.single_task<class swi_add>([=]()) {
    for (unsigned i = 0; i < 128; i++) {
        c[i] = a[i] + b[i];
    }
};
...
```



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

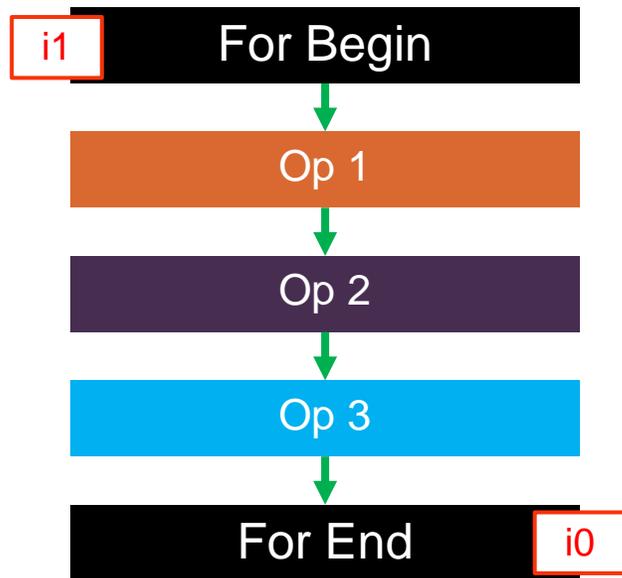
```
...
cgh.single_task<class swi_add>([=]()) {
    for (unsigned i = 0; i < 128; i++) {
        c[i] = a[i] + b[i];
    }
});
...
```



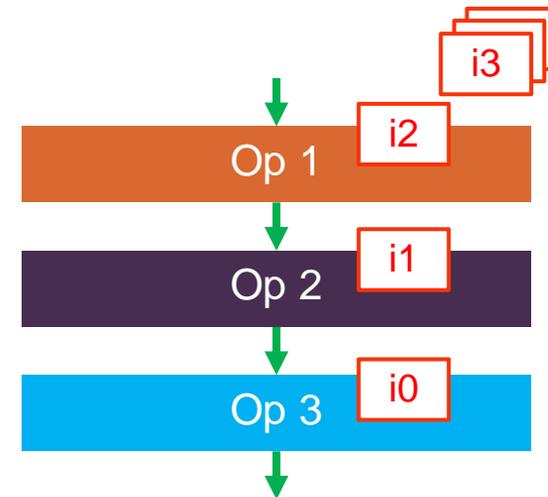
Loop Pipelining vs Serial Execution

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

Worst Case



Best Case

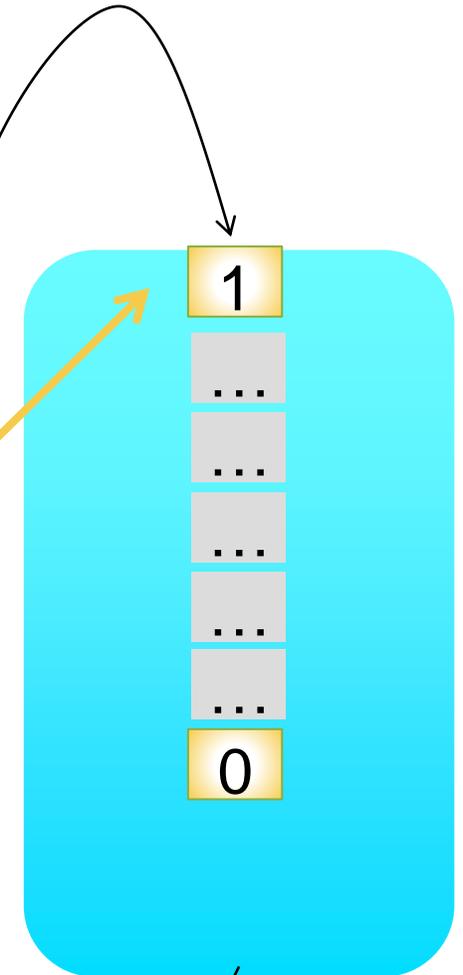


In-Between Scenario

- 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

Report: fpga_970fa3 - Mozilla Firefox

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

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

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

Value must be retrieved from global memory and incremented

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

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

```
[[intel FPGA::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

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

```
[[intel FPGA::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

```
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

Report: fpga_0cbd30 - Mozilla Firefox

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

Reports Summary Throughput Analysis Area Analysis System Viewers

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
66 // Kernel
67 cgh.single_task<class SimpleAdd>([=]() {
68     double a_int[ARRAY_SIZE];
69     double b_int[ARRAY_SIZE];
70     double r_int[ARRAY_SIZE];
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

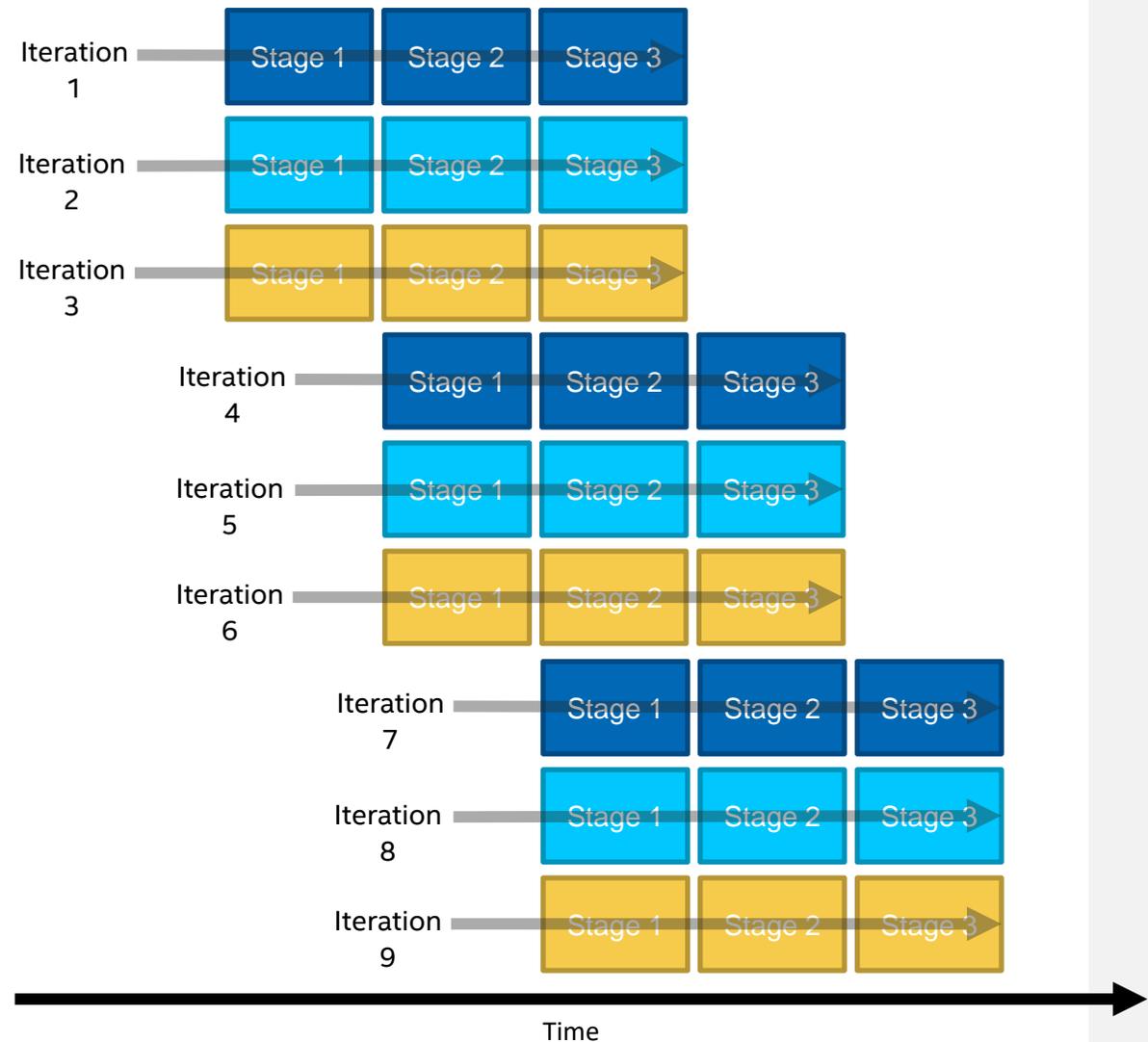
What Can You Do?

- 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];  
    }  
});
```



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

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

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

```
[[intel_fpga::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

```
[[intel FPGA::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

```
[[intel FPGA::max_concurrency(n)]]
```



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

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

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];

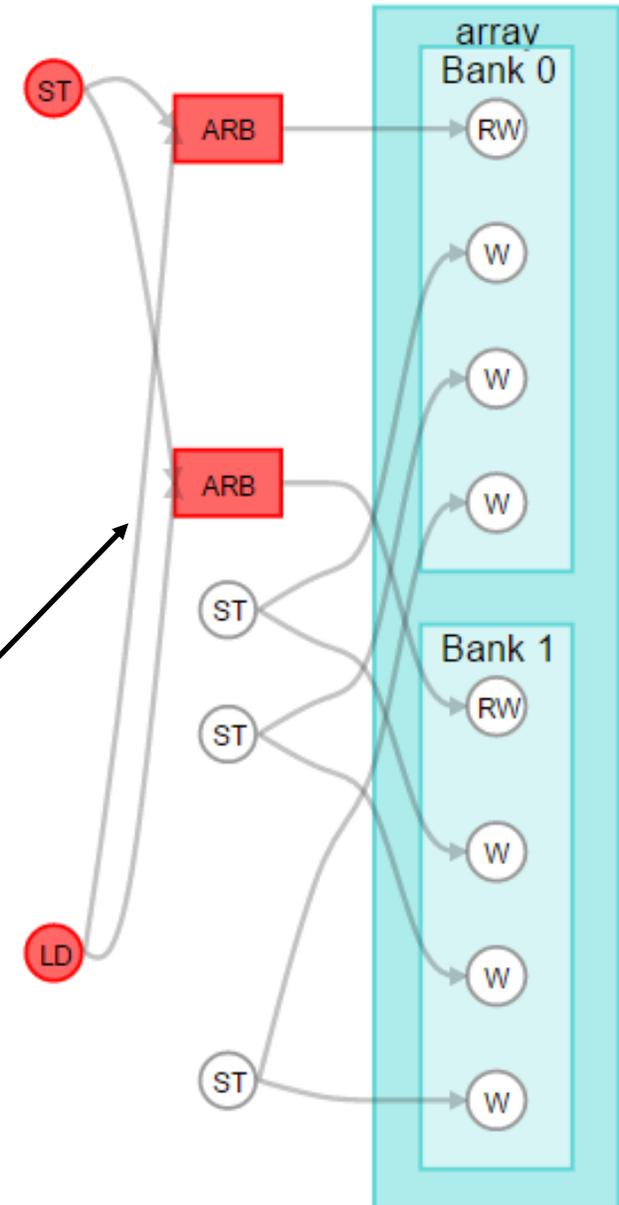
        for (unsigned i = 0; i < N; i++)
            B[N-i] = B[i];

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

Optimized

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 uses optimizations to minimize 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

```
//kernel scope
```

```
...
```

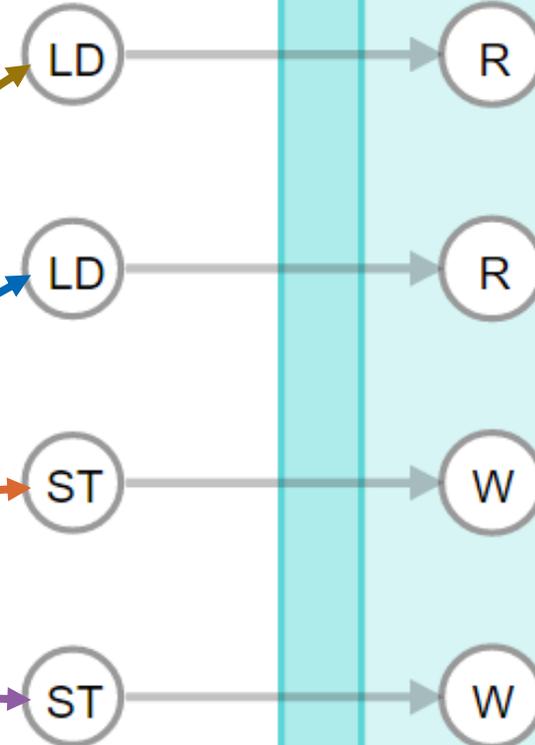
```
int array[1024];
```

```
array[ind1] = val;
```

```
array[ind1+1] = val;
```

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

```
...
```



Local Memory Replication Example

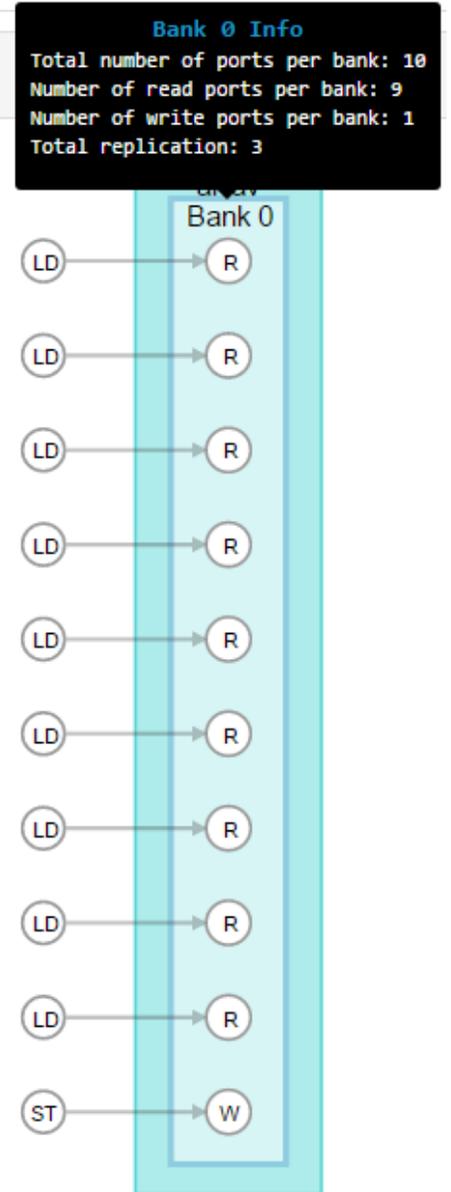
```
//kernel scope
...
  int array[1024];
  int res = 0;

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

  calc = res;
...
```

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;

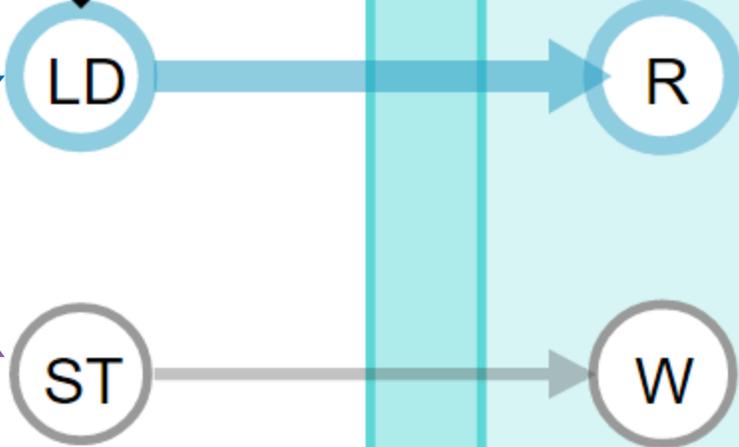
#pragma unroll
for (int i = 0; i < 4; i++)
    array[ind1*4 + i] = val;

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

calc = res;
...
```

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

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



Continuous addresses can be coalesced into wider accesses

Banking

- Divide the memory into independent fractional pieces (banks)

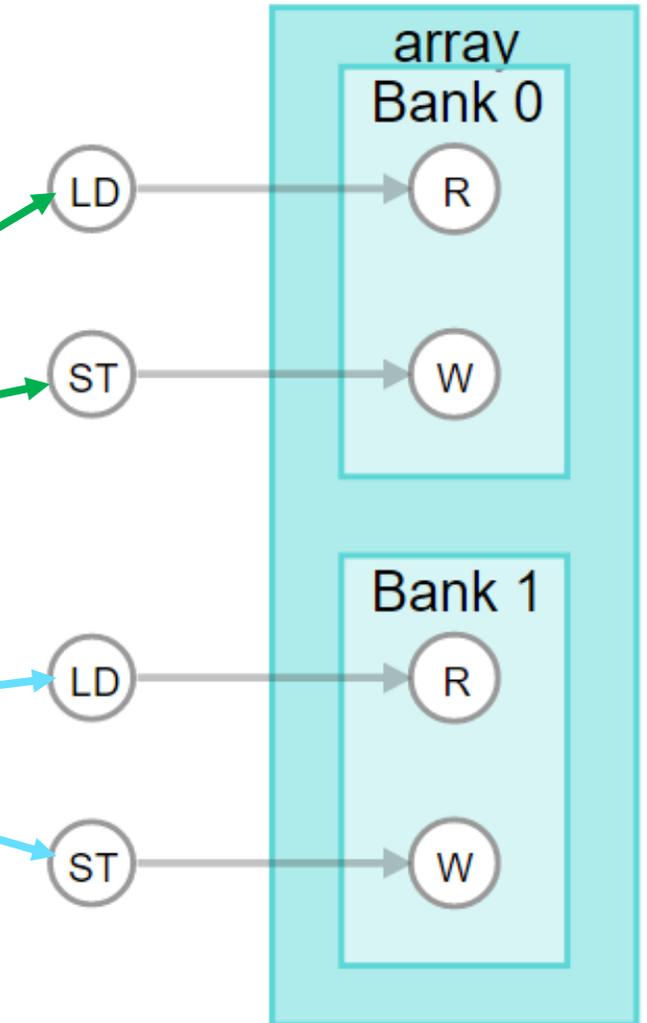
```
//kernel scope
```

```
...  
int array[1024][2];
```

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

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

```
...
```



Attributes for Local Memory Optimization

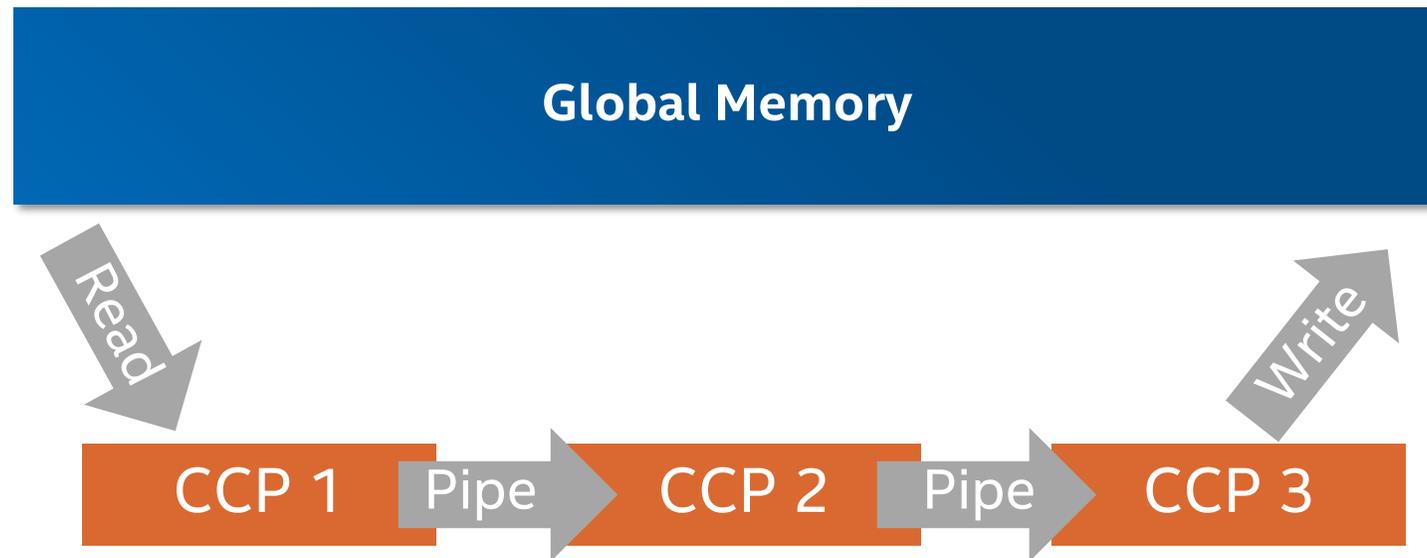
Note: Let the compiler try on it's own first.
It's very good at inferring an optimal structure!

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.

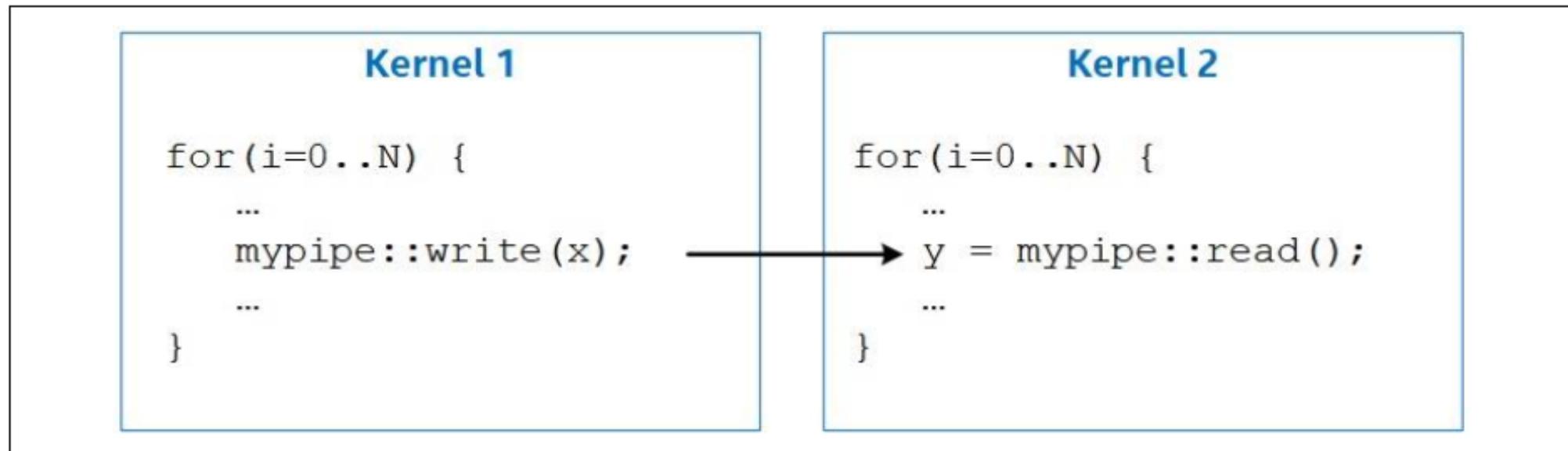
Pipes – Element the Need for Some Memory

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





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

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

Optimization Report – Throughput Analysis

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

The screenshot shows a web-based optimization report for a file named `hough_transform.cpp`. The report is titled "Reports" and includes a "Loops Analysis" section with a checkbox for "Show fully unrolled loops" which is checked. The analysis table shows the following data:

	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

Below the table is a "Details" section for `const::Hough_transform_kernel.B3:` with the following bullet points:

- 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

The code snippet on the right shows the following relevant lines:

```
97-   Hough_transform_kernel->[=]({
98-       for (uint y=0; y<HEIGHT; y++) {
99-           for (uint x=0; x<WIDTH; x++){
100-               unsigned short int increment = 0;
101-               if (_pixels[(WIDTH*y)+x] != 0) {
102-                   increment = 1;
103-               } else {
104-                   increment = 0;
105-               }
106-               for (int theta=0; theta<THETAS;
107-                   theta++){
108-                   int rho = x*_cos_table[theta] +
109-                       y*_sin_table[theta];
110-                   _accumulators[(THETAS*(rho+RHOS
111-                               )+theta] += increment;
112-               }
113-           }
114-       }
115-   });
```

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

The screenshot shows a web browser window titled "Report: fpga_970fa3 - Mozilla Firefox". The page displays an "Area Analysis of System" report. The report includes a table with columns for "ALUTs" and "FFs". The table shows "Function overhead" with 1338 ALUTs and 2411 FFs, and a highlighted "Private Variable: - 'theta' (hough_transform.cpp:105)" with 27 ALUTs and 43 FFs. To the right of the table is a code editor showing a snippet from "hough_transform.cpp" with lines 98-112. The code includes a loop for (int theta=0; theta<THETAS; theta++){...}. Below the code editor is a "Details" section for the private variable, listing its type as "Register" and providing two suggestions for resolution: "1 register of width 9 and depth 342 (depth was increased by a factor of 339 due to a loop initiation interval of 339.)" and "1 register of width 32 and depth 342 (depth was increased by a factor of 339 due to a loop initiation interval of 339.)".

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

```
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;
106-            theta++){
107-           int rho = x*_cos_table[theta] + y
108-               *_sin_table[theta];
109-           _accumulators[(THETAS*(rho+RHOS
110-               ))+theta] += increment;
111-       }
112-   };
```

Details

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

The screenshot displays the Intel System Viewer Graph Viewer interface. The browser window shows the report URL: `file:///home/student/fpga_trn/OCL_19_1/SimpleKernel/reports/report.html#view4`. The navigation tabs include Reports, Summary, Throughput Analysis, Area Analysis, and System Viewers. The System Viewer section shows a graph with a tooltip for a 'Store' component. The tooltip details are:

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

The code editor on the right shows the SimpleKernel.cl source code:

```
1 //ACL Kernel
2 _kernel
3 void SimpleKernel(__global const float * restrict in,
4                  __global const float * restrict in2, __global
5                  float * restrict out, uint N)
6 {
7     //Perform the Math Operation
8     for (uint index = 0; index < N; index++)
9         out[index] = in[index] * in2[index];
10 }
```

The details table at the bottom provides a structured view of the store's properties:

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

The screenshot displays the Intel Schedule Viewer interface. At the top, there are tabs for 'Reports', 'Summary', 'Throughput Analysis', 'Area Analysis', and 'System Viewers'. The main window is titled 'Schedule Viewer (alpha)' and shows a Gantt chart for the file 'hough_transform.cpp'. The chart has two x-axes: 'Cluster instruction schedule cycle' (0 to 6) and 'Absolute clock cycle' (2 to 8). The y-axis lists components: '...ter 0', 'i', '+', '...Comp.', and 'Xor'. A blue bar for '...ter 0' spans from cycle 0 to 6. Other components have shorter bars: 'i' from cycle 3 to 4, '+' from cycle 4 to 5, '...Comp.' from cycle 4 to 5, and 'Xor' from cycle 4 to 5. On the left, a 'Schedule List (alpha)' tree shows a hierarchy: 'System' > '_ZTSZZ4mai' > 'const::Hot' (multiple instances) > 'Cluster' > 'const::Hot' (multiple instances) > 'Cluster'. On the right, a code editor shows the source code for 'hough_transform.cpp', including headers, constants, and a main function.

```
1 #include <vector>
2 #include <CL/sycl.hpp>
3 #include <CL/sycl/intel/fpga_extensions.hpp>
4 #include <chrono>
5
6 // This file defines the sin and cos values for each degree up to 180
7 #include "sin_cos_values.h"
8
9 #define WIDTH 180
10 #define HEIGHT 120
11 #define IMAGE_SIZE WIDTH*HEIGHT
12 #define THETAS 180
13 #define RHOS 217 //Size of the image diagonally: (sqrt(180^2+120^2))
14 #define NS (1000000000.0) // number of nanoseconds in a second
15
16 using namespace std;
17 using namespace cl;
18
19 // This function reads in a bitmap and outputs an array of pixels
20 void read_image(char *image_array);
21
22 class Hough_Transform_kernel;
23
24 int main() {
25     //Declare arrays
26     ...
```

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

The screenshot displays the HTML Kernel Memory Viewer interface. At the top, there are navigation tabs: Reports, Summary, Throughput Analysis, Area Analysis, and System Viewers. The main area is divided into three sections:

- Memory List:** A tree view showing the system hierarchy. The selected item is 'accum' under the path 'System > _ZTSZZ4mai > _arg_ > AccessRa > AccessRa > accum'.
- Memory Viewer:** A diagram showing the memory architecture. It features two 'LD' (Load) and two 'ST' (Store) ports on the left, each connected to a 'SHARE' block. These 'SHARE' blocks are connected to a central 'accum_local Bank 0' block, which contains a read port 'R' and a write port 'W'.
- Code Snippet:** A vertical list of assembly instructions on the right, including '#ir', '#de', '#si', '#li', and '#int'.

Below the main viewer, there is a 'Details' section for the selected 'accum_local' memory:

accum_local:	
Requested size	156240 bytes
Implemented size	256 kilobytes = $2^{\text{ceil}(\log_2(\text{Req}))}$
Number of banks	1
Bank width (word size)	16 bits
Bank depth	131072 words



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**

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, absolute, 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

- 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

Tree-Balancing

- Floating-point operations are not associative
 - Rounding after each operation affects the outcome
 - i.e. $((a+b) + c) \neq (a+(b+c))$
- By default the compiler doesn't reorder floating-point operations
 - May create 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>

Lab: Optimizing the Hough Transform Kernel

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

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.

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.

intel®