



High Level Design languages for Intel® FPGAs

CNRS DAQ Seminar – Fréjus, November 2018

OBJECTIVES

- Introduce the high level design coding tools available for Intel® FPGAs to increase the abstraction level and boost your productivity
- Make the FPGAs more “friendly” to software programmers.
- We will cover OpenCL and HLS (High Level Synthesis).

HLS vs OpenCL™

HLS	OpenCL
Quickens design of blocks to fit into a traditional FPGA design	Quickens the development of a kernel to fit into a system with an FPGA accelerator card and host
Uses the C/C++ programming language for design of components	Uses the kernel C (similar to C) for kernel design, and a host API for interaction with the host

Meant to help you design a block to fit into a traditional FPGA design

Meant to help you create a FPGA accelerator to fit into an OpenCL compliant system



OpenCL™ on FPGAs for Software Programmers

CNRS DAQ Seminar – Frejus, November 2018

francisco.perez@intel.com

*OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission of Khronos

OBJECTIVES

- Introduce the concept of OpenCL™ for heterogeneous programming using Intel® FPGAs
- Understand how to develop kernels and how they are executed on Intel® FPGAs
- Know which are the singular features of OpenCL™ applied to Intel® FPGAs

Class Agenda

Types of Parallel Computing

Intro to OpenCL™ for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

The Intel FPGA SDK for OpenCL

Class Agenda

Types of Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

- SDK components

- Debug Tools

- FPGA-specific Features (channels, libraries)

Parallel Computing

“A form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently (in parallel)”

~ Highly Parallel Computing, Amasi/Gottlieb (1989)

Types of Parallelism

Data Parallelism

Task Parallelism

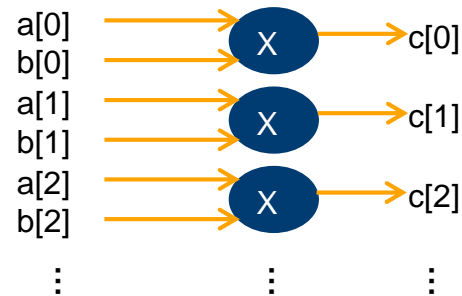
Pipeline Parallelism

Data Parallelism

Input data separated and sent to parallel resources, results recombined

- Same operation(s) applied across different data in parallel
- Single Program Multiple Data (SPMD)
- Single Instruction Multiple Data (SIMD)

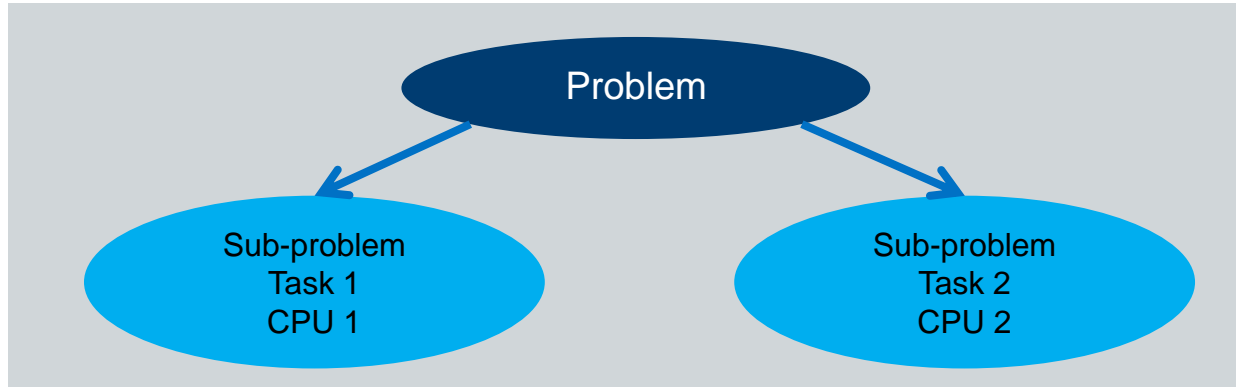
```
for (i = 0; i < N ; i++)  
    c[i] = a[i] * b[i]
```



Task Parallelism

Decompose problem into sub-problems (tasks). Divide and conquer

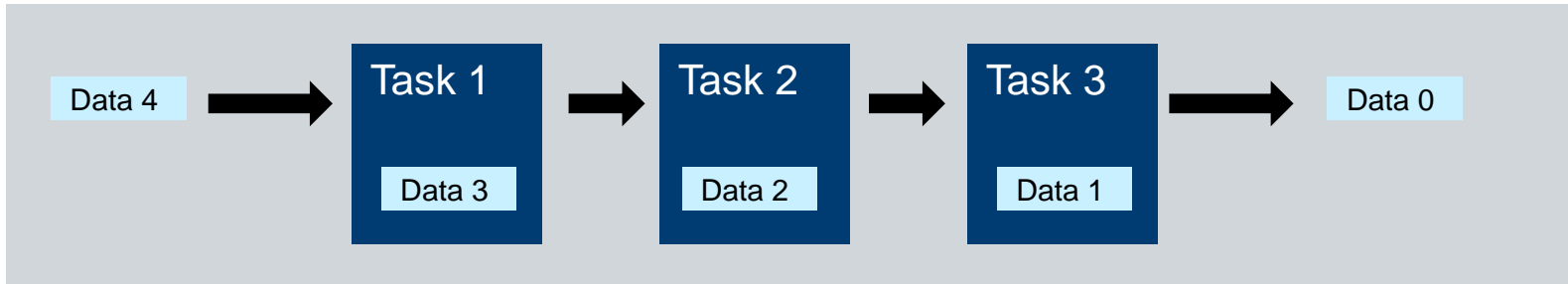
- Tasks operate on same or different data
- Example: Multi-CPU system where each CPU execute a different thread
- A.K.A. Simultaneous Multithreading (SMT), Thread/Function Parallelism



Pipeline Parallelism

Task parallelism where tasks have a producer consumer relationship

- Operates on pipelined data
 - Different tasks operate in parallel on different data
- Example
 - Task1 – FFT, Task 2 – Frequency Filter, Task3-Inverse FFT



Heterogeneous Computing Systems

Modern systems contain more than one kind of processor

- Applications exhibit different behaviors
 - Control intensive (Searching, parsing, etc...)
 - Data intensive (Image processing, data mining, etc...)
 - Compute intensive (Iterative methods, financial modeling, etc...)
- Gain performance by using specialized capabilities of different types of processors

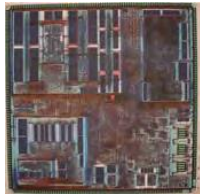
Example Heterogeneous System

Modern computing platform contains many dissimilar processors

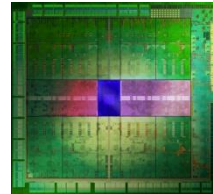
- Multi-core, general purpose, central processing units (CPUs)
- Digital Signal Processing (DSPs) processors
- Graphics Processing units (GPUs)
- Field Programmable Gate Arrays (FPGAs)



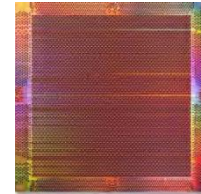
CPUs



DSPs



GPUs



FPGAs

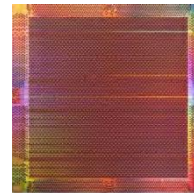
Challenge: How to build a software ecosystem for a heterogeneous platform?

Traditional Approach to Heterogeneous Computing

- Write software for each software programmable architecture CPU, GPU, DSP
 - Using different languages and vendor specific tools



- Develop custom parallel hardware for FPGA
 - Fine-grained parallelism
 - Write HDL
 - Simulation, timing closure, on-chip verification etc.



Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

- SDK components

- Debug Tools

- FPGA-specific Features (channels, libraries)

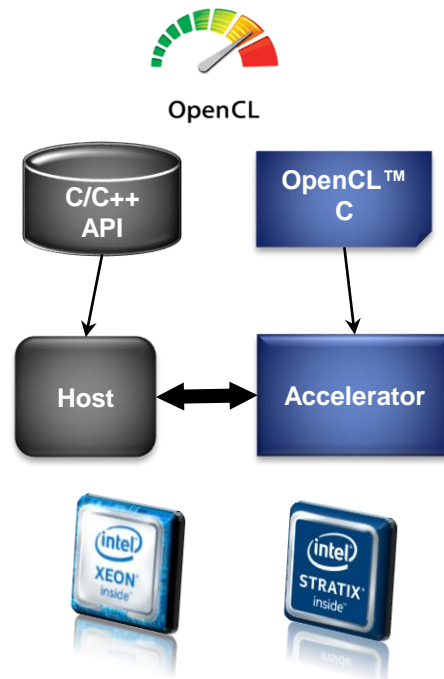
What is OpenCL™?

- Open Computing Language (OpenCL™) - Framework for heterogeneous computing
 - General purpose programming model for multiple platforms
 - Host API and kernel language
 - Low-level Programming language based on C/C++
 - Provides increased performance with hardware acceleration
- Open, royalty-free standard
 - Managed by Khronos* Group
 - Intel® is an active member
 - <http://www.khronos.org>



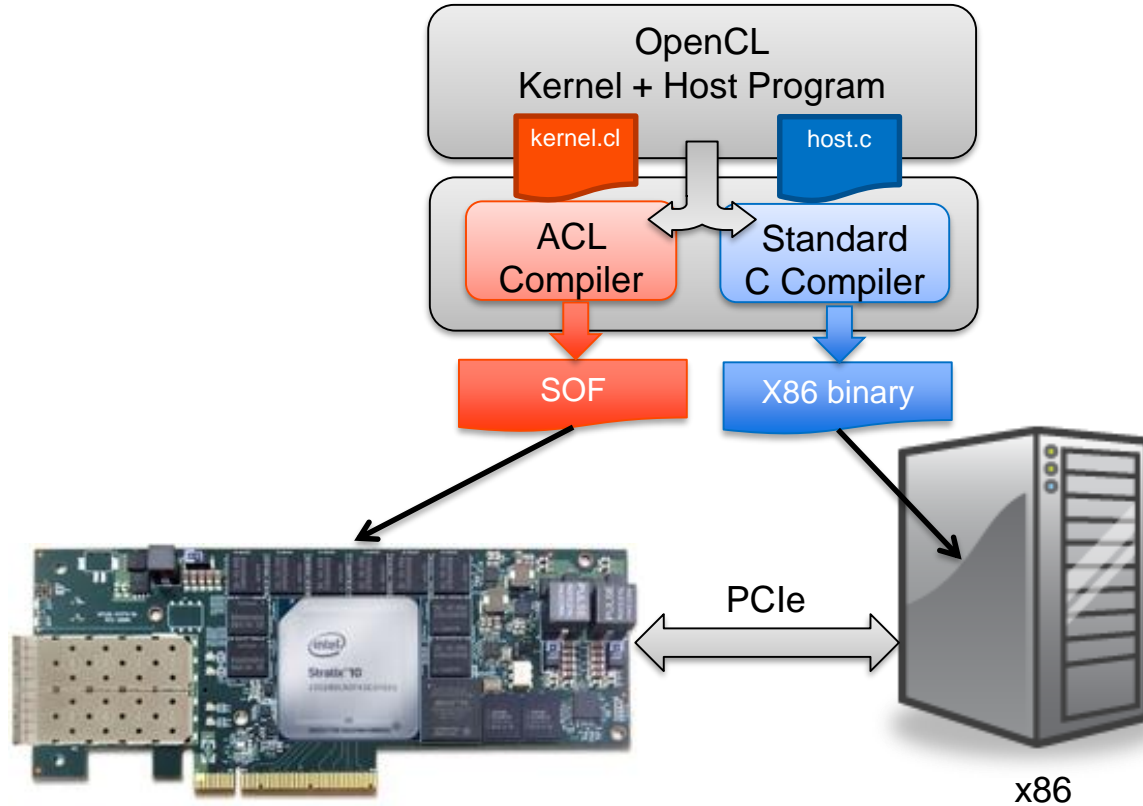
Two Sides of OpenCL™ Standard

- Kernel Function
 - OpenCL™ C
 - Software that runs on accelerators (OpenCL devices)
 - Usually used for computationally intensive tasks
- Host Program
 - Software running conventional microprocessor
 - Supports efficient plumbing of complicated concurrent programs with low overhead
 - Through OpenCL host API

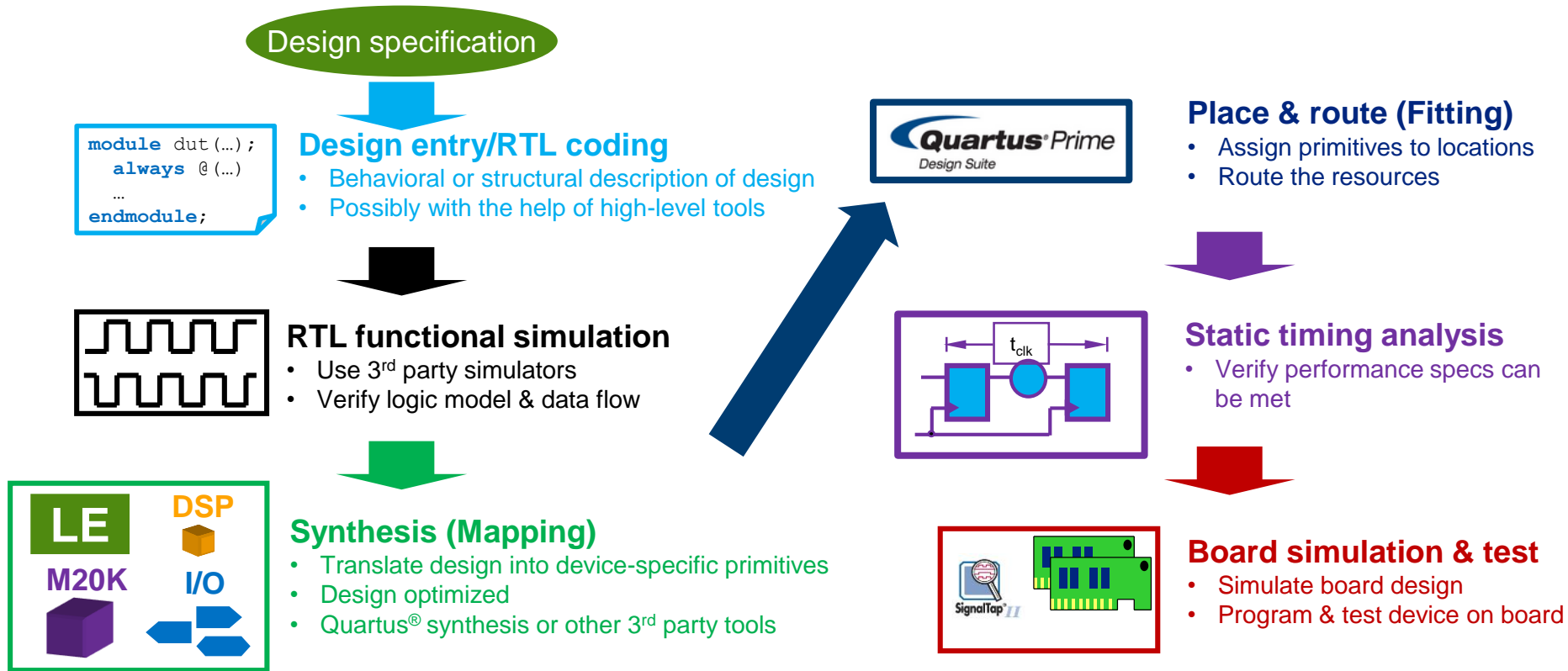


Used together to efficiently implement algorithms

Mapping OpenCL Programs



Traditional FPGA Design Flow



FPGA High Level Design with OpenCL™

Goal: Design FPGA custom hardware with C-based software language

```
__kernel void _foo (__global float *x) {  
    int i ...  
}
```

Intel® FPGA SDK for OpenCL™

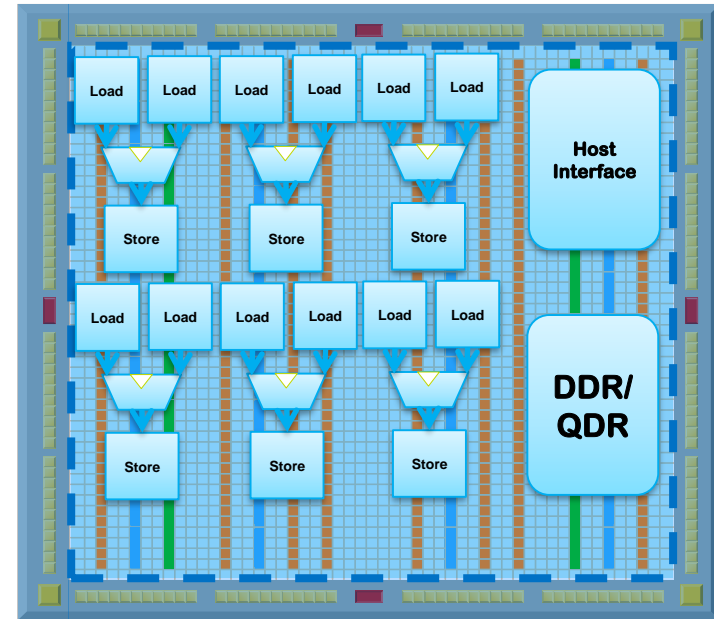


■ Benefits

- Makes FPGA acceleration available to software engineers
- Debug and optimize in a software-like environment
- Significant productivity gains compared to hardware-centric flow
- Easier to perform design exploration
- Abstracts away FPGA design flow and FPGA hardware

Compiling OpenCL™ to Intel® FPGA

- Custom hardware generated automatically for each kernel
 - Get the advantages of the FPGA without the lengthy design process
- Organized into functional units based on operation
- Able to execute OpenCL™ threads in parallel



Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

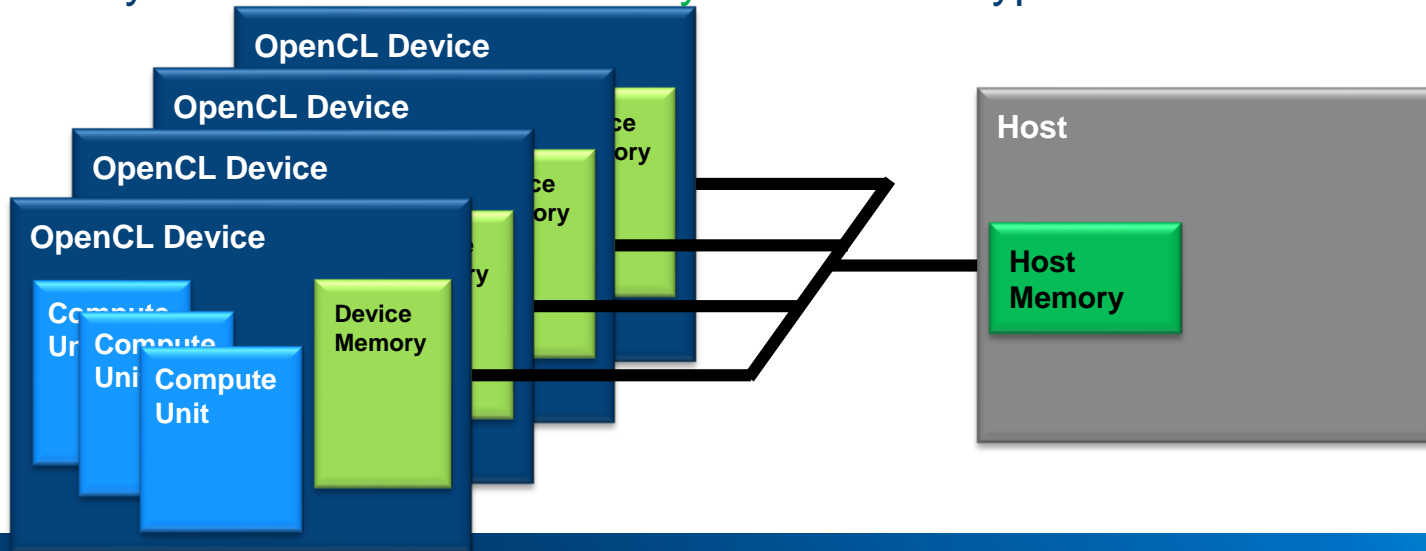
- SDK components

- Debug Tools

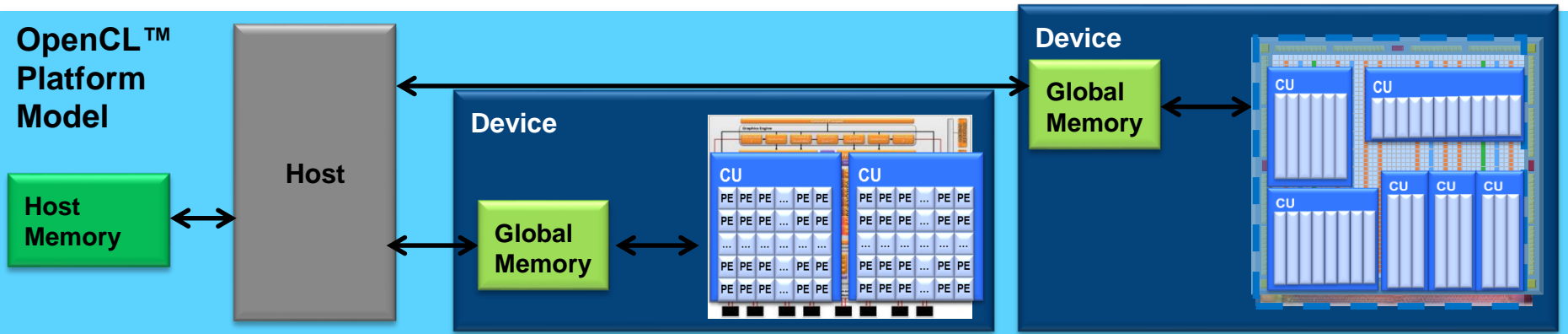
- FPGA-specific Features (channels, libraries)

OpenCL™ Platform Model

- One Host with one or more OpenCL™ Devices
 - Each Device is composed of one or more compute units
- Memory divided into Host Memory and various types of Device Memory



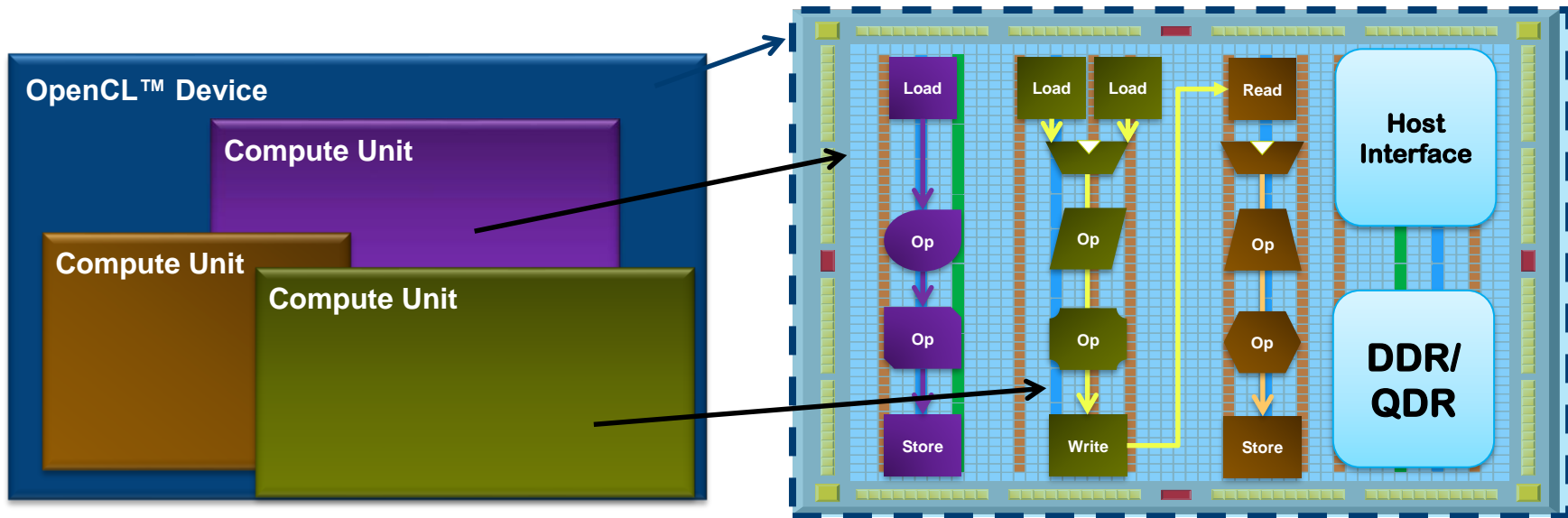
Heterogeneous Platform Model



Intel® FPGA OpenCL™ Device

Each device is made of many independent compute units

- Each compute unit is custom built from kernel code



OpenCL™ Platform Layer and Runtime Layer API

OpenCL™ API divided into two layers

- Platform Layer API
 - Discover platform and device capabilities
 - Setup execution environment
- Runtime Layer API
 - Executes compute kernels on devices
 - Manage device memory

Platform Layer API

Setup device execution environment

- Necessary to allow for heterogeneous environments and multiple devices

▪ Tasks

- Allows host to discover devices and capabilities
- Query, select and initialize compute devices
- Create compute contexts to manage OpenCL™ objects

Typical Platform Layer Steps

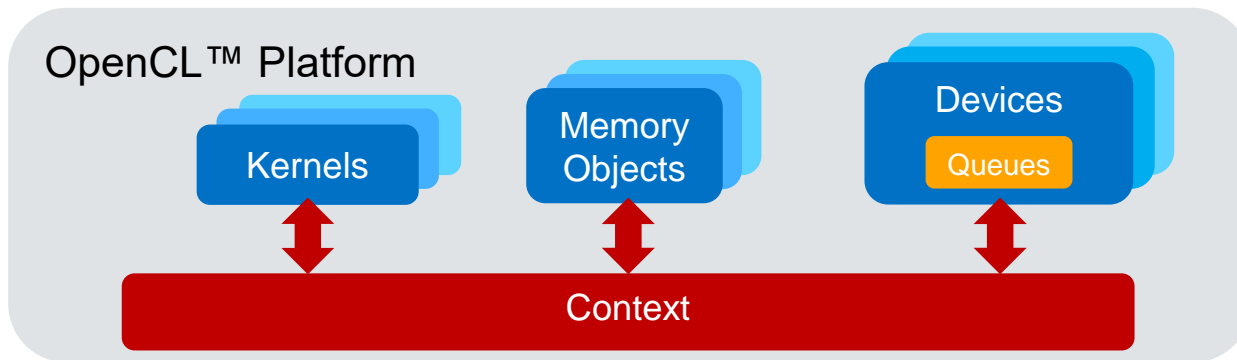
1. Query platforms
2. Query devices
3. Create a context for the devices

Context

Abstract containers that manage host device interaction

- Purpose

- Coordinates the mechanisms for host-device interaction
- Manages the device memory
- Keeps track of kernels to be executed on each device



Runtime Layer API

Execute kernels on the device

- Tasks
 - Memory management
 - Run kernels on the device
 - Host/device synchronization

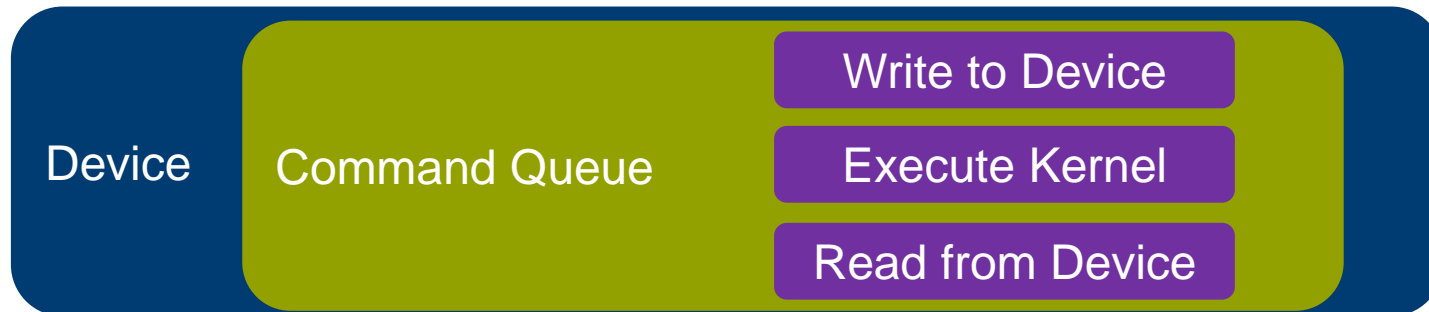
Typical Runtime Layer Steps

1. Create a command queue
2. Write to the device
3. Launch kernel
4. Read results back from the device

Command Queue

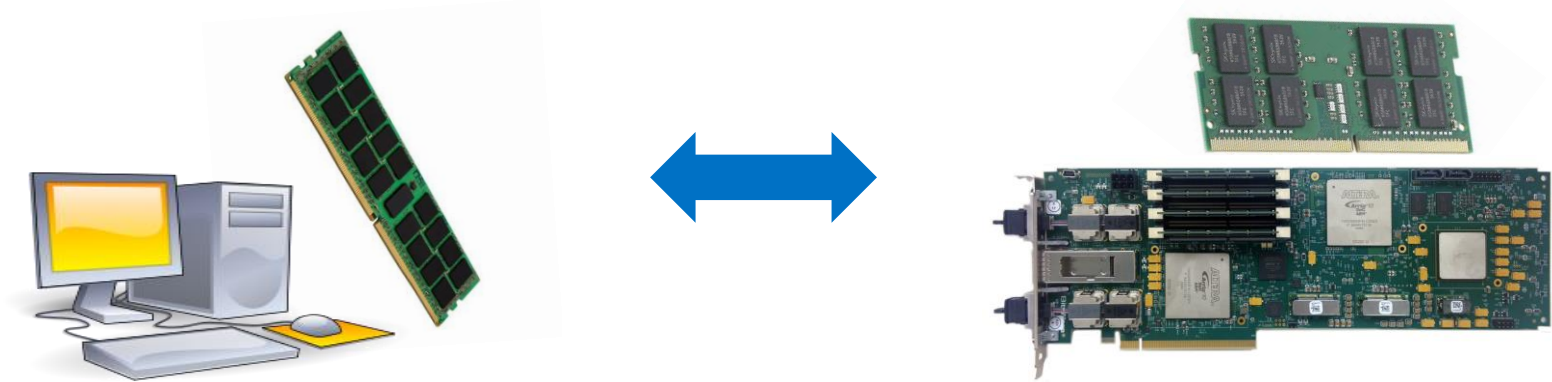
Mechanism for host to request action by the device

- Each command queue associated with one device
 - Each device can have one or more command queues
- Host submits commands to the appropriate queue
- Operations in the queue will execute in-order for Intel® FPGAs



Host / Device Physical Memory Space

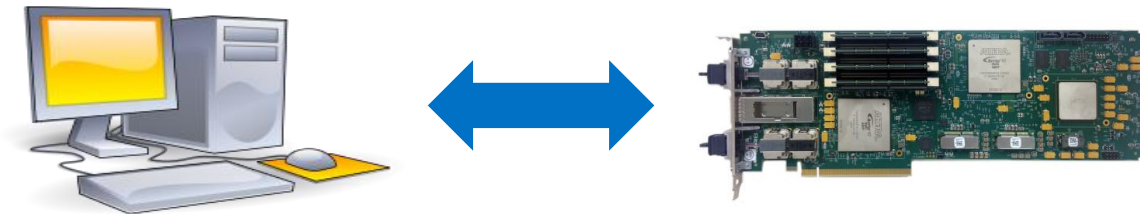
- The host and the device each has its own physical memory space
 - Data needs to be physically located on a device before kernel execution
- Use OpenCL™ API functions to allocate, transfer, and free device memory
 - Using **memory objects** through command queues



Data Transfers Calls

Use Read and Write Host API calls to explicitly transfer data from/to the device

- Commands placed on the command queue
- If kernel dependent on the buffer is executed on the accelerator device, buffer is transferred to the device
- Runtime determines precise timing of data movement



Terminology

OpenCL / Poker table

- Host -> Card dealer
- Context -> Table
- device -> Player
- cmd queue -> player hands
- kernel -> card



How they interact.....

Dealer sits at the card table and determine the player

The host selects the devices and places them in a context

The dealer selects cards from the deck and deals them (in hand)

Host select kernels from program, add them on the cmd_queue

Each player looks at their hand and decides what to do

Each device process kernel from the device queue

Dealer respond to host during the game

Host receive events from the devices and invokes event-handling queue

The dealer look at player and decide who won

Once all kernel are done the host receive the results

Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

- SDK components

- Debug Tools

- FPGA-specific Features (channels, libraries)

OpenCL™ Kernels

Functions that run on OpenCL™ devices

- Begins with the keyword `__kernel`
- Returns `void`
- Pointers in kernels should be qualified with an address space
 - `__private`, `__local`, `__global`, or `__constant`
- Kernel language derived from ISO C99 with certain restrictions

```
__kernel void my_kernel ( __global float *data) {  
}
```

Kernel Example

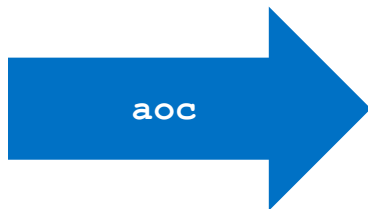
```
__kernel void my_kernel ( __global float *a,  
                          __global float *b,  
                          __global float *c,  
                          int N)  
{  
    int index;  
    for (index = 0; index < N; index++)  
        c[index] = a[index] + b[index];  
}
```

Compiling OpenCL™ Kernel to FPGAs

Kernels are compiled offline using an Offline Compiler (AOC)

- Kernels are first translated into an AOC Object file (.aoco)
 - Represents the FPGA hardware system
- Object file used to generate the AOC Executable file (.aocx)
 - Used to program the FPGA or Flash

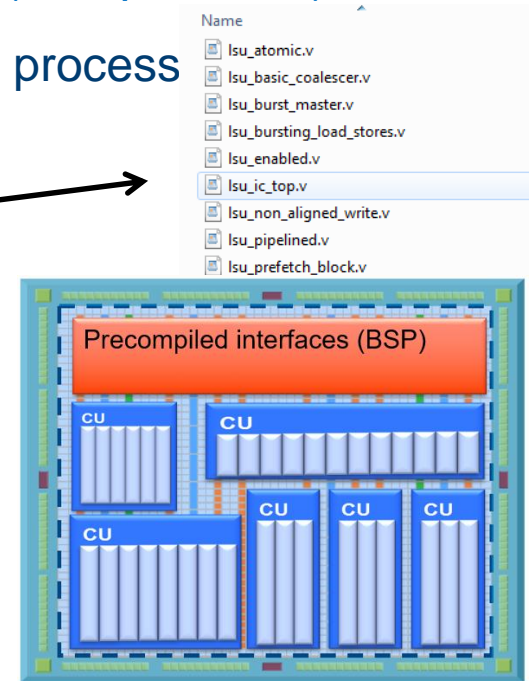
```
// kernel.cl
__kernel void KernelName(...)
{
    int index;
    for (index = 0; index < N; index++)
        c[index] = a[index] + b[index];
}
```



OpenCL™ Kernels to Dataflow Circuits

Each kernel is converted into custom dataflow hardware (Compute Unit)

- Gain the benefits of FPGAs without the lengthy design process
- Implement C operators as circuits
 - HDL code located in <SDK Installation>/ip
 - Load Store units to read/write memory
 - Arithmetic units to perform calculations
 - Flow control units
 - Connect circuits according to data flow in the kernel
- May replicate circuit to accelerate algorithm



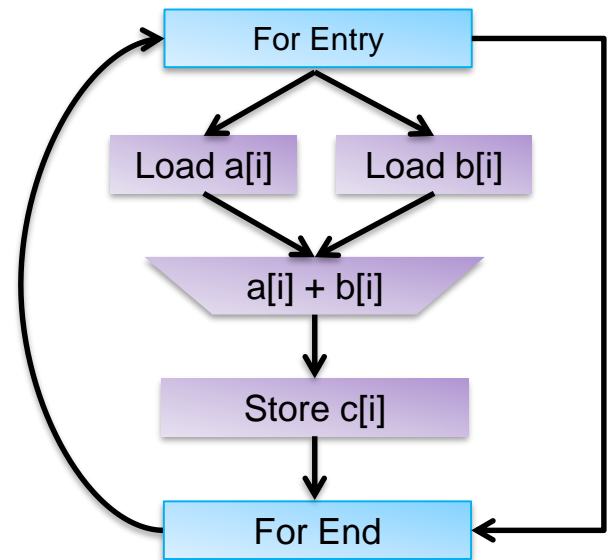
Compilation Example

Kernel compiled into dataflow circuit with flow control

- Includes branch and merge units

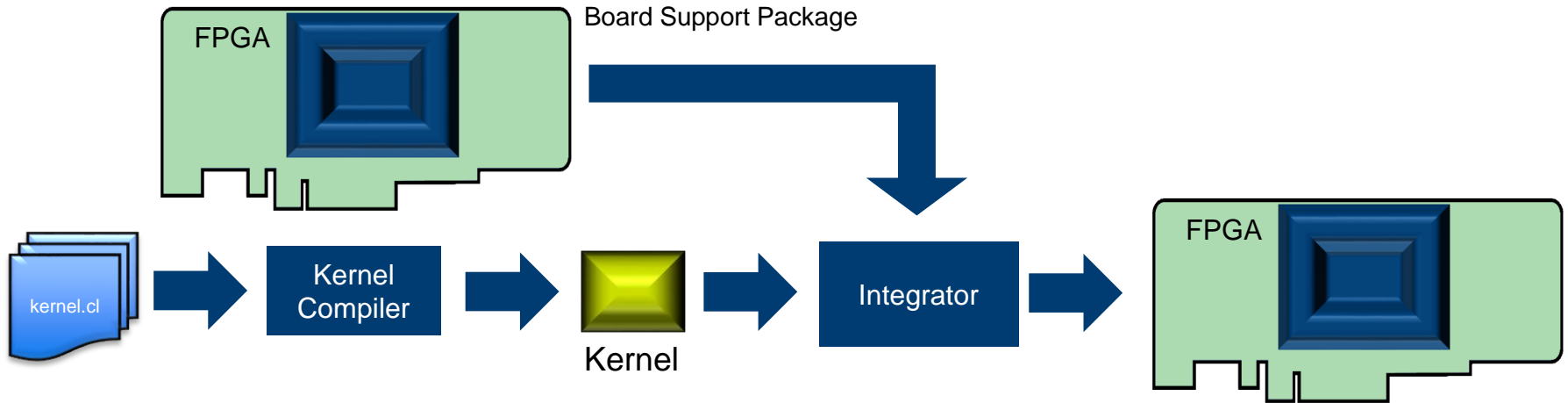
```
__kernel void my_kernel ( __global float *a,  
                          __global float *b,  
                          __global float *c,  
                          int N)  
{  
    int i;  
    for (i = 0; i < N; i++)  
        c[i] = a[i] + b[i];  
}
```

aoc



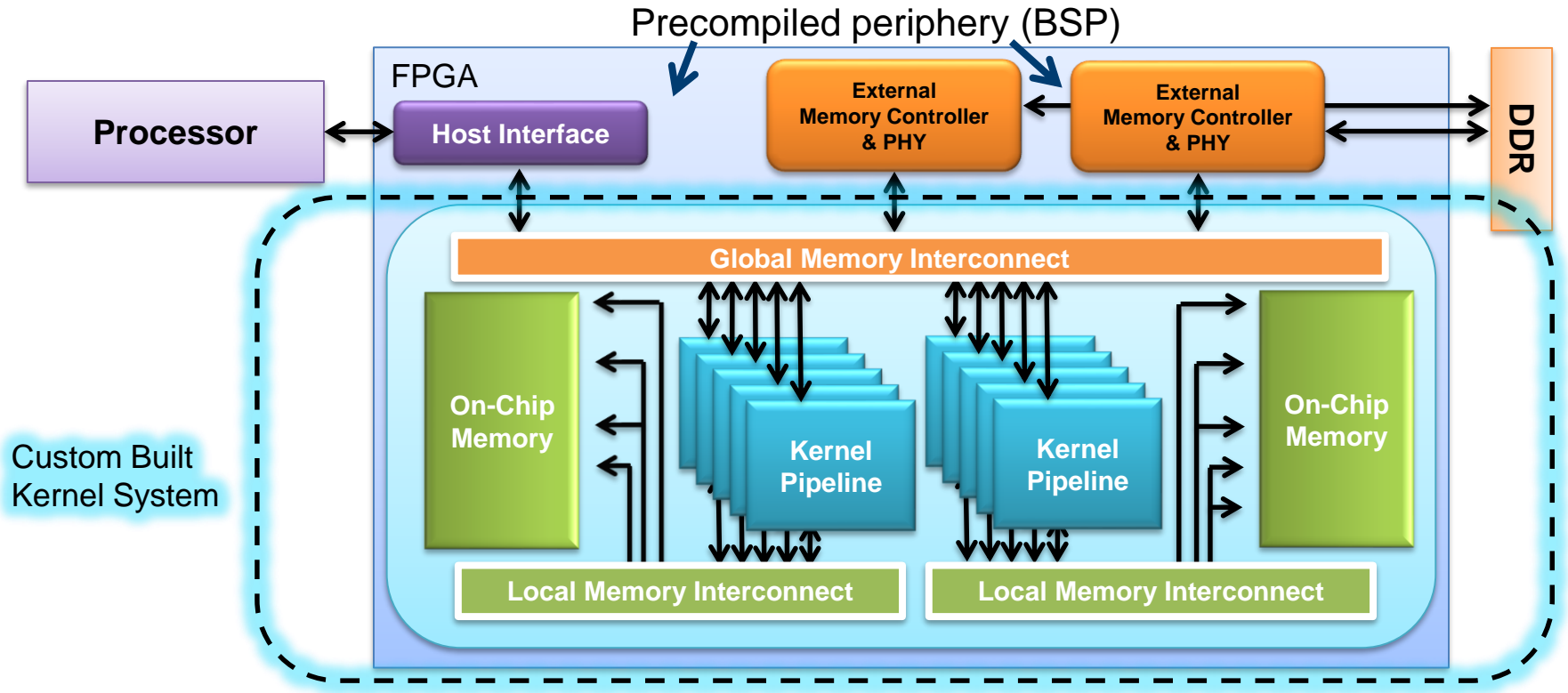
Altera's OpenCL Flow

Intel's OpenCL SDK for FPGA takes a system level view



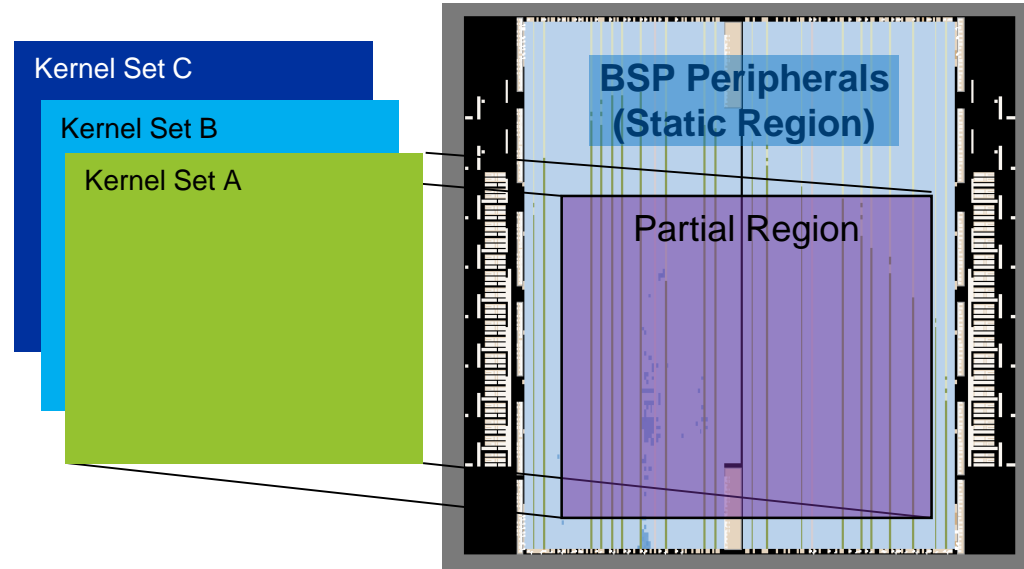
- Board Support Package (BSP)
 - “Chassis” to hold the newly created kernel
- Kernel Compiler
 - Optimized pipelines from C
- System Integrator
 - Merge all together and generate partial reconfiguration files for FPGA

FPGA Architecture for OpenCL™ Implementation



Partial Reconfiguration

- Reconfigures part of the FPGA while others continues operation
- Every aocx file represent a set of concurrent OpenCL kernels
- Allows kernels to be swapped while maintaining host-device communication



Intel FPGA Preferred Board for OpenCL

- Intel® FPGA Preferred Board for OpenCL™
 - Available for purchase from preferred partners and Intel
 - Passes conformance testing
- Download and install Intel FPGA OpenCL compatible BSP from vendor
 - Supplies board information required by the offline compiler
 - Provides software layer necessary to interact with the host code including drivers



Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels**

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

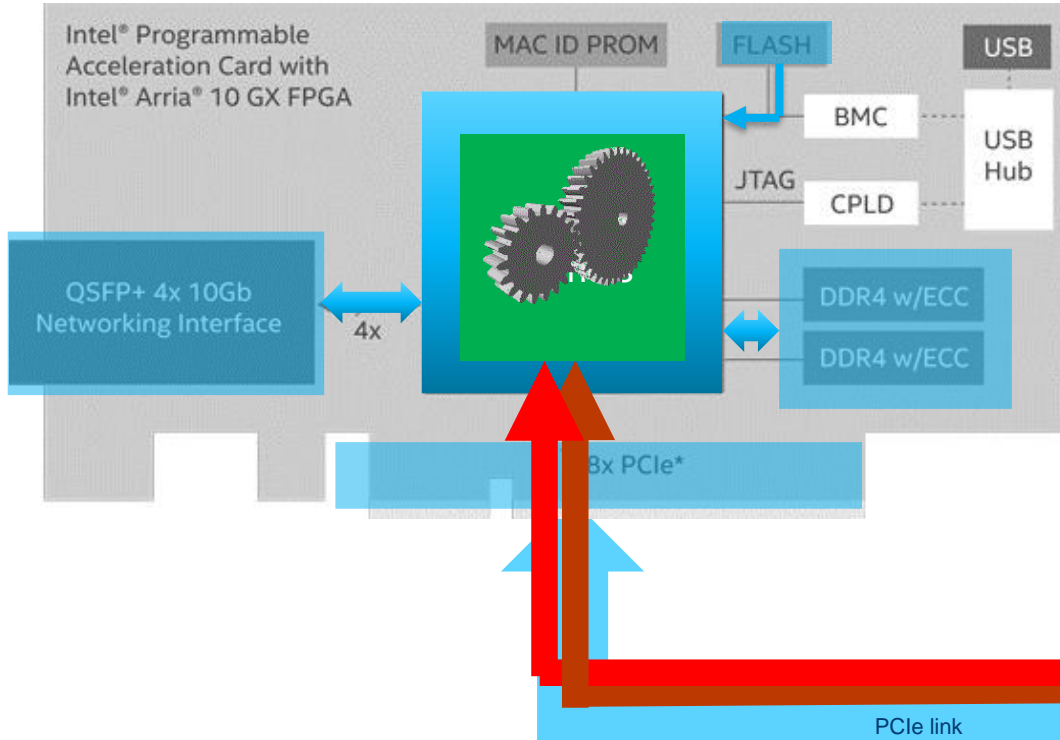
- SDK components

- Debug Tools

- FPGA-specific Features (channels, libraries)

OpenCL™ Execution Flow

Device



Host Machine



Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism**

The Intel FPGA SDK for OpenCL

- SDK components

- Debug Tools

- FPGA-specific Features (channels, libraries)

Mapping Multithreaded Kernels to FPGAs

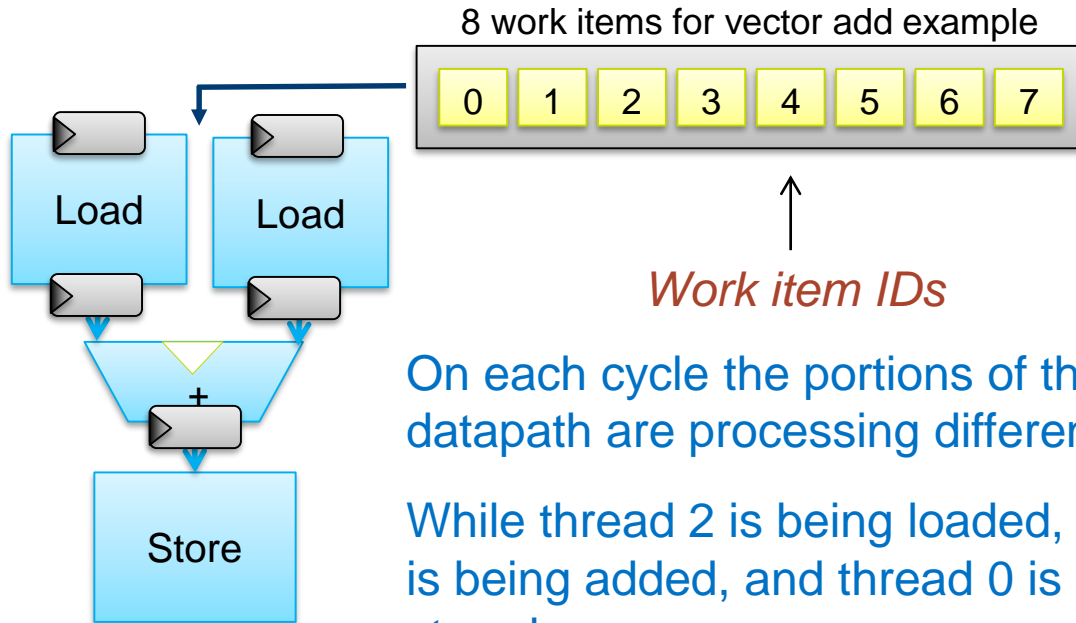
The most simple way of mapping kernel functions to FPGAs is to replicate the unrolled hardware for each thread

- Inefficient and wasteful

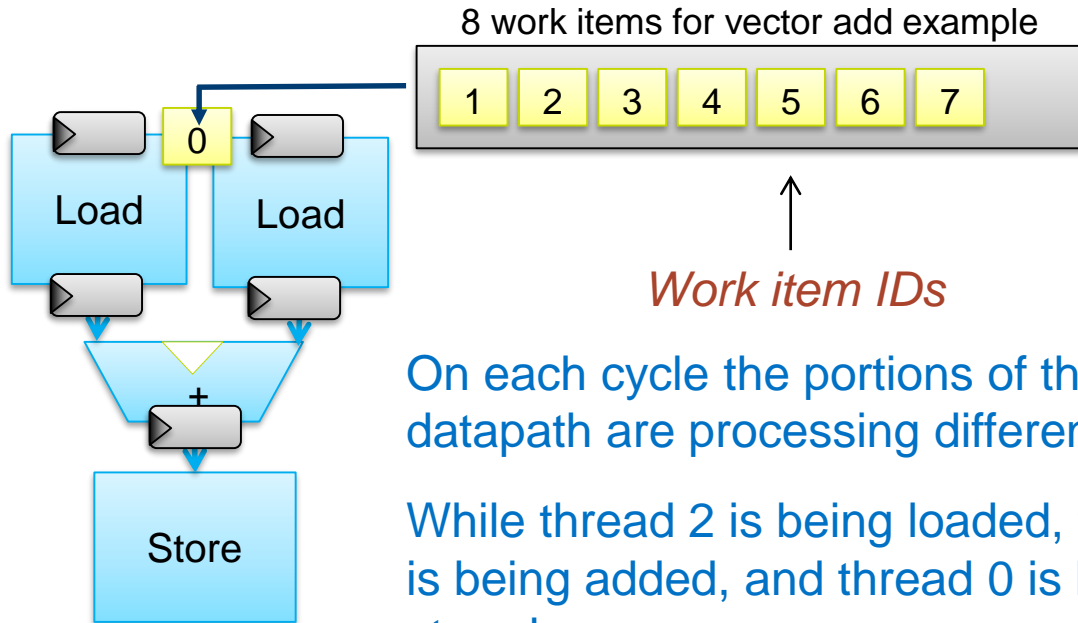
Better method involves taking advantage of **pipeline parallelism**

- Attempt to create a deeply pipelined representation of a kernel
- On each clock cycle, we attempt to send in input data for a new thread
- Method of mapping coarse grained thread parallelism to fine-grained FPGA parallelism

Example Datapath for Vector Add



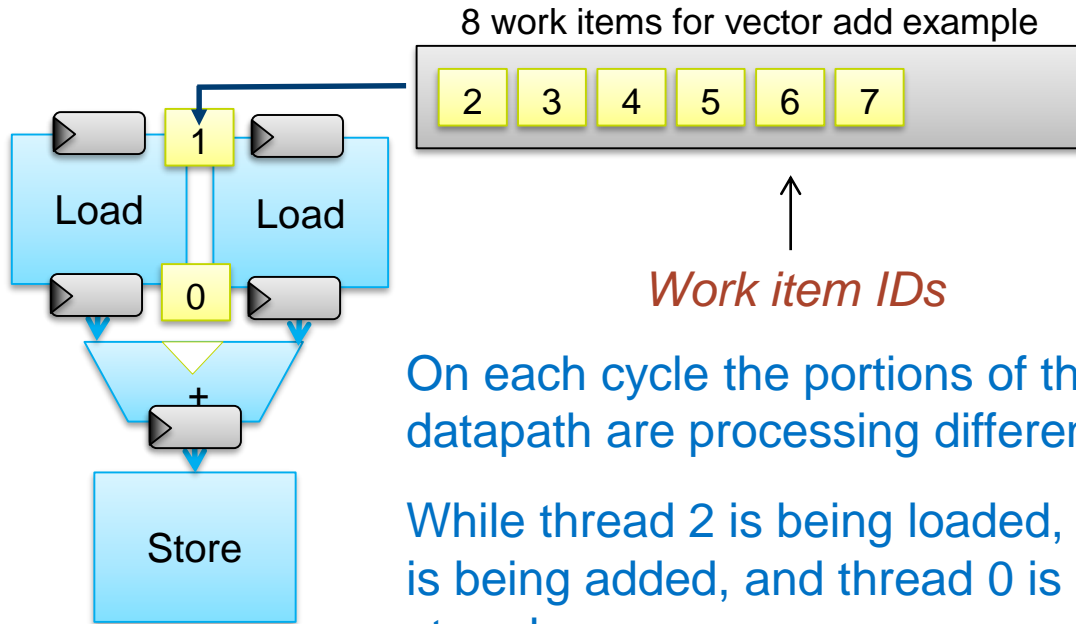
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

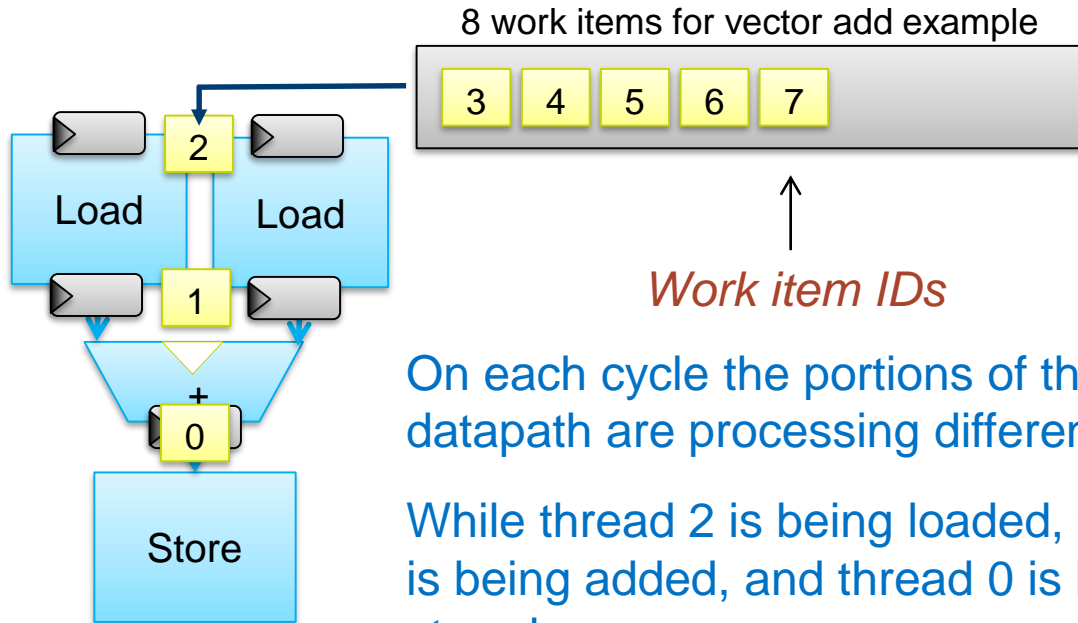
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

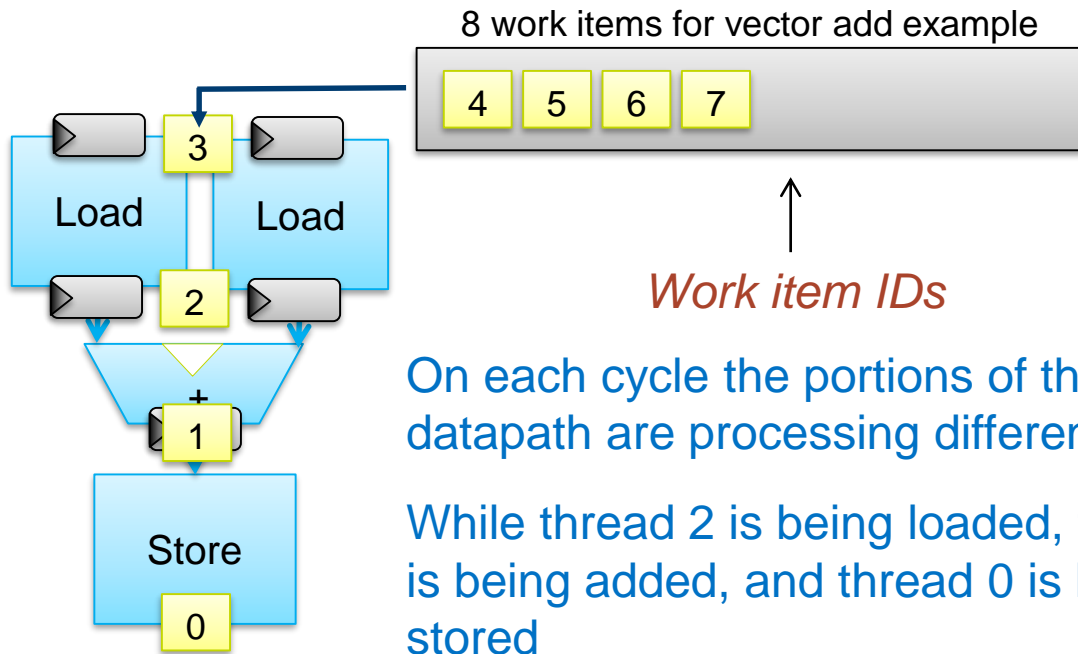
Example Datapath for Vector Add



On each cycle the portions of the datapath are processing different threads

While thread 2 is being loaded, thread 1 is being added, and thread 0 is being stored

Example Datapath for Vector Add

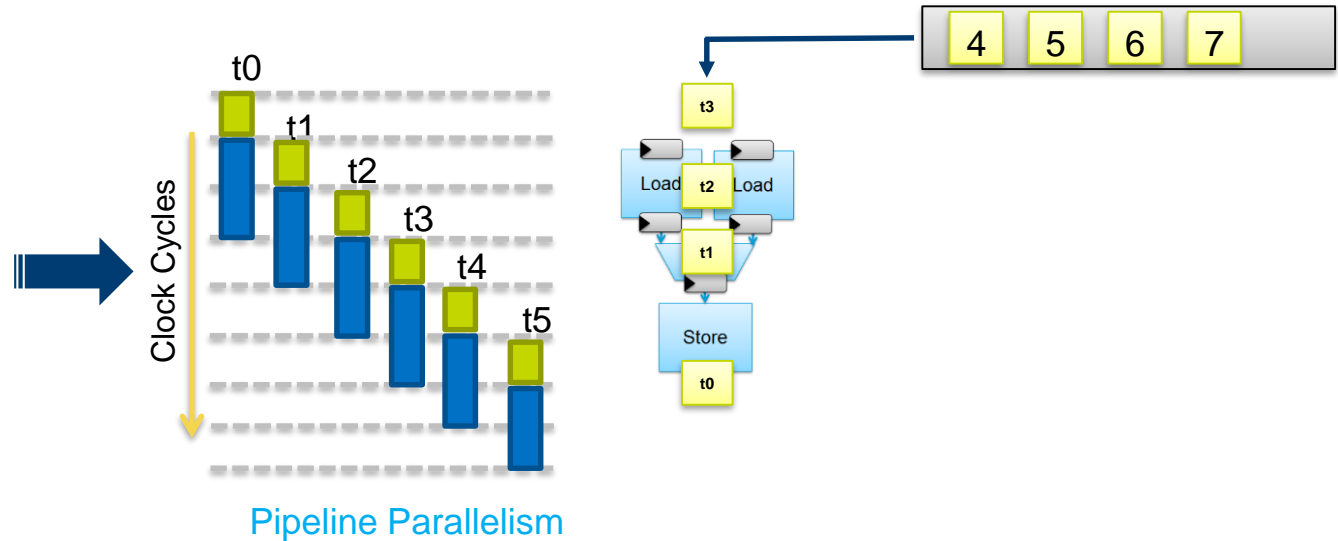


Silicon used efficiently at steady-state

Execution of Threads on FPGA

Better method involves taking advantage of *pipeline parallelism*

- *Throughput = 1 thread per cycle*



Pipeline parallelism execution

- A typical OpenCL™ kernel can have hundreds of pipeline stages
- This means, hundreds of simultaneous in-flight threads executing on the kernel.
- Very efficient usage of the inferred Hardware for maximize throughput

Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

- SDK components

- Debug Tools

- FPGA-specific Features

SDK Components

- Offline Compiler (AOC)
 - Translates your OpenCL™ C kernel source file into an Intel® FPGA hardware image
- Host Libraries
 - Provides the OpenCL host API to be used by OpenCL host applications
- AOCL Utility
 - Perform various tasks related to the board, drivers, and compile process

Intel® FPGA SDK for OpenCL™ Directory Structure

Directory	Description
bin	Main compiler and utility executables
windows64/bin linux64/bin	Runtime DLLs and other executables This should be in your path.
board	Design files related to specific supported boards
ip	IP cores required for kernel compilation
host	Files used by the compilation flow for user programs.
host/include	OpenCL™ API header files, and the interface files used to compile and link a user host program. Add this directory to the include file search path when compiling an OpenCL host program.
host/windows64/lib host/linux64/lib host/arm32/lib	The OpenCL host runtime libraries. Add this directory to the library file search path when linking an OpenCL host program.

Offline Kernel Compiler (aoc)

```
aoc -board=<my board> <my kernel file>
```

Option	Description
-help or -h	Help for the tool
-c	Creates .aoco object file and sets up a Quartus® Prime hardware design project
-rtl	Creates .aocr file that links all of the .aoco files
-board=<board name>	Compile for the specified board
-list-boards	Prints a list of available boards

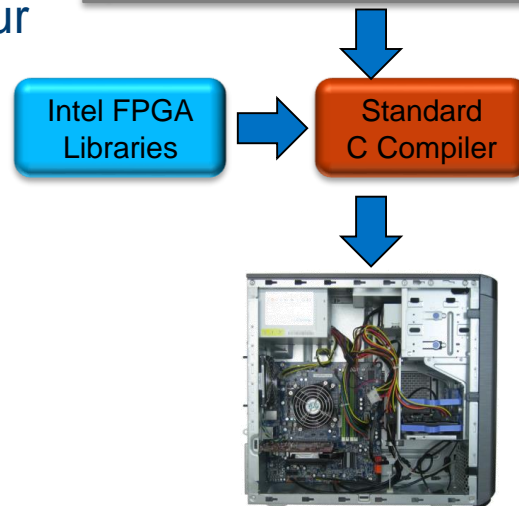
- Compiles kernels for a specific board defined by a board support package
- Generates aoco, and aocx files
- For detailed info on supported kernel constructs see the Intel® FPGA SDK for OpenCL™ programming Guide

There are many other debugging, optimization, and build options.

Compiling the Host Program

- Include `CL/opencl.h` or `CL/cl.hpp`
- Use a conventional C compiler (Visual Studio*/GCC)
- Add `$INTELFPGASDKROOT/host/include` to your file search path
 - Recommended to use `aocl compile-config`
- Link to Intel® FPGA OpenCL™ libraries
 - Link to libraries located in the `$INTELFPGASDKROOT/host/<OS>/lib` directory
 - Recommended to use `aocl link-config`

```
main() {  
    read_data( ... );  
    manipulate( ... );  
    clEnqueueWriteBuffer( ... );  
    clEnqueueNDRange( ..., sum, ... );  
    clEnqueueReadBuffer( ... );  
    display_result( ... );  
}
```



AOCL Utility

Host Compilation Commands (Use in your makefile)

<code>aocl compile-config</code>	Displays the compiler flags for compiling your host program
<code>aocl link-config</code>	Shows the link options needed by the host program to link with libraries
<code>aocl makefile</code>	Shows example Makefile fragments for compiling and linking a host program

Board Management Commands (Functionality Provided by BSP)

<code>aocl install</code>	Installs a board driver onto your host system
<code>aocl diagnose</code>	Runs the board vendor's test program
<code>aocl flash <.aocx></code>	Programs the on-board flash with the FPGA image over JTAG

View Kernel Compilation Report

<code>aocl report</code>	Displays kernel execution profiler data
--------------------------	---

Run `aocl help` or `aocl help <subcommand>` for detailed information about the tool

Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

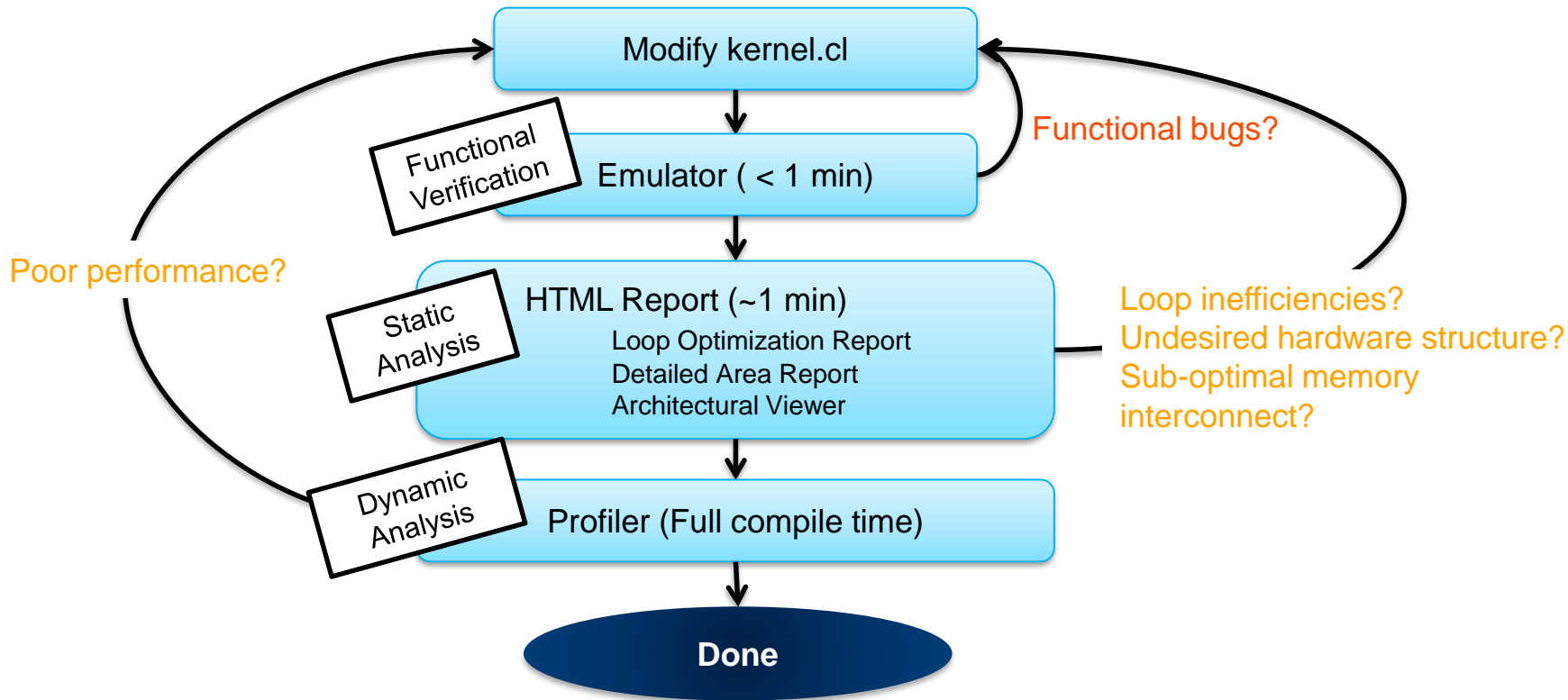
The Intel FPGA SDK for OpenCL

- SDK components

- Debug Tools**

- FPGA-specific Features

Kernel Development Flow and Tools



Emulator

Enable kernel functional debug on x86 systems

- Quickly generate x86 executables that represent the kernel

```
aoc -march=emulator <kernel file>
```

```
kernel void accel (...) {  
  ...  
  gid = get_global_id(0);  
  out[gid]=proc(data[gid]);  
  ...  
}
```



```
./kernel_tb...  
...  
Running ...
```

- Debug support for
 - Standard OpenCL™ syntax, Channels, Printf statements

HTML Report

Static report showing optimization, area, and architectural information

- Automatically generated with the object file (`aoc -rtl`)
 - Located in `<kernel file folder>\reports\report.html`
- Dynamic reference information to original source code
- Loop Analysis Optimization report
 - Information on how loops are implemented
- Area report
 - Detailed FPGA resource utilization by source code or system block
- Architectural viewer
 - Memory access implementation and kernel pipeline information

HTML Loop Analysis Optimization Report

- Actionable feedback on pipeline status of loops
 - Shows loop carried dependencies and bottlenecks
 - Especially important for single work-item kernels since they have an outer loop
- Shows loop unrolling status
- Shows loop nesting relationship

The screenshot displays a web browser window titled "Report: summation - Mozilla Firefox". The address bar shows the file path: `file:///home/student/fpga_trn/OpenCL_SW/OCLSW_18_0/summation/reports/report.ht`. The page content includes a "Reports" section with a "View reports..." link. Below this is a "Loops analysis" table with a checkbox for "Show fully unrolled loops" which is checked. The table has columns for "Kernel", "Loop", "Pipeline", "II", "Bottleneck", and "Details".

Kernel	Loop	Pipeline	II	Bottleneck	Details
summation (summation.cl:6)					Single work-item exec...
summation.B1 (summation.cl:8)		Yes	>+1	n/a	
summation.B2 (summation.cl:10)		Yes	-1	n/a	It is an approximation

Below the table is a "Details" section for "summation.B2:" which states: "It is an approximation due to the following stallable instruction:" followed by a bullet point: "Load Operation (summation.cl:12)".

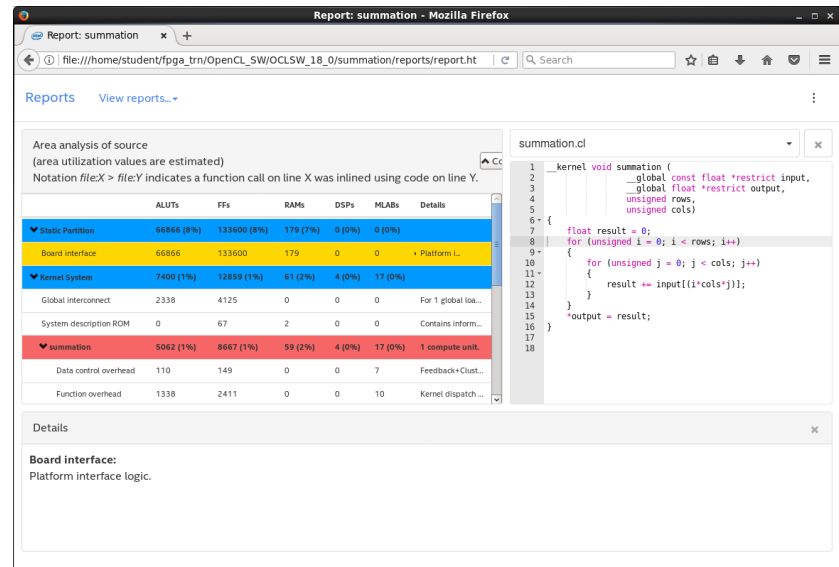
On the right side of the browser window, the source code for "summation.cl" is displayed:

```
1 kernel void summation (  
2     __global const float *restrict input,  
3     __global float *restrict output,  
4     unsigned rows,  
5     unsigned cols)  
6 {  
7     float result = 0;  
8     for (unsigned i = 0; i < rows; i++)  
9     {  
10        for (unsigned j = 0; j < cols; j++)  
11        {  
12            result += input[i*cols+j];  
13        }  
14    }  
15    *output = result;  
16 }  
17  
18
```

HTML Area Report

Generate detailed estimated area utilization report of kernel code

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



Report: summation - Mozilla Firefox

file:///home/student/fpga_trn/OpenCL_SW/OCLSW_18_0/summation/reports/report.ht

Reports View reports...

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

	ALUTs	FFs	RAMs	DSPs	MLABs	Details
▼ Static Partition	66866 (8%)	133600 (8%)	179 (7%)	0 (0%)	0 (0%)	
Board interface	66866	133600	179	0	0	Platform L...
▼ Kernel System	7400 (1%)	12859 (1%)	61 (2%)	4 (0%)	17 (0%)	
Global interconnect	2338	4125	0	0	0	For 1 global loa...
System description ROM	0	67	2	0	0	Contains inform...
▼ summation	5062 (1%)	8667 (1%)	59 (2%)	4 (0%)	17 (0%)	1 compute unit.
Data control overhead	110	149	0	0	7	Feedback+Clust...
Function overhead	1338	2411	0	0	10	Kernel dispatch...

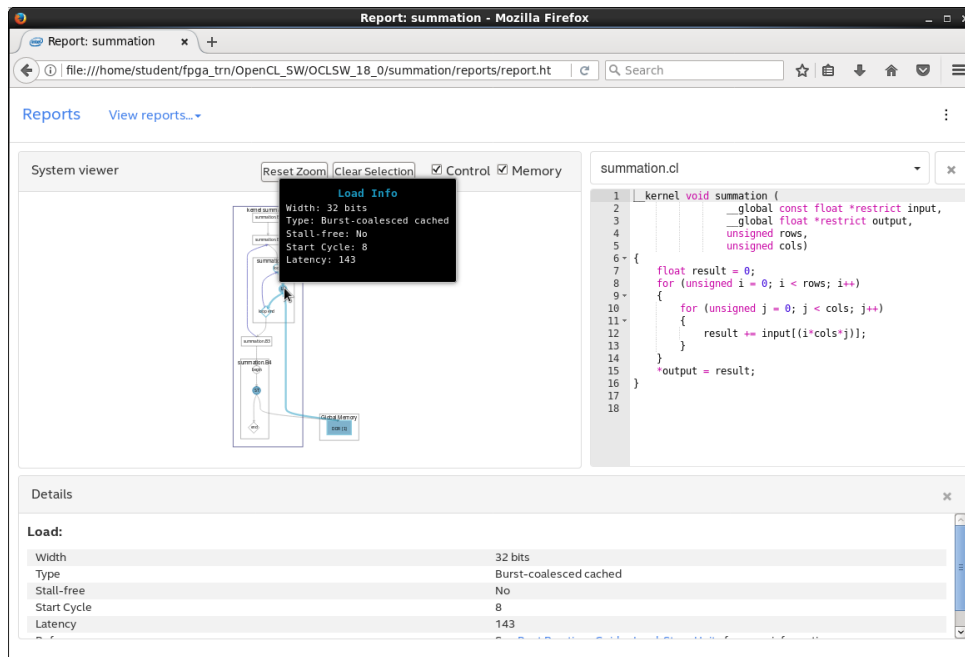
Details

Board interface:
Platform interface logic.

```
summation.cl
1 _kernel void summation (
2     __global const float *restrict input,
3     __global float *restrict output,
4     unsigned rows,
5     unsigned cols)
6 {
7     float result = 0;
8     for (unsigned i = 0; i < rows; i++)
9     {
10        for (unsigned j = 0; j < cols; j++)
11        {
12            result += input[(i*cols+j)];
13        }
14    }
15    *output = result;
16 }
17
18
```

HTML System Viewer

- Displays kernel pipeline implementation and memory access implementation
- Visualize
 - Off-chip memory
 - Load-store units
 - Accesses
 - Stalls
 - Latencies
 - On-chip memory
 - Implementation
 - Accesses



The screenshot displays the HTML System Viewer interface in a Mozilla Firefox browser window. The main area is divided into three sections:

- System viewer:** A diagram showing the kernel pipeline implementation. A tooltip for a 'Load Info' operation is visible, listing: Width: 32 bits, Type: Burst-coalesced cached, Stall-free: No, Start Cycle: 8, and Latency: 143.
- Code editor:** Shows the source code for 'summation.cl', which is a kernel function for matrix multiplication.
- Details panel:** A table providing detailed information for the selected 'Load' operation.

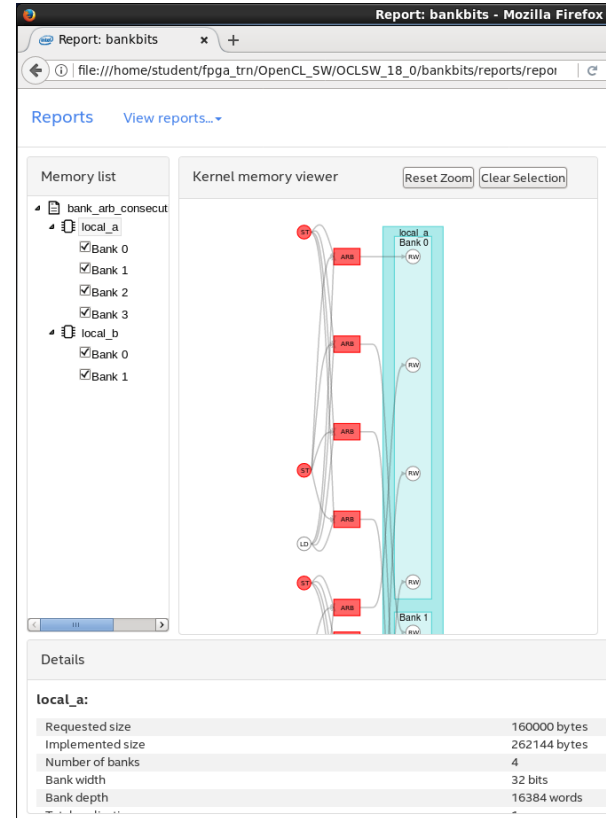
```
1 kernel void summation (
2     __global const float *restrict input,
3     __global float *restrict output,
4     unsigned rows,
5     unsigned cols)
6 {
7     float result = 0;
8     for (unsigned i = 0; i < rows; i++)
9     {
10        for (unsigned j = 0; j < cols; j++)
11        {
12            result += input[i+cols*j]);
13        }
14    }
15    *output = result;
16 }
17
18
```

Load:	
Width	32 bits
Type	Burst-coalesced cached
Stall-free	No
Start Cycle	8
Latency	143

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 shows the HTML Kernel Memory Viewer interface. The browser title is "Report: bankbits - Mozilla Firefox". The address bar shows the file path: "file:///home/student/fpga_trn/OpenCL_SW/OCLSW_18_0/bankbits/reports/repor". The interface includes a "Reports" section with a "View reports..." link. The "Memory list" section shows a tree view with the following structure:

- bank_arb_consecut
 - local_a
 - Bank 0
 - Bank 1
 - Bank 2
 - Bank 3
 - local_b
 - Bank 0
 - Bank 1

The "Kernel memory viewer" section displays a diagram of the memory banks. The diagram shows two vertical bars representing memory banks, labeled "local_a Bank 0" and "Bank 1". The "local_a Bank 0" bar is light blue and has four "RW" (Read/Write) ports. The "Bank 1" bar is light green and has two "RW" ports. On the left side, there are five "ARB" (Arbitration) ports, each connected to one of the "RW" ports. The "ARB" ports are labeled with "ST" (Read) and "LD" (Write) symbols. The diagram also shows "Reset Zoom" and "Clear Selection" buttons.

The "Details" section provides the following information for "local_a":

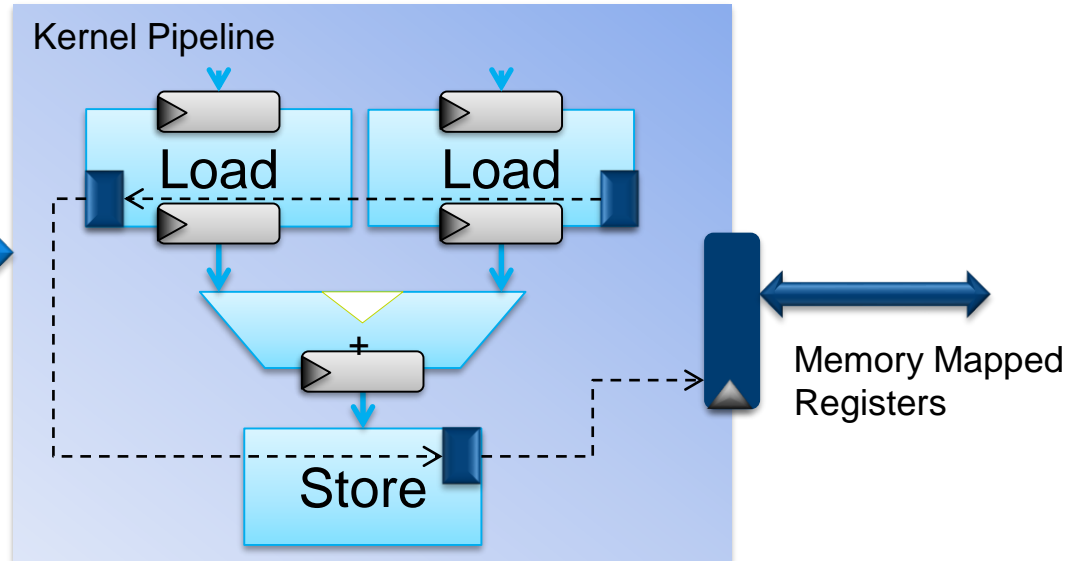
Property	Value
Requested size	160000 bytes
Implemented size	262144 bytes
Number of banks	4
Bank width	32 bits
Bank depth	16384 words

Profiler

- Inserts counters and profiling logic into the HW design
- Dynamically reports the performance of kernels

```
aoc --profile <kernel file>
```

```
kernel void accel(...) {  
    ...  
    gid = get_global_id(0);  
    out[gid] = a[gid]+b[gid];  
    ...  
}
```



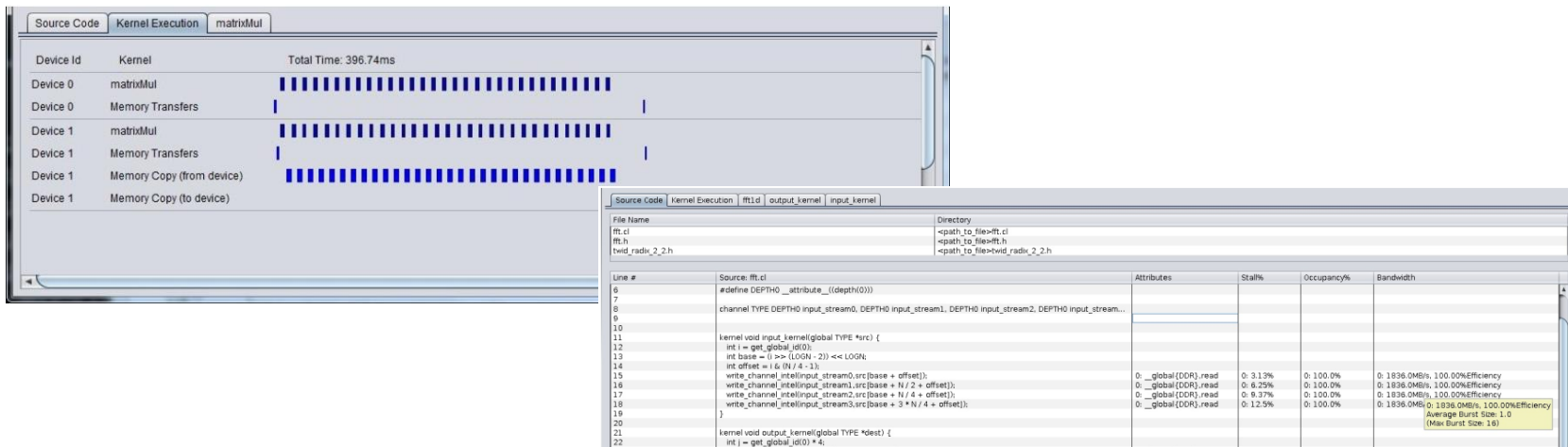
Collecting and Viewing Profile Information

- Compile kernel with `aoc --profile` option
 - `.source` file generated containing source information
- Run host application with generated `aocx` file
 - Performance counters will collect profile information
 - Host saves a `profile.mon` monitor description file to working directory
- View statistical data using the profiler GUI
 - Optionally provide `.source` file to view source code of profiled application

```
aocl report <kernel file>.aocx profile.mon [<kernel file>.source]
```


Profiler Reports

- Get runtime information about kernel performance
- Reports bottlenecks, bandwidth, saturation, and pipeline occupancy
 - At data access points



Class Agenda

Heterogeneous Parallel Computing

Intro to OpenCL for Intel FPGAs

OpenCL™ Platform model and Host-side Software

Executing OpenCL Kernels

- Writing & compiling kernels

- Launching kernels

- Harnessing Pipeline parallelism

The Intel FPGA SDK for OpenCL

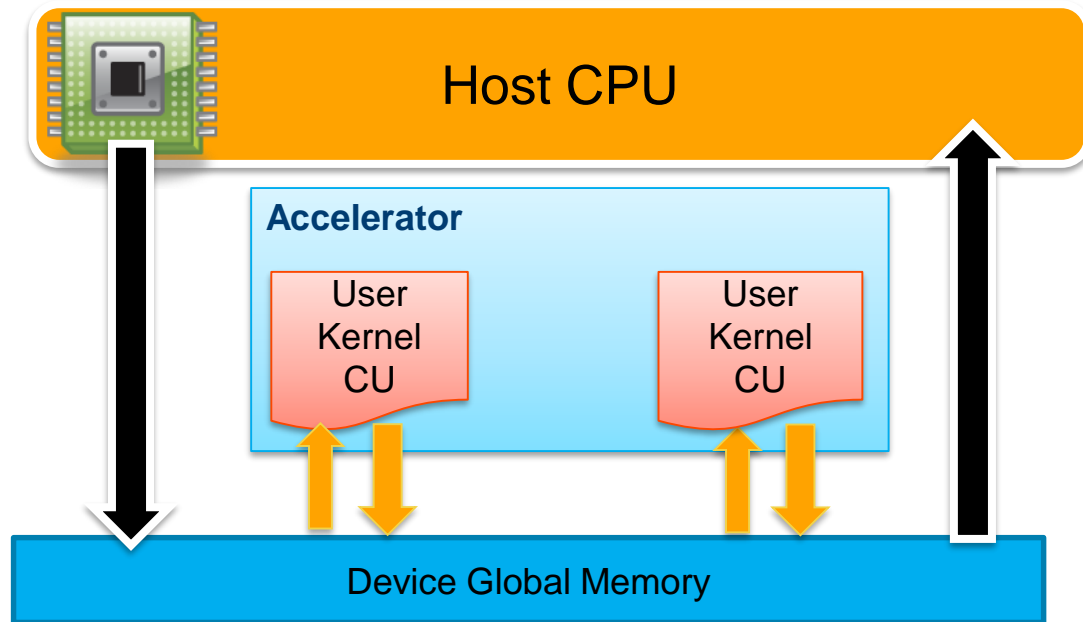
- SDK components

- Debug Tools

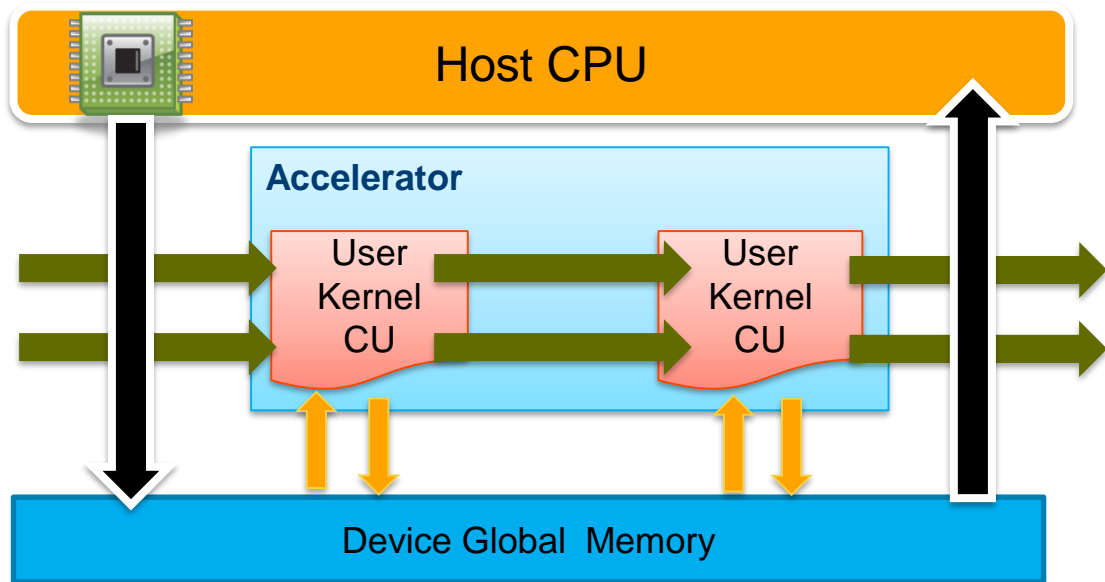
- FPGA-specific Features (channels, libraries)

Traditional OpenCL™: Host-Centric Architecture

All communication to/from kernels done through global memory



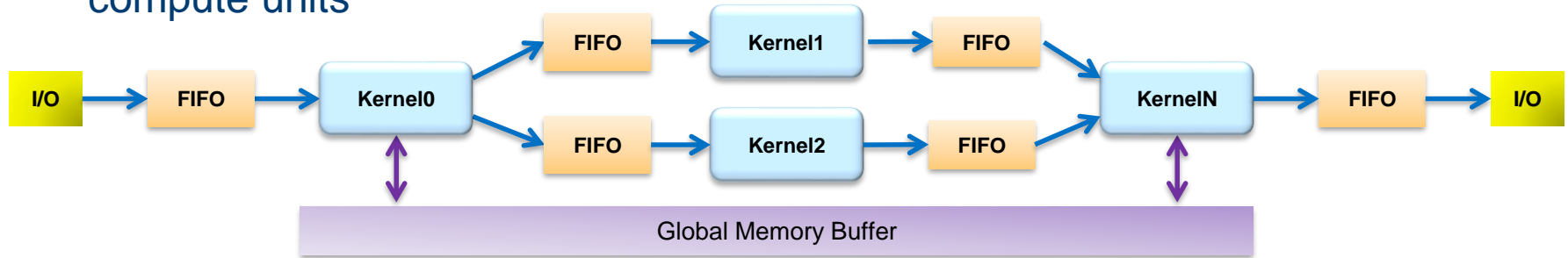
Idea: Communication without Global Memory



- Kernel-to-kernel communication done directly on-chip
- IO-to-kernel communication done without the host

Implementing FIFOs with Channels / Pipes

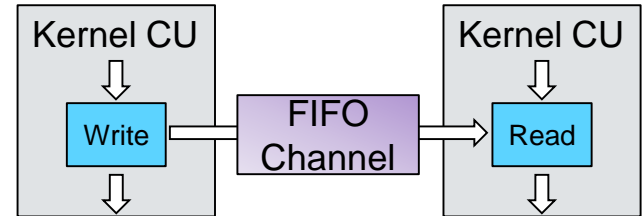
- Use FIFOs instead of global memory for efficient communication to/from kernel compute units



- Supported with Intel® FPGA's channels extension and OpenCL™ 2.0 pipes
- Works well with applications fitting into the general streaming template
 - E.g. Wireline Processing, Financial HFT Applications, Video Pipelines

Channels / Pipe Features

- Provides FIFO-like communication mechanism
- Each call site is unidirectional
- Allows BSP-specific I/O communication with kernel compute units
- Advantages
 - Leverage internal bandwidth of the FPGA
 - Avoid the bottleneck of using off-chip memory
 - Reduces overall latency by allowing concurrent Kernel execution
 - Reduce storage requirements when data is consumed as it is produced



Kernel-to-Kernel Channel Performance Gains

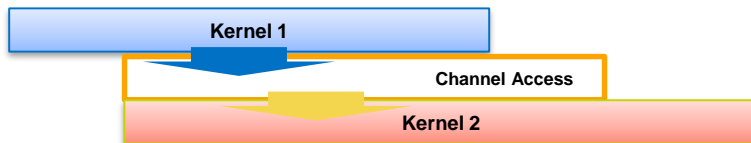
- Standard

- If communication between kernels is required, host forced to launch kernels sequentially
- Kernel 1 writes to global memory, kernel 2 reads from global memory



- With channels

- Host can launch kernels in parallel
- kernel 1 writes to channel as kernel 2 reads from it



IO Channel Performance Gains

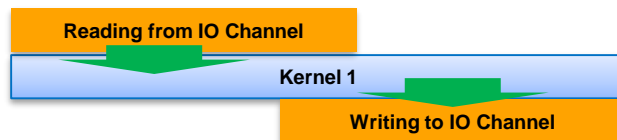
- Standard

- Data needs to be written to global memory first before kernel can process it and then read back after processing
- Limited by PCIe* bandwidth and memory throughput



- With IO channels

- Kernel can run while data flows across network interface
- System running at speed of network interface



I/O Channels

- Channels used with input or output features of a board
 - E.g., network interfaces, PCIe interfaces, camera interfaces, etc.
- Behavior defined by the Board Support Package (check `board_spec.xml`)

```
<channels>  
  <interface name="udp_0" port="udp0_out" type="streamsource" width="256" chan_id="eth0_in"/>  
  <interface name="pcie" port="tx" type="streamsink" width="32" chan_id="pcie_out" />  
</channels>
```

- Declaration of I/O channel using the `io` attribute

```
channel QUADWord udp_in_IO __attribute__((io("eth0_in")));  
channel float data __attribute__((io("pcie_out")));
```

- Usage same as other channels

- `data = read_channel_intel(udp_in_IO);`

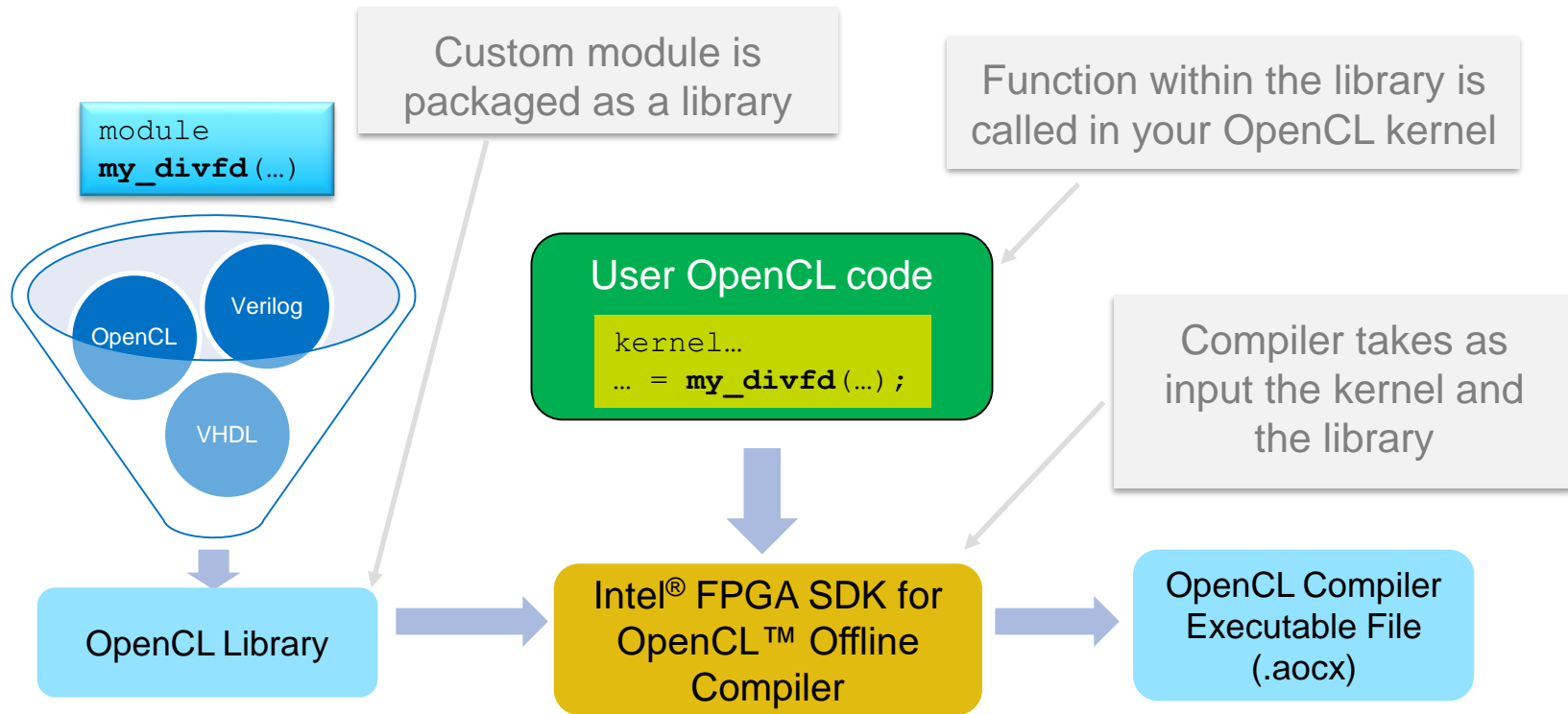
OpenCL™ Libraries

Create libraries from RTL or OpenCL™ source and call those library functions from user OpenCL code

Why use RTL modules?

- You want to use optimized and verified RTL modules in OpenCL™ kernels without rewriting the modules as OpenCL functions
- You want to implement OpenCL kernel functionality that you cannot express effectively in OpenCL

OpenCL™ Libraries



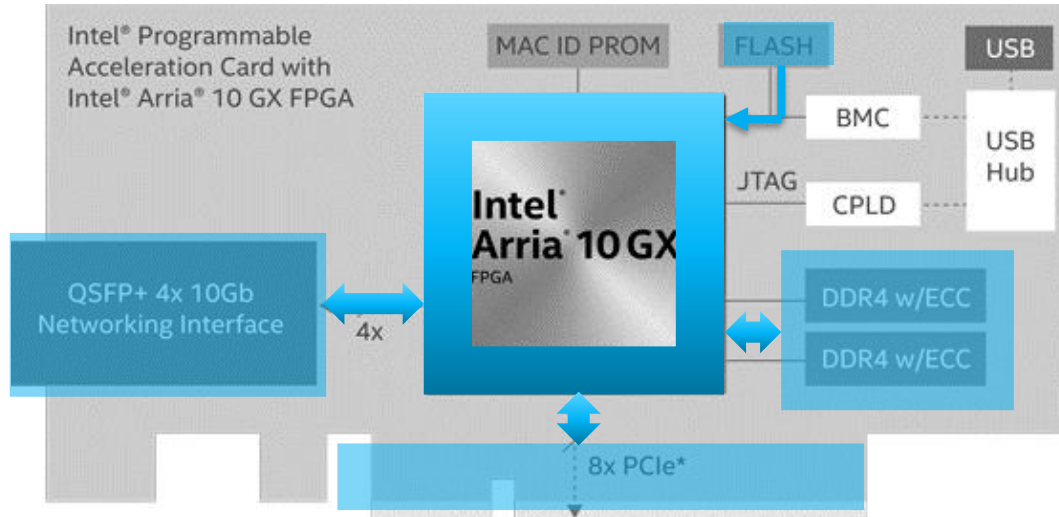
Developing a custom Board Support Package

When you need (or want) to use your own boards with OpenCL

- Framework of host software and FPGA interface design to enable the use of OpenCL™ on a custom board
- FPGA design, software, and board bring up skills required
- Custom BSP provides
 - Timing-closed Hardware
 - MMD software layer (drivers)
 - Some AOCL utility function

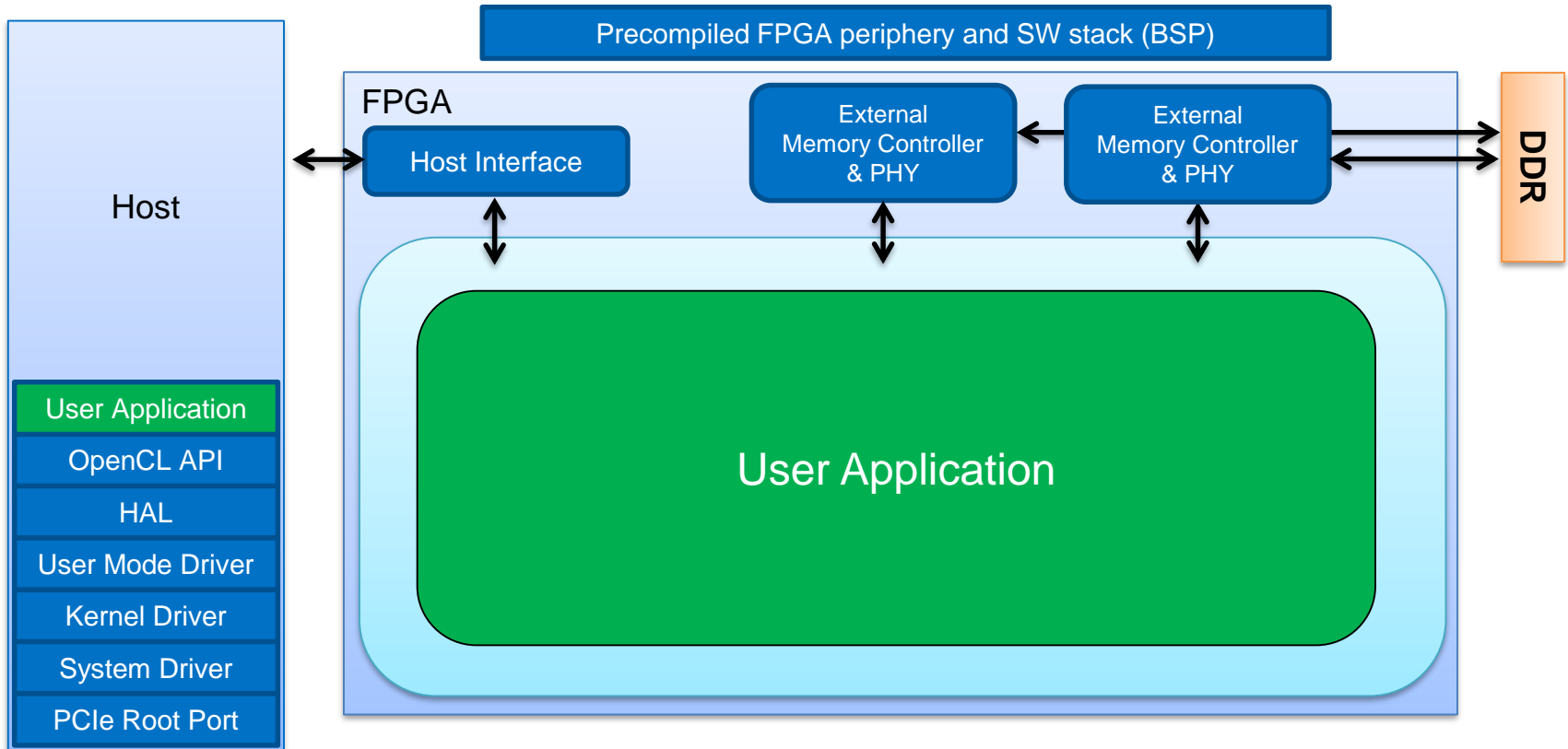
Working with custom boards

Remember the concept of BSP



- Board Support Package (BSP)
 - Initial FPGA image as “Chassis” to hold the newly created kernel

BSP includes some software stuff as well...



References and Documentation

- Intel® FPGA OpenCL collateral
 - <https://www.intel.com/content/www/us/en/software/programmable/sdk-for-openc/overview.html>
 - Intel FPGA SDK for OpenCL™ Getting Started Guide
 - Intel FPGA SDK for OpenCL Programming Guide
 - Intel FPGA SDK for OpenCL Best Practices Guide
 - Free Intel FPGA OpenCL Online Trainings
- Khronos* Group OpenCL Page
- OpenCL Reference Card
 - <https://www.khronos.org/files/OpenCLPP12-reference-card.pdf>

SUMMARY

- High-level parallel computing as the way to solve performance bottlenecks problems of your processing systems.
- OpenCL™ SDK with Intel® FPGAs facilitates the adoption of heterogeneous computing.
- We went through the basics of the OpenCL™ standard and how compile and run OpenCL™ programs using the available Intel® FPGA tools.



High-Level Synthesis with Intel® FPGAs

CNRS DAQ Seminar – Frejus, November 2018

OBJECTIVES

- Understand the concept of high-level synthesis for Intel® FPGAs
- Use the Intel HLS Compiler to synthesize, functionally verify, and simulate design IP for Intel FPGA
- Understand how the component executes on the FPGA

Class Agenda

Introduction to high-level synthesis with the Intel® HLS Compiler

HLS flow

HLS interfaces for integration in Platform Designer

Class Agenda

Introduction to high-level synthesis with the Intel® HLS Compiler

HLS flow

HLS interfaces for integration in Platform Designer

High Level Synthesis

Synthesize a C/C++ function in to an RTL implementation

- Develop the component in a software environment
- Functionally verify the component within a software environment
- Seamlessly integrate with hardware simulation environment
- Optimize design using software-centric tools and reports
- Integrate generated IP easily within traditional FPGA design tools

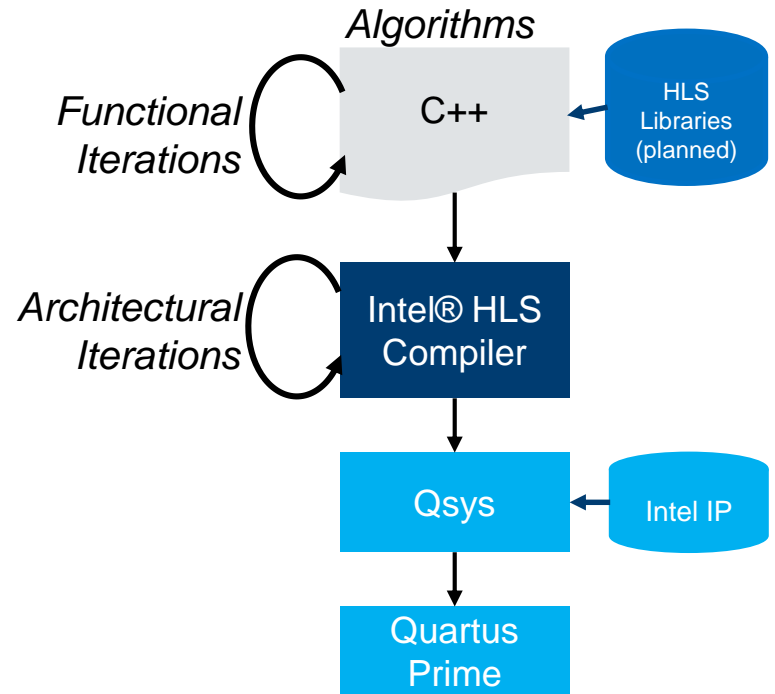
Intel® HLS Compiler

Accelerated Development

- Untimed C++ to optimized RTL
- Fast functional debug iterations
- Export to Platform Designer IP Library

Optimized Results

- Increased Fmax with Pipeline insertion
- Increased throughput with Parallelism
- Map to device hardware resources
- **Ability to target hard floating-point blocks with Intel FPGAs**



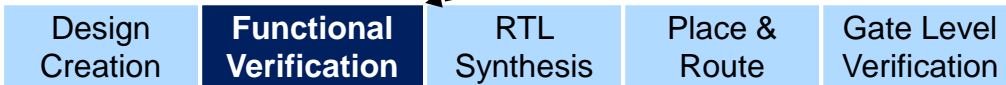
Accelerate FPGA design by raising abstraction layers to C++

Accelerate FPGA Development Cycles

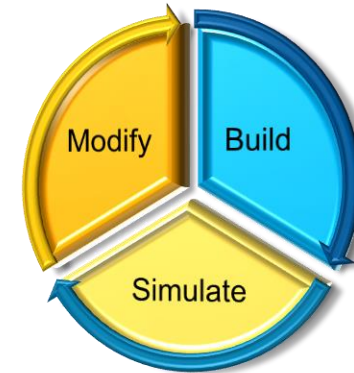
Traditional RTL Design Methodology



HLS Design Methodology



Fast Functional iteration Cycle



RTL vs Untimed C++ Functional Verification Times

Design	RTL Sim Time	C Sim Time	Acceleration
AES Encryption	22 mins	46 ms	29,000x
Huffman Encoding	13 mins	52ms	15,000x
Optical Flow	~2 Days	10 seconds	12,000x
Complex FIR Filter	4.5 min	63 ms	4,200x

Benchmark performed using the following hardware & software

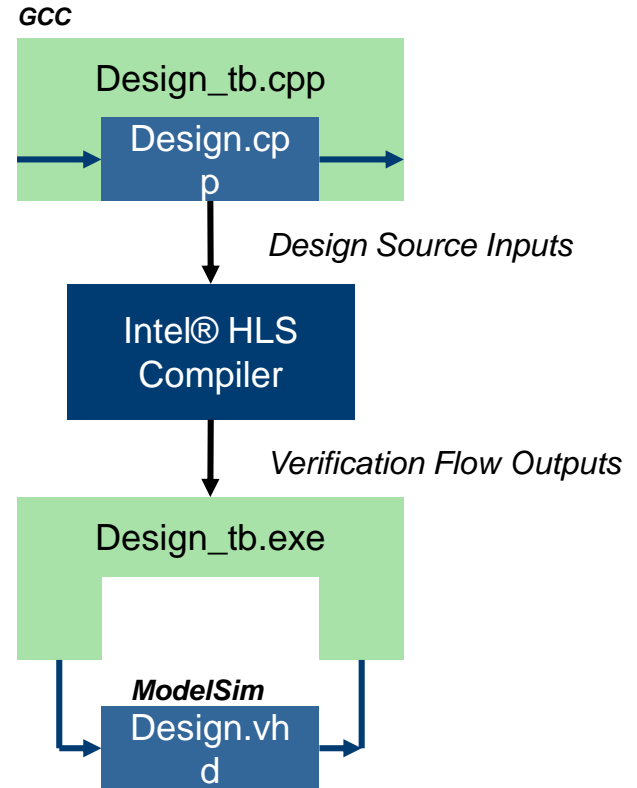
- Intel® HLS v0.9, ModelSim-SE-64 10.4d, Hardware: 2x8-core Intel Xeon ES-2680 @2.7 GHz, 256 GB RAM

Tests measure performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks

Automatically Verified RTL

Generated RTL is verified to the original C++ System Model

- New top-level C++ testbench executable is generated that supports ModelSim co-simulation
- Simulation files automatically generated and executed



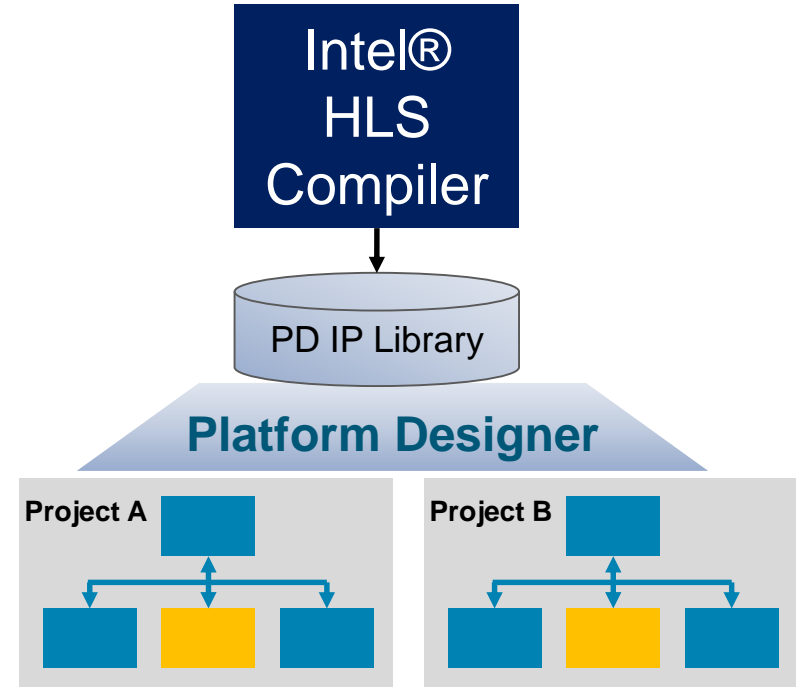
Easier Design Reuse Enabled through Abstraction

Easily reuse C++ based IP in multiple projects (building libraries)

- Parametrize with directives
 - Performance
 - Interfaces
 - Memories

C++ Source easier to modify vs RTL

Generate Library IP for use by Qsys Pro System design environment



Class Agenda

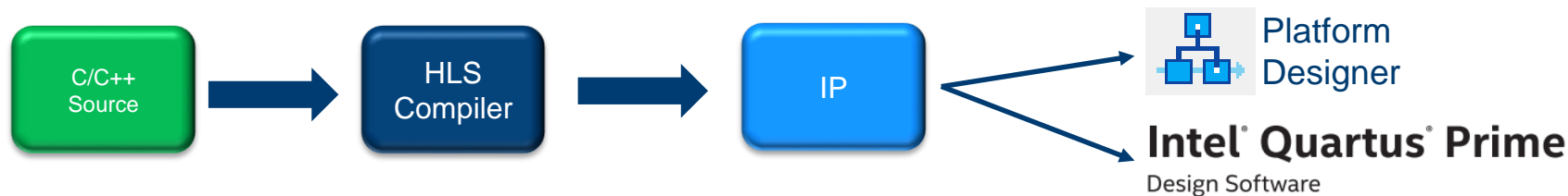
Introduction to high-level synthesis with the Intel® HLS Compiler

HLS flow

HLS interfaces for integration in Platform Designer

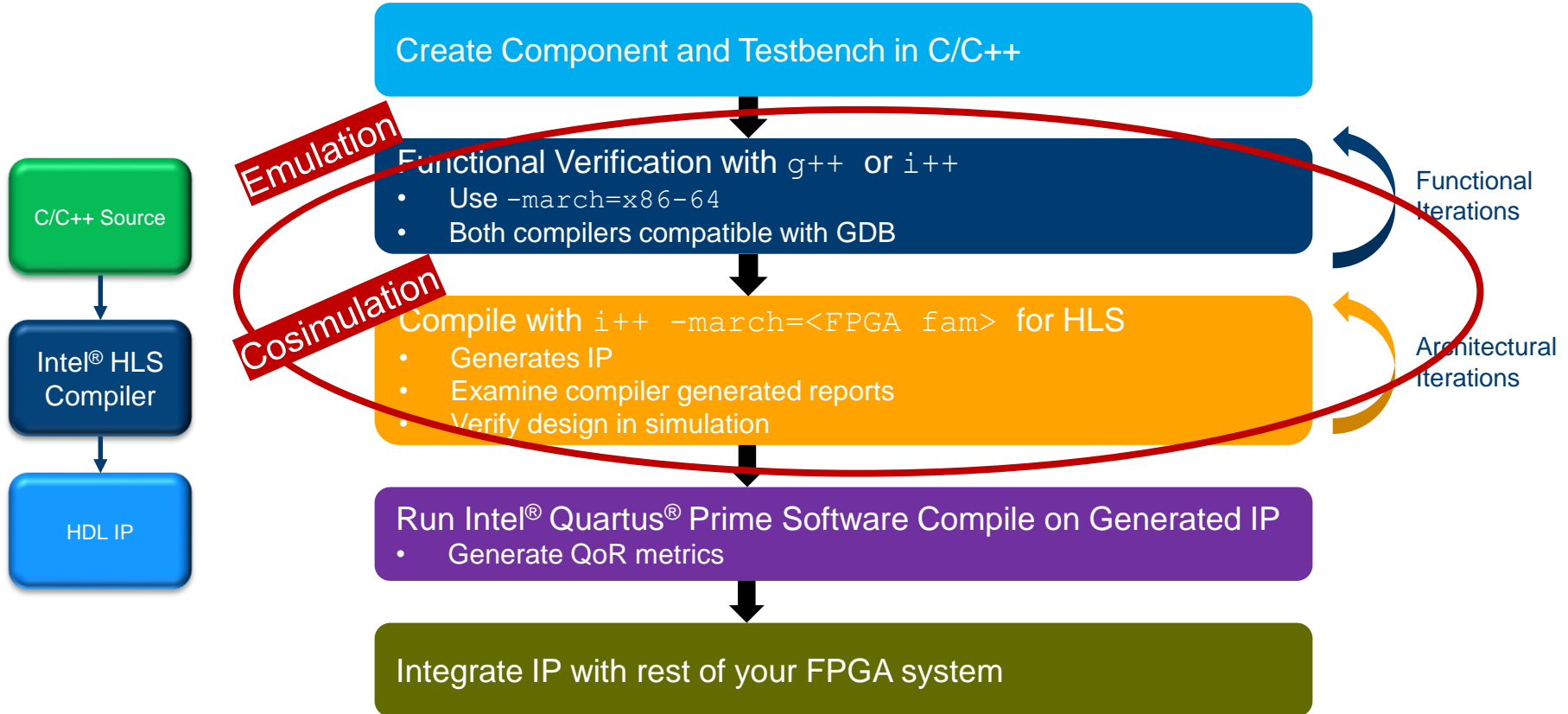
Intel® HLS Compiler

- Targets Intel® FPGAs
- Command-line executable: `i++`
- Builds an IP block
 - To be integrated into a traditional FPGA design using FPGA tools



- Leverages standard C/C++ development environment
- Goal: Same performance as hand-coded RTL with 10-15% more resources

HLS Procedure



Example Component/Testbench Source

```
#include "HLS/hls.h"
#include "assert.h"
#include "HLS/stdio.h"
#include "stdlib.h"

component int accelerate(int a, int b) {
    return a+b;
}

int main() {
    srand(0);
    for (int i=0; i<10; ++i) {
        int x=rand() % 10;
        int y=rand() % 10;
        int z=accelerate(x, y);
        printf("%d + %d = %d\n", x, y, z);
        assert(z == x + y);
    }
    return 0;
}
```

```
i++ -march=<fpga family> --component accelerate mysource.cpp
```

accelerate() becomes an FPGA component

- Use `--component i++` argument or component attribute in source

main() becomes testbench for component accelerate()

Cosimulation

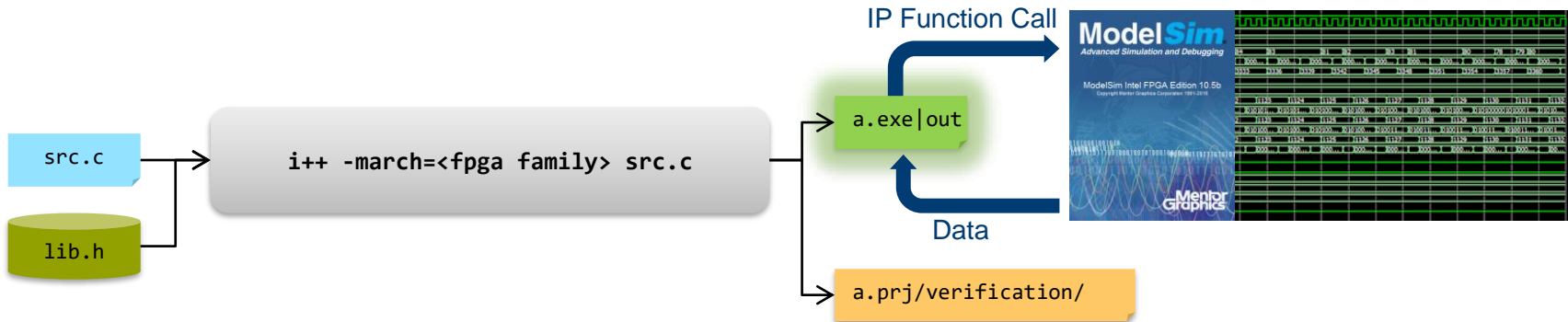
Cosimulation: combines x86 testbench with RTL simulation

- HDL code for the component runs in an RTL Simulator
 - Verilog
 - RTL testbench automatically created from software
- `main()` and everything else called from `main` runs on x86 as the testbench
- Communication using SystemVerilog Direct Programming Interface (DPI)
 - Allows C/C++ to interface SystemVerilog
 - Inter-process communication (IPC) library used to pass testbench input data to RTL simulator, and returns the data back to the x86 testbench

Cosimulation Verifying HLS IP

The Intel® HLS compiler automatically compiles and links C++ testbench with an instance of the component running in an RTL simulator

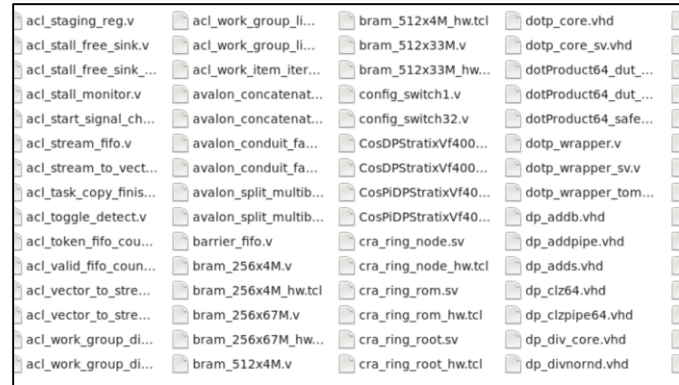
- To verify RTL behavior of IP, just run the executable generated by the HLS compiler targeting the FPGA architecture
 - Any calls to the component function becomes calls the simulator through DPI



C/C++ Functions to Dataflow Circuits

Each component function is converted into custom dataflow hardware

- Gain the benefits of Intel® FPGAs without the lengthy design process
- Implement C/C++ operators as circuits
 - HDL code located in <HLS Installation>\ip →
 - Load Store units to read/write memory
 - Arithmetic units to perform calculations
 - Flow control units
 - Connect circuits according to data flow in the function



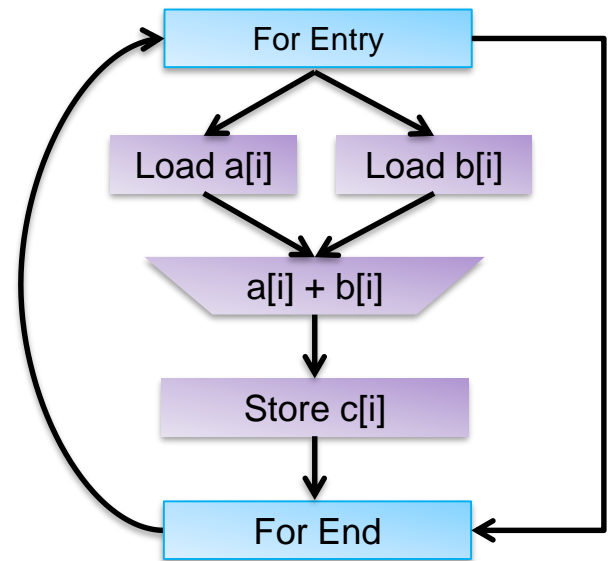
Compilation Example

Software compiled into dataflow circuit with flow control

- Include branch and merge units

```
void my_component (    int *a,  
                      int *b,  
                      int *c,  
                      int N)  
{  
    int i;  
    for (i = 0; i < N; i++)  
        c[i] = a[i] + b[i];  
}
```

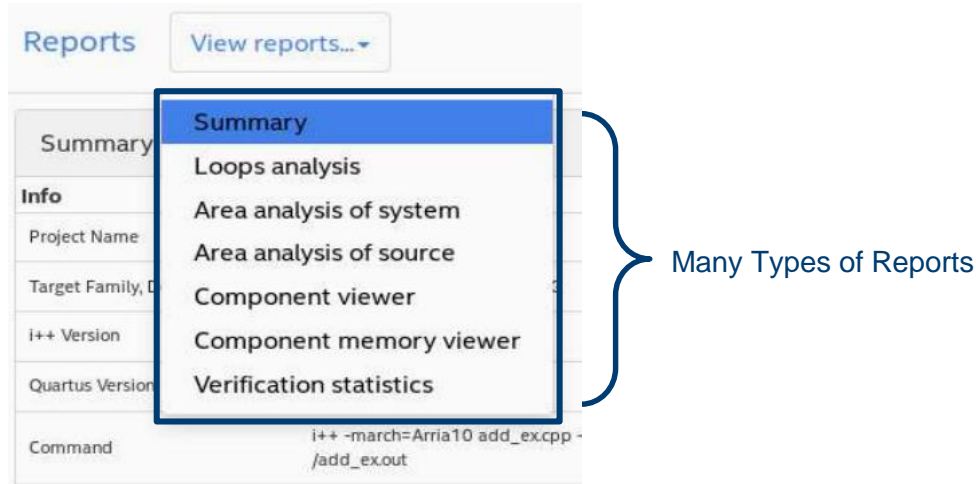
i++



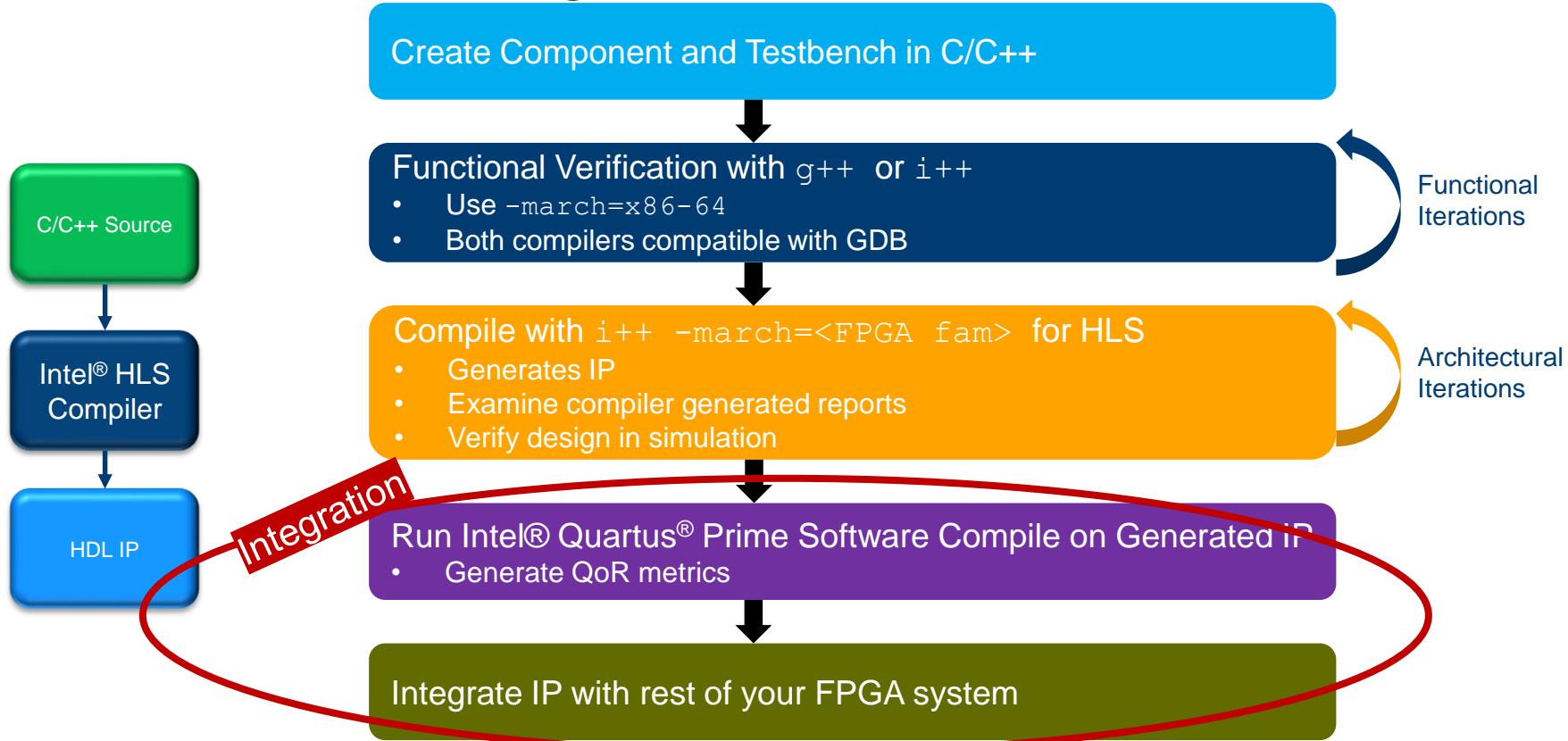
Main HTML Report

The Intel® HLS Compiler automatically generates HTML report that analyzes various aspects of your function including area, loop structure, memory usage, and system data flow

- Located at a `.prj/reports/report.html`



HLS Procedure: Integration



Intel® Quartus® Software Integration

- `a.prj/components` directory contains all the files to integrate
 - One subdirectory for each component
 - Portable, can be moved to a different location if desire
- 2 use scenarios
 1. Instantiate in HDL
 2. Adding IP to a Platform Designer system

HDL Instantiation

- Add Components to Intel® Quartus® Software Project
 - `<component>.qsys` to Standard Edition
 - `<component>.ip` to Pro Edition
- Instantiate component module in your design
 - Use template

`a.prj/components/<component>/<component>_inst.v`



```
add add_inst (
  // Interface: clock (clock end)
  .clock      ( ), // 1-bit clk input
  // Interface: reset (reset end)
  .resetn     ( ), // 1-bit reset_n input
  // Interface: call (conduit sink)
  .start      ( ), // 1-bit valid input
  .busy       ( ), // 1-bit stall output
  // Interface: return (conduit source)
  .done       ( ), // 1-bit valid output
  .stall      ( ), // 1-bit stall input
  // Interface: returndata (conduit source)
  .returndata( ), // 32-bit data output
  // Interface: a (conduit sink)
  .a          ( ), // 32-bit data input
  // Interface: b (conduit sink)
  .b          ( ) // 32-bit data input
);
```

Platform Designer System Integration Tool



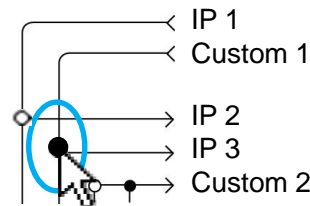
Catalog of available IP

- Interface protocols
- Memory
- DSP
- Embedded
- Bridges
- PLL
- Custom Components
- Custom Systems

Accelerate development

Use	Connections	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		clk	Clock Source		
<input checked="" type="checkbox"/>		clk_in	Clock Input	clk	exported
<input checked="" type="checkbox"/>		clk_in_reset	Reset Input	reset	
<input checked="" type="checkbox"/>		clk	Clock Output	Double-click to export	clk
<input checked="" type="checkbox"/>		clk_reset	Reset Output	Double-click to export	
<input checked="" type="checkbox"/>		pll	Altera PLL		
<input checked="" type="checkbox"/>		refclk	Clock Input	Double-click to export	clk
<input checked="" type="checkbox"/>		reset	Reset Input	Double-click to export	
<input checked="" type="checkbox"/>		outclk0	Clock Output	Double-click to export	sys_clk
<input checked="" type="checkbox"/>		outclk1	Clock Output	Double-click to export	ssram
<input checked="" type="checkbox"/>		locked	Conduit	Double-click to export	
<input checked="" type="checkbox"/>		reset_debounce	Reset Button Debounce		
<input checked="" type="checkbox"/>		clock	Clock Input	Double-click to export	sys_clk
<input checked="" type="checkbox"/>		button_debounce_in	Reset Input	Double-click to export	[dock]
<input checked="" type="checkbox"/>		button_qual	Reset Output	Double-click to export	[dock]
<input checked="" type="checkbox"/>		button_qual_n	Reset Output	Double-click to export	[dock]
<input checked="" type="checkbox"/>		button_switch	Button Switch PIO		
<input checked="" type="checkbox"/>		clock	Clock Input	Double-click to export	sys_clk
<input checked="" type="checkbox"/>		reset	Reset Input	Double-click to export	[dock]
<input checked="" type="checkbox"/>		buttonreg	Avalon Memory Mapped Slave	Double-click to export	[dock]
<input checked="" type="checkbox"/>		button_conduit	Conduit	Double-click to export	[dock]
<input checked="" type="checkbox"/>		switch_conduit	Conduit	Double-click to export	[dock]
<input checked="" type="checkbox"/>		av_sm_master	Avalon State Machine Master		buttons
<input checked="" type="checkbox"/>		clock	Clock Input	Double-click to export	switches

Connect custom IP and systems



Simplify integration

Automate integration tasks

HDL

Platform Designer Integration

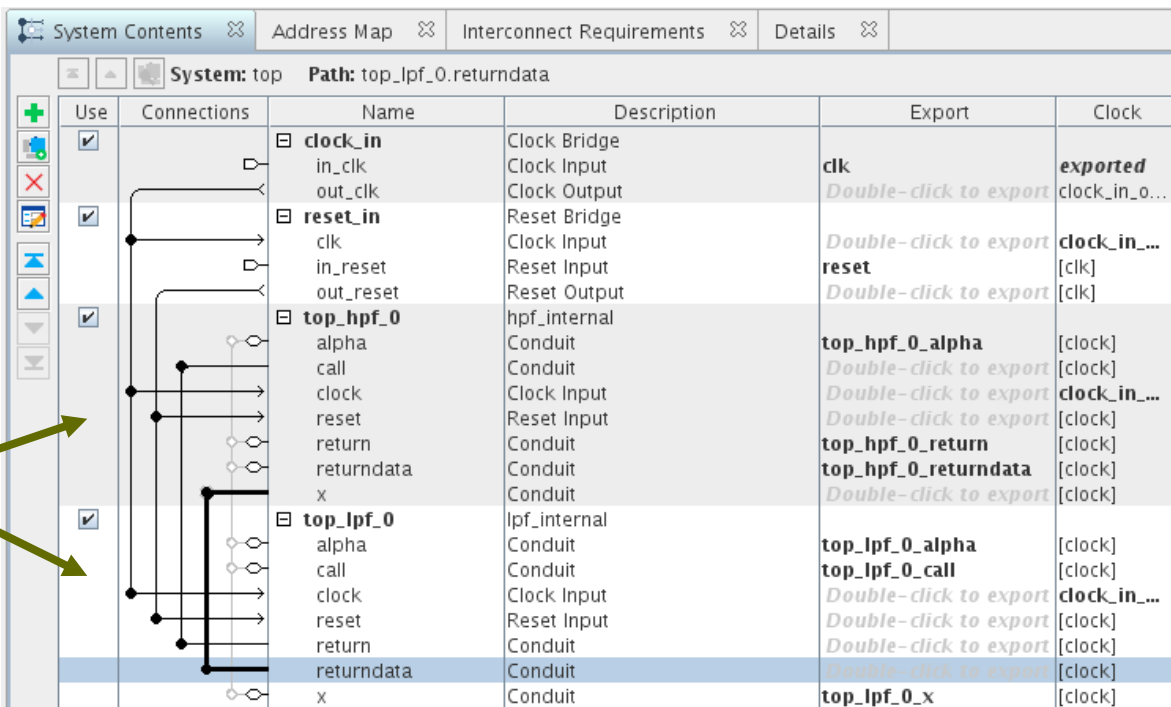
- Platform Designer component generated for each component:
 - For PD Standard – `a.prj/components/<component>/<component>.qsys`
 - For Platform Designer – `a.prj/components/<component>/<component>.ip`
- In Platform Designer, instantiate component from the IP Catalog in the HLS project directory
 - Add IP directory to IP Catalog Search Locations
 - May use `a.prj/components/**/*`
 - Can be stitched with other user IP or Intel® FPGA IP with compatible interfaces
- See tutorials under `tutorials/usability`

Platform Designer HLS Component Example

■ Example

- Cascaded low-pass filter and high-pass filter

HLS Components



Use	Connections	Name	Description	Export	Clock
<input checked="" type="checkbox"/>		clock_in	Clock Bridge		
		in_clk	Clock Input	clk	exported
		out_clk	Clock Output	Double-click to export	clock_in_o...
<input checked="" type="checkbox"/>		reset_in	Reset Bridge		
		clk	Clock Input	Double-click to export	clock_in_...
		in_reset	Reset Input	Double-click to export	[clk]
		out_reset	Reset Output	Double-click to export	[clk]
<input checked="" type="checkbox"/>		top_hpf_0	hpf_internal		
		alpha	Conduit	top_hpf_0_alpha	[clock]
		call	Conduit	Double-click to export	[clock]
		clock	Clock Input	Double-click to export	clock_in_...
		reset	Reset Input	Double-click to export	[clock]
		return	Conduit	top_hpf_0_return	[clock]
		returndata	Conduit	top_hpf_0_returndata	[clock]
		x	Conduit	Double-click to export	[clock]
<input checked="" type="checkbox"/>		top_lpf_0	lpf_internal		
		alpha	Conduit	top_lpf_0_alpha	[clock]
		call	Conduit	top_lpf_0_call	[clock]
		clock	Clock Input	Double-click to export	clock_in_...
		reset	Reset Input	Double-click to export	[clock]
		return	Conduit	Double-click to export	[clock]
		returndata	Conduit	Double-click to export	[clock]
		x	Conduit	top_lpf_0_x	[clock]

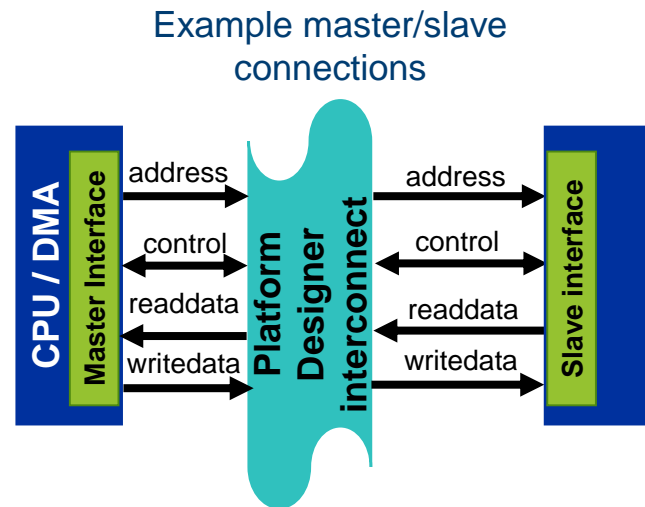
Avalon® Interfaces

Easily connects components in an Intel® FPGA to simplify system design

- Standard interfaces design for interoperability
- HLS compiler generates Avalon® interfaces around HLS components
- Avalon Streaming Interface (Avalon-ST)
 - Unidirectional flow of data, simple flexible interface
- Avalon Memory Mapped Interface (Avalon-MM)
 - Address-based read/write interface typical of master-slave connections
- Other Interfaces
 - Conduit, Tri-State Conduit, Interrupt, Clock, Reset

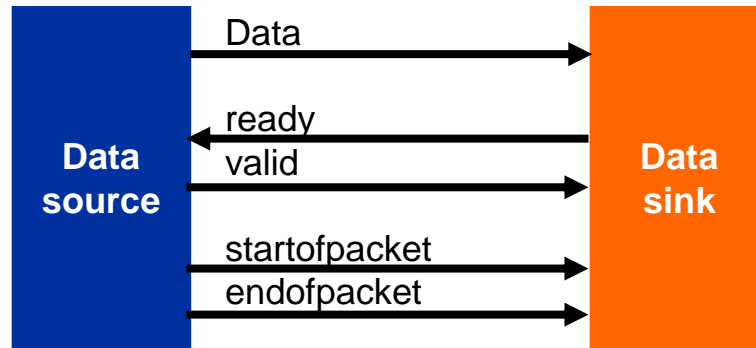
Avalon[®]-MM Interfaces

- Address-based (memory-mapped) protocol that allows components to communicate using read/write requests
- Master interface
 - Initiates read/write transfers targeting specific address
- Slave interface
 - Accepts and responds to transfer requests
- Interconnect handles decoding of master address request to actual slave interface, backpressure, clocking differences, etc.
- Associated with a clock interface



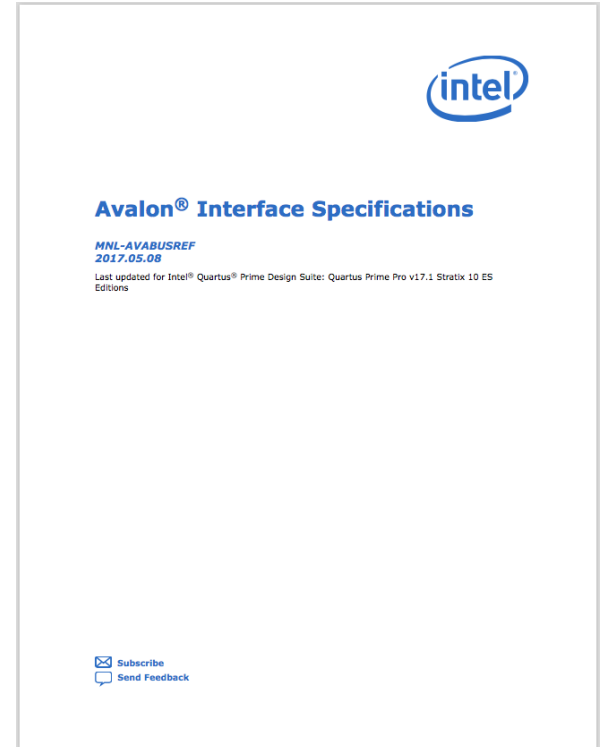
Avalon®-ST Interfaces

- Standard, flexible, and modular protocol for transfer of data
 - Unidirectional
 - Point-to-point connections
 - Fully synchronous
 - Supports simple and complex interface requirements



Avalon® Interface Specification

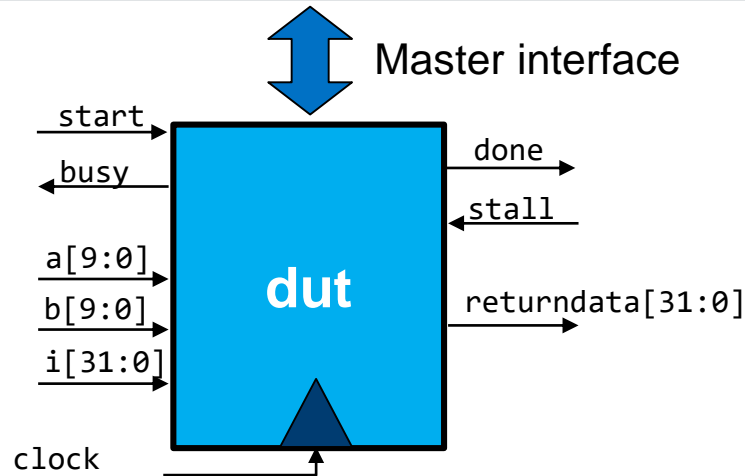
- Defines the entire Avalon interface standard, including all variations
- Provides reference information on additional transfer types
 - Use cases
 - Waveform diagrams
- http://www.altera.com/literature/manual/mnl_avalon_spec.pdf



Explicit MM Master Interface

```
component int dut(ihc::mm_master<int, ihc::aspace<2>, ihc::latency<0>,
                 ihc::awidth<10>, ihc::dwidth<32> > &a,
                 ihc::mm_master<int, ihc::aspace<2>, ihc::latency<0>,
                 ihc::awidth<10>, ihc::dwidth<32> > &b,
                 int i) {
    return a[i]*b[i];
}
```

- Explicitly declare Avalon-MM Master interfaces using `mm_master<>` class
 - Greater control over interface
 - Specify attributes through parameters

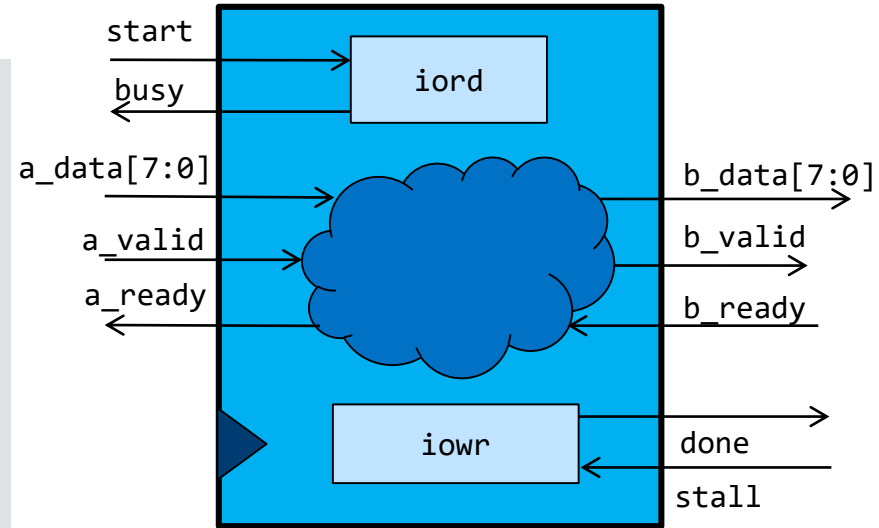


Streaming Interfaces

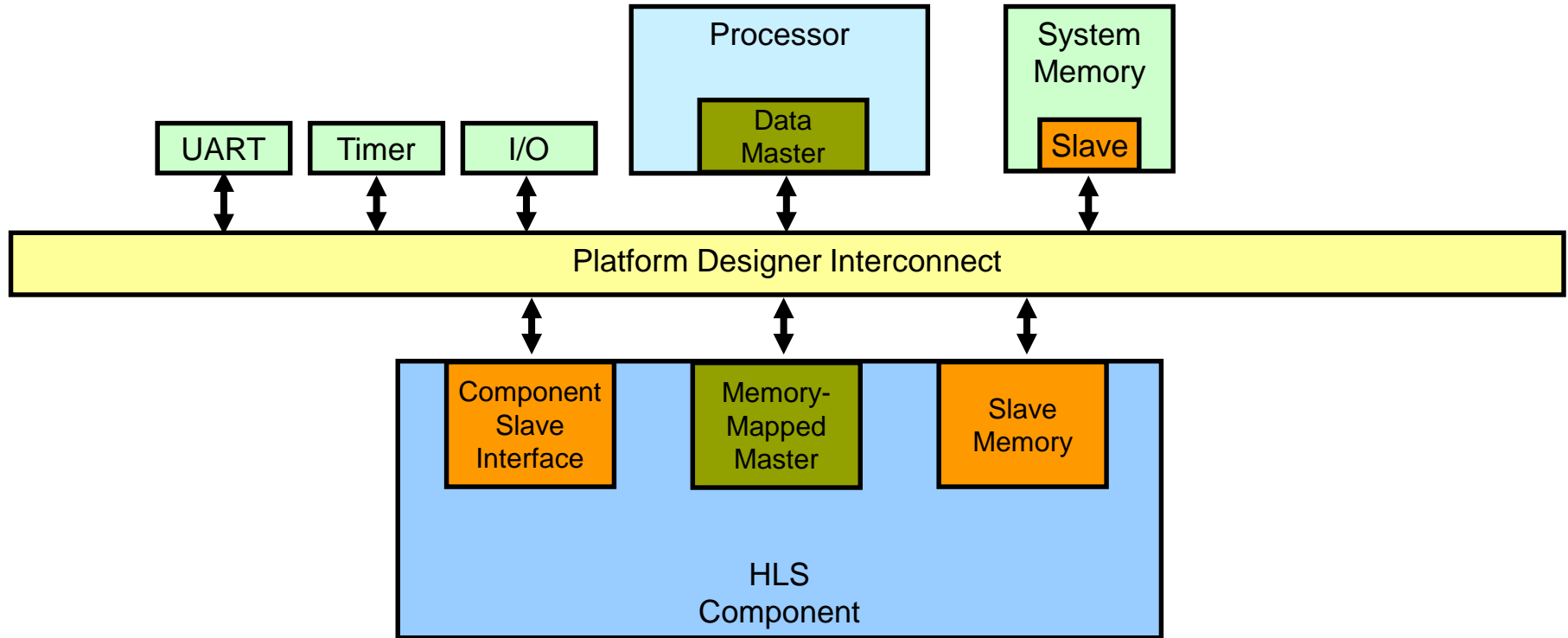
- Scalar function arguments become pipelined input ports on the HDL module
 - Avalon Streaming interface associated with `start` and `busy` inputs
 - Implicit
- Explicit Streaming Interfaces
 - Use `ihc::stream_in<>` and `ihc::stream_out<>` template classes
 - Pass by reference
 - Creates Avalon Streaming interface with valid and ready signals
 - Explicit control over interface

Explicit Streaming Interface Example

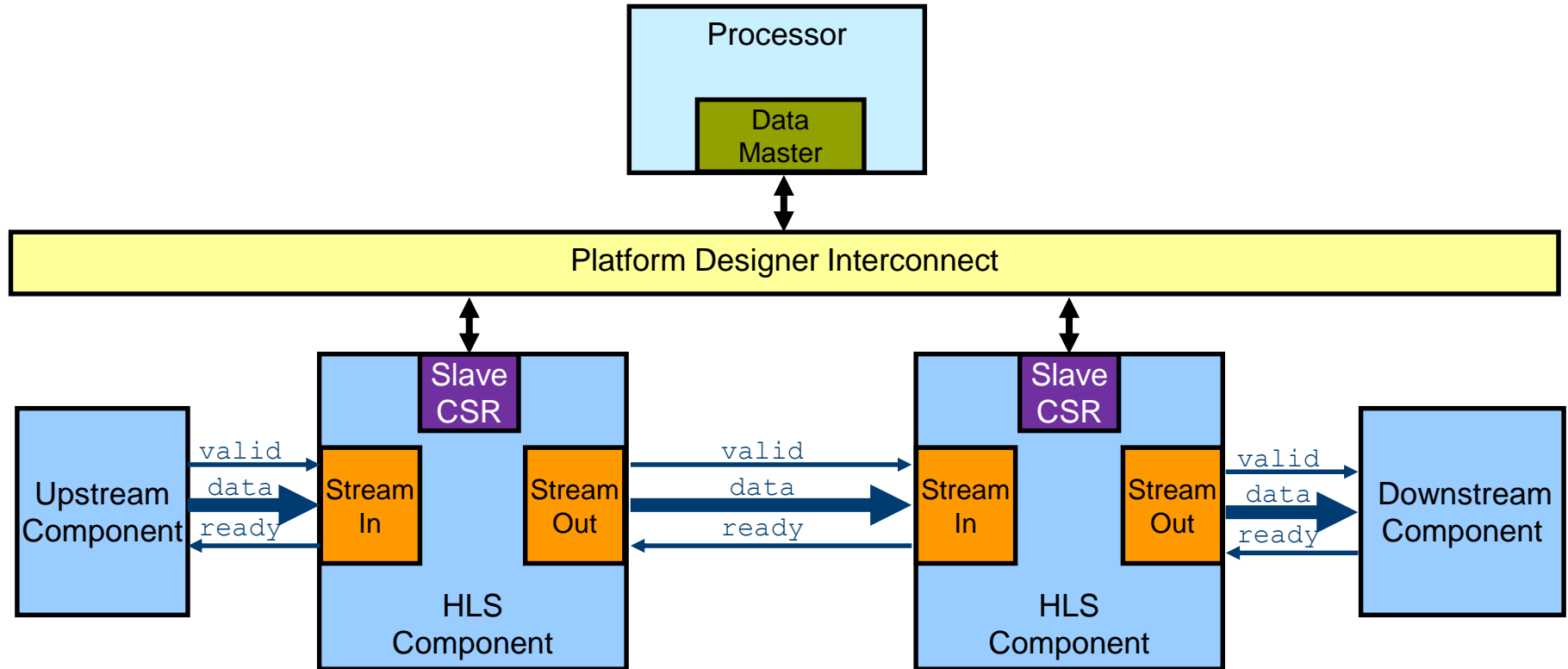
```
component
void dut(ihc::stream_in<unsigned char> &a,
        ihc::stream_out <unsigned char> &b)
{
    for (int i = 0; i < N; i++) {
        unsigned char input = a.read();
        input = 255 - input;
        b.write(input);
    }
}
```



Memory-Mapped HLS Component in a System



MM HLS Component with Streaming Interfaces



References and Documentation

- [Intel® FPGA high-level design tools landing page](#)
- [Intel HLS Compiler support page](#)
- References
 - *Intel HLS Compiler User Guide*
 - *Intel HLS Compiler Getting Started Guide*
 - *Intel HLS Compiler Reference Manual*
 - *Intel HLS Compiler Best Practices Guide*

SUMMARY

- Intel® HLS Compiler increases the designer productivity by raising the design entry abstraction from RTL to C++
- Shortens development time through accelerated verification
- Implements FPGA specific optimization techniques to deliver great quality of results

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

