Using Intel® oneAPI Toolkits with FPGAs*

Prof. Ricardo Menotti (menotti@ufscar.br)
Federal University of Sao Carlos (UFSCar)

*Special thanks to Susannah Martin for the material and support

Tutorial Objectives

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

TUTORIAL AGENDA

The Basics

The oneAPI Toolkit

Introduction to Data Parallel C++

Lab: Overview of DPC++

Using FPGAs with the Intel® oneAPI Toolkits

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

Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

Lab: Practice the FPGA Development Flow

Optimizing Your Code for FPGAs

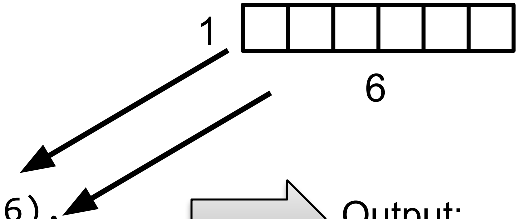
Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Lab: Optimizing the Hough Transform Kernel

KERNEL Model

parallel_for(num_work_items)

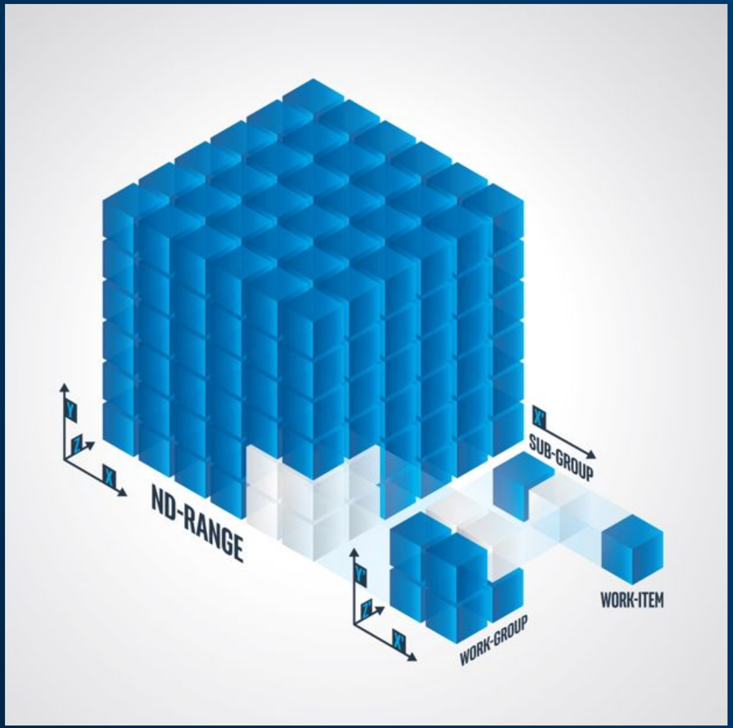
- Execute kernel in parallel over a 1, 2, or 3 dimensional index space
- Work-item can query ID and range of invocation (num_work_items)




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

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

Output:

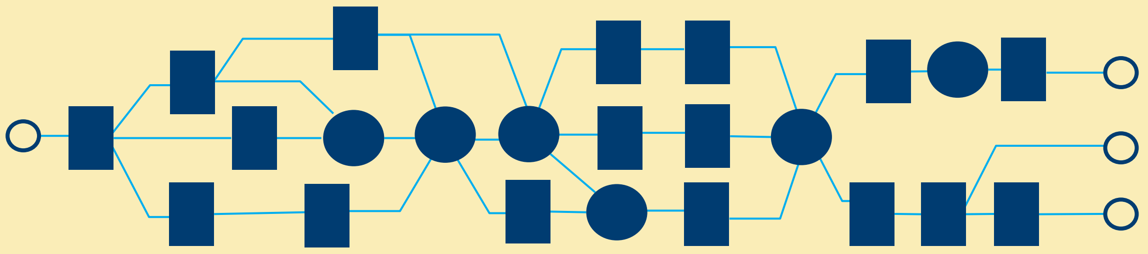
```
id<1>{ 0 }
id<1>{ 1 }
id<1>{ 2 }
id<1>{ 3 }
id<1>{ 4 }
id<1>{ 5 }
```



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

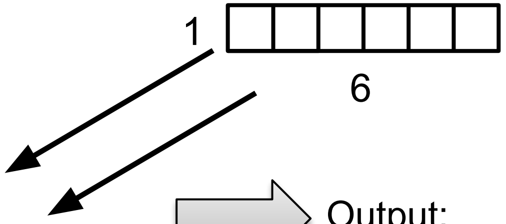
Copyright © 2019, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

KERNEL Model



single_task( )

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



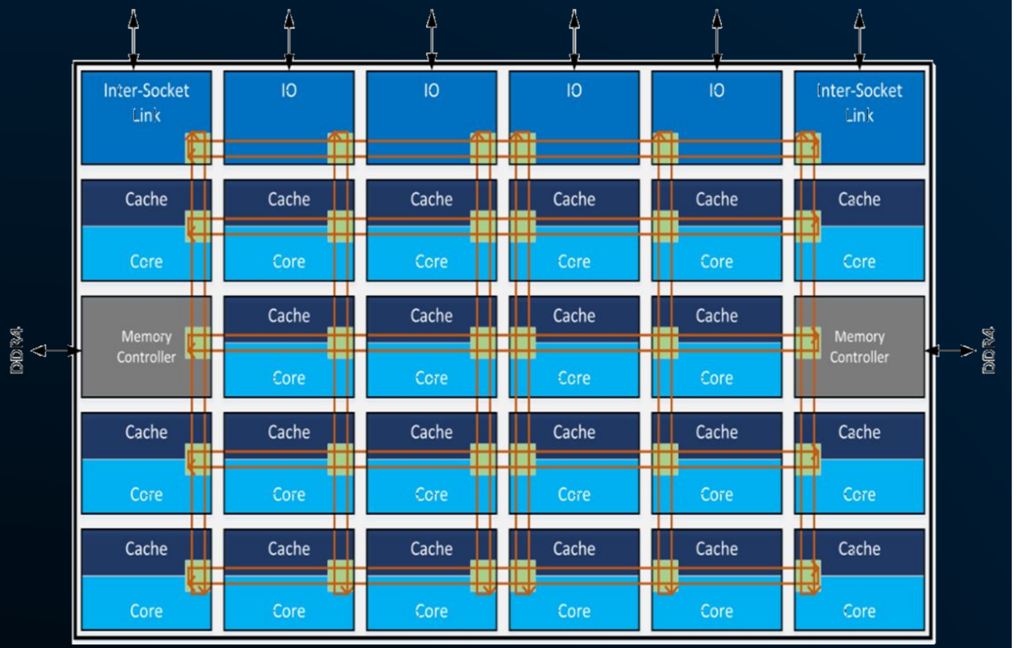

```
myQueue.submit([&](handler & cgh) {
    stream os(1024, 80, cgh);

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

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

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

Copyright © 2019, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

How it maps to CPU, GPU, FPGA

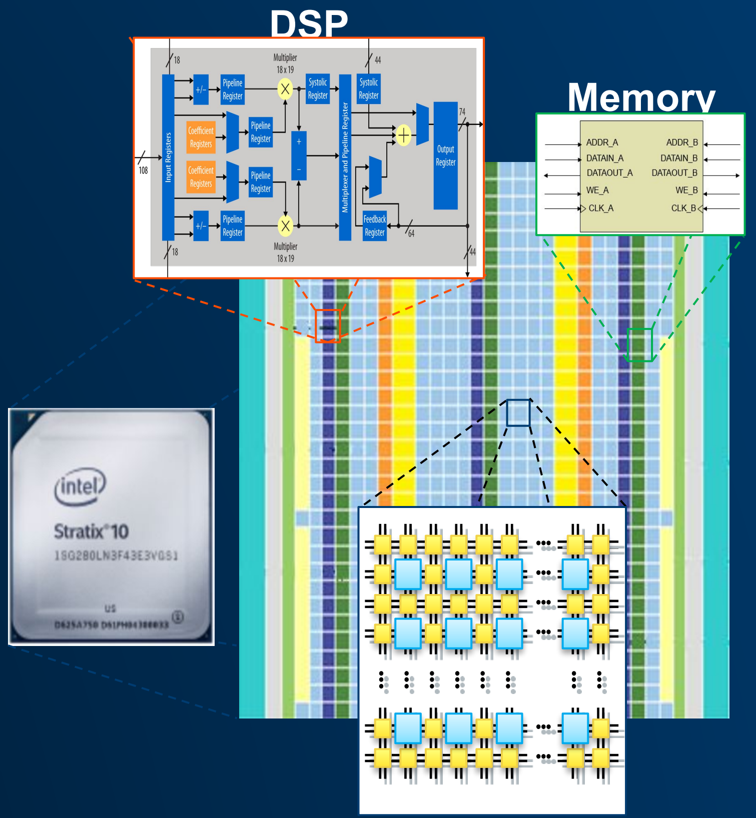


CPU

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

GPU

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



FPGA

- Custom Pipeline
- MULTI-CORE (pipeline)

Copyright © 2019, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.

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

A Brief Introduction

What is an FPGA?

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

“Field-Programmable Gate Array” (FPGA)

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

FPGA = Programmable Hardware

Reconfigurable Computing

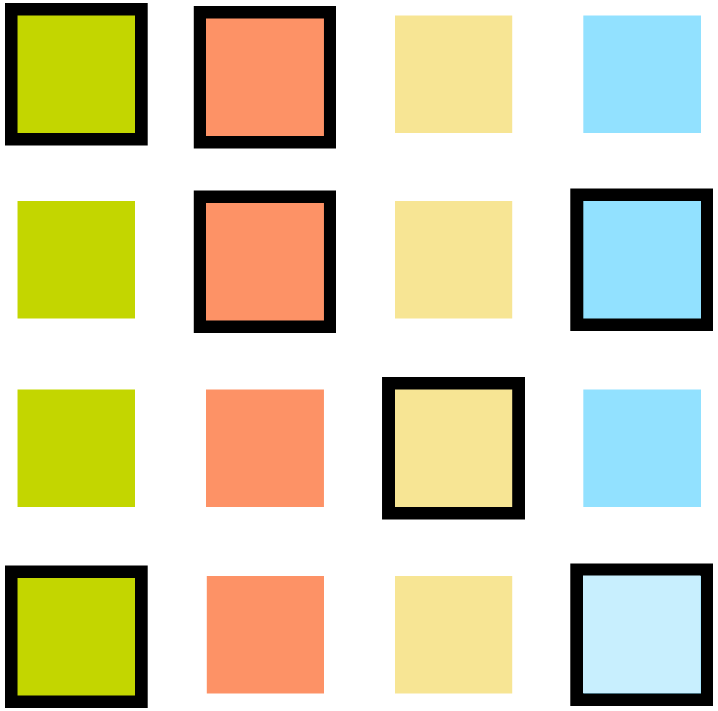
How an FPGA Becomes What You Want It To Be

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

How an FPGA Becomes What You Want It To Be

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

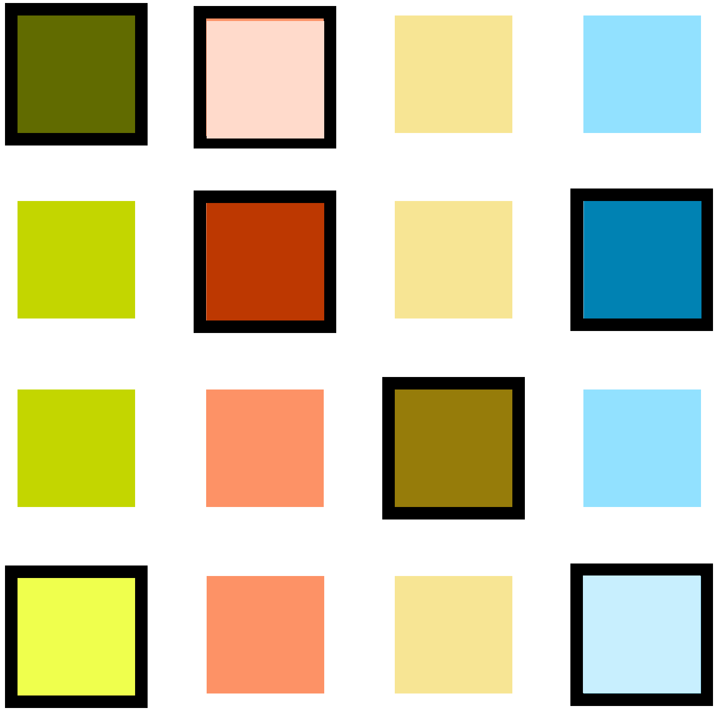
- The building blocks you **choose**



How an FPGA Becomes What You Want It To Be

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

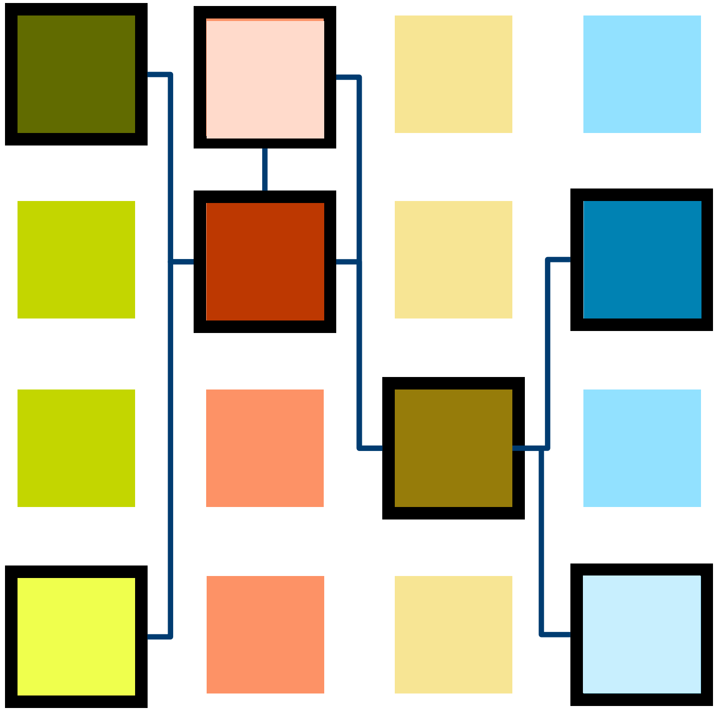
- The building blocks you **choose**
- How you **configure** them



How an FPGA Becomes What You Want It To Be

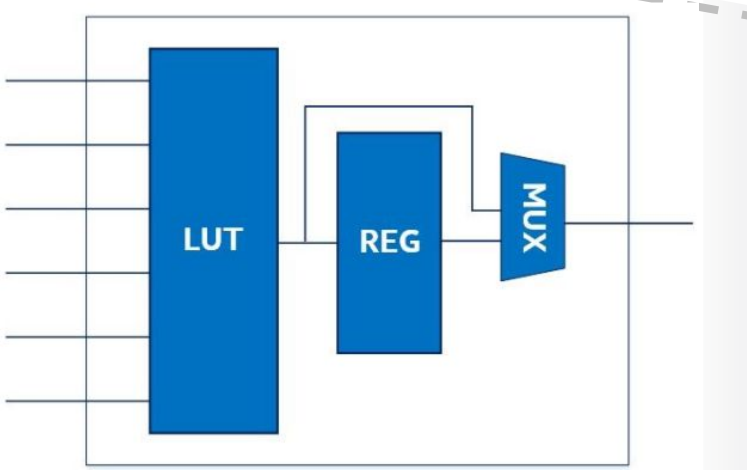
The FPGA is made up of small building blocks of logic and other functions

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

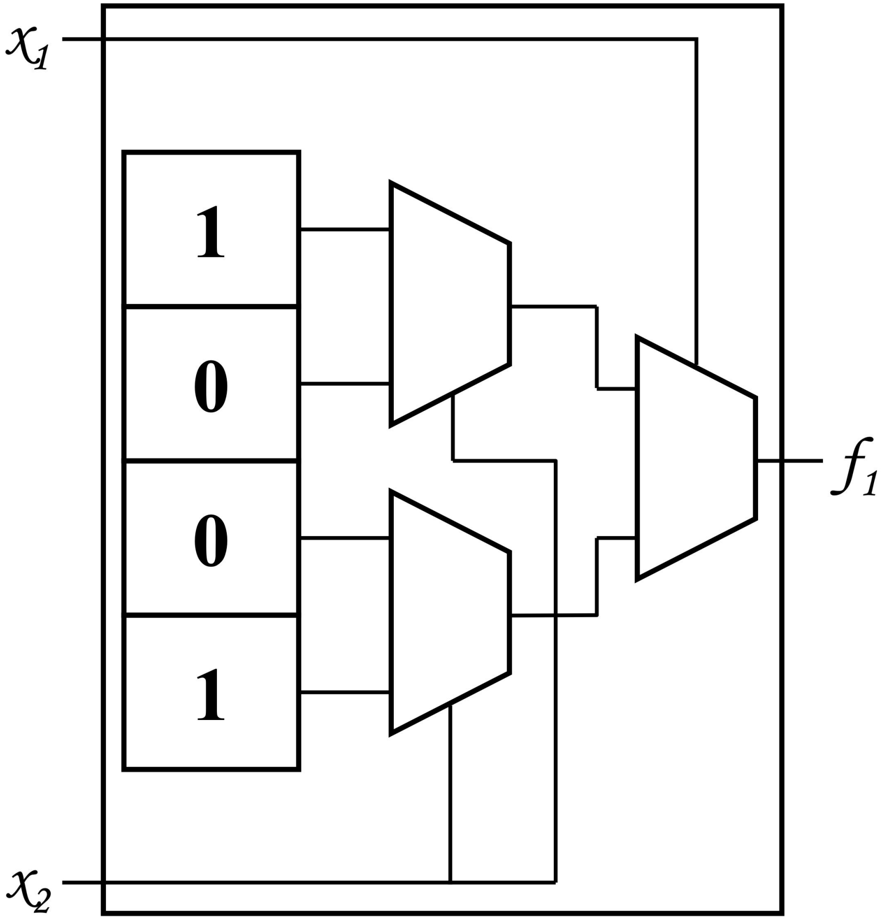


Determine what function the FPGA performs

Blocks Used to Build What You've Coded

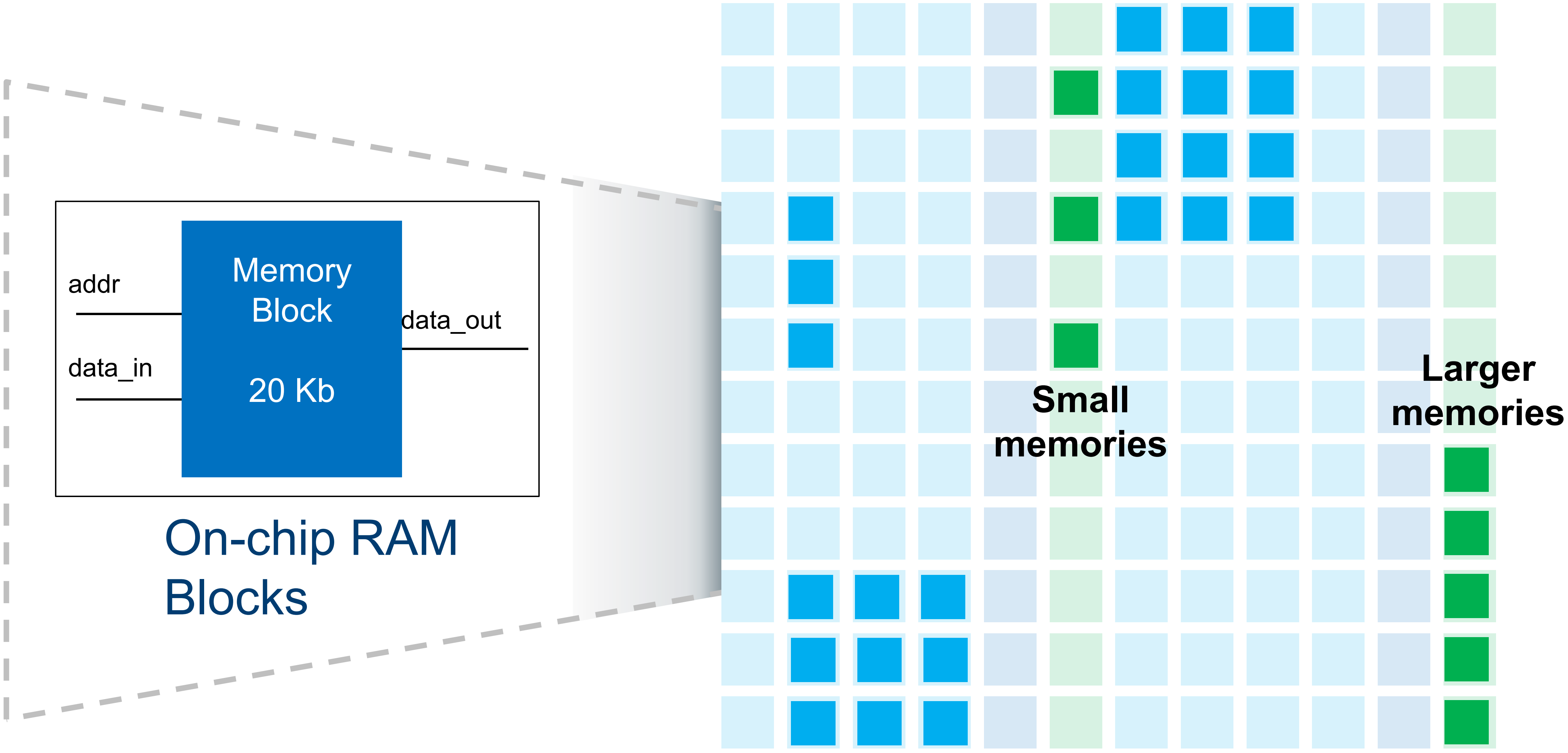


Look-up Tables
and Registers

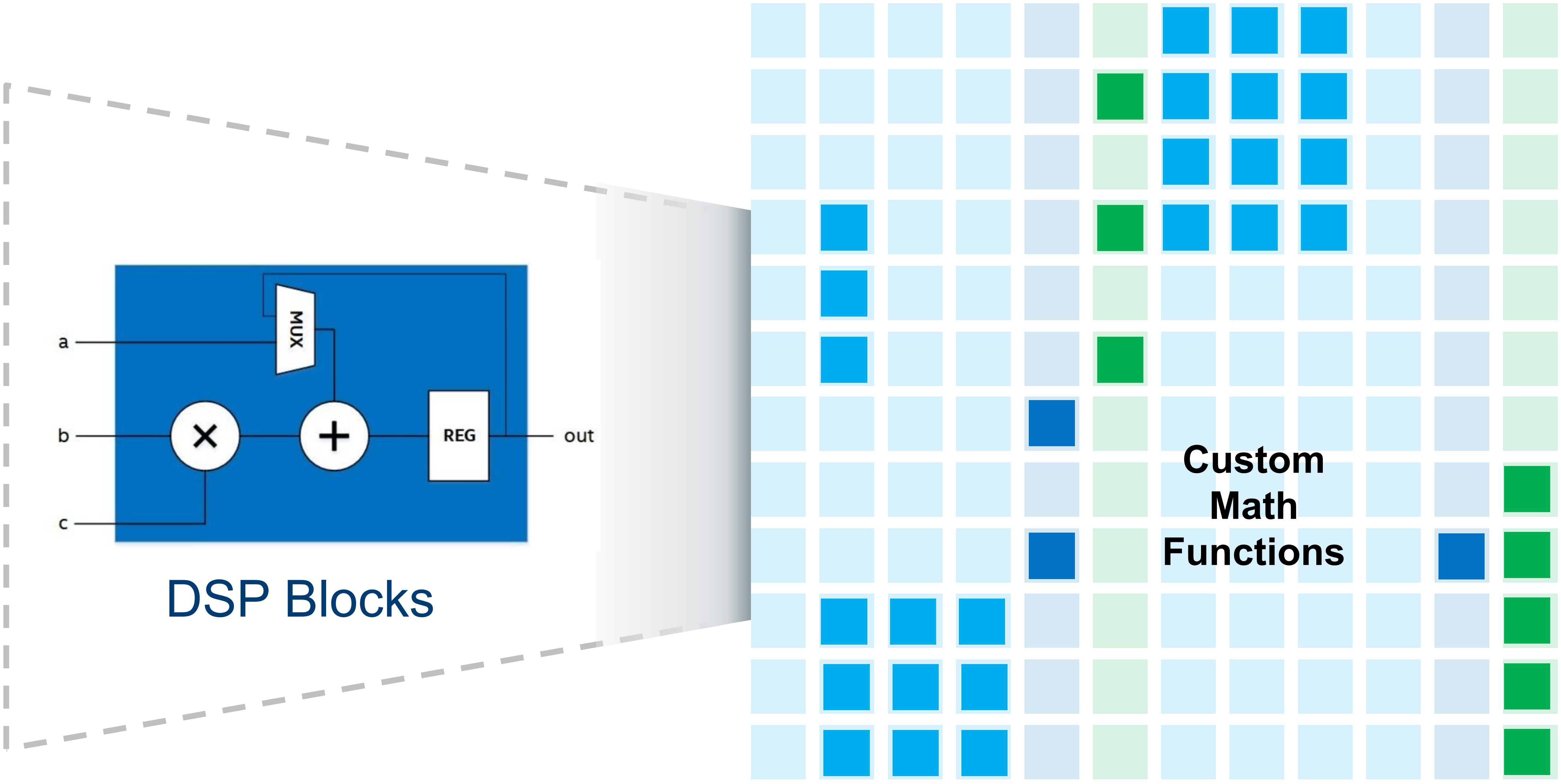


tate
e
encode

Blocks Used to Build What You've Coded

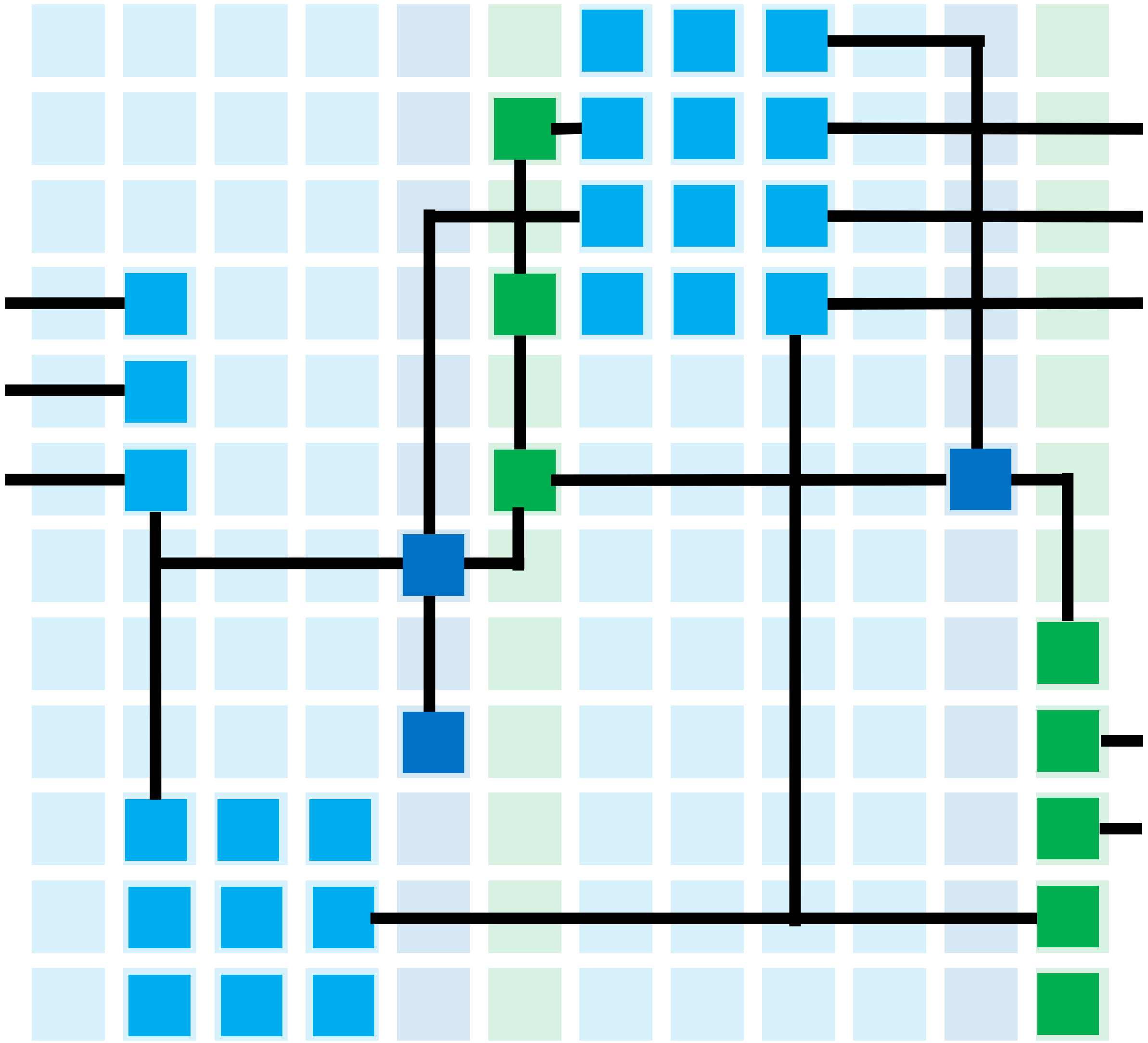


Blocks Used to Build What You've Coded

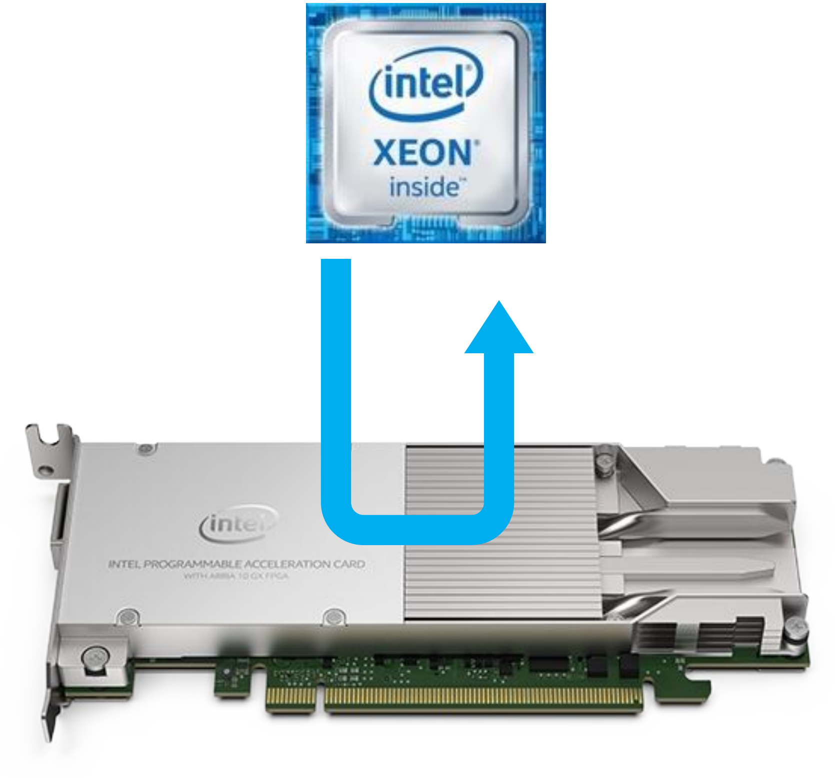


Then, It's All Connected Together

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



What About Connecting to the Host?

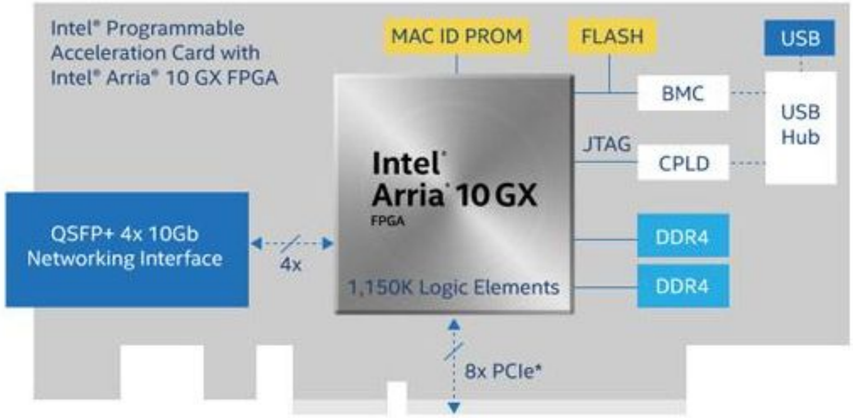


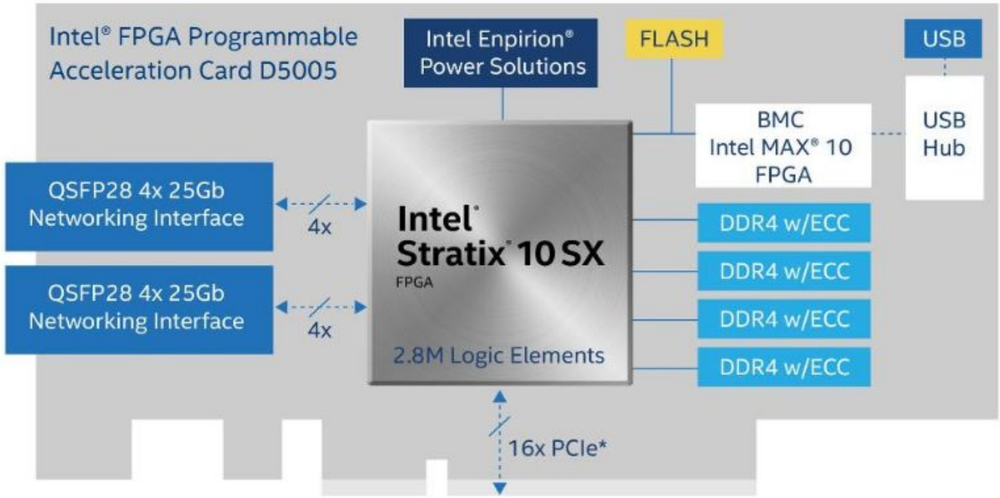
Accelerated functions run on a PCIe attached FPGA card

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

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

Intel® FPGAs Available





Why should I care about programming for an FPGA?

It all comes down to the advantage of custom hardware.

First, some impressive examples…

IMAGE LOSSLESS COMPRESSION:
ACCELERATING PERFORMANCE

GENOMICS SEQUENCING

intel select solution

Sample FPGA Workloads

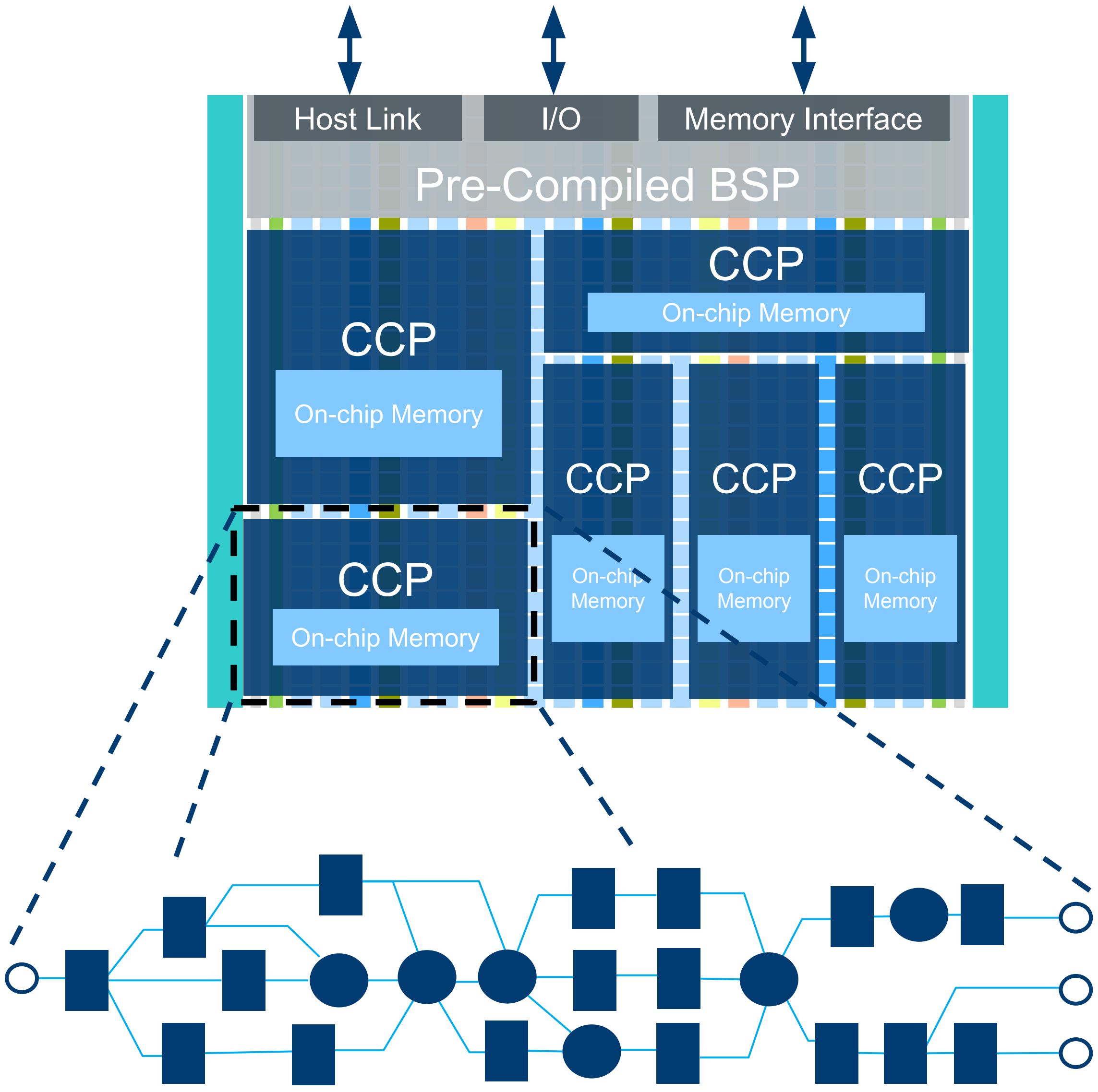
KEY VALUE STORE
ACCELERATING THROUGHPUT

DATABASE ANALYTICS
ACCELERATION

Code to Hardware: An Introduction

Intel® FPGAs

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

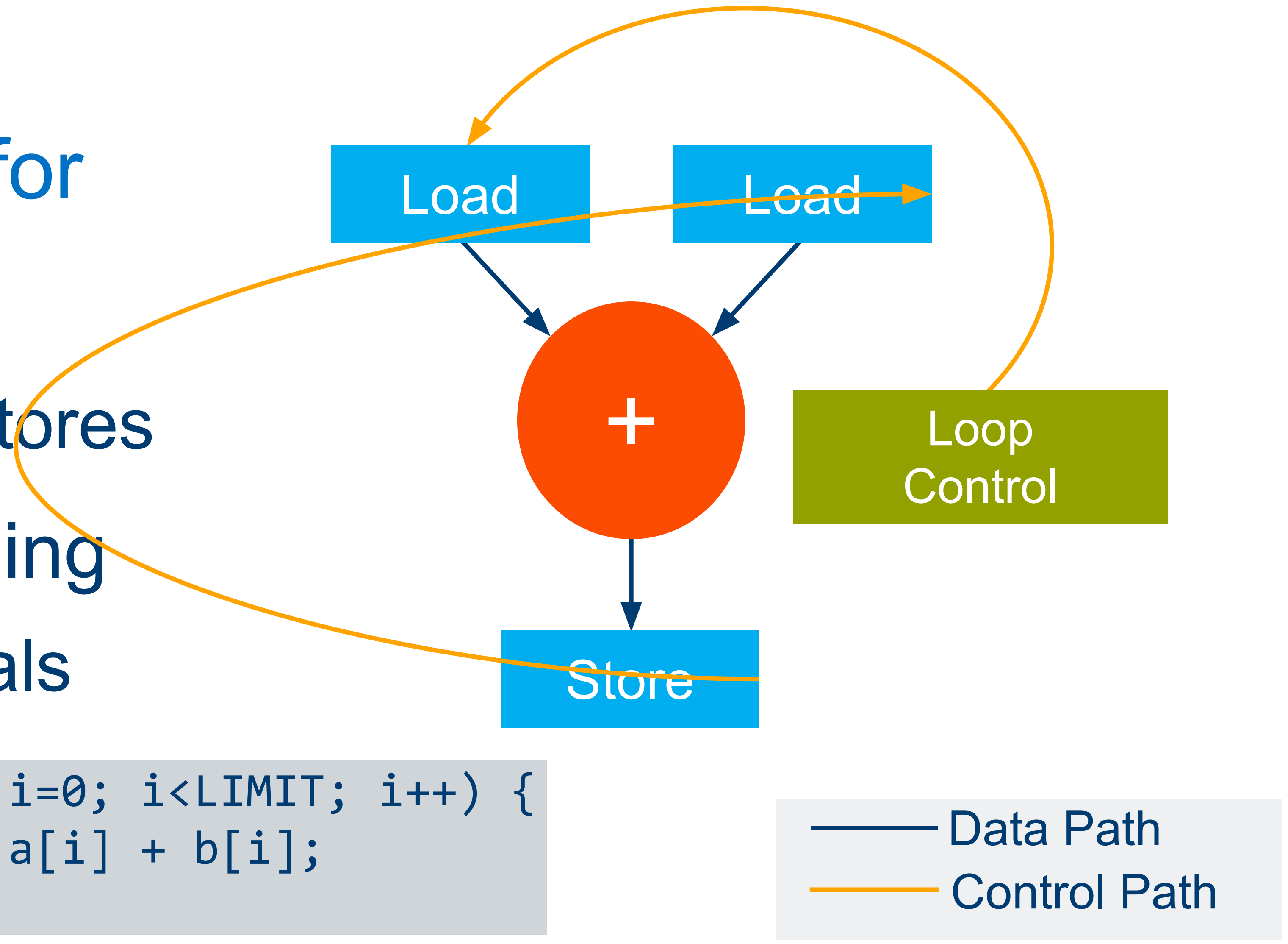


Custom Compute Pipeline

How Is a Pipeline Built?

Hardware is added for

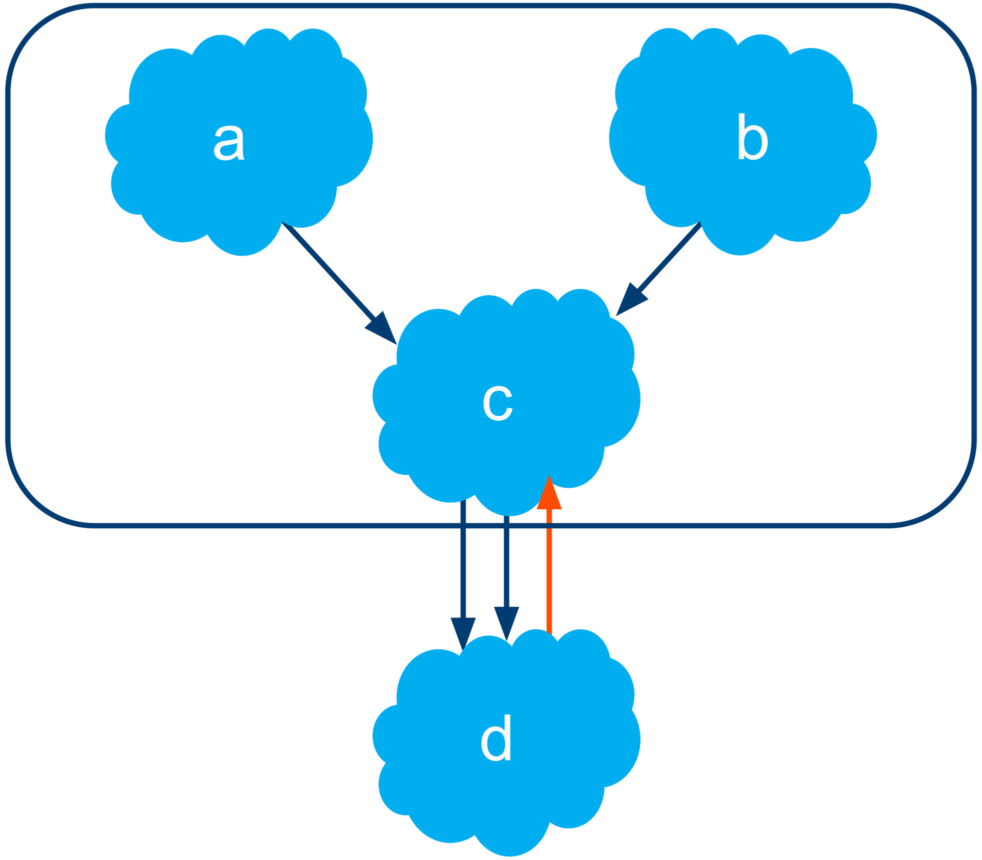
- Computation
- Memory Loads and Stores
- Control and scheduling
  – Loops & Conditionals




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

Connecting the Pipeline Together

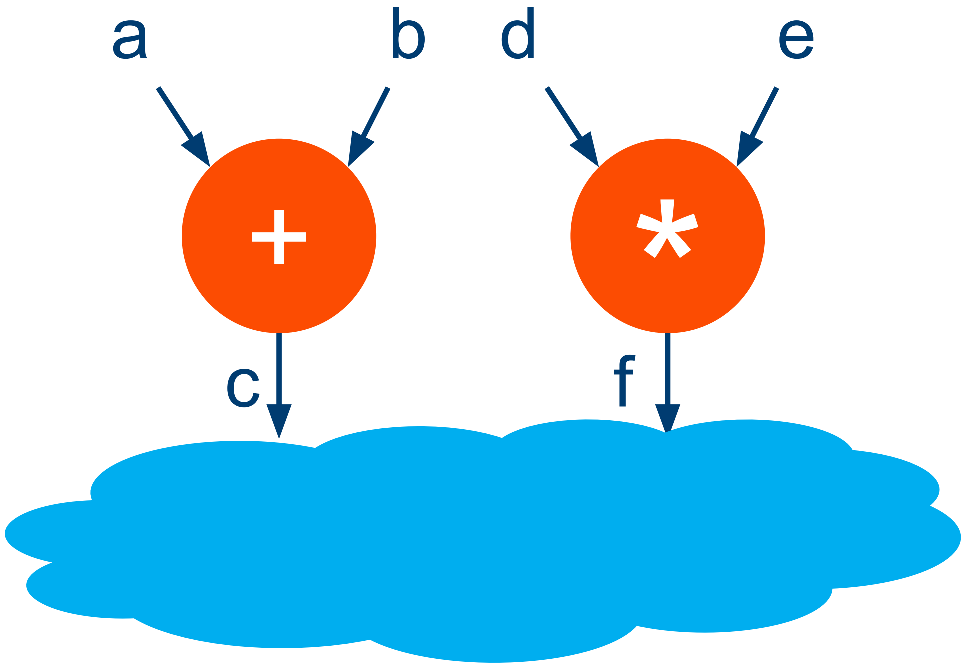
- Handshaking signals for variable latency paths
- Operations with a fixed latency are clustered together
- Fixed latency operations improve
  - Area: no handshaking signals required
  - Performance: no potential stalling due to variable latencies



Simultaneous Independent Operations

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

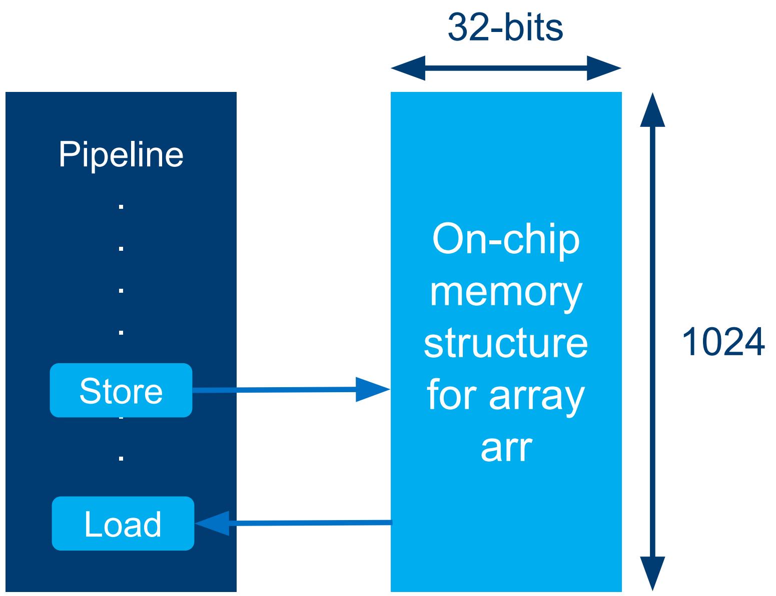
```
c = a + b;
f = d * e;
```



On-Chip Memories Built for Kernel Scope Variables

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

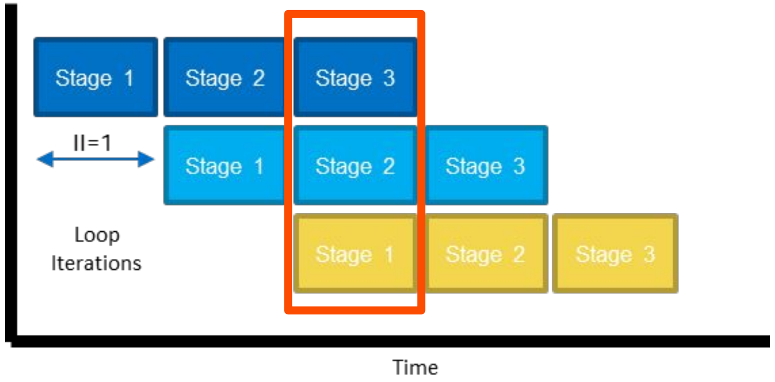
```
//kernel scope
cgh.single_task<>([=]() {
  int arr[1024];
  …
  arr[i] = …; //store to memory
  …
  … = arr[j] //load from memory
  …
} //end kernel scope
```

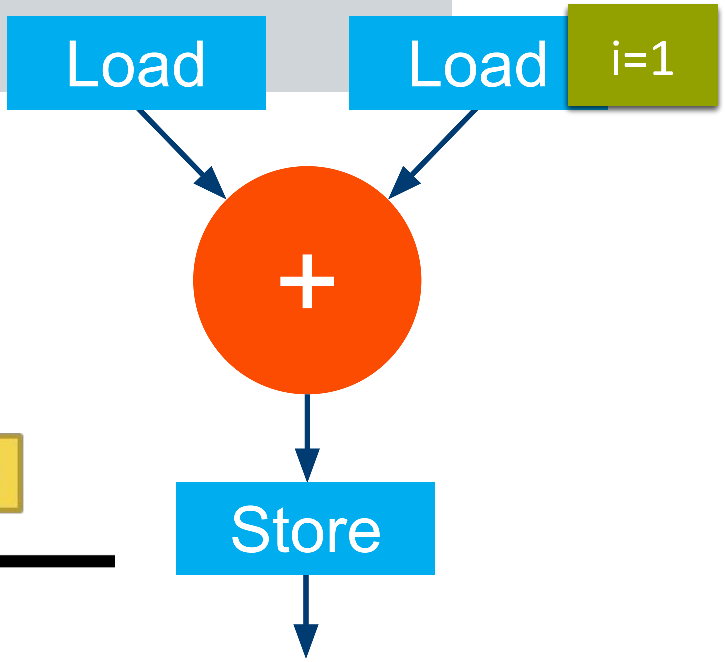


Pipeline Parallelism for Single Work-Item Kernels

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

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



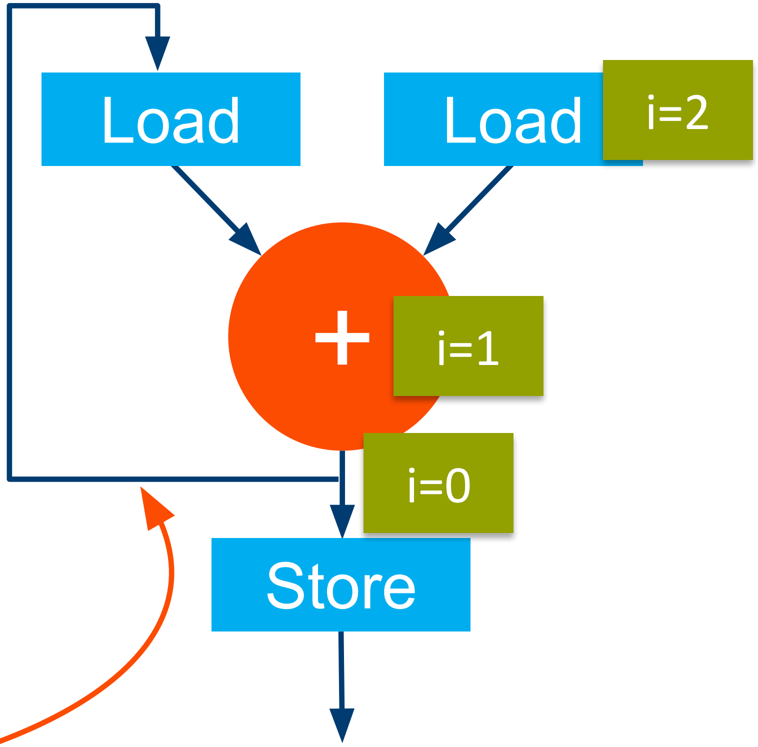


Dependencies Within the Single Work-Item Kernel

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

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

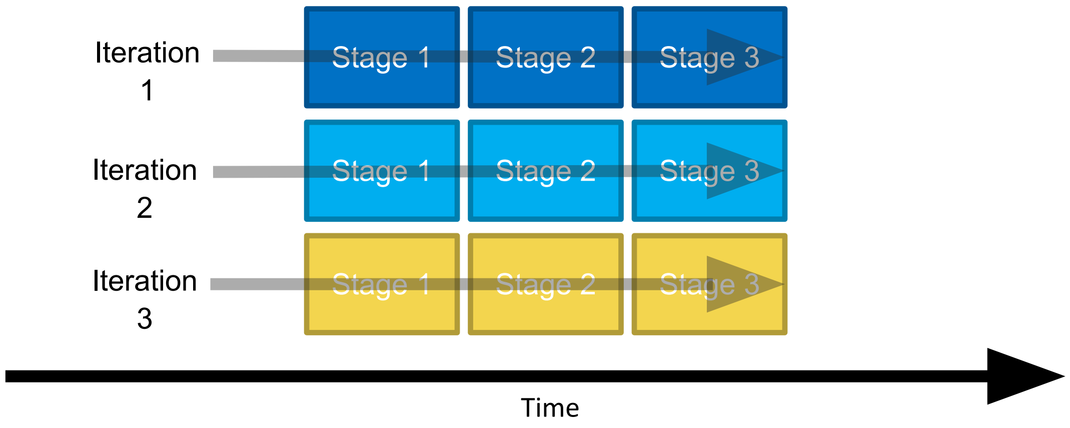
```
handle.single_task<>([=]() {
  int b = 0;
  for (int i=0; i<LIMIT; i++) {
    b += a[i];
  }
});
```



How Do I Use Tasks and Still Get Data Parallelism?

The most common technique is to unroll your loops

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

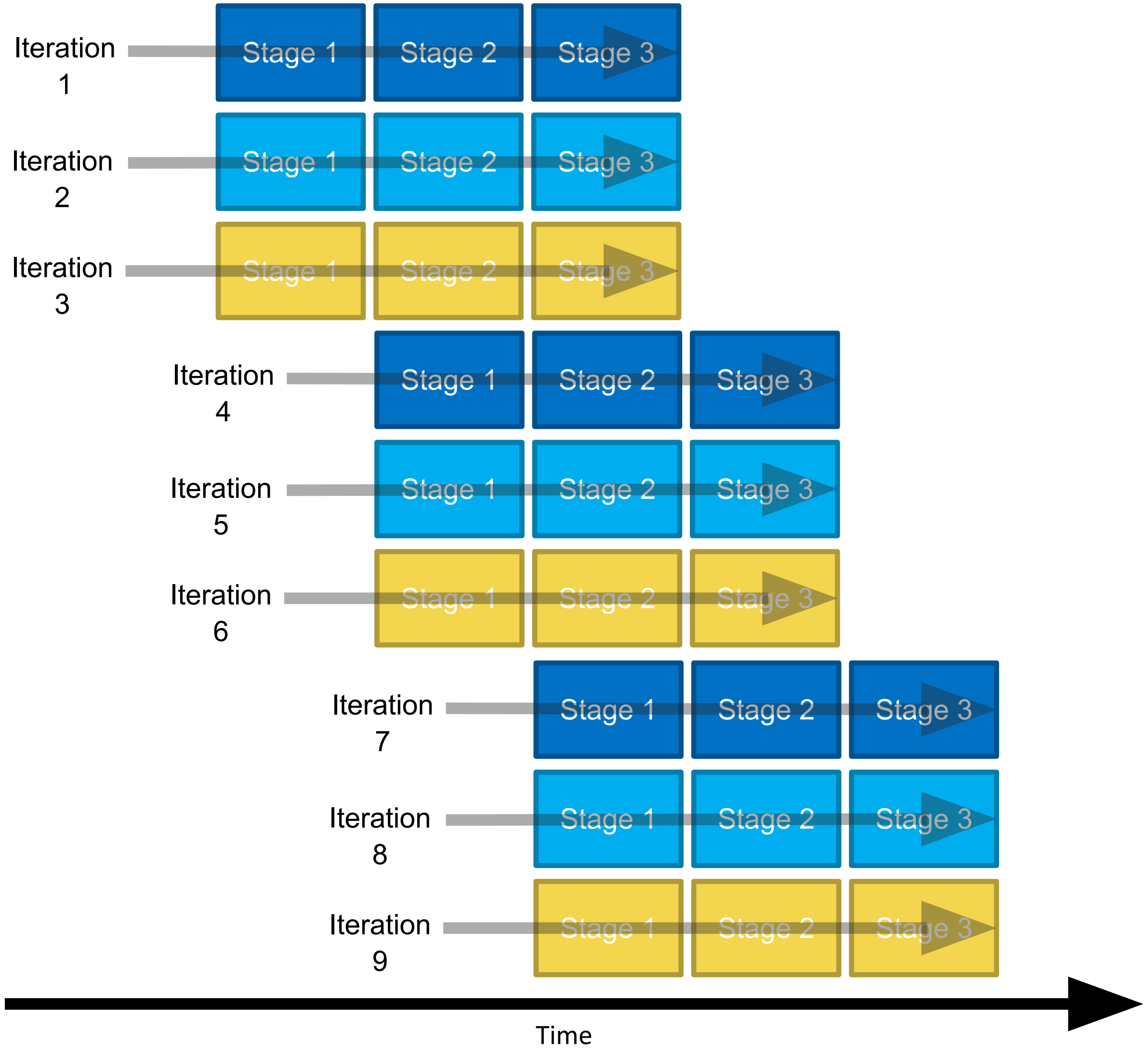


Unrolled Loops Still Get Pipelined

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

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

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



What About Task Parallelism?

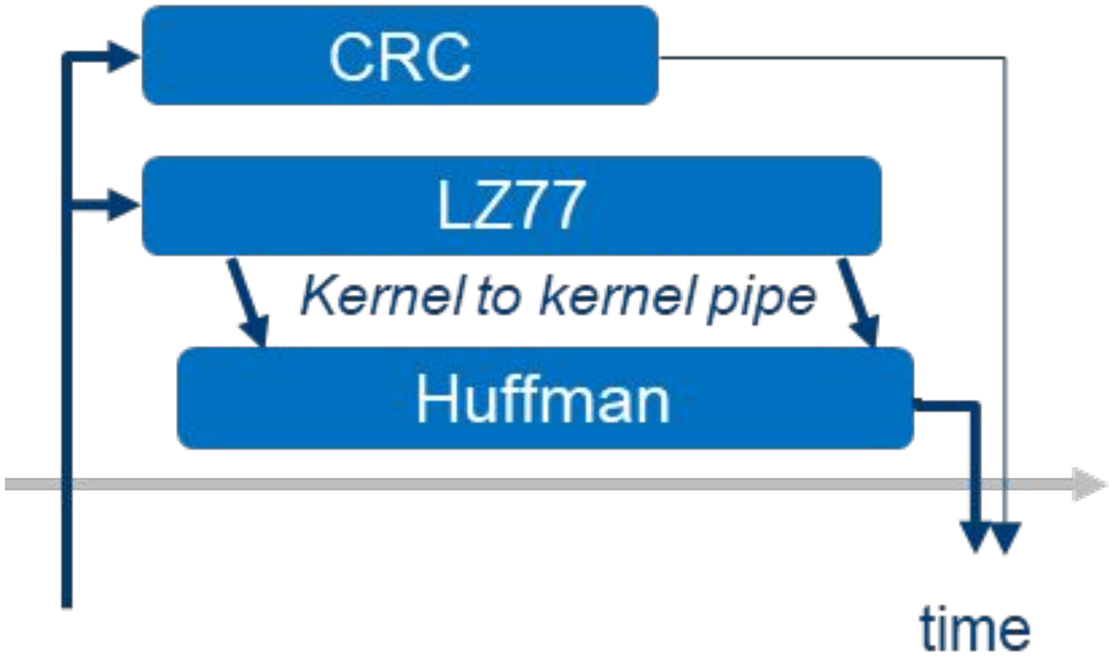
FPGAs can run more than one kernel at a time

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

Data can be passed between kernels using pipes

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

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



So, Can We Build These? NDRange Kernels

- Kernels launched parallel_for() or parallel_for_work_group() with a NDRange/work-group size of >1

```
…//application scope

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

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

});

…//application scope
```

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

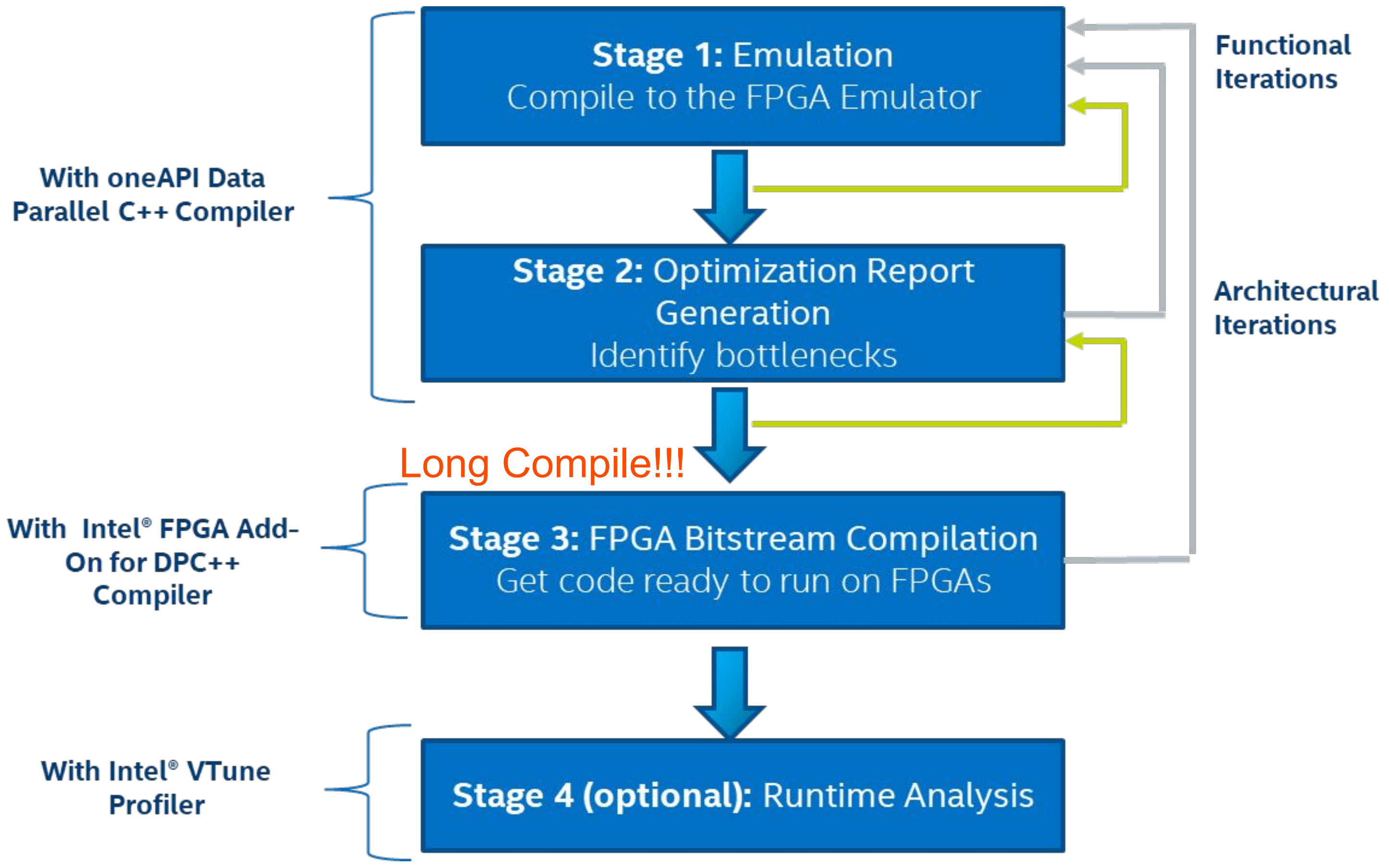
But, **tasks** usually imply more optimal pipeline structures.

The loop optimizations are limited for NDRange kernels.

Development Flow for Using FPGAs with the Intel® oneAPI Toolkits

FPGA Development Flow with oneAPI

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



Anatomy of a Compiler Command Targeting FPGAs

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

- Language
  DPCPP = Data Parallel C++
- Target Platform
- Input Files
  source or object
- Link Options
- FPGA-Specific Arguments

Emulation

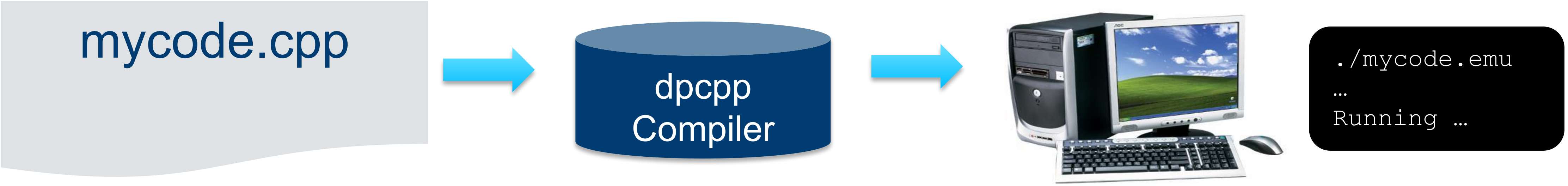
Get it Functional

Does my code give me the correct answers?

Emulation

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

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



Explicit Selection of Emulation Device

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

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

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

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

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

Using the Static Optimization Report

Get it Optimized

Where are the bottlenecks?

Compiling to Produce an Optimization Report

Two Step Method:

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

One Step Method:

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

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

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

– We will discuss more about the report later

FPGA Bitstream Compilation

Check Runtime Behavior

Check what you can't check
during static analysis

Compile to FPGA Executable with Profiler Enabled

Two Step Method:

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

One Step Method:

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

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

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

Compiling FPGA Device Separately and Linking

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

has_kernel.cpp

host_only.cpp

Partition code

This is the long compile

Then run this command to compile the FPGA image:

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

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

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

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

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

References and Resources

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

Lab: Practice the FPGA Development Flow

Introduction to Optimizing FPGAs with the Intel oneAPI Toolkits

Agenda

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

Reports

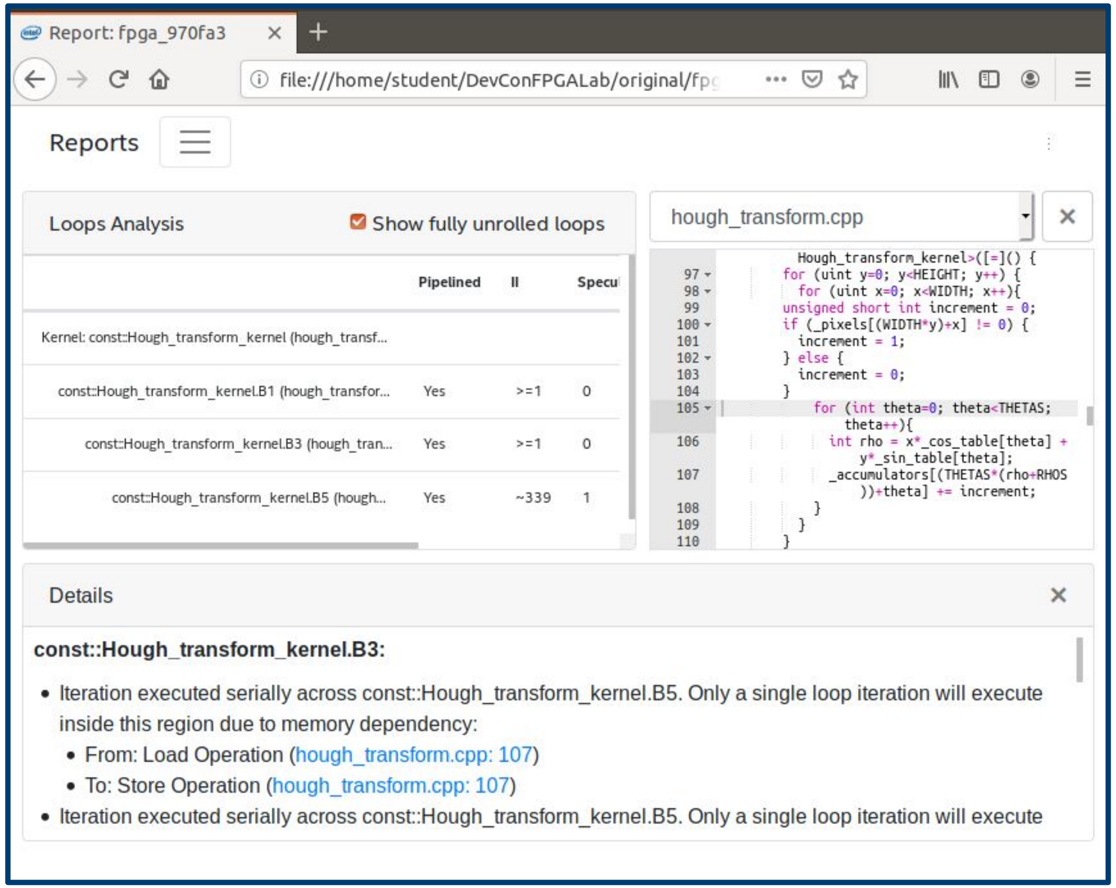
HTML Report

Static report showing optimization, area, and architectural information

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

Optimization Report – Throughput Analysis

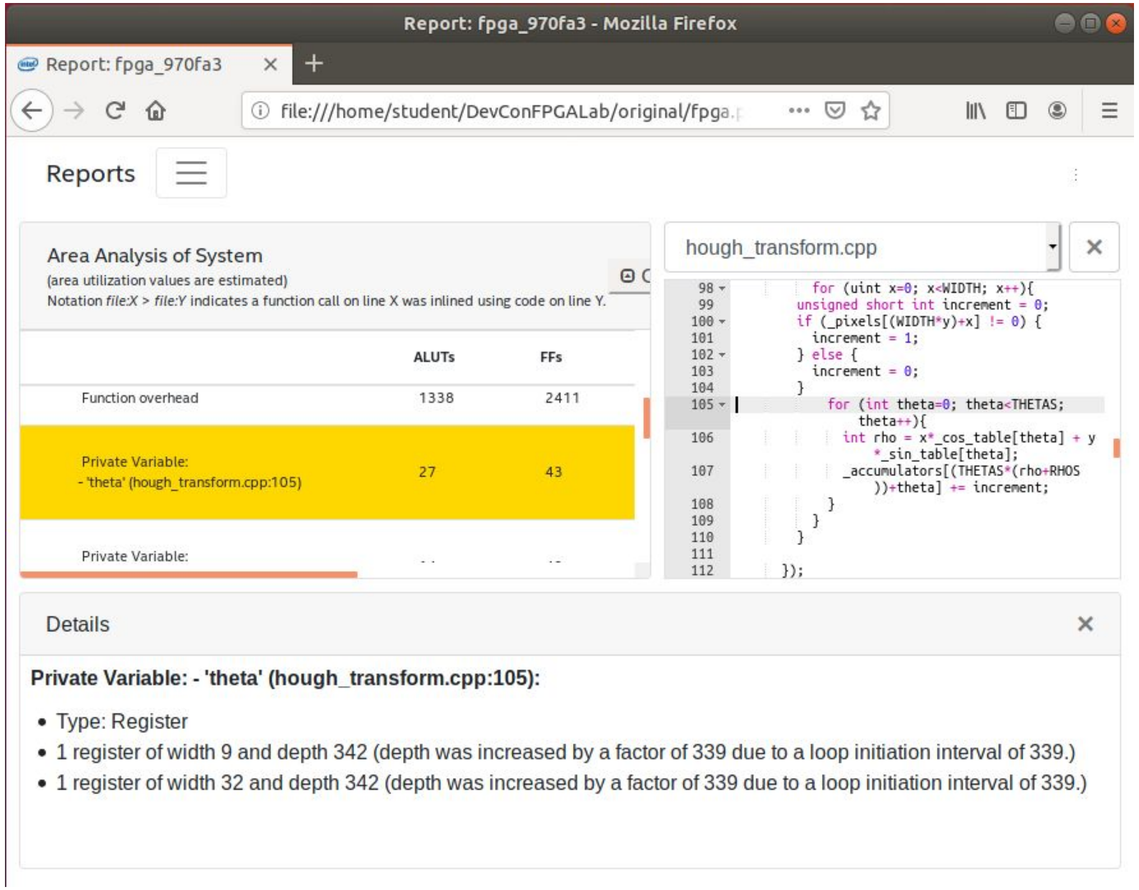
- Loops Analysis and Fmax II sections
- Actionable feedback on pipeline status of loops
- Show estimated Fmax of each loop



Optimization Report – Area Analysis

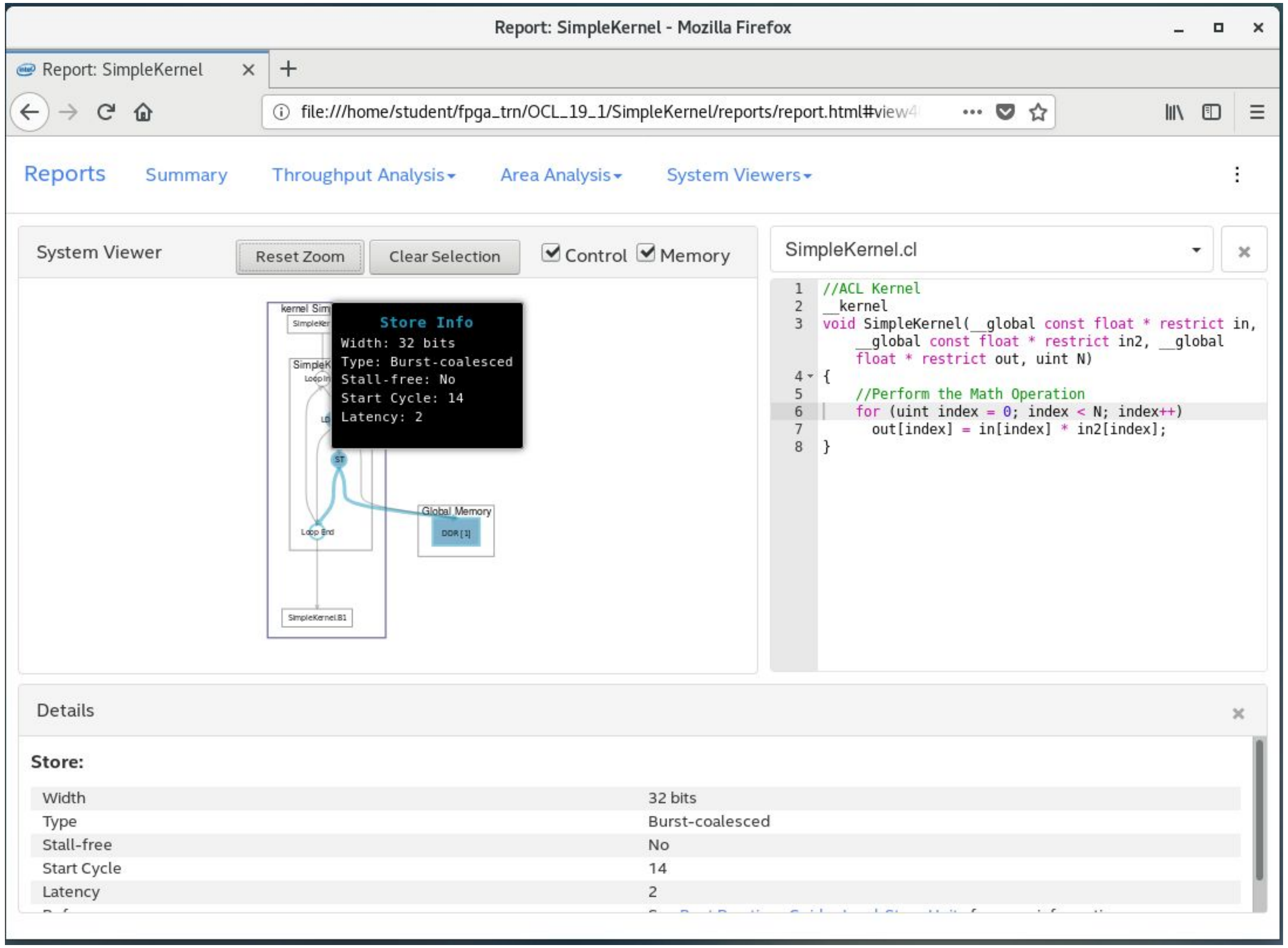
Generate detailed estimated area utilization report of kernel scope code

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



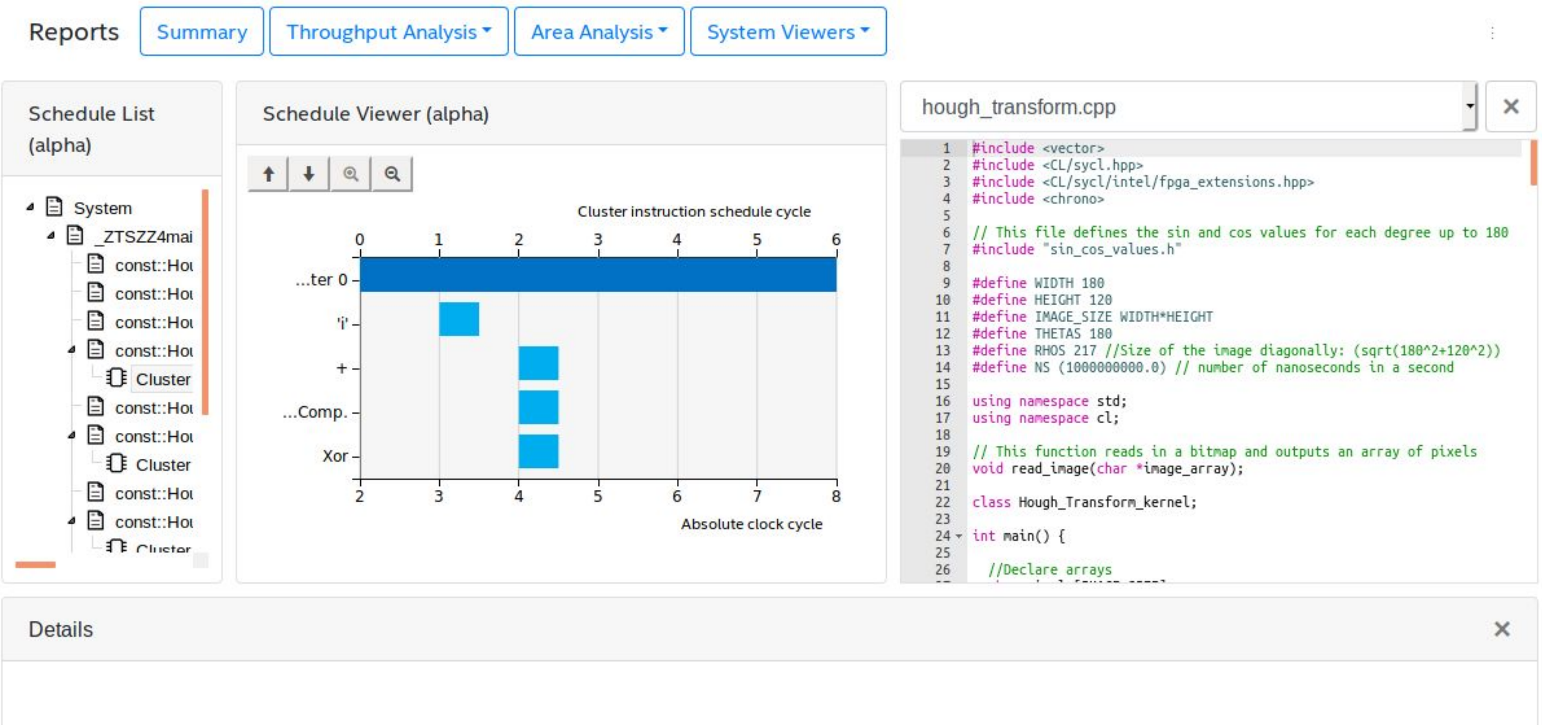
Optimization Report – Graph Viewer

- 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



Optimization Report – Schedule Viewer

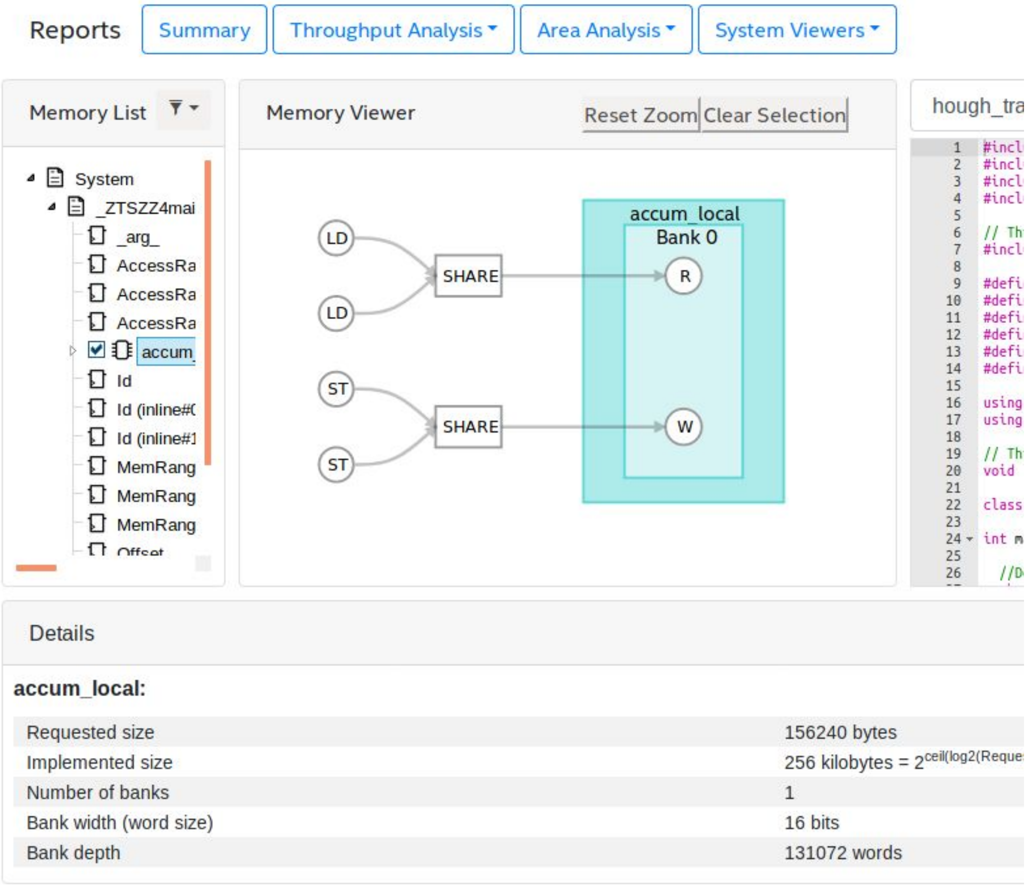
- Schedule in clock cycles for different blocks in your code



HTML Kernel Memory Viewer

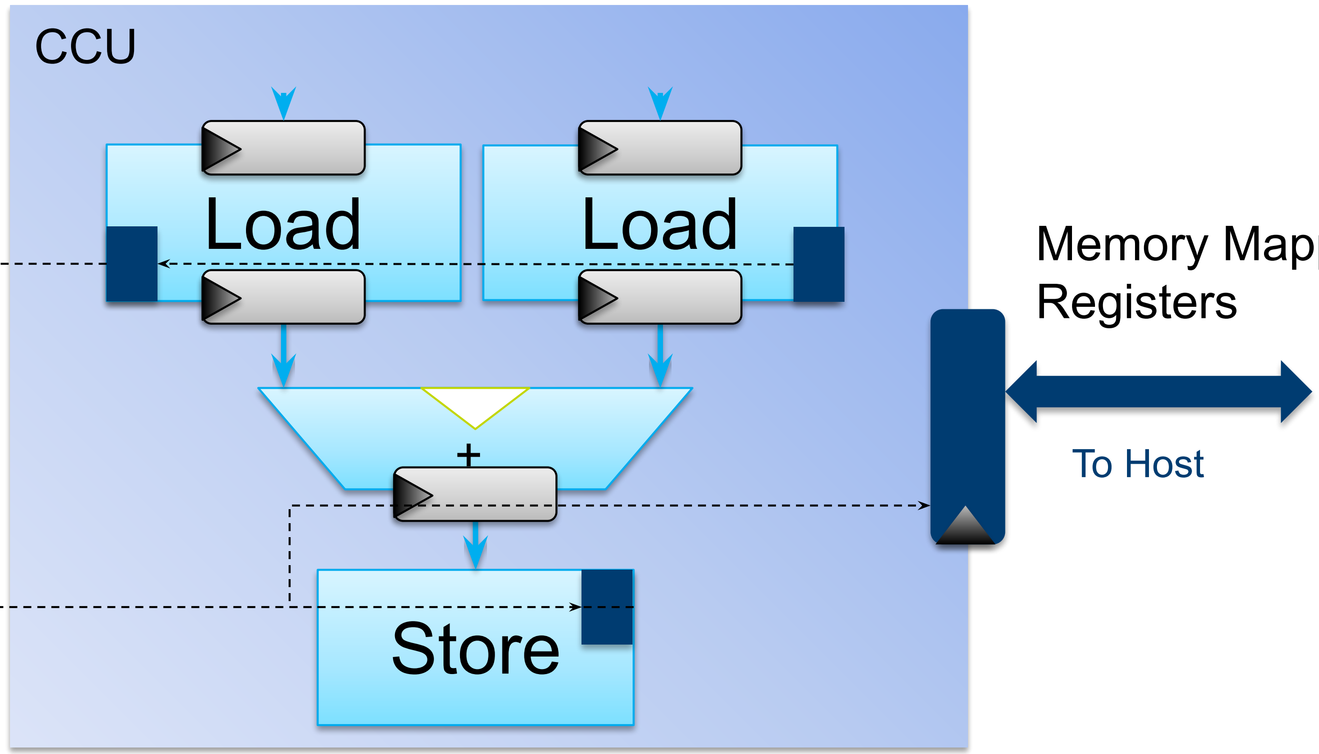
Helps you identify data movement bottlenecks in your kernel design. Illustrates:

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



Profiler

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



Collecting Profiling Data

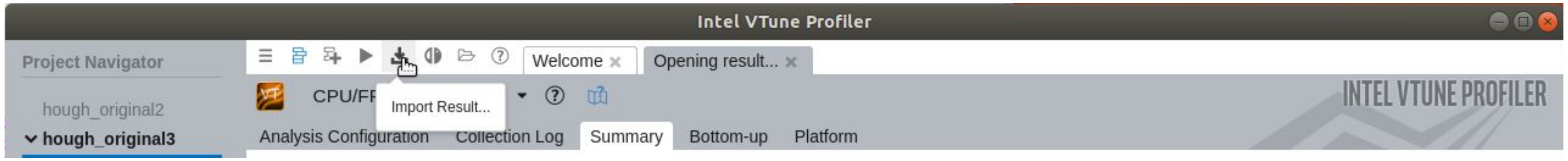
- Run the executable that integrates the kernel with the profiler using

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

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

Importing Profile Data into Intel® Vtune™ Profiler

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



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

Loop Optimization

Types of Kernels (Review)

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

Single Work-Item Kernels

- Single work items kernels are kernels that contain no reference to the work item ID.
- Usually launched with the group handler member function single_task().
- Or, launched with other functions and given a work-group/NDRange size of 1.
- Almost always contain an outer loop.

```
…//application scope

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

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

});

…//application scope
```

NDRange Kernels

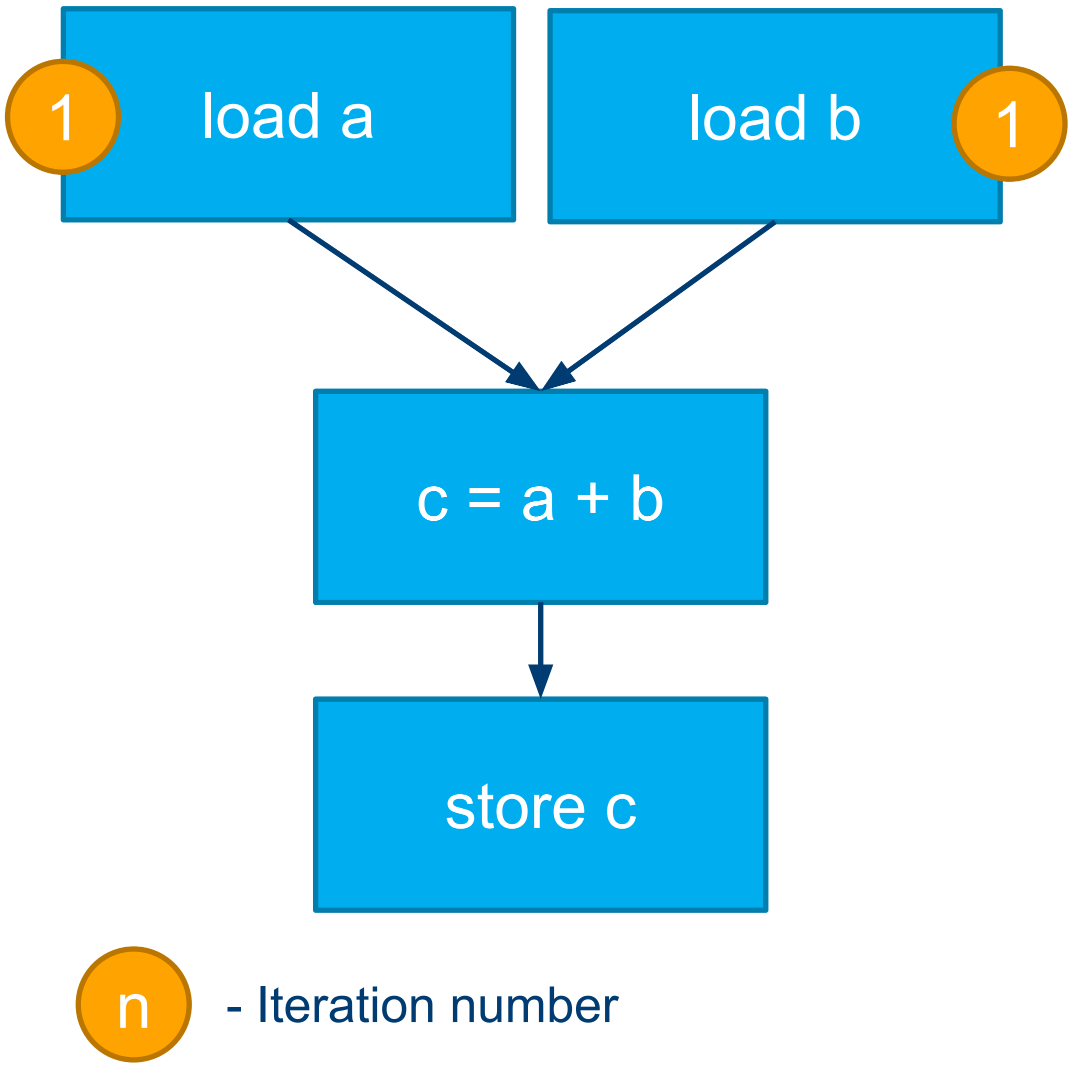
- Kernels launched with the command group handler member function parallel_for() or parallel_for_work_group() with a NDRange/work-group size of >1.
- Much of this section will not apply to NDRange kernels

```
…//application scope

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

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

});

…//application scope
```

Understanding Initiation Interval

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

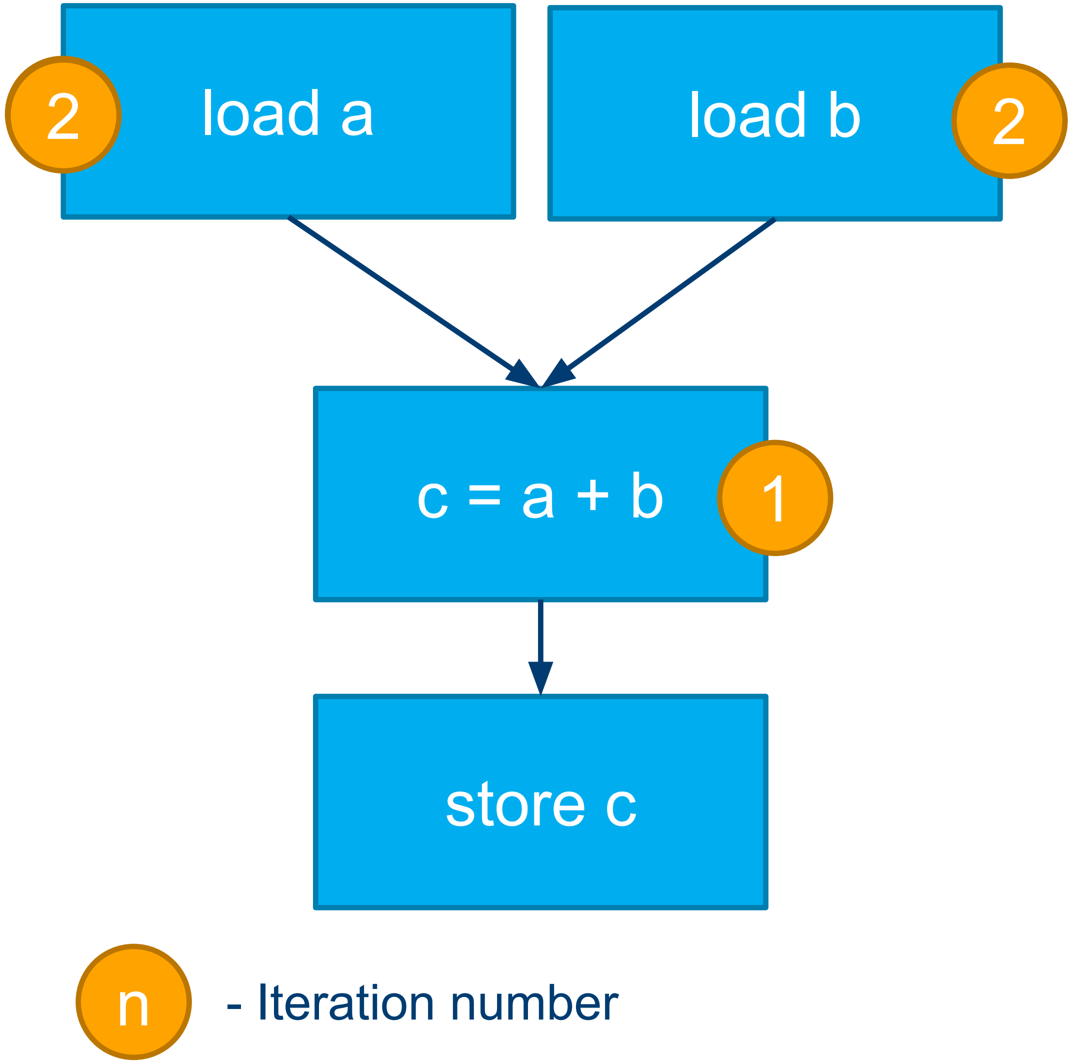
```
…
cgh.single_task<class swi_add>([=]() {
    for (unsigned i = 0; i < 128; i++) {
      c[i] = a[i] + b[i];
    }
   });
…
```



Understanding Initiation Interval

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

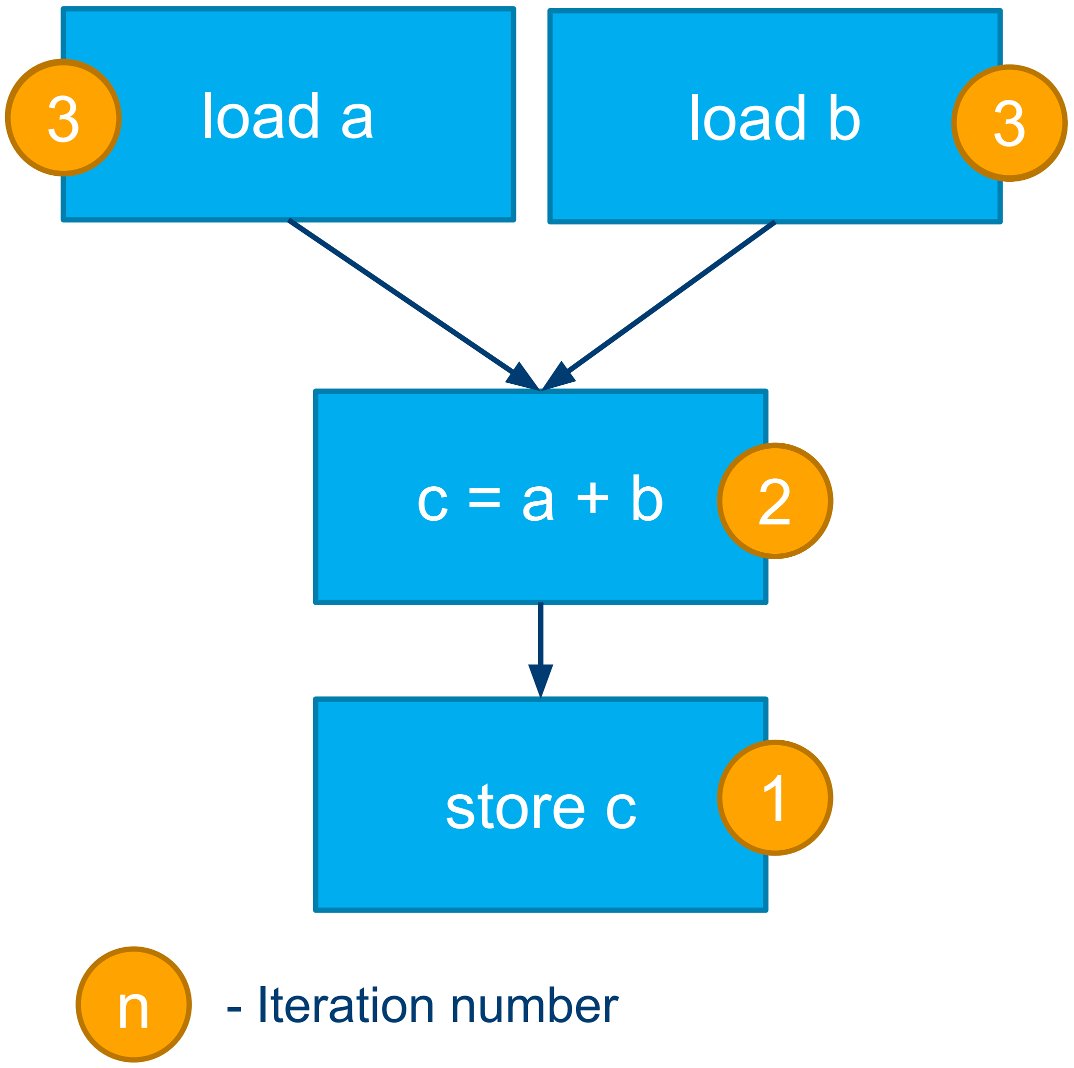
```
…
cgh.single_task<class swi_add>([=]() {
    for (unsigned i = 0; i < 128; i++) {
      c[i] = a[i] + b[i];
    }
   });
…
```



Understanding Initiation Interval

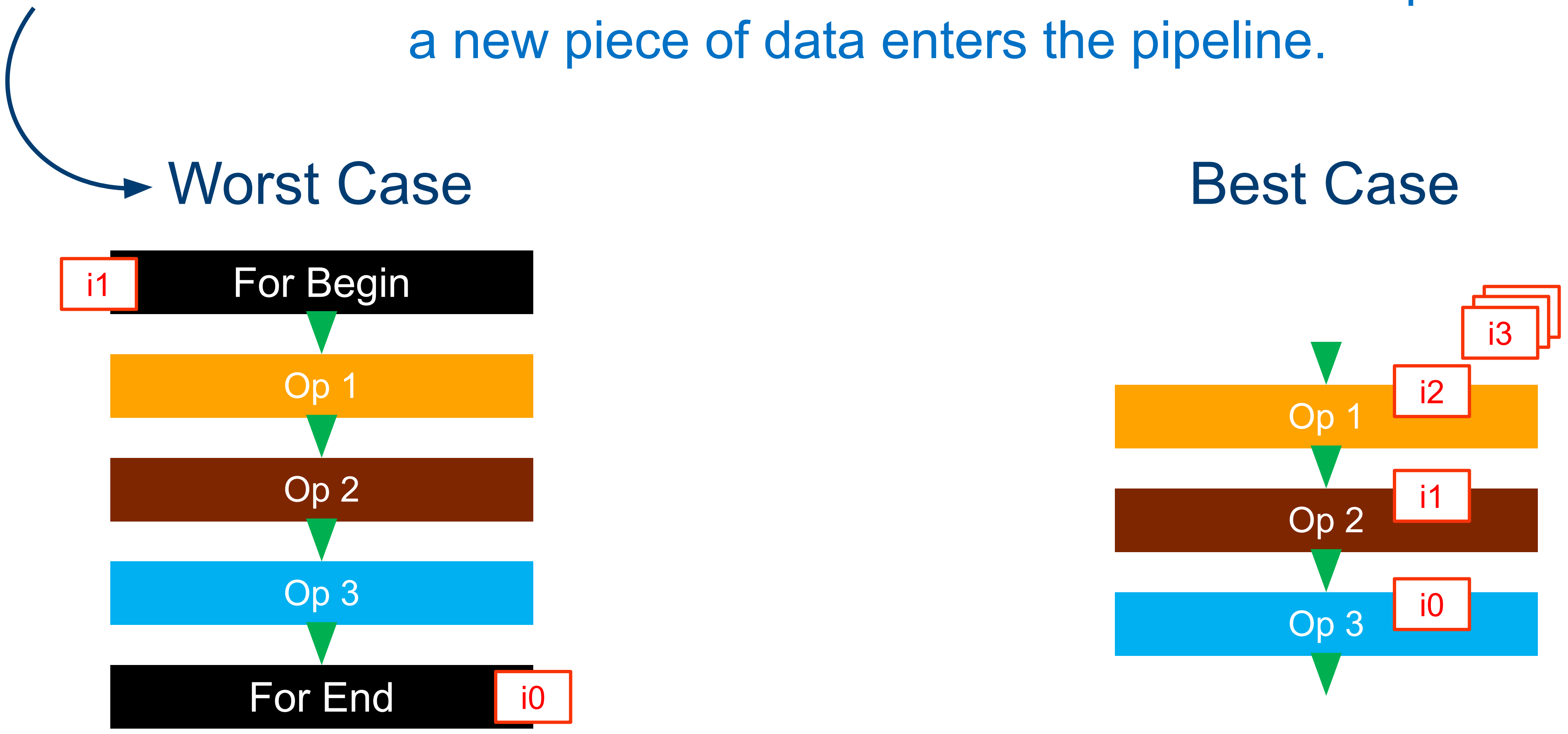
- dpcpp will infer pipelined parallel execution across loop iterations
  - Different stages of pipeline will ideally contain different loop iterations
- Best case is that a new piece of data enters the pipeline each clock cycle

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



Loop Pipelining vs Serial Execution

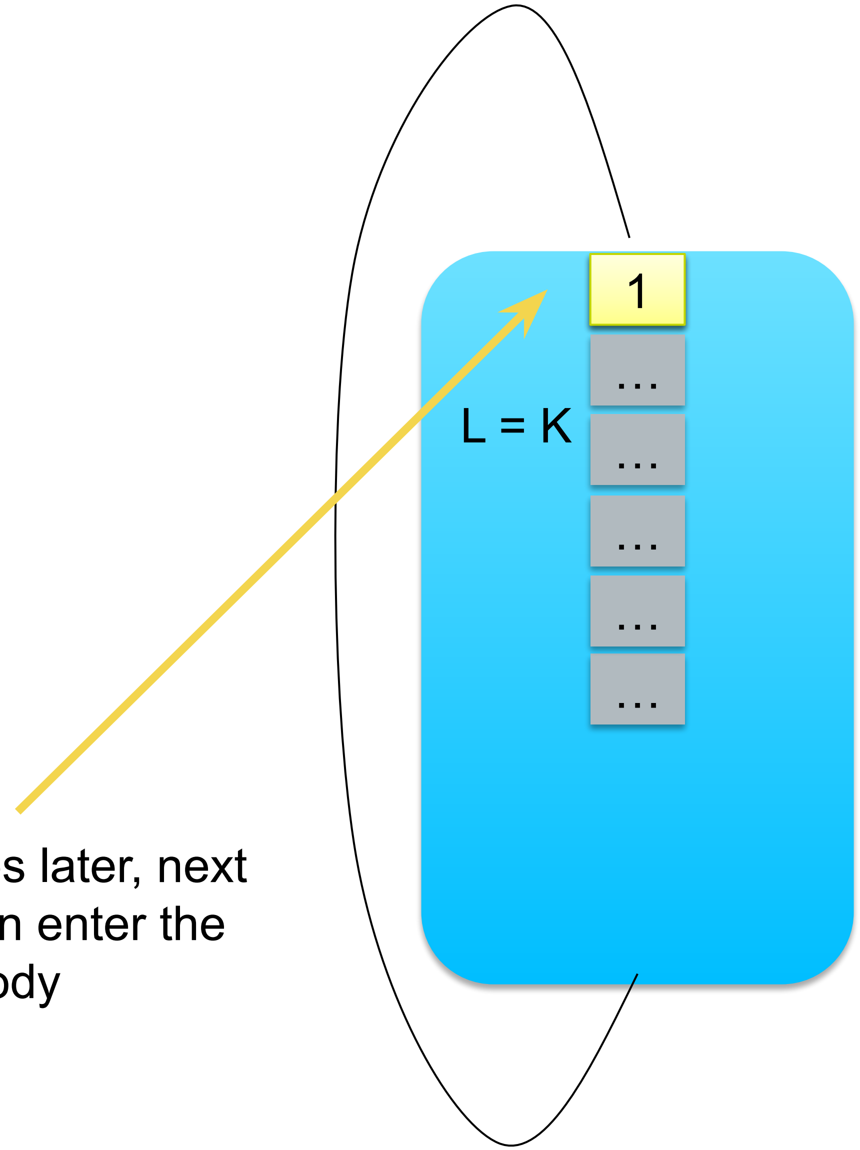
Serial execution is the worst case. One iteration needs to complete **fully** before a new piece of data enters the pipeline.



In-Between Scenario

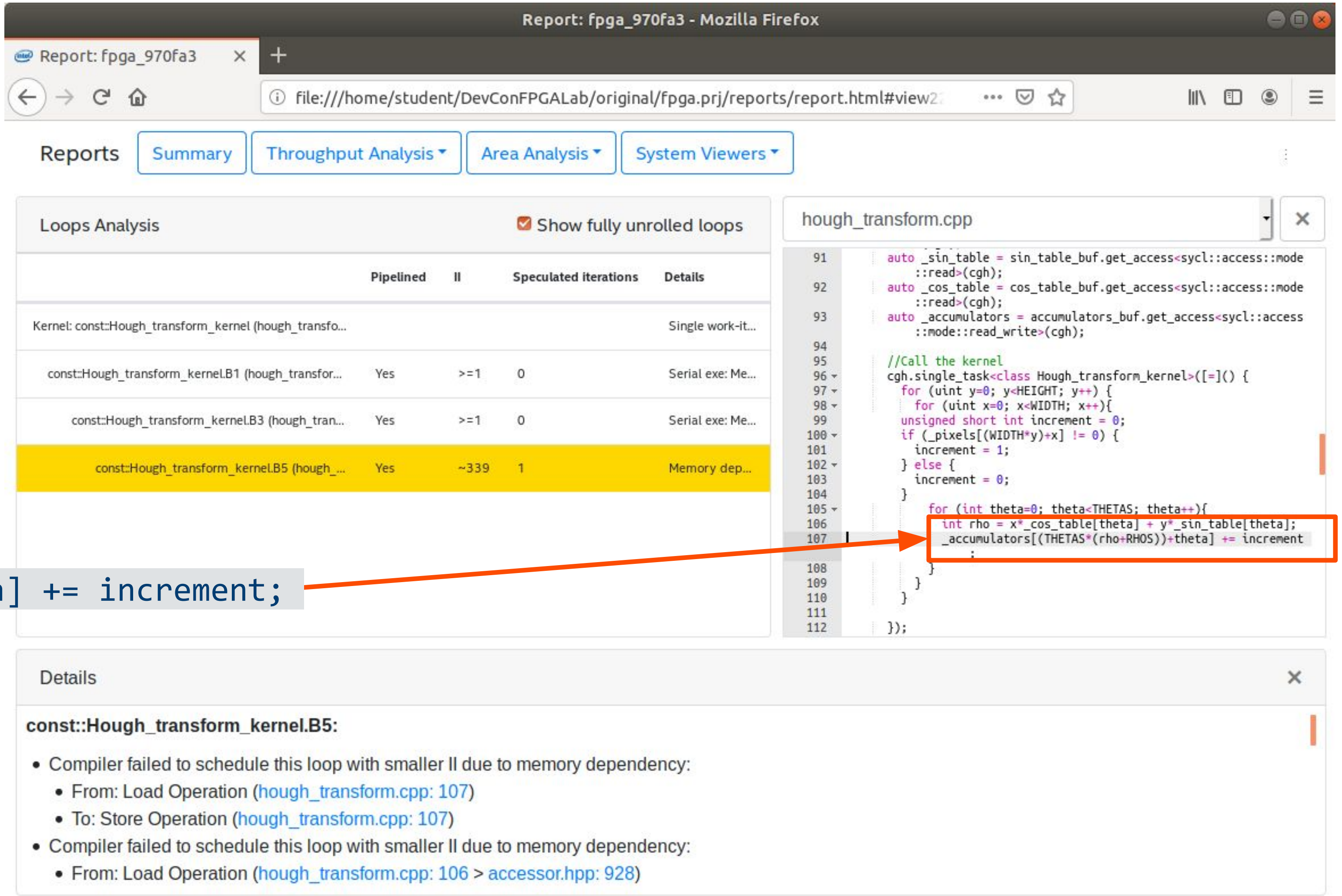
- Sometimes you must wait more than one clock cycle to input more data
- Because dependencies can't resolve fast enough
- How long you have to wait is called **Initiation Interval** or **II**
- Total number of cycles to run kernel is about (loop iterations)*II
  - (neglects initial latency)
- Minimizing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

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



Why Could This Happen?

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




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

Value must be retrieved from global memory and incremented

What Can You Do? Use Local Memory

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

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

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

});
...
```

Non-optimized

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

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

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

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

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

});
...
```

Optimized

What Can You Do? Tell the Compiler About Independence

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

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

Dependency ignored for A and B array

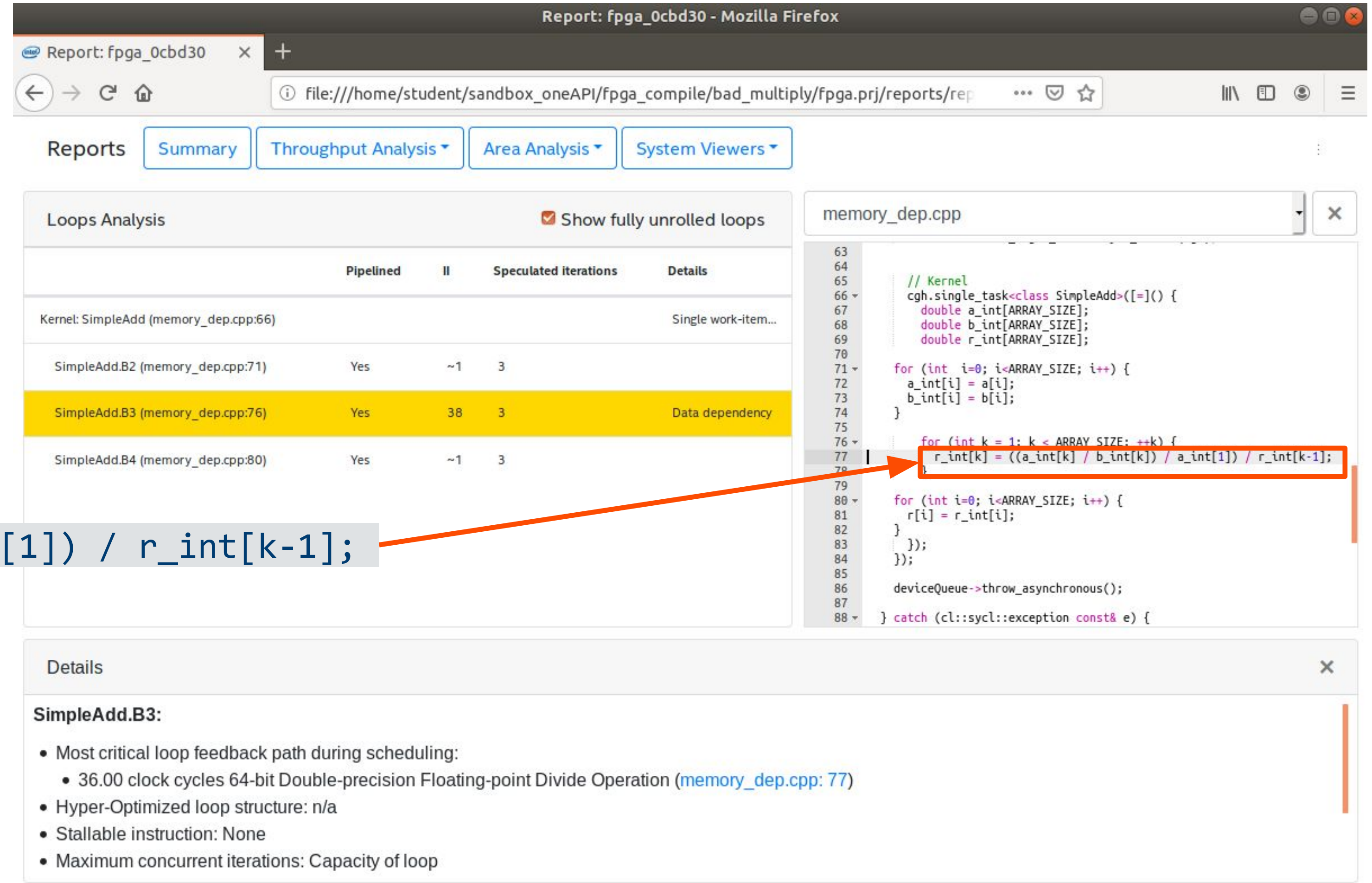
- `[[intelfpga::ivdep(array_name)]]`
  - Dependency ignored for only `array_name` accesses

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

Dependency ignored for A array
Dependency for B still enforced

Why Else Could This Happen?

- Data Dependency
  - Kernel cannot complete a calculation fast enough




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

Difficult double precision floating point operation must be completed

What Can You Do?

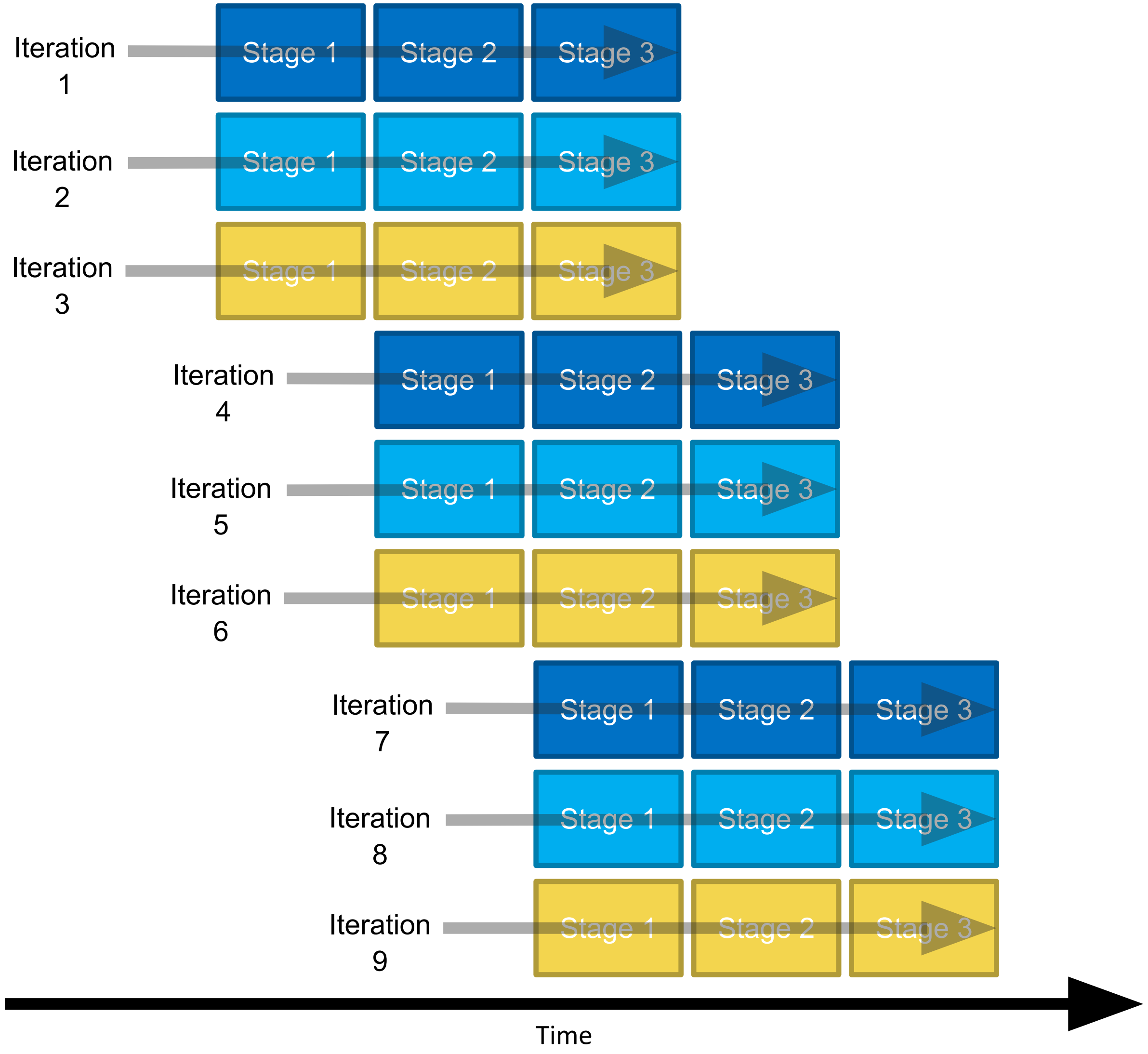
- Do a simpler calculation
- Pre-calculate some of the operations on the host
- Use a simpler type
- Use floating point optimizations (discussed later)
- Advanced technique: Increase time (pipeline stages) between start of calculation and when you use answer
  - See the “Relax Loop-Carried Dependency”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

How Else to Optimize a Loop? Loop Unrolling

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

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

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



Fmax

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

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

How Can You Tell This Is a Problem?

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

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

f_{MAX} II Report

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

What Can You Do?

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

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

Area

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

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

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

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

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

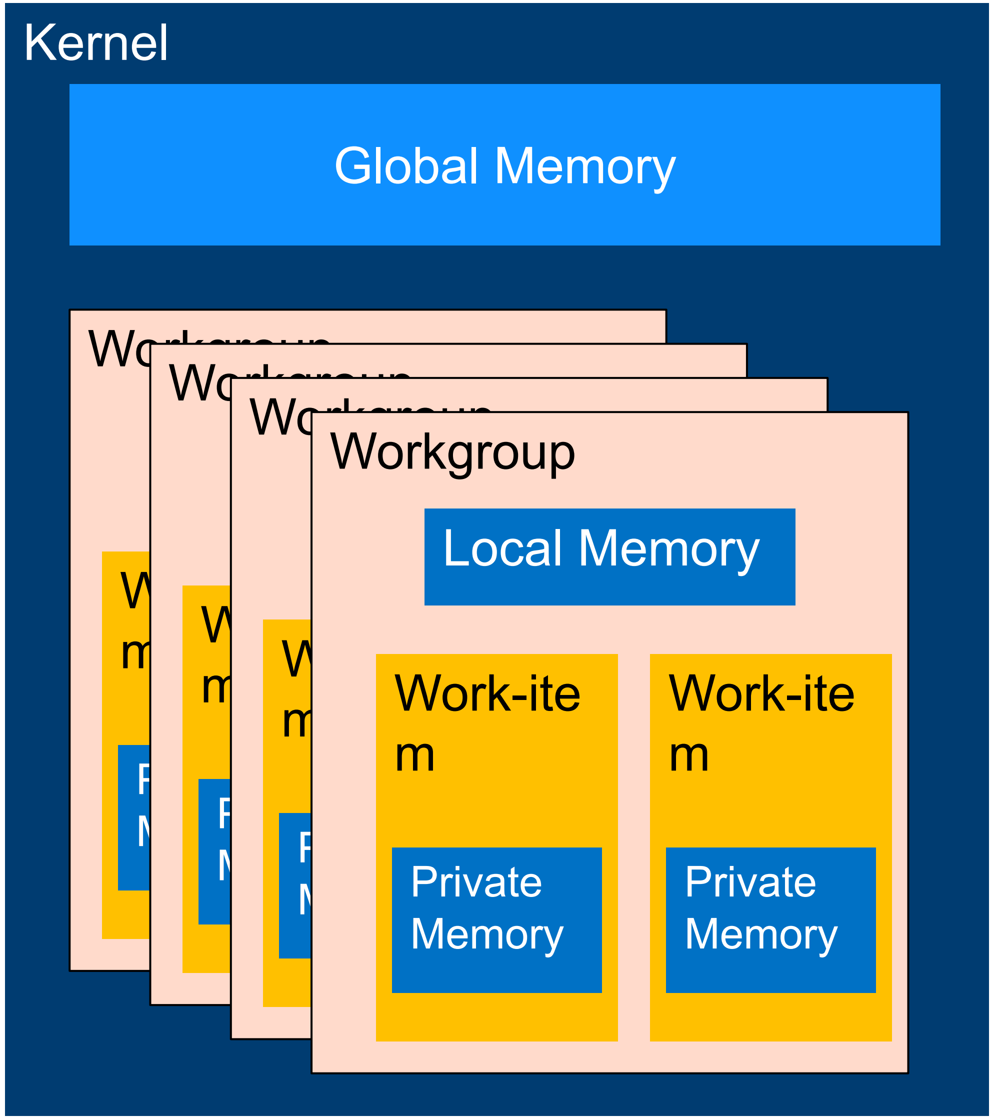
Memory Optimization

Memory Model

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

These are the same for single_task kernels

- Global Memory
  - Visible to all workgroups



Understanding Board Memory Resources

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

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

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

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

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

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

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

});
…
```

Non-optimized

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

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

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

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

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

});
…
```

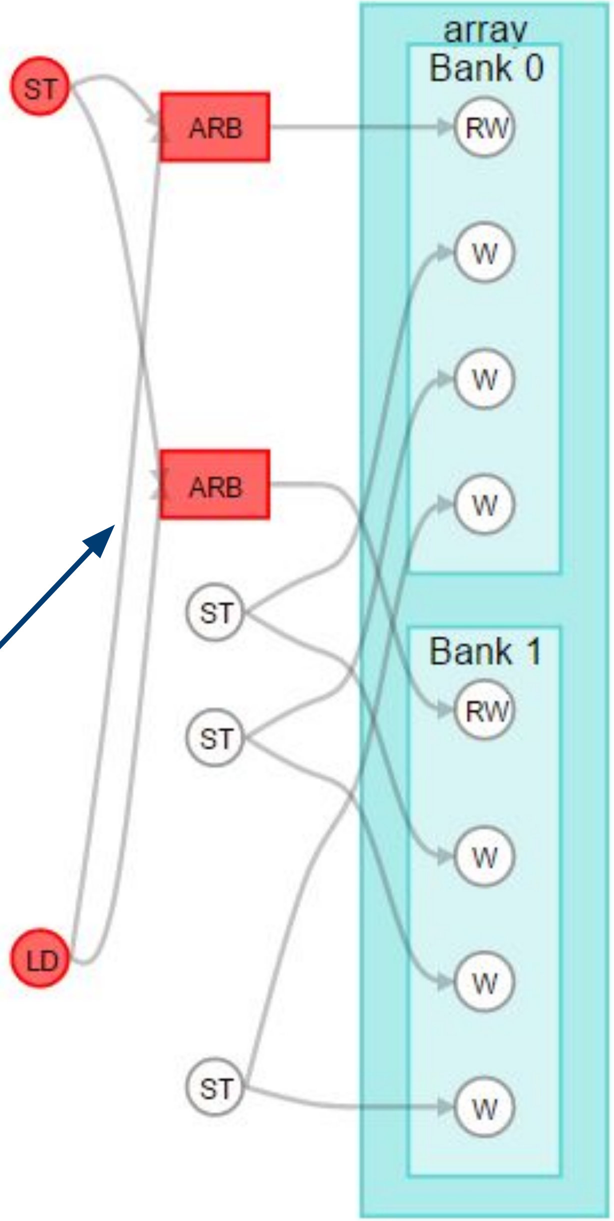
Optimized

Local Memory Bottlenecks

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

These can stall, so are a potential bottleneck

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



Local Memory Bottlenecks

Natively, the memory architecture has 2 ports

The compiler optimizes memory accesses to map to these without arbitration

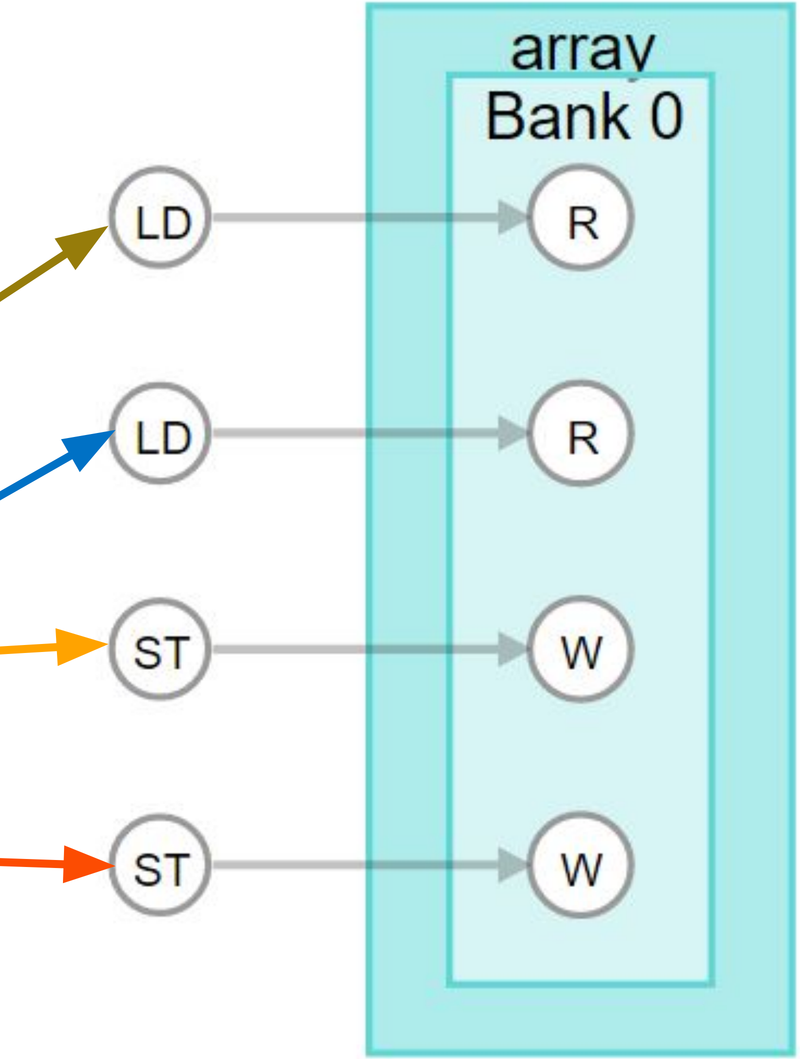
Your job is to write code the compiler can optimize

Double-Pumped Memory Example

Increase the clock rate to 2x

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

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

  array[ind1] = val;

  array[ind1+1] = val;

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

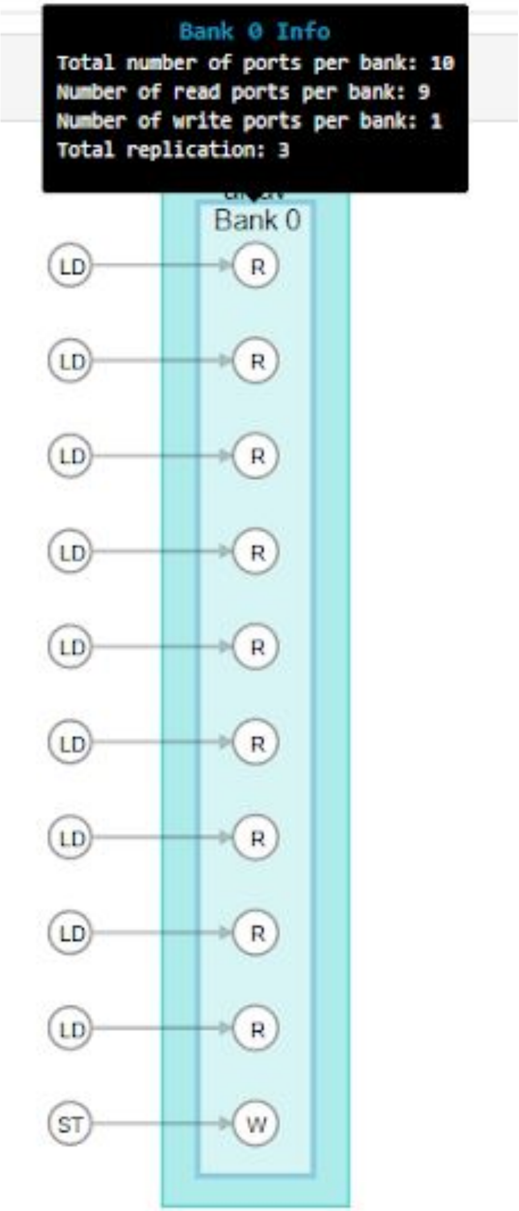


Local Memory Replication Example

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

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

   calc = res;
...
```

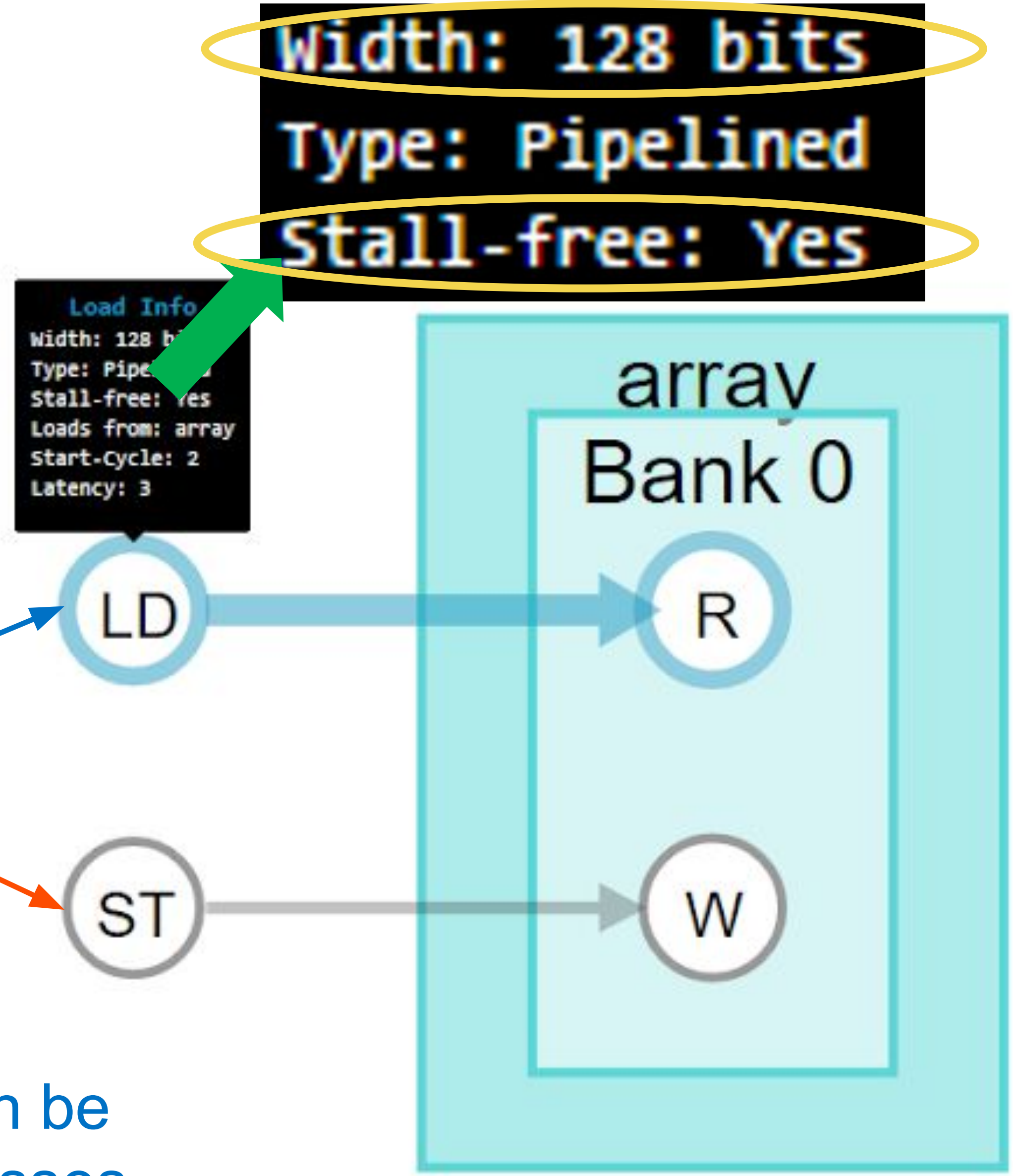
Turn 4 ports of double-pumped memory to unlimited ports

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;
...
```

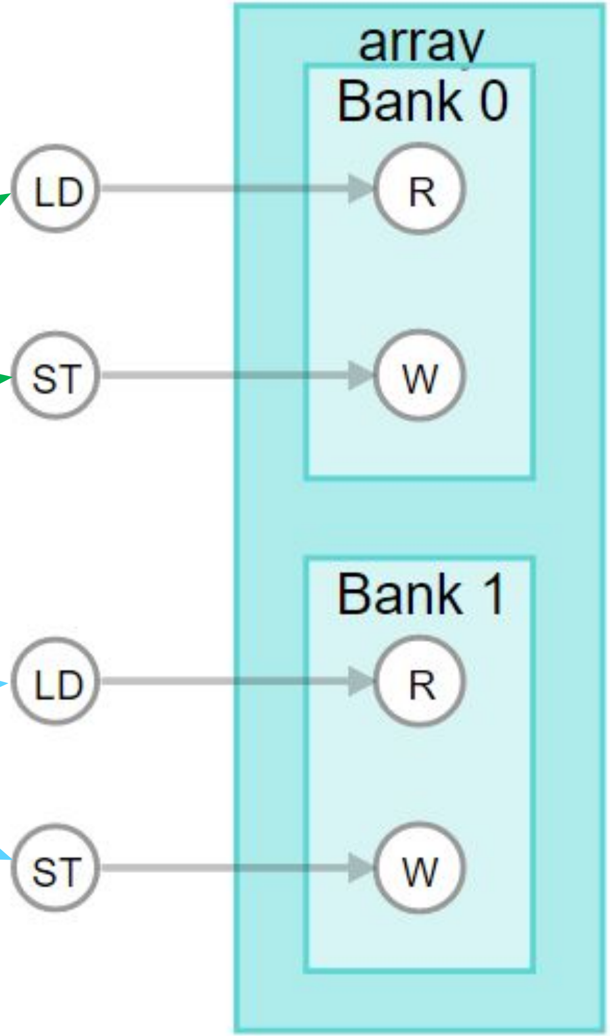


Continuous addresses can be coalesced into wider accesses

Banking

Divide the memory into independent fractional pieces (banks

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

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

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



Compiler looks at lower indices by default

Indices for banking must be a power of 2 size

Attributes for Local Memory Optimization

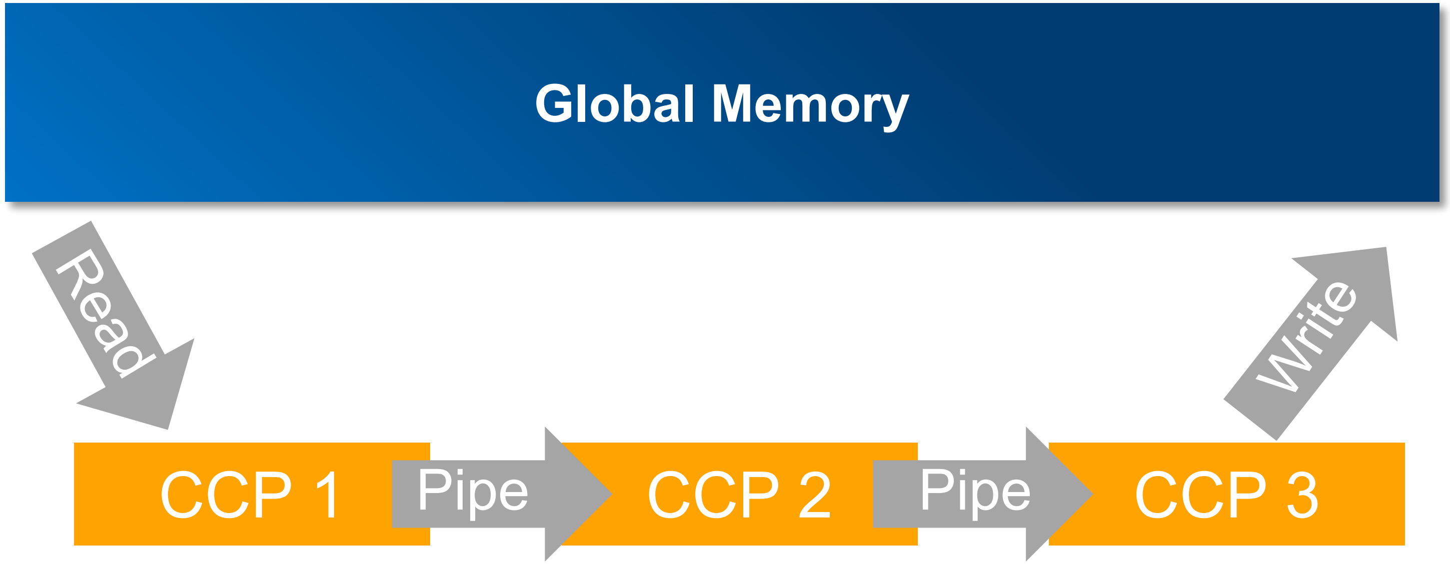
Note: Let the compiler try on it's own first. It's very good at inferring an optimal structure!

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

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

Pipes – Element the Need for Some Memory

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

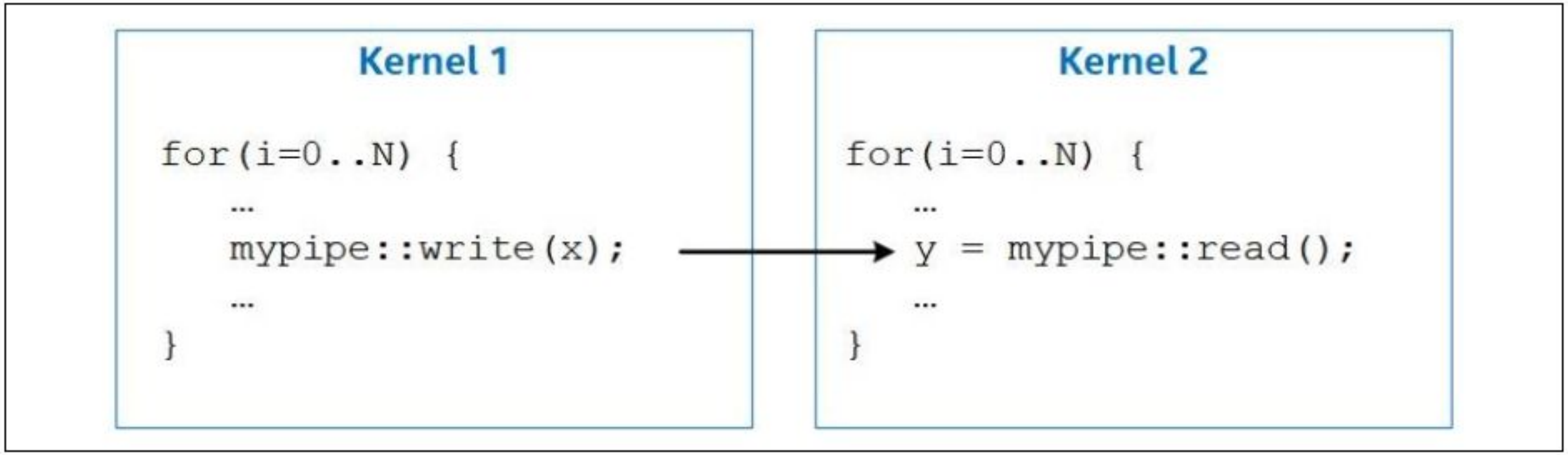


Task Parallelism By Using Pipes

Launch separate kernels simultaneously

Achieve synchronization and data sharing using pipes

Make better use of your hardware



Lab: Optimizing the Hough Transform Kernel

Other Optimization Techniques

Avoid Expensive Functions

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

Inexpensive Functions

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

Use Least-“Expensive” Data Type

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

Floating-Point Optimizations

- Apply to `half`, `float` and `double` data types

- Optimizations will cause small differences in floating-point results
  - **Not** IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008) compliant

- Floating-point optimizations:
  - Tree Balancing
  - Reducing Rounding Operations

Tree-Balancing

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

Rounding Operations

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

References and Resources

References and Resources

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

Upcoming Training

These online trainings are being developed throughout 2020

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

…and others!

Legal Disclaimers/Acknowledgements

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

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

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

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

© Intel Corporation

Notices & Disclaimers

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

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

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

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

Copyright ©, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Core, VTune, and OpenVINO are trademarks of Intel Corporation or its subsidiaries in the U.S. and other countries.

Optimization Notice

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

Copyright © 2019, Intel Corporation. All rights reserved.
*Other names and brands may be claimed as the property of others.