SDAccel Development Environment Methodology Guide

Performance Optimization

UG1207 (v1.0) February 16, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
02/16/2016	1.0	Initial Xilinx release.

Table of Contents

Introduction

Overview

To achieve the highest possible acceleration of a software application, recent advances have included the development of multi-core and heterogeneous computing platforms. These architectures enable the software engineer to more effectively trade-off performance and power for different form factors and computational loads. The one challenge in using these new computing architectures is the programming model of each platform. All multi-core and heterogeneous computing platforms require the programmer to rethink the problem to be solved in terms of explicit parallelism.

Recognizing the programming challenge of multi-core and heterogeneous compute platforms, the Khronos™ Group industry consortium has developed the OpenCL™ programming standard [Ref 1]. The OpenCL specification for multi-core and heterogeneous compute platforms defines a single consistent programming model and system-level abstraction for all hardware platforms that support the standard. This means that a software engineer learns a single programming model and directly uses it on devices from multiple vendors.

Xilinx® is an active member of the Khronos Group, collaborating on the specification of OpenCL, and supports the compilation of OpenCL programs for Xilinx FPGA devices. SDAccel™ is the Xilinx® development environment for compiling OpenCL programs to execute on Xilinx FPGA devices.

The OpenCL standard guarantees functional portability but not performance portability. Therefore, even though the same code will run on every platform supporting OpenCL, the performance achieved will vary depending on coding style and capabilities of the underlying hardware. Optimizing for an FPGA using the SDAccel tool chain requires the same effort as code optimization for a CPU/GPU. The one difference in optimization for these platforms is that in a CPU/GPU, the programmer is trying to get the best mapping of an application onto a fixed architecture. For an FPGA, the programmer is concerned with guiding the compiler to generate optimized compute architecture for each accelerator (referred to as a *kernel*) in the application.

As specified by the OpenCL standard, any code that complies with the OpenCL specification is functionally portable and will execute on any computing platform that supports the standard. Therefore, any code changes are for performance optimization. To aid the user in these optimizations, SDAccel offers performance profiling capabilities integrated into the run-time. This profiling helps the user analyze the achieved performance and pinpoint any potential bottlenecks that need to be addressed.

Guide Organization

This User Guide employs the integrated profiling in SDAccel to analyze and understand the implications of OpenCL constructs on FPGA performance. This guide uses a few key designs as vehicles to illustrate performance characteristics and in turn, suggests design techniques to write OpenCL accelerators using FPGAs. The chapters in this guide are organized as follows:

Chapter 2: What is an FPGA?

This chapter introduces the computational elements available on an FPGA and how they compare to a processor. It covers topics such as FPGA memory hierarchy, logic elements, and how these elements interrelate.

Chapter 3: What is OpenCL?

This chapter introduces the basic concepts of the OpenCL programming standard. It provides an overview of the standard, provides definitions of terminologies used in the standard, and describes how FPGAs are uniquely suited for the parallel computational aspects of the standard.

Chapter 4: Data Path Optimizations

This chapter describes a matrix adder kernel example and steps through the performance optimizations applied to the design. These optimizations are primarily targeted to improve data path performance, and their effects on the overall design performance are described in detail.

Chapter 5: Memory Access Optimizations

This chapter steps through a second design example implementing the Smith-Waterman algorithm. This algorithm performs local sequence alignment and is often applied to protein or nucleic acid sequences. The optimizations applied to this design primarily improve memory accesses, and their effects on the overall design performance are described in detail.

Chapter 6: Putting It All Together

This chapter provides a third design example accelerating a 3x3 median filter. This algorithm is applied to images and is excellent for removing certain types of noise. The optimizations used to improve this design target both data path and memory accesses. These optimizations are gradually applied to improve the kernel, and their effects are described in detail.

Chapter 7: Performance Checklist

This chapter provides a starting checklist that can generally be applied to any OpenCL kernels targeting FPGA devices. It contains a list of suggested items to consider when improving the performance of your kernel. This chapter also provides a few tool flow suggestions which leverage the many capabilities of SDAccel.

Appendix A: Improving Data Path Performance

This chapter highlights a few key techniques to improve the data path performance within OpenCL kernels. These techniques take the form of Tcl parameters, data types, or attributes in the OpenCL C kernel source code. This chapter describes their usage in SDAccel and demonstrates their effects on performance.

Appendix B: Improving Memory Efficiency

This chapter highlights some key techniques to improve memory efficiency of OpenCL kernels. These techniques take the form of Tcl parameters, usage of local memories, and other architectural concepts for OpenCL C kernels. This chapter describes their usage in SDAccel and demonstrates their effects on performance.

What is an FPGA?

Overview

An FPGA is an integrated circuit (IC) that can be programmed for different algorithms after fabrication. Modern FPGA devices consist of up to two million logic cells that can be configured to implement a variety of software algorithms. Although the traditional FPGA design flow is more similar to a regular IC than a processor, an FPGA provides significant cost advantages in comparison to an IC development effort and offers the same level of performance in most cases. Another advantage of the FPGA when compared to the IC is its ability to be dynamically reconfigured. This process, which is the same as loading a program in a processor, can affect part or all of the resources available in the FPGA fabric.

When using SDAccel, it is important to have a basic understanding of the available resources in the FPGA fabric and how they interact to execute a target application. This chapter presents fundamental information about FPGAs, which is required to guide SDAccel to the best computational architecture for any algorithm.

FPGA Architecture

Xilinx FPGAs are heterogeneous compute platforms that include Block RAMs, DSP Slices, PCI Express support, and programmable fabric. They enable parallelism and pipelining of applications across the entire platform as all of these compute resources can be used simultaneously. SDAccel is the tool provided by Xilinx to target and enable these compute resources for OpenCL programs.

The basic structure of an FPGA is composed of the following elements:

- Look-up table (LUT) - This element performs logic operations.
- Flip-Flop (FF) - This register element stores the result of the LUT.
- Wires - These elements connect elements to one another.
- Input/Output (I/O) pads - These physical ports get data in and out of the FPGA.

The combination of these elements results in the basic FPGA architecture shown in Figure 2-1. Although this structure is sufficient for the implementation of any algorithm, the efficiency of the resulting implementation is limited in terms of computational throughput, required resources, and achievable clock frequency.

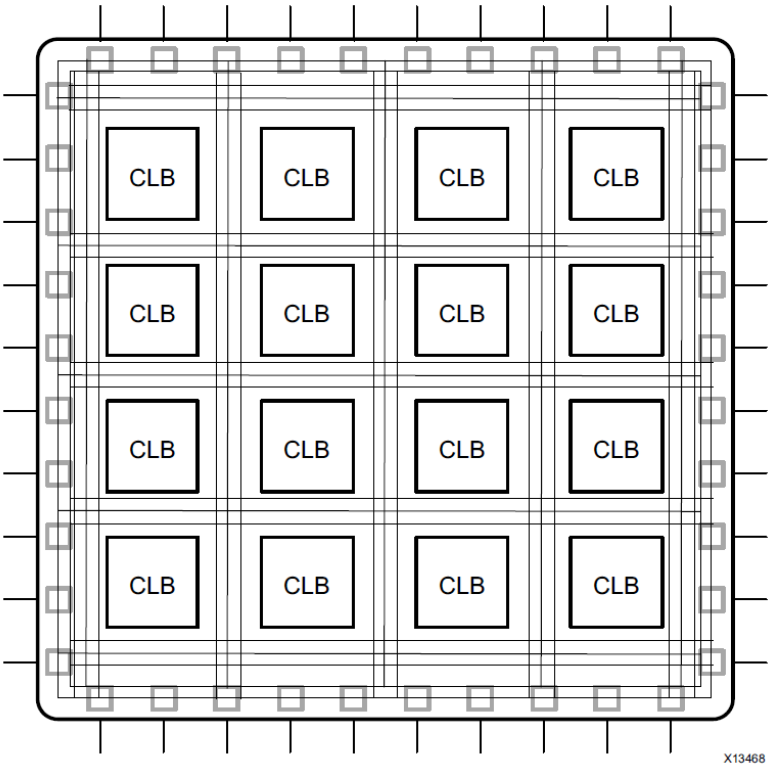

Figure 2-1: **Basic FPGA Architecture**

Contemporary FPGA architectures incorporate the basic elements along with additional computational and data storage blocks that increase the computational density and efficiency of the device. These additional elements, which are discussed in the following sections, include:

- Embedded memories for distributed data storage
- Phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates
- High-speed serial transceivers
- Off-chip memory controllers
- Multiply-accumulate blocks

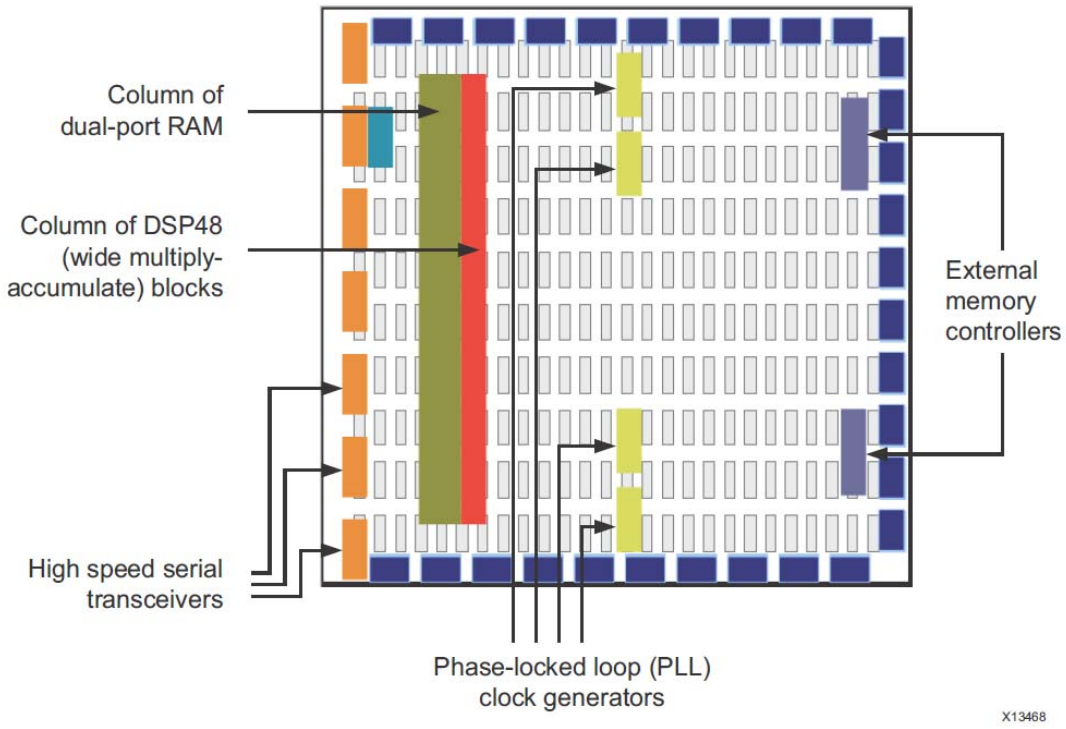

Figure 2-2: **Contemporary FPGA Architecture**

Figure 2-2 shows the combination of these elements on a contemporary FPGA architecture. This provides the FPGA with the flexibility to implement any software algorithm running on a processor. Note that all of these elements across the entire FPGA device can be used concurrently, creating a unique compute platform for OpenCL applications.

LUT

The LUT is the basic building block of an FPGA and is capable of implementing any logic function of N Boolean variables. Essentially, this element is a truth table in which different combinations of the inputs implement different functions to yield output values. The limit on the size of the truth table is N, where N represents the number of inputs to the LUT. For the general N-input LUT, the number of memory locations accessed by the table is 2^N. This allows the table to implement 2^{N^N} functions. Note that a typical value for N in Xilinx FPGA devices is 6.

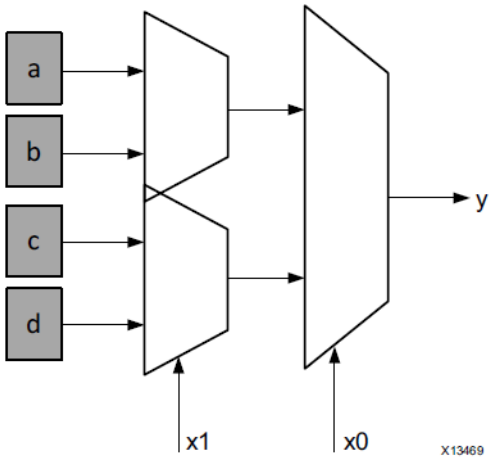

Figure 2-3: **Functional Representation of a LUT as a Collection of Memory Cells**

The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers. Figure 2-3 shows this functional representation of the LUT. The inputs to the LUT act as selector bits on the multiplexer to select the result at a given point in time. It is important to keep this representation in mind, because a LUT can be used as both a function compute engine and a data storage element.

Flip Flop

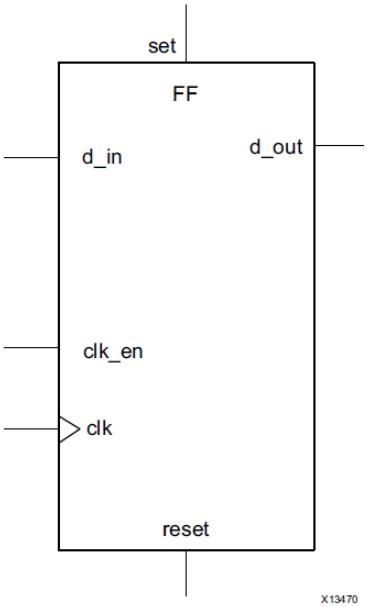

Figure 2-4: **Structure of a Flip-Flop**

The flip-flop is the basic storage unit within the FPGA fabric. This element is always paired with a LUT to assist in logic pipelining and data storage. As shown in Figure 2-4, page 11, the basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold a specific value for more than one clock pulse. New data inputs are only latched and passed to the data output port when both clock and clock enable are equal to one.

DSP48 Block

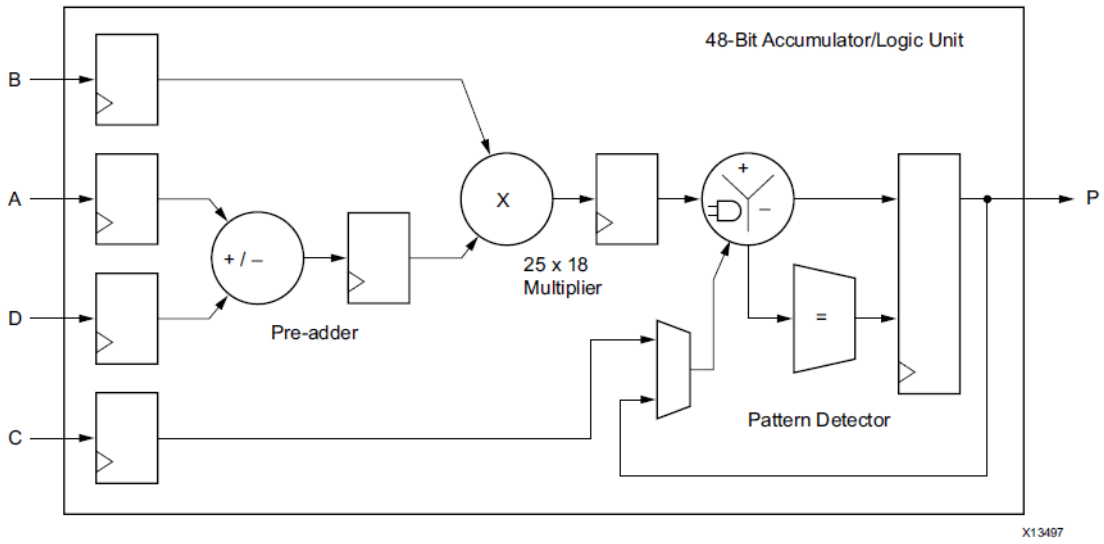

Figure 2-5: **Structure of a DSP48 Block**

The most complex computational block available in a Xilinx FPGA is the DSP48 block, which is shown in Figure 2-5. The DSP48 block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA and is composed of a chain of three different blocks. The computational chain in the DSP48 contains an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine. This chain allows a single DSP48 unit to implement functions of the form:

```
P=B×(A+D)+C
```

Or

```
P +=B×(A+D)
```

The DSP48 block can be utilized by SDAccel to perform a lot of the computational load within OpenCL kernels. The synthesis flow inside the SDAccel tool targets this block automatically.

BRAM and Other Memories

The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs, and shift registers.

The BRAM is a dual-port RAM module instantiated into the FPGA fabric to provide on-chip storage for a relatively large set of data. The two types of BRAM memories available in a device can hold either 18k or 36k bits, and the available amount of these memories is device specific. The dual-port nature of these memories allows for parallel, same-clock-cycle access to different locations.

In OpenCL code, BRAMs can implement either a RAM or a ROM, covering on-chip global, local, and private memory types. In a RAM configuration, the data can be read and written at any time during the runtime of the circuit. In contrast, in a ROM configuration, data can only be read during the runtime of the circuit. The data of the ROM is written as part of the FPGA configuration and cannot be modified in any way.

As previously discussed, the LUT is a small memory in which the contents of a truth table are written during device configuration. Due to the flexibility of the LUT structure in Xilinx FPGAs, these blocks can be used as 64-bit memories and are commonly referred to as distributed memories. This is the fastest kind of memory available on the FPGA device, because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit.

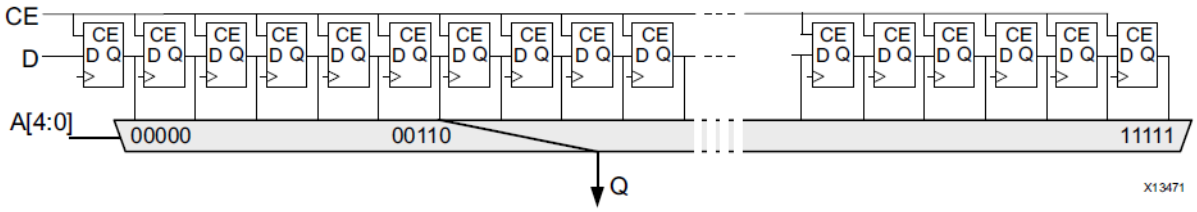

Figure 2-6: **Structure of an Addressable Shift Register**

The shift register is a chain of registers connected to each other, as shown in Figure 2-6. The purpose of this structure is to provide data reuse along a computational path, such as with a filter. For example, a basic filter is composed of a chain of multipliers that multiply a data sample against a set of coefficients. By using a shift register to store the input data, a built-in data transport structure moves the data sample to the next multiplier in the chain on every clock cycle.

FPGA Parallelism Versus Processor Architectures

When compared with processor architectures, the structures that comprise the FPGA fabric enable a high degree of parallelism in application execution. The custom processing architecture generated by SDAccel for an OpenCL kernel presents a different execution paradigm. This must be taken into account when deciding to port an application from a processor to an FPGA. To examine the benefits of the FPGA execution paradigm, this section provides a brief review of processor program execution.

Program Execution on a Processor

A processor, regardless of its type, executes a program as a sequence of instructions that translate into useful computations for the software application. This sequence of instructions is generated by processor compiler tools, such as the GNU Compiler Collection (GCC), which transform an algorithm expressed in C/C++ into assembly language constructs that are native to the processor. The job of a processor compiler is to take a C function of the form:

```
z=a+b;
```

and transform it into assembly code as follows:

```
ADD $R1,$R2,$R3
```

The assembly code above defines the addition operation to compute the value of z in terms of the internal registers of a processor. The input values for the computation are stored in registers R1 and R2, and the result of the computation is stored in register R3. The assembly code above is simplified as it does not express all the instructions needed to compute the value of z. This code only handles the computation after the data has arrived at the processor. Therefore, the compiler must create additional assembly language instructions to load the registers of the processor with data from a central memory and to write back the result to memory. The complete assembly program to compute the value of z is as follows:

```
LD      a,$R1
LD      b,$R2
ADD     R1,$R2,$R3
ST      $R3,c
```

This code shows that even a simple operation, such as the addition of two values, results in multiple assembly instructions. The computational latency of each instruction is not equal across instruction types. For example, depending on the location of a and b, the LD operations take a different number of clock cycles to complete. If the values are in the processor cache, these load operations complete within a few tens of clock cycles. If the values are in the main, double data rate (DDR) memory, these operations take hundreds of clock cycles to complete. If the values are on a hard drive, the load operations take even longer to complete. This is why software engineers with cache hit traces spend so much time restructuring their algorithms to increase the spatial locality of data in memory to increase the cache hit rate and decrease the processor time spent per instruction [2].

Program Execution on an FPGA

The FPGA is an inherently parallel processing fabric capable of implementing any logical and arithmetic function that can run on a processor. The main difference is that the Vivado® High-Level Synthesis (HLS) compiler [3], which is used by SDAccel to transform OpenCL software descriptions into RTL, is not hindered by the restrictions of a cache and a unified memory space.

The computation of z is compiled by HLS into several LUTs required to achieve the size of the output operand. For example, assume that in the original software program the variable a, b, and z are defined with the short data type. This type, which defines a 16-bit data container, gets implemented as 16 LUTs by HLS. As a general rule, 1 LUT is equivalent to 1 bit of computation.

The LUTs used for the computation of z are exclusive to this operation only. Unlike a processor, where all computations share the same ALU, an FPGA implementation instantiates independent sets of LUTs for each computation in the software algorithm.

In addition to assigning unique LUT resources per computation, the FPGA differs from a processor in both memory architecture and the cost of memory accesses. In an FPGA implementation, the HLS compiler arranges memories into multiple storage banks as close as possible to the point of use in the operation. This results in an instantaneous memory bandwidth, which far exceeds the capabilities of a processor. For example, the Xilinx Kintex®-7 410T device has a total of 1,590 18k-bit BRAMs available. In terms of memory bandwidth, the memory layout of this device provides the software engineer with the capacity of 0.5M-bits per second at the register level and 23T-bits per second at the BRAM level.

With regard to computational throughput and memory bandwidth, the HLS compiler exercises the capabilities of the FPGA fabric through the processes of scheduling, pipelining, and dataflow. Although transparent to the user, these processes are integral stages of the software compilation process that extract the best possible circuit-level implementation of the software application.

Scheduling

Scheduling is the process of identifying the data and control dependencies between different operations to determine when each will execute. In traditional FPGA design, this is a manual process also referred to as parallelizing the software algorithm for a hardware implementation.

HLS analyzes dependencies between adjacent operations as well as across time. This allows the compiler to group operations to execute in the same clock cycle and to set up the hardware to allow the overlap of function calls. The overlap of function call executions removes the processor restriction that requires the current function call to fully complete before the next function call to the same set of operations can begin. This process is called pipelining and is covered in detail in the following section and remaining chapters.

Pipelining

Pipelining is a digital design technique that allows the designer to avoid data dependencies and increase the level of parallelism in an algorithm hardware implementation. The data dependence in the original software implementation is preserved for functional equivalence, but the required circuit is divided into a chain of independent stages. All stages in the chain run in parallel on the same clock cycle. The only difference is the source of data for each stage. Each stage in the computation receives its data values from the result computed by the preceding stage during the previous clock cycle. For example, consider the following function:

```
y=(a*x)+b+c
```

The HLS compiler instantiates one multiplier and two adder blocks to implement this function.

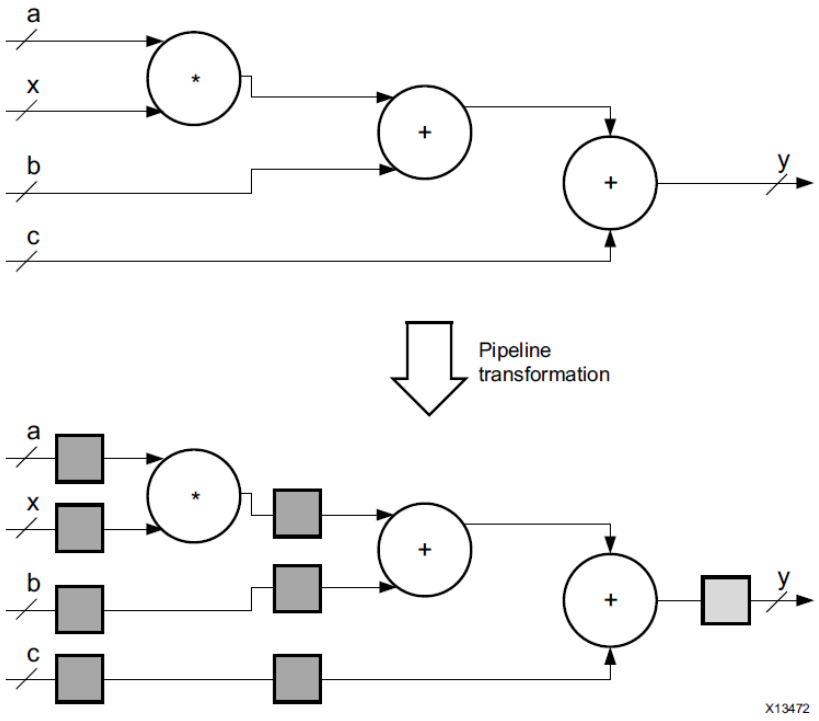

Figure 2-7: **FPGA Implementation of a Compute Function**

Figure 2-7 shows this compute structure and the effects of pipelining. It shows two implementations of the example function. The top implementation is the data path required to compute the result y without pipelining. This implementation behaves similarly to the corresponding software function in that all input values must be known at the start of the computation, and only one result y can be computed at a time. The bottom implementation shows the pipelined version of the same circuit.

The boxes in the data path in the above figure represent registers that are implemented by flip-flop blocks in the FPGA fabric. Each box can be counted as a single clock cycle. Therefore, in the pipelined version, the computation of each result y takes three clock cycles. By adding the register, each block is isolated into separate compute sections in time.

This means that the section with the multiplier and the section with the two adders can run in parallel and reduce the overall computational latency of the function. By running both sections of the data path in parallel, the block is essentially computing the values y and y' in parallel, where y' is the result of the next execution of the equation for y above. The initial computation of y, which is also referred to as the pipeline fill time, takes three clock cycles. However, after this initial computation, a new value of y is available at the output on every clock cycle. The computation pipeline contains overlapped data sets for the current and subsequent y computations.

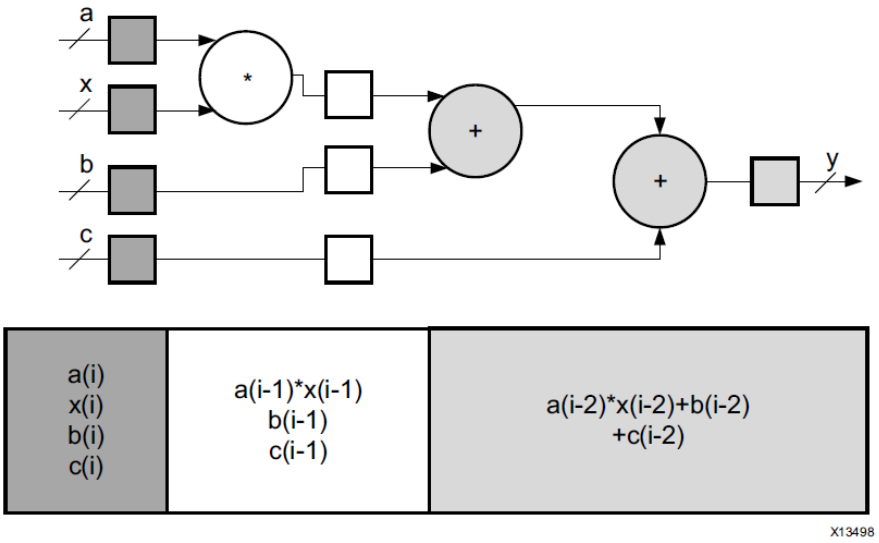

Figure 2-8: **Pipelined Architecture**

Figure 2-8 shows a pipelined architecture in which raw data (dark gray), semi-computed data (white), and final data (light gray) exist simultaneously, and each stage result is captured in its own set of registers. Thus, although the latency for such computation is in multiple cycles, a new result can be produced on every cycle.

Dataflow

Dataflow is another digital design technique, which is similar in concept to pipelining. The goal of dataflow is to express parallelism at a coarse-grain level. In terms of software execution, this transformation applies to parallel execution of functions within a single program.

SDAccel extracts this level of parallelism by evaluating the interactions between different functions of a program based on their inputs and outputs. The simplest case of parallelism is when functions work on different data sets and do not communicate with each other. In this case, SDAccel allocates FPGA logic resources for each function and then runs the blocks independently. The more complex case, which is typical in software programs, is when one function provides results for another function. This case is referred to as the consumer-producer scenario.

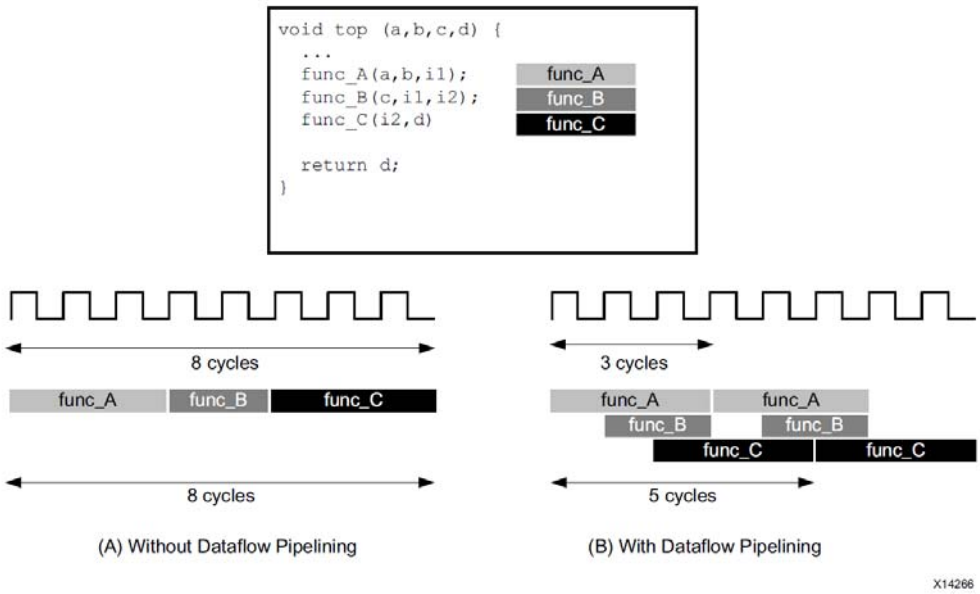

Figure 2-9: **Dataflow Optimization**

Figure 2-9 shows a conceptual view of dataflow pipelining. After synthesis, the default behavior is to execute and complete `func_A`, then `func_B`, and finally `func_C`. However, you can use the Vivado HLS `DATAFLOW` directive to schedule each function to execute as soon as data is available. In this example, the original function has a latency and interval of 8 clock cycles. When you use dataflow optimization, the interval is reduced to only 3 clock cycles. The tasks shown in this example are functions, but you can perform dataflow optimization between functions, between functions and loops, and between loops.

SDAccel supports two use models for the consumer-producer scenario. In the first use model, the producer creates a complete data set before the consumer can start its operation. Parallelism is achieved by instantiating a pair of BRAM memories arranged as memory banks ping and pong. Each function can access only one memory bank, ping or pong, for the duration of a function call. When a new function call begins, the HLS-generated circuit switches the memory connections for both the producer and the consumer. This approach guarantees functional correctness but limits the level of achievable parallelism to across function calls.

In the second use model, the consumer can start working with partial results from the producer, and the achievable level of parallelism is extended to include execution within a function call. The HLS-generated modules for both functions are connected through the use of a first in, first out (FIFO) memory circuit. This memory circuit, which acts as a queue in software programming, provides data-level synchronization between the modules. At any point during a function call, both hardware modules are executing their programming. The only exception is that the consumer module waits for some data to be available from the producer before beginning computation. In HLS terminology, the wait time of the consumer module is referred to as the interval or initiation interval (II).

What is OpenCL?

Overview

The OpenCL standard for parallel programming has been developed by the Khronos Group industry consortium to address the challenges of programming multi-core and heterogeneous compute platforms [Ref 1]. The OpenCL specification defines a single programming model and a set of system-level abstractions that are supported by all hardware platforms conforming to the standard. This means that a software engineer can learn a single programming model and use it directly on devices from multiple vendors.

OpenCL provides a programming language and runtime API to support the development of close-to-the-metal software which is both efficient and portable. Additionally, OpenCL provides low-level hardware abstractions that allow OpenCL implementations to expose many details of underlying hardware. These low-level abstractions are the platform, memory, and executions models described in the OpenCL specification. Understanding how these concepts translate into physical implementations on an FPGA is necessary for application optimization.

This chapter provides a review of the OpenCL platform model and its extensions to FPGA devices. It explains the mapping of the OpenCL platform and memory model into an SDAccel generated implementation. This chapter will also mention how contemporary FPGAs can be leveraged to achieve high levels of performance using the Xilinx SDAccel tool.

OpenCL Platform Model

The OpenCL platform model defines the logical representation of all hardware capable of executing an OpenCL program. OpenCL platforms are defined by the grouping of a host processor and one or more OpenCL compute devices. The host processor, which runs the OS for the system, is also responsible for the general bookkeeping and task launch duties associated with the execution of OpenCL applications. The device is the hardware element in the system on which the compute kernels of an OpenCL application are executed. Each device is further divided into a set of compute units. The number of compute units depends on the target hardware. A compute unit is further subdivided into processing elements. A processing element is the fundamental computation engine in the compute unit, which is responsible for executing the operations of one work item.

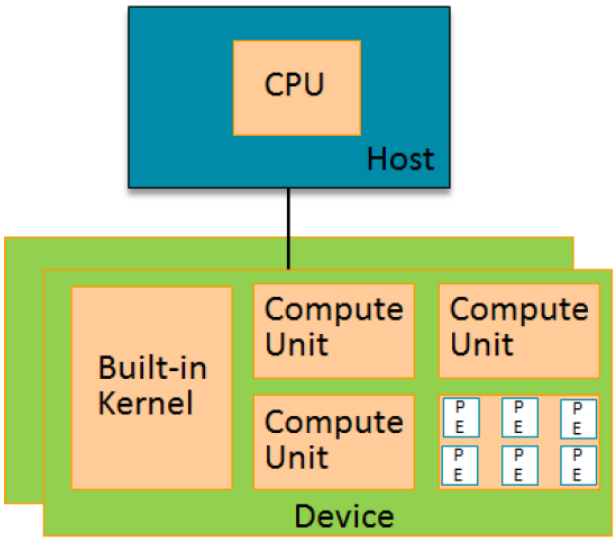

Figure 3-1: **OpenCL Platform Model**

A conceptual view of the OpenCL platform model is shown in Figure 3-1. An OpenCL platform always starts with a host processor. For platforms created with Xilinx® devices, the host processor is an x86 based processor communicating to the devices using PCIe®. The host processor has the following responsibilities:

- Manage the operating system and enable drivers for all devices.
- Execute the application host program.
- Set up all global memory buffers and manage data transfer between the host and the device.
- Monitor the status of all compute units in the system.

In all OpenCL platforms, the host processor tasks are executed using a common set of OpenCL API. The implementation of the OpenCL API functions is provided by the hardware vendor and is referred to as the OpenCL runtime library. The OpenCL runtime library is responsible for translating user commands described by the OpenCL API into hardware specific commands for a given device. For example, when the application programmer allocates a memory buffer using the clCreateBuffer API, it is the responsibility of the runtime library to keep track of where the allocated buffer physically resides in the system, and of the mechanism required for buffer access. It is important for the application programmer to keep in mind that the OpenCL API is portable across vendors, but the runtime library provided by a vendor is not. Therefore, OpenCL applications have to be linked at compile time with the runtime library that is paired with the target execution device.

The other component of a platform is the device. A device in the context of OpenCL is the physical collection of hardware resources onto which the application kernels are executed.

A platform must have at least one device available for the execution of kernels. Also, per the OpenCL platform model, all devices in a platform do not have to be of identical type.

OpenCL Devices and FPGAs

In the context of CPU and GPU hardware, the attributes of an OpenCL device are fixed and the programmer has very little influence on what the device looks like. An advantage of this characteristic of CPU/GPU systems makes it relatively easy to obtain and use off-the-shelf hardware. This advantage is also a major limitation when compared to FPGA based OpenCL devices. CPU and GPU based systems typically have fixed data paths, memory systems, and I/O architectures. It is not possible, for example, to directly attach high-speed I/O to an OpenCL compute kernel. Similarly, efficient data movement is only performed using bulk memory based transfers.

An OpenCL device for an FPGA is not limited by the constraints of a CPU/GPU device. By taking advantage of the fact that the FPGA starts off as a blank computational canvas, the user can decide the level of device customization that is appropriate to support a single application or a class of applications. In determining the level of customization in a device, the programmer can take advantage of the fact that kernel compute units are not placed in isolation within the FPGA fabric.

FPGA devices capable of supporting OpenCL programs can include, but are not limited to, the following components:

- DMA engines
- I/O peripherals such as PCIe and Ethernet
- Memory controllers
- Custom interconnects
- OpenCL compute units
- RTL-based accelerators

The creation of Xilinx FPGA based OpenCL devices requires FPGA design expertise and is beyond the scope of SDAccel itself. Devices for SDAccel are created using the Xilinx Vivado® design suite for FPGA designers. SDAccel provides pre-defined devices as well as allows users to augment the tool with third party created devices. A methodology guide describing how to create a device for SDAccel is available upon request from Xilinx.

The devices available in SDAccel are for Virtex®-7, Kintex®-7, and Kintex-UltraScale® devices from Xilinx. These devices are available in a PCIe form factor. The PCIe form factor assumes that the host processor is an x86 based processor and that the FPGA is used for the implementation of compute units.

OpenCL Memory Model

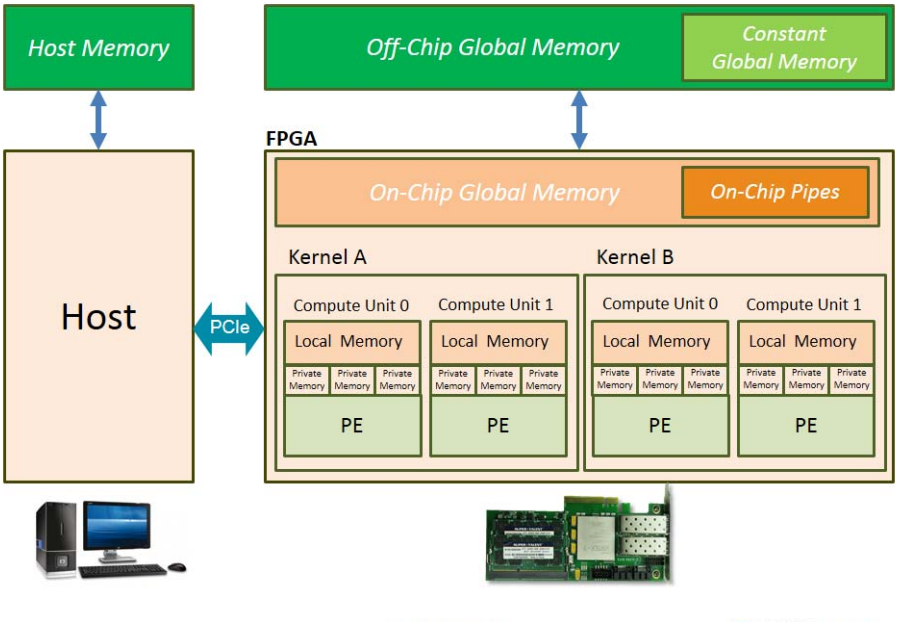

Figure 3-2: **OpenCL Memory Model**

The OpenCL memory model defines the behavior and hierarchy of memory that can be used by OpenCL applications, as shown in Figure 3-2. This hierarchical representation of memory is common across all OpenCL implementations, but it is up to individual vendors to define how the OpenCL memory model maps to specific hardware. This section defines the mapping used by SDAccel.

Host Memory

The host memory is defined as the region of system memory that is directly accessible from the host processor. Any data needed by compute kernels must be transferred to and from OpenCL device global memory using the OpenCL API.

Global Memory

The global memory is defined as the region of device memory that is accessible to both the OpenCL host and device. Global memory permits read/write access to the host processor as well to all compute units in the device. As shown above, Xilinx OpenCL platforms may further divide the global memory space between on-chip and off-chip memories. The host is responsible for the allocation and de-allocation of buffers in this memory space. There is a handshake between host and device over control of the data stored in this memory. The host processor transfers data from the host memory space into the global memory space. Then, once a kernel is launched to process the data, the host loses access rights to the buffer in global memory. The device takes over and is capable of reading and writing from

the global memory until the kernel execution is complete. Upon completion of the operations associated with a kernel, the device turns control of the global memory buffer back to the host processor. Once it has regained control of a buffer, the host processor can read and write data to the buffer, transfer data back to the host memory, and de-allocate the buffer.

Constant Global Memory

Constant global memory is defined as the region of system memory that is accessible with read and write access for the OpenCL host and with read only access for the OpenCL device. As the name implies, the typical use for this memory is to transfer constant data needed by kernel computation from the host to the device.

Local Memory

Local memory is a region of memory that is local to a single compute unit. The host processor has no visibility and no control on the operations that occur in this memory space. This memory space allows read and write operations by all the processing elements with a compute units. This level of memory is typically used to store data that must be shared by multiple work-items. Operations on local memory are un-ordered between work-items but synchronization and consistency can be achieved using barrier and fence operations. In SDAccel, the structure of local memory can be customized to meet the requirements of an algorithm or application.

Private Memory

Private memory is the region of memory that is private to an individual work-item executing within an OpenCL processing element. As with local memory, the host processor has no visiblilty into this memory region. This memory space can be read from and written to by all work-items, but variables defined in one work-item's private memory are not visible to another work-item. In SDAccel, the structure of private memory can be customized to meet the requirements of an algorithm or application.

For devices using an FPGA device, the physical mapping of the OpenCL memory model is the following:

- Host memory is any memory connected to the host processor only.
- Global and constant memories are any memory that is connected to the FPGA device. These are usually memory chips (e.g. SDRAM) that are physically connected to the FPGA device, but might also include distributed memories (e.g. BlockRAM) within the FPGA fabric. The host processor has access to these memory banks through infrastructure provided by the FPGA platform.
- Local memory is memory inside of the FPGA device. This memory is typically implemented using registers or BlockRAMs in the FPGA fabric.

- Private memory is memory inside of the FPGA device. This memory is typically implemented using registers or BlockRAMs in the FPGA fabric.

OpenCL Execution Model

The OpenCL execution model defines how kernels execute. The most important concept to understand is NDRange execution. When OpenCL kernels are submitted for execution on an OpenCL device, they execute within the computer science concept of an index space. An example of an index space which is easy to understand is a for loop in C/C++. In the for loop defined by the statement "for(int i=0; i<10; i++)", any statements within this loop will execute ten times, with i=0,1,2...,9. In this case the index space of the loop is [0,1,2,...,9]. In OpenCL, index spaces are called NDRanges, and can have 1, 2, or 3-dimensions.

OpenCL kernel functions are executed exactly one time for each point in the NDRange index space. This unit of work for each point in the NDRange is called a work-item. Unlike for loops in C, where loop iterations are executed sequentially and in-order, an OpenCL runtime and device is free to execute work-items in parallel and in any order. It is this characteristic of OpenCL execution model that allows the programmer to take advantage of parallel compute resources.

Work-items are not scheduled for execution individually onto OpenCL devices. Instead, work-items are organized into work-groups, which are the unit of work scheduled onto compute units. Because of this, work-groups also define the set of work-items that may share data using local memory.

When a user submits a kernel for execution, they also provide the NDRange. This is called the *global size* in the OpenCL API. The user may also set the work-group size at runtime. This is called the *local size* in the OpenCL API. The user may also let the runtime select the local size based on the properties of the kernel and selected device. Once the work-group size (local size) has been determined, the NDRange (global size) is divided automatically into work-groups, and the work-groups are scheduled for execution on the device.

Optionally, a kernel programmer can set the work-group size at kernel compile time.

IMPORTANT: *In the case of an FPGA implementation, the specification of the work-group size is highly recommended as it can be used for performance optimization during the generation of the custom logic for a kernel.*

The work-group size of a kernel can be specified using the following OpenCL C attribute:

```
__kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
```

In this example, the only work-group size supported by the kernel is the tuple (256, 1, 1). SDAccel will therefore generate a specialized compute unit supporting only this size work-group.

OpenCL supports one-dimensional, two-dimensional, and three-dimensional NDRanges and work-groups.

One-Dimensional NDRange

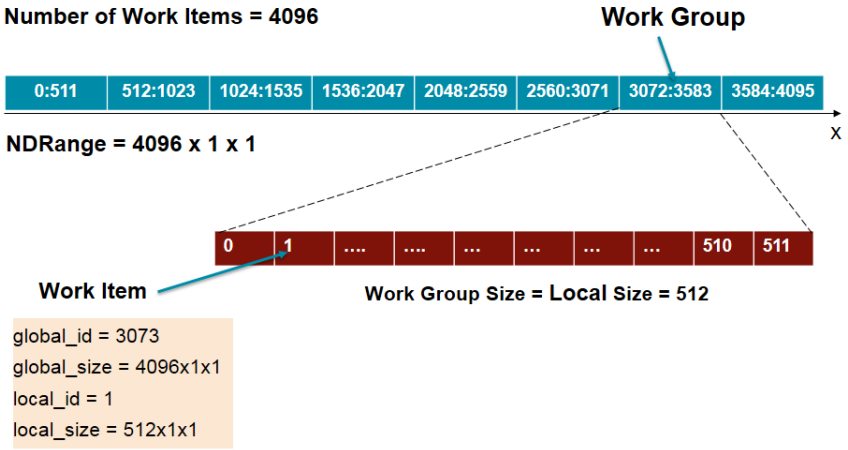

Figure 3-3: **One-Dimensional Work Size**

Figure 3-3 illustrates an example of one-dimensional NDRange with global size = (4096, 1, 1) and local size = (512, 1, 1). This allows the computation to be broken down into eight work-groups, each with 512 work-items.

Now consider a simple vector adder kernel written with a work size of (1, 1, 1):

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd(__global const int* a, __global const int* b, __global int* c) {
  int i;
  for (i=0; i < 4096; i++) {
    c[i] = a[i] + b[i];
  }
}
```

In this example, the kernel is written in sequential C style. The length of the data is 4096, and the function iterates over the data using an explicit loop. In OpenCL C, however, it is better to write the kernel as shown below:

```
__kernel __attribute__ ((reqd_work_group_size(512, 1, 1)))
void vadd(__global const int* a, __global const int* b, __global int* c) {
  int i = get_global_id(0);
  c[i] = a[i] + b[i];
}
```

This produces the NDRange and work group sizes shown above. Because this example allows the OpenCL compiler and runtime to control the iteration over the 4096 data items, it allows a simpler coding style and enables the compiler to make better optimization decisions to parallelize the operations. The call to get_global_id(0) provides the current

location in the NDRange and is analogous to the index of a for loop. This is a simple example but is extensible to other larger work sizes. When using SDAccel, it is sometimes useful to think of the above code as transformed into the following form by the SDAccel compiler:

```
__kernel void vadd(global const int* a, global const int* b, global int* c) {
  localid_t id;
  for (id[0] = 0; id[0] < 512; id[0]++) {
    for (id[1] = 0; id[1] < 1; id[1]++) {
      for (id[2] = 0; id[2] < 1; id[2]++) {
        c[id[0]] = a[id[0]] + b[id[0]];
      }
    }
  }
}
```

Note that the code written within the kernel is surrounded by three nested loops to traverse the entire work-group size. These three for loops are conceptually introduced by SDAccel into the kernel to handle the three-dimensional space of the NDRange. The SDAccel compiler exploits NDRange parallelism by pipelining and vectorizing these conceptual loops.

The conceptual loop nest introduced by SDAccel can have either variable or fixed loop bounds. By setting the reqd_work_group_size attribute, the programmer is setting the loop boundaries on this loop nest. Fixed boundaries allow the kernel compiler to optimize the size of local memory in the compute unit and to provide latency estimates. If the work size is not specified, SDAccel might assume a large size for private memory, which can hinder the number of compute units that can be instantiated in the FPGA fabric.

Two-Dimensional NDRange

These concepts can be extended to a two-dimensional NDRanges. This type of NDRange works well with two-dimensional data such as matrices. Consider the following matrix adder kernel:

```
__kernel __attribute__ ((reqd_work_group_size(2, 2, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
  int index = get_global_id(1)*get_global_size(0) + get_global_id(0);

  output[index] = a[index] + b[index];
}
```

This kernel defines a local work size of 2x2, specified as a required size of (2, 2, 1). The calls to get_global_id() provide the index in the global work size, while get_global_size() provides the total range value (e.g., 64 for a 64x64 matrix). Alternatively, the kernel could also index the local work indices and sizes using get_local_id() and get_local_size().

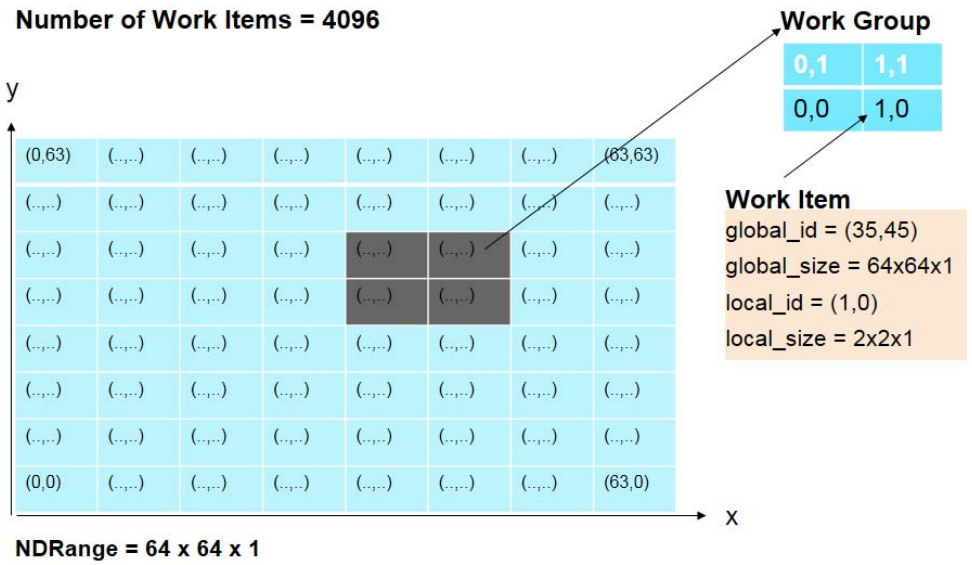

Figure 3-4: **Two-Dimensional NDRange**

Figure 3-4 illustrates how this two-dimensional space is defined and indexed. While the ND range is 64x64x1, the local work size is 2x2x1. Similar to the one-dimensional work size, this enables simpler coding as well as concurrent implementation across the vast resources of the FPGA.

Three-Dimensional NDRange

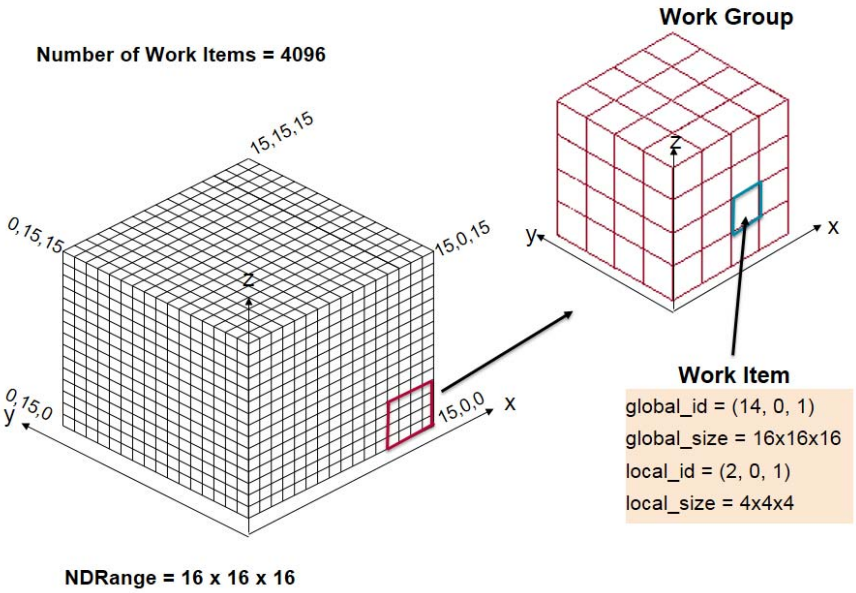

Figure 3-5: **Three-Dimensional Work Size**

The concept of work size can be extended to a three-dimensional space. Figure 3-5 illustrates this work size as a three-dimensional cube of size 16x16x16. While the total number of work items is again 4096, the work space is now defined across three different

dimensions. This works well for applications that can be defined across three dimensions such as 3D computer graphics and data mining algorithms. Similar to the one- and two-dimensional cases, three dimensional work-items can be implemented to operate in a concurrent fashion on the FPGA device.

OpenCL Region

An SDAccel device contains a customization area called the OpenCL region (OCL Region). Although not defined in the OpenCL standard, the OCL Region is an important concept in SDAccel. The compute units generated from user kernel functions are placed in this region. These compute units are highly specialized to execute a single kernel function and internally contain parallel execution resources to exploit work-group level parallelism. By placing multiple compute units of the same type in the OCL Region, developers can easily scale the performance of single kernels across larger NDRange sizes. By placing multiple compute units of different types in the OCL Region, developers can leverage task parallelism between disparate kernels. In this way, the massive amounts of parallelism available in the FPGA device can be customized and harnessed by the SDAccel developer. This is different from CPU and GPU implementations of OpenCL which contain a fixed set of general purpose resources.

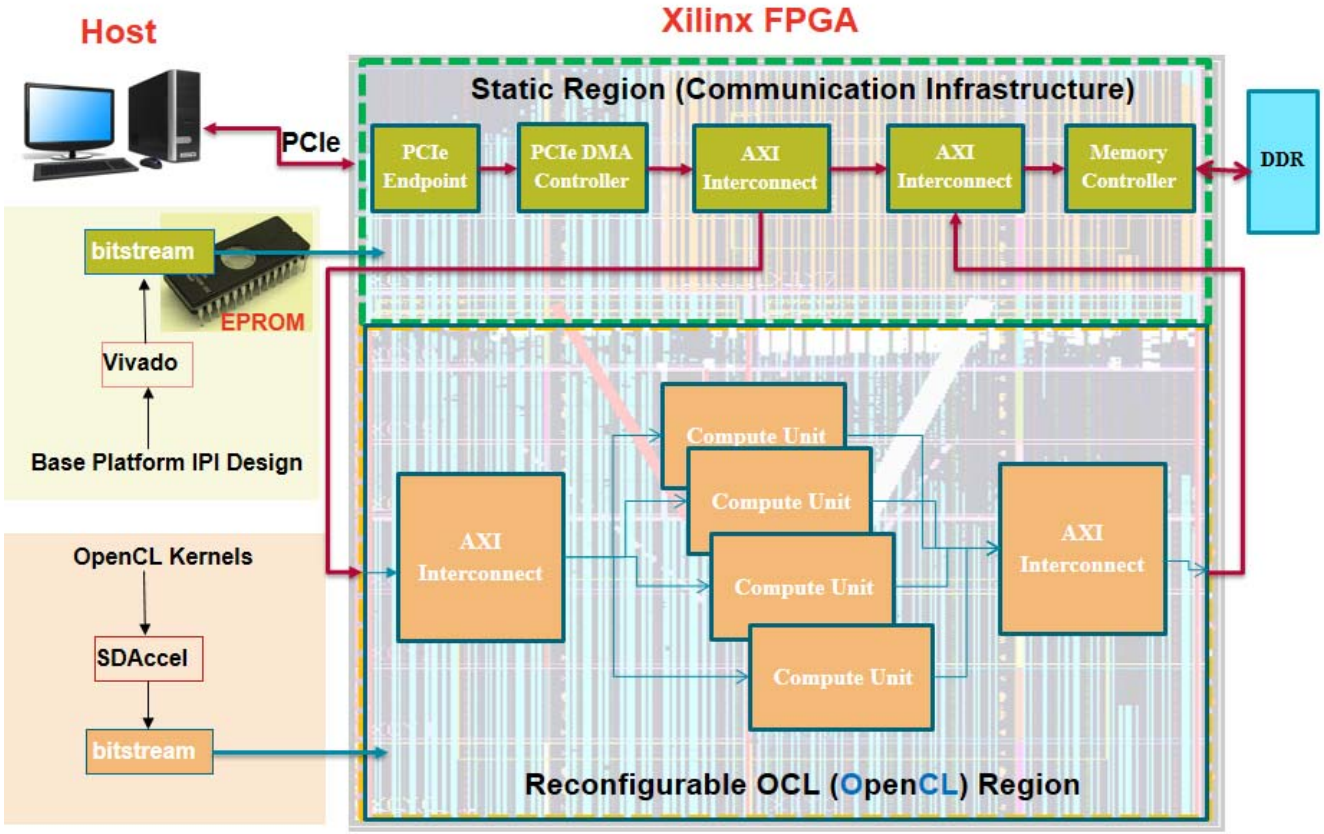

Figure 3-6: **Block Diagram of Example Xilinx SDAccel Platform**

Figure 3-6 shows how an OCL region fits into an example Xilinx SDAccel platform. The OCL region contains the customized compute units which implement the user-defined accelerator kernels. SDAccel automatically adds the necessary interconnects for these compute units to communicate with the rest of the platform. Also contained on the FPGA device is a static region containing all the necessary circuitry for communication between host, compute units, and off-chip global memory. This static region is a pre-defined base platform which can be flashed onto an EPROM on the board. The FPGA would then be configured with this base platform upon power-up and is always there and accessible for the user. As shown in the above figure, communication to the host is performed over PCIe, a fast, standard interface used to connect and link with boards.

OpenCL C Example

To understand the benefits of FPGAs for OpenCL, consider the following OpenCL C kernel code:

```
#define LENGTH 64

__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vmult(__global const int* a, __global const int* b, __global int* c) {
  local int bufa[LENGTH];
  local int bufb[LENGTH];
  local int bufc[LENGTH];

  event_t evt[3];
  evt[0] = async_work_group_copy(bufa, a, LENGTH, 0);
  evt[1] = async_work_group_copy(bufb, b, LENGTH, 0);
  wait_group_events(2, evt);

  for (int i=0; i < LENGTH; i++) {
    bufc[i] = bufa[i] * bufb[i];
  }

  barrier(CLK_LOCAL_MEM_FENCE);
  event_t e = async_work_group_copy(c, bufc, LENGTH, 0);
  wait_group_events(1, &e);
}
```

There are a number of FPGA resources leveraged by SDAccel to perform the functionality in this kernel. This includes the following:

- **Loops** - These are common elements in kernel functionality and are implemented using a variety of FPGA resources including LUTs and flip-flops. These loops can be unrolled and pipelined based on resource and performance requirements (refer to Loop Unrolling and Loop Pipelining in Appendix A for more information). How loops are implemented can have a major impact on overall kernel performance.
- **Arrays** - The arrays bufa, bufb, and bufc are typically implemented in BRAMs, utilizing the distributed local storage on the FPGA.
- **Operators** - The multiplication of each element in the vectors can be performed by either LUTs or DSP48 Blocks. The same is true for other common operators such as addition, subtraction, comparators, etc.
    - If desired to improve performance, the loop could be partially or fully unrolled. A high number of multiplications would then be performed concurrently.
- **Communication** - The high-speed communication between this kernel and the rest of the device would be implemented using LUTs and flip-flops. This includes the interconnect and memory controller to handle the calls to `async_work_group_copy` using high-bandwidth burst data transfers.

Data Path Optimizations

Overview

Now that you have an understanding of the acceleration capabilities of FPGA devices and the OpenCL programming environment that enables them, it would be helpful to step through a few design examples. The hope is that these designs are complex enough to provide example characteristics yet simple enough to be easily understood. These designs take advantage of various attributes and architectural techniques, and step through their usage and impact. The profiling capabilities integrated into SDAccel are used to visually demonstrate these performance impacts.

This chapter focuses on a matrix adder kernel which computes the addition of two 64x64 matrices. While these are relatively small matrices, the techniques described herein are applicable to much larger matrices.

Table 4-1: **Comparison of Kernel Execution Times for 64x64 Matrix Adder**

Design	Kernel Execution Time (msec)
Un-Optimized	4.16
Workgroup Pipelined	1.64
Multiple Compute Units	1.26

Three different architectures for a 64x64 matrix adder were created and compared. Table 4-1 lists the kernel execution times as reported by the profiling summary. As you can see, the different architectures and settings had a significant impact on the overall performance of the system as the run-time to compute one matrix was considerably reduced. Below are details on how these kernels were created including the attributes and architectures for each.

Unoptimized Design

```
#define RANK 64

__kernel __attribute__ ((reqd_work_group_size(RANK, RANK, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
int index = get_local_id(1)*get_local_size(0) + get_local_id(0);

  output[index] = a[index] + b[index];
}
```

Above is the kernel source code for the un-optimized adder. No attributes were specified for this design other than the work size equal to the size of the matrices (i.e., 64x64). That is, iterating over an entire workgroup will fully add the input matrices a and b and output the result to output. All three are global integer pointers, which means each value in the matrices is four bytes and is stored in off-chip DDR global memory.

This local work size of (64, 64, 1) is the same as the global work size. It should be noted that this setting creates a total work size of 4096.

IMPORTANT: *This is the largest work size that SDAccel supports with the standard OpenCL attribute* `reqd_work_group_size`. *SDAccel supports work size larger than 4096 with the Xilinx attribute* `xcl_max_work_group_size`.

Any matrix larger than 64x64 would need to only use one dimension to define the work size. That is, a 128x128 matrix could be operated on by a kernel with a work size of (128, 1, 1), where each invocation operates on an entire row or column of data.

Data Transfer: Kernels and Global Memory

Transfer Type	Number Of Transfers	Transfer / Execution Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)
READ	1026	6.268	0.098	1.024	738.421
WRITE	1024	6.255	0.098	1.024	1075.010

Figure 4-1: **Kernel Data Transfer Summary of Un-optimized Matrix Adder Kernel**

Figure 4-1 shows the kernel data transfer summary table for the un-optimized matrix adder kernel. This table is shown in the profile summary in the SDAccel user interface (UI). There are 8K read transfers, each with a size of 64 bytes. This is a single 512-bit memory word. Since read transfers do not use byte enables, we can deduce from our code that this is one 32-bit word per transfer. The kernel reads two matrices, each containing 4K integer values. Writing the output matrix involves 4K transfers, each with a size of 4 bytes. This is one integer value per transfer.

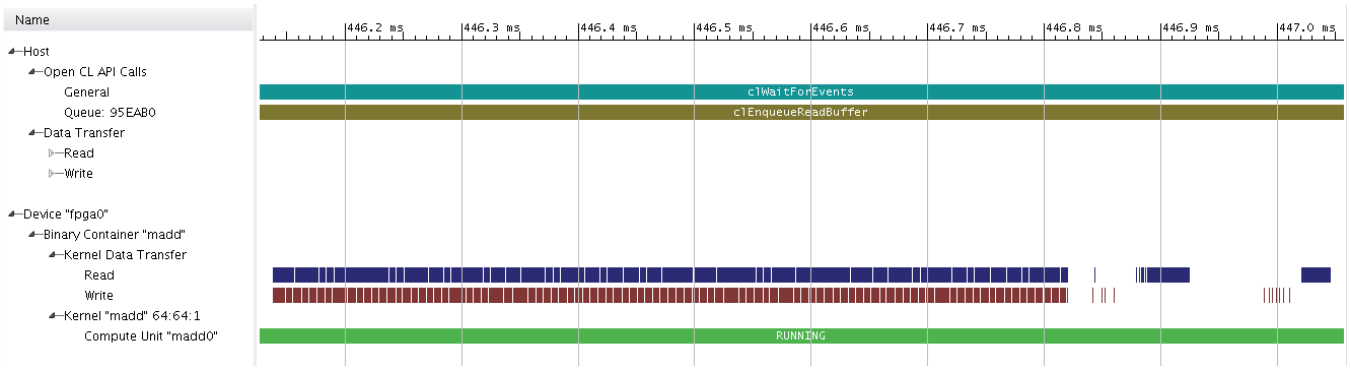

Figure 4-2: **Timeline Trace of Un-Optimized Matrix Adder Kernel**

Figure 4-2 shows a timeline trace of the un-optimized matrix adder. As expected from the profile summary in Figure 4-1, page 33, the timeline shows a high number of data transfers initiated by the kernel. The kernel is clearly spending a lot of time on data transfers.

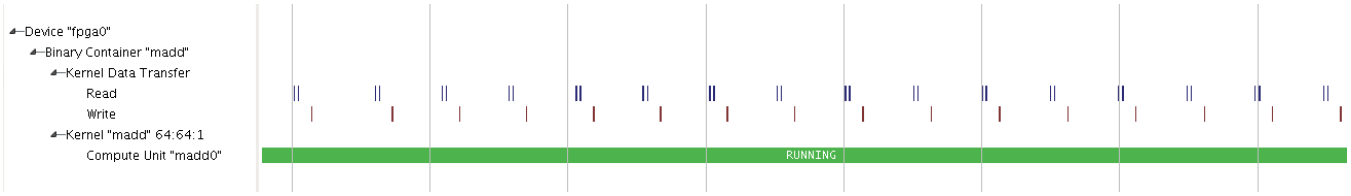

Figure 4-3: **Details of Timeline Trace for Un-optimized Matrix Adder Kernel**

Figure 4-3 displays the details of this timeline trace. The data transfers are performed as two reads, followed by time to add the two values, then a single write. The read transfers are then initiated for the next work item, and the process repeats again until all work items have been processed.

However, since the sizes of these data transfers is so small, this leads to a very inefficient kernel execution. This fact is also reflected in the high kernel execution time, as shown in Table 4-1, page 32, and low bandwidth utilization, as shown in Figure 4-1, page 33. This can be improved in a number of ways, including burst data transfers and improvements in the kernel data path. For this chapter, we are going to focus on the data path improvements.

Workgroup Pipelined

We can take advantage of the fact that the matrix adder kernel uses multiple workgroups. Across multiple invocations in the workgroup, we can expose a number of optimizations to the Vivado HLS compiler. One important one is workgroup pipelining. This involves pipelining the activity across all of the work items.

As described in NDRange Kernels, it is best to think of workgroups as nested for loops to cover the entire range of (x,y,z) values. Pipelining the activity across the entire range keeps as much of the kernel implementation as busy as possible throughout the execution. For more information, see Work Item Pipelining, page 67.

```
#define RANK 64

__kernel __attribute__ ((reqd_work_group_size(RANK, RANK, 1)))
void madd(__global int* a, __global int* b, __global int* output) {
  __local unsigned int bufa[RANK*RANK];
  __local unsigned int bufb[RANK*RANK];

  // Global
  int width = get_global_size(0);
  int xg = get_global_id(0);
  int yg = get_global_id(1);
  // Local
  int xl = get_local_id(0);
  int yl = get_local_id(1);
  // Indeces
  int index1 = yl*RANK + xl;
  int index2 = yg*width + xg;

  __attribute__((xcl_pipeline_workitems)) {
    bufa[index1] = a[index2];
    bufb[index1] = b[index2];
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  __attribute__((xcl_pipeline_workitems)) {
    output[index2] = bufa[index1] + bufb[index1];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
```

The above code shows an initial improvement to the matrix adder kernel. The attribute `xcl_pipeline_workitems` was added twice to wrap the data transfers and pipeline the read and write requests. As shown in Figure 4-3, page 34, the read transfers are requested in pairs in order to obtain one value of each input matrix. The next read pair does not happen until the other pair has completed. This can be improved by pipelining these read requests. The same can be done to improve the integer addition and write transfers.

To accommodate the local storage of these pipelined read requests, we added local memories `bufa` and `bufb`. These memories as well as the global pointers are accessed accordingly using global and local indeces. We also added calls to barrier(`CLK_LOCAL_MEM_FENCE`) to ensure all values have been read before we begin the next stage. For a matrix adder kernel where each value can be computed independently, this is not absolutely necessary. However, it provides good design practice as a method of isolating read and write data transfers.

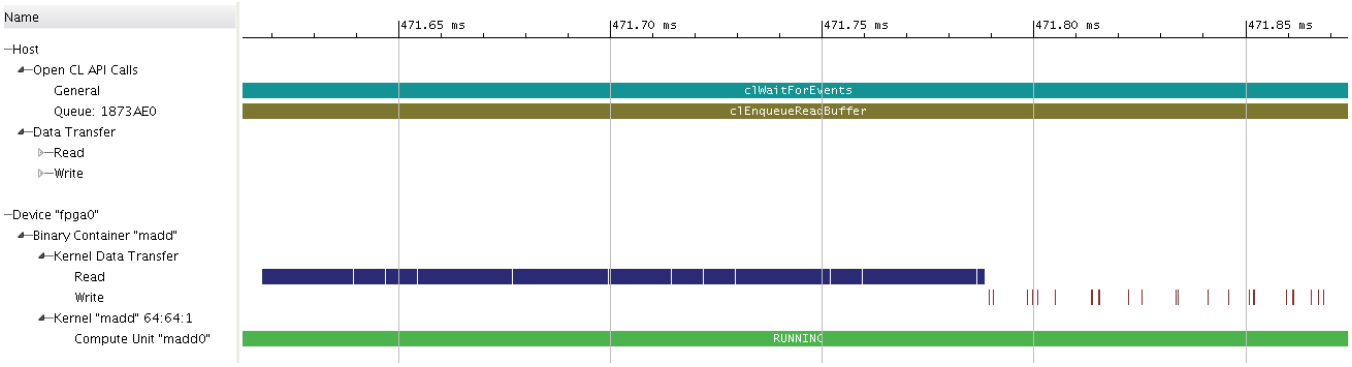

Figure 4-4: **Timeline Trace of Pipelined Matrix Adder Kernel**

Figure 4-4 shows the timeline trace of the pipelined matrix adder kernel. Rather than the kernel performing two read transfers then a single write, it now first performs all read transfers. Once all are completed, then the kernel performs the addition and subsequent write transfers to DDR memory.

While this is a more efficient kernel design with a lower execution time, we can further take advantage of the workgroups. Since the matrix addition involved separate operations for each work item, we can do a lot more in parallel and take advantage of the resources on the FPGA. To do this, we can include multiple compute units on the device.

Multiple Compute Units

If we take a closer look at the matrix addition computation, we see that it can be broken down into smaller chunks which can be performed independently. This allows us to perform them concurrently. Using OpenCL, the mechanism to do this is called workgroups. We can enqueue the madd kernel using a local work size which is different from the global work size.

The global work size is still 64x64, which is the size of the matrices. However, we now specify a required local work size of 16x16 for the kernel. For the matrix adder, we essentially use the same source code above in Workgroup Pipelined, page 34; however, we modify the value of `RANK` to be 16 (i.e., `#define RANK 16`). That informs the SDAccel compiler to break down the computation into groups of 16x16 for each call to a compute unit. If multiple compute units are on the device, then this computation can be done simultaneously.

We take advantage of this parallelization by including eight compute units on our device. This exploits the multitude of resources available on an FPGA. An SDAccel Tcl script in order to request eight compute units would include the following:

```
create_opencl_binary bin_madd
set_property region "OCL_REGION_0" [get_opencl_binary bin_madd]
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd0
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd1
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd2
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd3
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd4
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd5
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd6
create_compute_unit -opencl_binary [get_opencl_binary bin_madd] -kernel [get_kernels
madd] -name madd7
```

With a global size of 64x64 and a local size of 16x16, there will be (64/16) x (64/16) = 16 invocations to these compute units. We chose eight units since that would ideally allow each compute unit to be used twice. Note that SDAccel supports up to ten compute units on one device.

In the host code, we still only enqueue the kernel once. Only a slight modification is required to specify different global and local sizes. The host code would look like the following:

```
global[0] = 64;
global[1] = 64;
local[0] = 16;
local[1] = 16;
err = clEnqueueNDRangeKernel(commands, kernel, 2, NULL,
                             (size_t*)&global, (size_t*)&local, 0, NULL, NULL);
```

Compute Unit Utilization

Search:

Compute Unit	Kernel	Global Work Size	Local Work Size	Number Of Calls	Total Time (ms)	Minimum Time (ms)	Average Time (ms)	Maximum Time (ms)
madd0	madd	64:64:1	16:16:1	2	0.847	0.217	0.424	0.630
madd1	madd	64:64:1	16:16:1	2	0.912	0.276	0.456	0.636
madd2	madd	64:64:1	16:16:1	2	1.011	0.426	0.505	0.585
madd3	madd	64:64:1	16:16:1	2	1.062	0.468	0.531	0.593
madd4	madd	64:64:1	16:16:1	2	1.064	0.493	0.532	0.570
madd5	madd	64:64:1	16:16:1	2	1.150	0.546	0.575	0.604
madd6	madd	64:64:1	16:16:1	2	1.156	0.572	0.578	0.585
madd7	madd	64:64:1	16:16:1	2	1.147	0.536	0.573	0.611

Figure 4-5: **Compute Unit Utilization Table for Matrix Adder Kernel with Eight Compute Units**

Figure 4-5 shows the compute unit utilization table as displayed in the SDAccel profile summary. As expected, each compute unit is called twice. With eight compute units, this involves 16 total calls. While the global work size is confirmed as 64:64:1 for each compute unit, the local work size is now 16:16:1 as specified. Since we have broken down the computation into smaller local work sizes, the call times for each compute unit is much smaller than the kernel execution time of the single workgroup pipelined kernel. This is entirely because the work size is smaller.

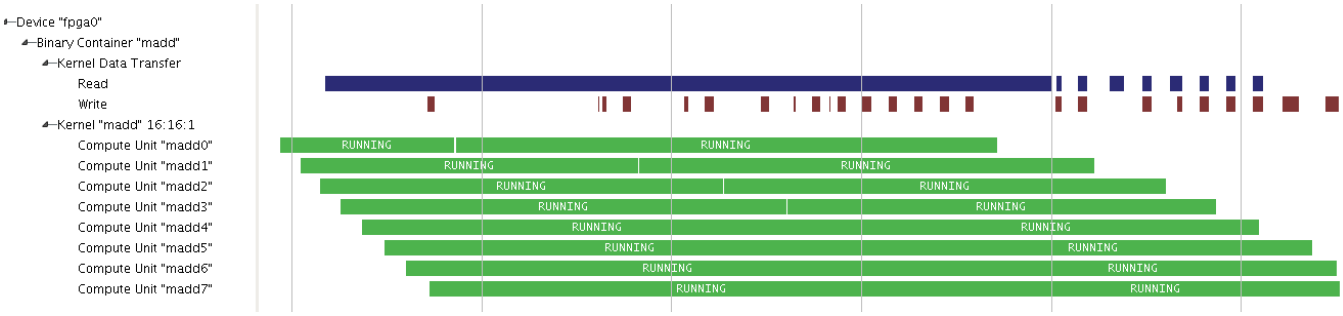

Figure 4-6: **Timeline Trace for Matrix Adder Kernel with Eight Compute IUnits**

The timeline trace confirms this operation. Figure 4-6 shows the timeline trace of the matrix adder with eight compute units. The calls to the compute units are run simultaneously thus providing an improved execution time of the kernel, as shown in Table 4-1, page 32. Their calls are also staggered to accommodate for the overhead of starting and stopping each unit.

Memory Access Optimizations

Overview

Improving memory accesses is a second category of optimizations that can be made to a kernel. Efficient memory accesses are critical to the performance of OpenCL kernels running on an FPGA since there is an inherent latency overhead to read and write data from off-chip DDR DRAM. A well-designed kernel will minimize this latency impact. A few suggested techniques include the following:

- On-Chip Memories - These memories utilize the BlockRAMs on the FPGAs and are physically located near the kernel computation. In the OpenCL memory architecture, these can either be on-chip global, local, or private memories. They allow one-cycle reads and writes, thus drastically improving memory access performance. Copying the data from DDR to these memories can be done very quickly using burst transactions (see below).
- On-Chip Pipes - These memories are implemented in BlockRAMs similarly to on-chip global memories. They provide efficient communication channels in between kernels.
- Multiple Memory Ports per Kernel - This setting informs the SDAccel tool to maximize the number of memory ports per kernel. This provides a much more efficient method to read and write data instead of sharing a memory port across multiple interfaces.
- Adjustable Bit Width for Memory Ports - This setting takes advantage of the wider memory path on the device. For example, the Alpha Data board [Ref 7] uses a 512-bit memory path, which can enable access to sixteen 32-bit integers in a single memory word.
- Burst Transfers from Off-Chip Global Memory - This architectural technique utilizes large bursts of data between the kernel and the off-chip global memory. The benefit is much more efficient memory accesses as the overhead of the access is shared across a large amount of data being transferred.

All of the memory access optimizations listed above are described in more detail in Appendix B, Improving Memory Efficiency.

This chapter steps though an implementation of the Smith-Waterman algorithm [Ref 5]. This is a database search algorithm that performs local sequence alignment. It determines similar regions between two strings and is often used to compare protein or nucleic sequences. The algorithm compares segments of all possible lengths and optimizes a similarity measurement. Two version of the design, one un-optimized and the other optimized, were created and their profile results were compared and contrasted.

Un-Optimized Design

An un-optimized Smith-Waterman kernel, shown in the code snippet below, was created and profiled. It was designed for functionality only and no attempt was made to optimize any parts of the design. This can be considered an "off the shelf" design created by an FPGA novice. It also provides a good starting point to understand the profiling capabilities in SDAccel.

```
#include "constants.h"

__attribute__ ((reqd_work_group_size(1, 1, 1)))
kernel void smithwaterman (global int *matrix, global int *maxIndex, global const
char *s1, global const char *s2) {
    short north = 0;
    short west = 0;
    short northwest = 0;
    int maxValue = 0;

    for (short index = N; index < N * N; index++) {
        short dir = CENTER;
        short val = 0;
        short j = index % N;
        if (j == 0) { // Skip the first column
            west = 0;
            northwest = 0;
            continue;
        }
        short i = index / N;
        short2 temp = matrix[index - N];
        north = temp.x;
        const short match = (s1[j] == s2[i]) ? MATCH : MISMATCH;
        short val1 = northwest + match;

        if (val1 > val) {
            val = val1;
            dir = NORTHWEST;
        }
        val1 = north + GAP;
        if (val1 > val) {
            val = val1;
            dir = NORTH;
        }
```

```
            val1 = west + GAP;
            if (val1 > val) {
                val = val1;
                dir = WEST;
            }
            temp.x = val;
            temp.y = dir;
            matrix[index] = as_int(temp);
            west = val;
            northwest = north;
            if (val > maxValue) {
                *maxIndex = index;
                maxValue = val;
            }
        }
    }
```

Note that this design compares two strings located at the `s1` and `s2` global constant pointers. This kernel assumes that the two string sequences have already been written to that location in shared global memory by the host processor. As we compare these two sequences and replace values according to the algorithm, the results are stored in global memory specified by the `matrix` pointer. The pointer `maxIndex` is used to store the index of the maximum value in the sequence.

Data Transfer: Kernels and Global Memory

Transfer Type	Number Of Transfers	Transfer / Execution Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)
READ	21168	174.109	2.720	0.064	332.501
WRITE	7098	3.649	0.057	0.004	12.202

Figure 5-1: **Kernel Data Transfer Summary Table of Un-Optimized Smith-Waterman Kernel**

This design was compiled with N=85 and run on an Alpha Data Virtex-7 board. Figure 5-1 shows the kernel data transfer summary table reported by the integrated profiling in SDAccel. The inefficiency of this design is expressed in a few different values in this table. First, the number of read and write transfers is much more than it needs to be. For example, consider the fact that there are only 2*85 = 170 unique values in the two sequences, `s1` and `s2`. However, there are 21,168 read transfers, or 124.5x more than there are unique values.

Second, one reason for the large number of transfers is expressed in the small average size of 64 bytes per transfer (listed in the table as 0.064 KB). This corresponds to a single 512-bit value. This is a very inefficient way of accessing this data since each read data transfer to DDR can take 50-70 clock cycles.

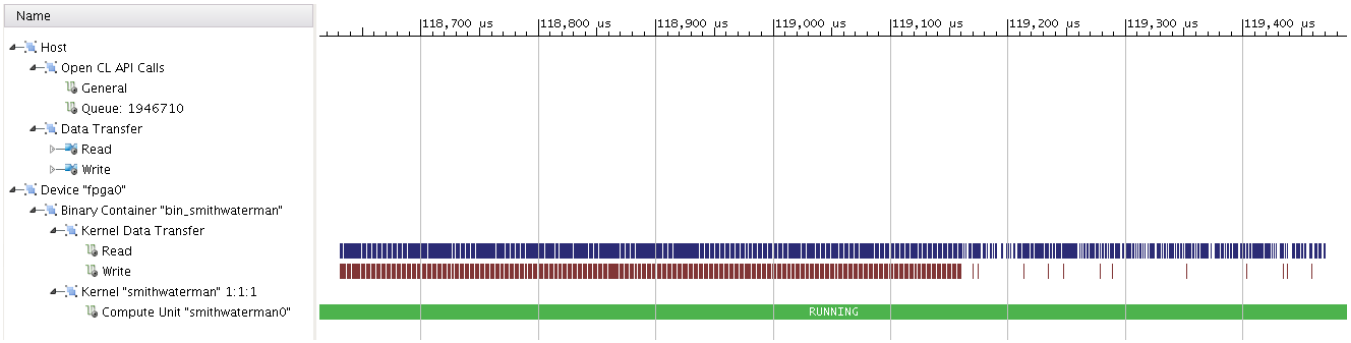

Figure 5-2: **Timeline Trace of Un-optimized Smith-Waterman Kernel**

The high number of data transfers is confirmed in the timeline trace. Figure 5-2 shows a partial view of the timeline trace for this un-optimized kernel. While this kernel is busy with many data transfers, each transfer only involves a burst length of one. Since these values are stored in sequential addresses in global memory, this is a poor method of reading this data.

Optimized Design

```
__attribute__ ((reqd_work_group_size(1, 1, 1)))
kernel void smithwaterman (global int *matrix, global int *maxIndex, global const
char *s1, global const char *s2) {
    short north = 0;
    short west = 0;
    short northwest = 0;
    int maxValue = 0;
    int localMaxIndex = 0;
    int gid = get_global_id(0);

    // Local memories using BlockRAMs
    local char localS1[N];
    local char localS2[N];
    local int localMatrix[N*N];

    async_work_group_copy(localS1, s1, N, 0);
    async_work_group_copy(localS2, s2, N, 0);
    async_work_group_copy(localMatrix, matrix, N*N, 0);

    __attribute__((xcl_pipeline_loop)
    for (short index = N; index < N * N; index++) {
      short dir = CENTER;
      short val = 0;
      short j = index % N;
        if (j == 0) { // Skip the first column
          west = 0;
          northwest = 0;
          continue;
        }
      short i = index / N;
      short2 temp = localMatrix[index - N];
      north = temp.x;
      const short match = (localS1[j] == localS2[i]) ? MATCH : MISMATCH;
```

```
        short val1 = northwest + match;

        if (val1 > val) {
          val = val1;
          dir = NORTHWEST;
        }
        val1 = north + GAP;
        if (val1 > val) {
          val = val1;
          dir = NORTH;
        }
        val1 = west + GAP;
          if (val1 > val) {
            val = val1;
            dir = WEST;
          }0   temp.x = val;
        temp.y = dir;
        localMatrix[index] = as_int(temp);
        west = val;
        northwest = north;
        if (val > maxValue) {
          localMaxIndex = index;
          maxValue = val;
        }
      }

      *maxIndex = localMaxIndex;
      async_work_group_copy(matrix, localMatrix, N*N, 0);
}
```

The source code above shows the optimized Smith-Waterman kernel. Key changes are highlighted in red. To increase the efficiency of the memory accesses, burst reads and writes were added using calls to the `async_work_group_copy()` function. This enables large amounts of data to be transferred between the kernel and global memory. To temporarily store the data locally, three local memories were added: `localS1`, `localS2`, and `localMatrix`. These memories involve local storage with fast single-cycle accesses and provide buffer interfaces between global memory and the data processing in the kernel.

Data Transfer: Kernels and Global Memory

Transfer Type	Number Of Transfers	Transfer / Execution Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)
READ	2	2.871	0.045	0.256	89153.000
WRITE	1	1.436	0.022	0.256	178306.000

Figure 5-3: **Kernel Data Transfer Summary Table of Optimized Smith-Waterman Kernel**

Figure 5-3 shows the kernel data transfer summary table of the optimized Smith-Waterman kernel. All of the data can be transferred between kernel and global memory in three burst transfers: two read and one write. The transfers of `s1` and `s2` are completed in a single transfer, while the values of `matrix` are read in the second read transfer. The new values of `matrix` are written to global memory in the single write transfer. The `Transfer Rate` is low, however, which is expected because of the small number of transfers. This is typical of a compute-intensive algorithm such as Smith-Waterman.

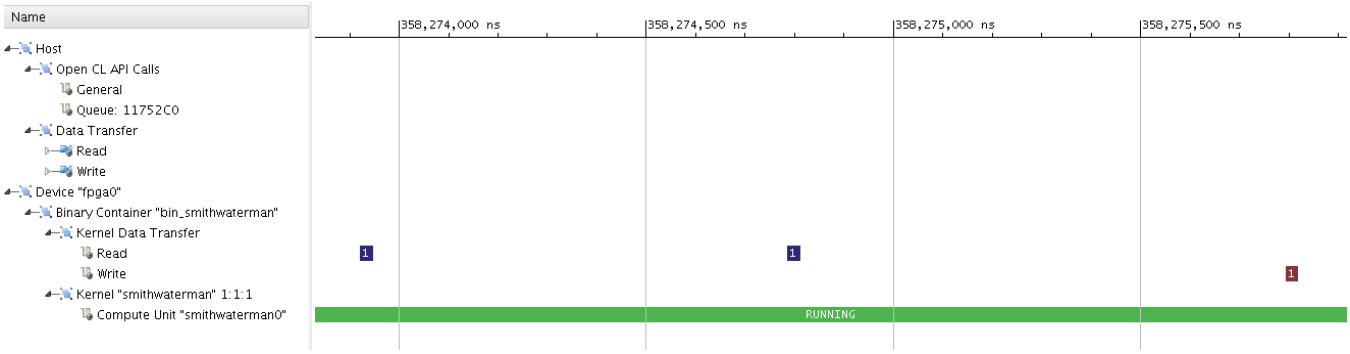

Figure 5-4: **Timeline Trace of Optimized Smith-Waterman Kernel**

The timing of the small number of data transfers is confirmed using trace. Figure 5-4 shows the timeline trace of the optimized Smith-Waterman kernel. Two read transfers are executed, then processing is performed on the data according to the algorithm. Once this computation is completed, then the write transfer is executed.

Note that the performance of the processing portion of the kernel was also improved by applying a `xcl_pipeline_loop` attribute to the kernel source code, as shown above. See Loop Pipelining, page 65 for more details on this attribute.

In summary, we used the SDAccel profiling to deduce that the poor performance of our Smith-Waterman kernel was the inefficient memory transfers. To improve on this, we took advantage of bursts to transfer large amounts of data from the off-chip global memory. In order to accommodate this data, we added local memories to store the data in the kernel. This enabled the kernel to perform much faster reads and writes during processing. With the optimizations discussed above, the kernel execution rate improved 41.0x to 0.19 msec.

Putting It All Together

Overview

This chapter describes a median filter kernel which applies a 3x3 filter window to an image. This application is typically used in noise reduction and removal [Ref 6]. Three different versions of the design are described: one un-optimized, one partially optimized, and one fully optimized. The optimizations applied to this kernel include both data path and memory access optimizations, typical of a more complex design. The profiling results of each design will be analyzed and the process to interpret these results and improve the kernel will be discussed.

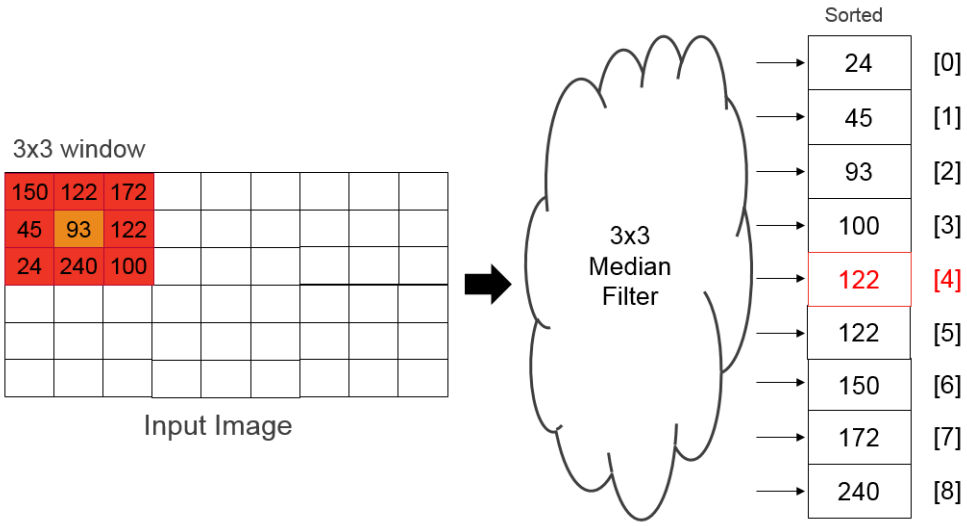

Figure 6-1: **3x3 Median Filter Example**

A median filter is a popular image and video processing design used for noise removal. Figure 6-1 shows how the filter uses a two-dimensional aperture or window surrounding a pixel in the image or frame and then calculates the median value within that aperture. The pixel is then replaced by this median value. Using the example values shown in the above figure, this new median value would be 122. This value is calculated after comparing and sorting the pixel values within the filter window. For a color image that contains three components (i.e., red, green, and blue), this median value is calculated separately for all three components.

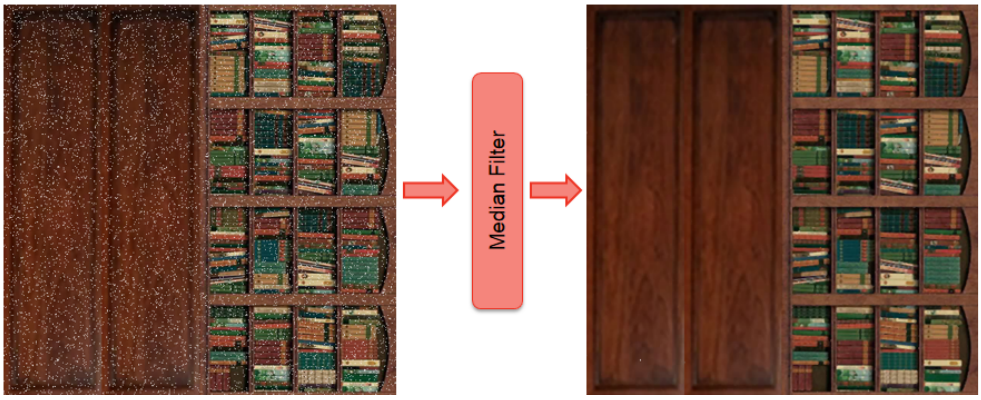

Figure 6-2: **Original and Filtered Images To Demonstrate the Effects of the Median Filter**

Figure 6-2 shows how the median filter is applied to an image as well as its effects on noise removal. On the left is the example Bookcase image with white noise added. After the 3x3 median filter is applied to this image, the actual resulting image read from hardware is shown on the right of the above figure. While this filter performs a smoothing of the values, you can see how it is also excellent at removing the high and low values typical of noise.

Table 6-1: **Comparison of Kernel Execution Times for Median Filter Designs Using 512x512 Images**

Median Filter Design	Kernel Execution Time (msec)
Un-optimized	445.9
Partially Optimized (Line buffers only)	167.6
Fully Optimized	6.0

Table 6-1 lists the three median filter designs that will be described and compared in this chapter. The kernel execution times were taken from the profile summary report as run on an Alpha Data Virtex-7 board [Ref 7]. As shown in the above table, the optimizations about to be described had a high impact on the performance of the median filter kernel. This chapter will describe the series of analyses and subsequent optimizations applied to this kernel.

Un-Optimized Design

The code snippet below displays the first implementation, which is a capture of the median filter algorithm directly into kernel code:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void median(__global const uint* input, __global uint* output, int width, int height)
{
  for (int y=0; y < height; y++) {
    int offset = y * width;
    int prev = offset - width;
    int next = offset + width;

    for (int x=0; x < width; x++) {
      // Get pixels within 3x3 aperture
      uint rgb[SIZE];
      rgb[0] = input[prev + x - 1];
      rgb[1] = input[prev + x];
      rgb[2] = input[prev + x + 1];

      rgb[3] = input[offset + x - 1];
      rgb[4] = input[offset + x];
      rgb[5] = input[offset + x + 1];

      rgb[6] = input[next + x - 1];
      rgb[7] = input[next + x];
      rgb[8] = input[next + x + 1];

      uint result = 0;

      // Iterate over all color channels
      for (int channel = 0; channel < CHANNELS; channel++) {
        result |= getMedian(channel, rgb);
      }

      // Store result into memory
      output[offset + x] = result;
    }
  }
}
```

It was designed for functional correctness but not optimized in any way. There are two `for` loops to traverse the entire 2-D frame. For every pixel in the image, a 3x3 aperture of pixel values is read and copied locally. Each pixel is 24 bits (e.g., 8 bits for red, green, blue). Another for loop then computes the median value of the aperture for each of the three color channels.

Data Transfer: Kernels and Global Memory

Device	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)
fpga0-0	READ	2359296	178.542	2.789	0.064	310.019
fpga0-0	WRITE	2048	1.240	0.019	0.512	347148.000

Figure 6-3: **Kernel Data Transfer Summary for Un-Optimized Median Filter Kernel**

Figure 6-3 shows the kernel data transfer summary table as taken from the SDAccel profile summary. Note the large amount of read transfers, each only involving 64 bytes. Each transfer actually involves reading a single 4-byte value representing one pixel. The total number of read transfers is exactly as expected for a 512x512 image using a 9-pixel aperture (512 * 512 * 9 = 2359296). That is, the kernel reads the 3x3 aperture before computing the median values for every pixel in the image. This is the equivalent of reading the entire image nine times. Clearly, this is an inefficient design.

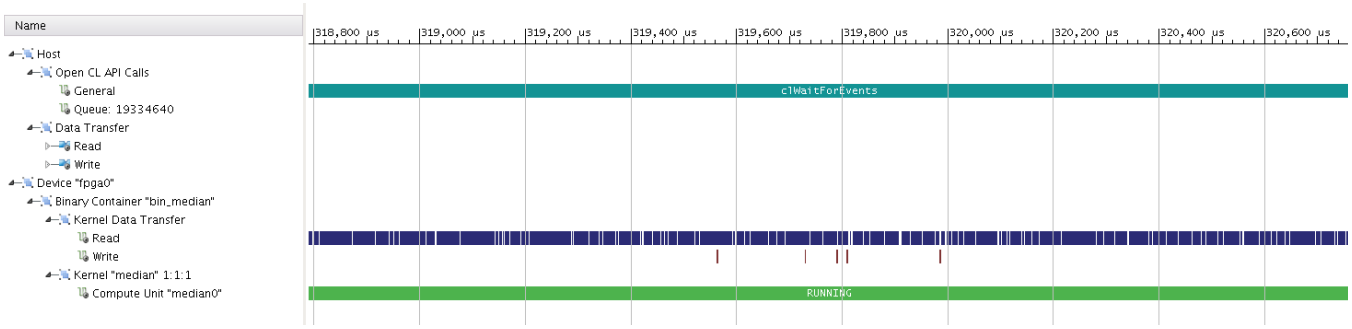

Figure 6-4: **Timeline Trace of Un-Optimized Median Filter Kernel**

Figure 6-4 shows a portion of the timeline trace for the un-optimized median filter design. Focusing on the `Kernel Data Transfer - Read`, this timeline confirms the lack of efficiency in the DDR memory reads. The creation of the aperture (i.e., the 9 values in the local `rgb` array) involves nine individual reads from off-chip global memory since that is where the global input pointer references. While reading these values is necessary to create this aperture, there are two problems with this kernel design approach:

1. Reads One Value at a Time - Only one value is read on each data transfer. The transaction, therefore, has a burst size of 1. This is very inefficient since nearby values in memory will also be used in the same as well as subsequent apertures.
2. Reads Values Multiple Times - Each pixel value is read from the DDR nine different times for each of the apertures it is a part of. This is also inefficient as the data values are not stored and re-used in subsequent apertures.

Partially Optimized Design

One popular technique in video design is to use line buffers to store all pixel values from an entire line of the image or frame. These buffers are loaded once per line (e.g., 512 pixel

values for a 512x512 image) and can also be traversed using common, incremental addressing to create the apertures.

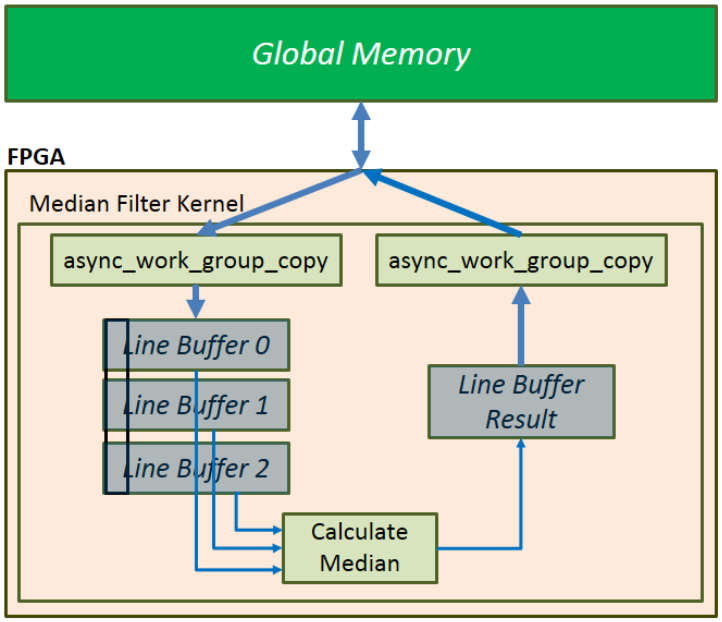

Figure 6-5: **Block Diagram of Partially Optimized Median Filter Kernel**

Figure 6-5 shows how line buffers were added to the median filter kernel. This modification addressed the two above mentioned concerns and provided the following benefits for the median filter design:

1. Reads Bursts of Data - Calls to `async_work_group_copy()` were used to fill up the line buffers with a row of image data. This uses burst access to the data stored in DDR memory and thus, provides the best performance.
2. Reads Every Value Only Once - After the line buffer is filled, the data is continually used as we sweep across its contents to provide different apertures. We also re-use that line buffer and only re-fill it once its current contents are no longer needed. This enables the kernel to read every pixel value only once.

Note that the usage of these line buffers in the median filter is a bit different than typical line buffers as pixel values do not shift across the buffer on every clock cycle. However, they do store an entire row, and accessing their values is done using simple incremental addressing.

Below is a code snippet of a partially optimized median filter kernel:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void median(__global const uint* input, __global uint* output, int width, int height)
{
  local uint linebuf0[MAX_WIDTH];
  local uint linebuf1[MAX_WIDTH];
  local uint linebuf2[MAX_WIDTH];
  local uint lineres[MAX_WIDTH];

  for (int line = 0; line < height; line++) {
    if (line == 0) {
      async_work_group_copy(linebuf0, input, width, 0);
      async_work_group_copy(linebuf1, input, width, 0);
      async_work_group_copy(linebuf2, input + width, width, 0);
    }
    else if (line < height-1) {
      if (line % 3 == 0)
        async_work_group_copy(linebuf0, input + (line+1)*width, width, 0);
      else if (line % 3 == 1)
        async_work_group_copy(linebuf1, input + (line+1)*width, width, 0);
      else if (line % 3 == 2)
        async_work_group_copy(linebuf2, input + (line+1)*width, width, 0);
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    for (int x=0; x < width; x++) {
      uint rgb[SIZE];
      rgb[0] = (x == 0) ? linebuf0[x] : linebuf0[x - 1];
      rgb[1] = linebuf0[x];
      rgb[2] = (x == width-1) ? linebuf0[x] : linebuf0[x + 1];

      rgb[3] = (x == 0) ? linebuf1[x] : linebuf1[x - 1];
      rgb[4] = linebuf1[x];
      rgb[5] = (x == width-1) ? linebuf1[x] : linebuf1[x + 1];

      rgb[6] = (x == 0) ? linebuf2[x] : linebuf2[x - 1];
      rgb[7] = linebuf2[x];
      rgb[8] = (x == width-1) ? linebuf2[x] : linebuf2[x + 1];

      uint result = 0;
      for (int channel = 0; channel < CHANNELS; channel++) {
        result |= getMedian(channel, rgb);
      }
      lineres[x] = result;
    }

    async_work_group_copy(output + line*width, lineres, width, 0);
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

Four line buffers were added to the kernel: three for pre-filtered input and one for the filtered result. Calls to `async_work_group_copy()` were also added to perform write and read transfers between these buffers and DDR memory.

Data Transfer: Kernels and Global Memory

Device	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)
fpga0-0	READ	2052	6.270	0.098	0.512	350.899
fpga0-0	WRITE	2048	6.258	0.098	0.512	541.340

Figure 6-6: **Kernel Data Transfer Summary for Partially Optimized Median Filter Kernel**

Figure 6-6 shows the data transfer summary table between DDR global memory and the partially optimized median filter kernel. Again, the image size is 512x512. The number of read transfers was drastically reduced to 2052. This was partly due to the burst transfers, where the average size was increased to 512 bytes. Also, since the kernel re-used values in the line buffers, there was a 9x reduction in the total amount of data that the kernel read.

While the memory accesses have clearly been optimized, the transfer rate and bandwidth utilization are still very low. To decipher why this is the case, utilize the timeline trace to show details of the kernel activity.

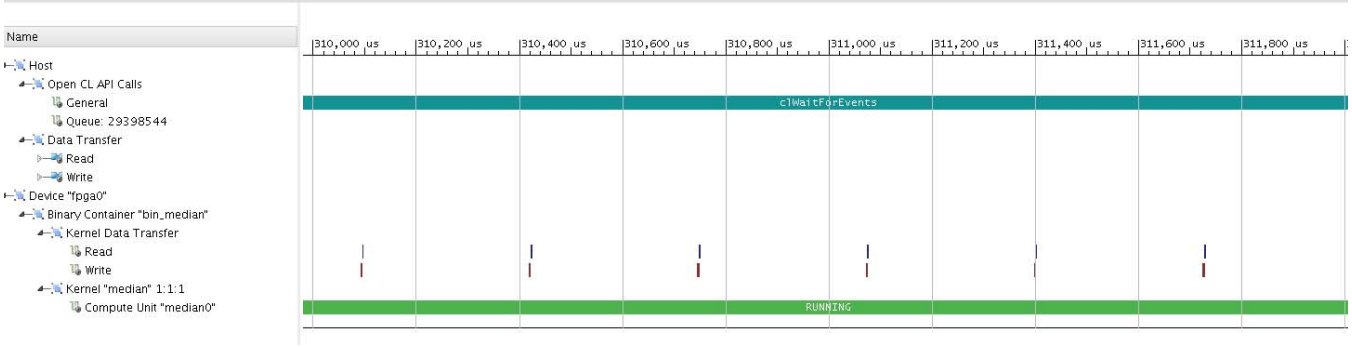

Figure 6-7: **Timeline Trace of Partially Optimized Median Filter Kernel**

Figure 6-7 shows the timeline trace for this partially optimized median filter kernel. The reads and writes using `async_work_group_copy()` are very efficient as a line of 512 pixels is transferred in two transactions, each with a burst size of 16. Each line of pixels is 512 pixels * 32 bits = 16,384 bits per line. The DDR memory controller burst transfers 512 bits * 16 beats for a total of 8,192 bits per burst. Therefore, two burst transactions are required to completely transfer one line of 512 pixels.

While the data transfer issues were optimized, the inefficiency is in the computation or processing time. The design uses 372.6 usec to process a row of pixels, as measured from the end of the second read transfer to the beginning of the first write transfer. This is an unacceptably long period to process the data. The next section addresses how this can be optimized.

Fully Optimized Design

There are two data path optimizations that can be performed on the median filter kernel:

1. Local memory reads and processing can be pipelined - The kernel operation involves reads, computation, and writes. It reads from the line buffers and creates the aperture by storing it in an array of size 9 called `rgb`. It then calculates the median value of this aperture for each of the color channels and writes the results to another line buffer. All of these operations were pipelined by including the `xcl_pipeline_loop` attribute before the `x for` loop. This informs the compiler to keep the implementation of reads, computation, and writes to be as busy as possible.
2. Color channels can be processed in parallel - We have a `for` loop that computes the median value for all three color channels. Since there are no dependencies across color channels, we can compute these three, separate median values in parallel. We added an `opencl_unroll_hint` attribute before the channel for loop. This informs the compiler to fully unroll this loop and add hardware such that all three median values are calculated concurrently. Since this loop is only of size three, a considerable speedup can be achieved with only a slight increase in FPGA resources.

Below is a code snippet of a fully optimized median filter kernel:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void median(__global const uint* input, __global uint* output, int width, int height)
{
  :
  for (int line = 0; line < height; line++) {
    :
    __attribute__((xcl_pipeline_loop))
    for (int x=0; x < width; x++) {
      :
      __attribute__((opencl_unroll_hint))
      for (int channel = 0; channel < CHANNELS; channel++) {
        result |= getMedian(channel, rgb);
      }
      lineres[x] = result;
    }

    async_work_group_copy(output + line*width, lineres, width, 0);
    barrier(CLK_LOCAL_MEM_FENCE);
  }
}
```

This kernel uses the same source code as the partially optimized kernel except for two additions: the `x for` loop was pipelined; and the channel `for` loop was unrolled. These two lines of code inform HLS to perform the two data path optimizations listed above.

Data Transfer: Kernels and Global Memory

Device	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Average Time (ns)
fpga0-0	READ	2052	175.918	2.748	0.512	347.554
fpga0-0	WRITE	2048	175.575	2.743	0.512	541.633

Figure 6-8: **Kernel Transfer Summary for Fully Optimized Median Filter Kernel**

Figure 6-8 shows the data transfer summary table for the fully optimized median filter kernel. Note the number of read transfers was again 2052 and the average size was still 512 bytes. This informs you that the memory access optimizations were still in place and did not change. However, transfer rate and bandwidth utilization values are much higher. This tells you the kernel execution time was full of efficient memory transfers.

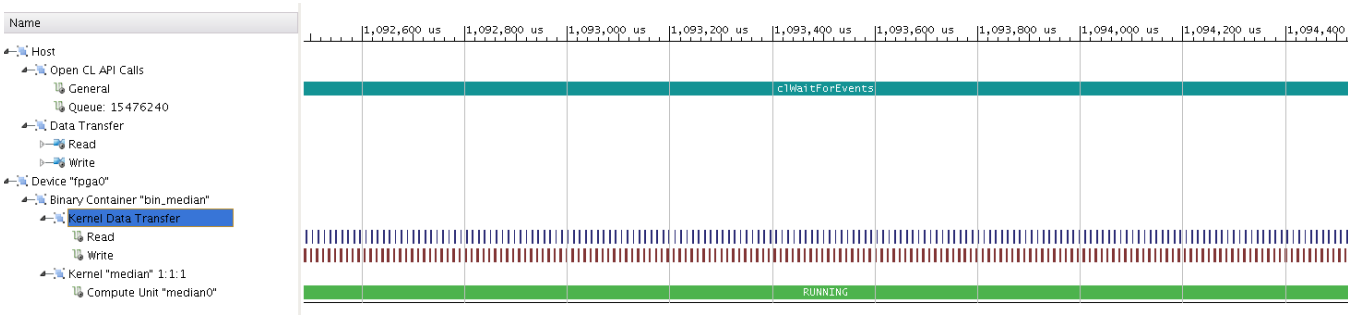

Figure 6-9: **Timeline Trace of Fully Optimized Median Filter Kernel**

The timeline trace confirms this efficiency. Figure 6-9 shows the timeline trace of the fully optimized median filter design. Note that the continuous write data transfers demonstrates that the hardware is constantly kept busy throughout the processing. Table 6-1, page 46 shows that the run-time of this fully optimized median filter design is 6.0 msec to process a 512x512 image. This performance is 74.3 times faster than the un-optimized kernel and 27.9 times faster than the partially optimized kernel.

Data Transfer Analysis

Looking back on the progressive optimizations of the median filter design, an interesting question arises: what is the maximum throughput of this design? A related question would be: when do you stop optimizing? These are good question and get to the heart of the optimizations performed. To arrive at answers, understanding how to optimize both the data transfers and the computation in your design is critical.

To that end, one method to accomplish this would be to create various test kernels that separate out the data transfer from the computation. This actually teaches you a few things about both. Specific to the median filter, consider the test kernel below:

```
void test(__global const int* input, __global int* output) {
  local int linebuf0[WIDTH];
  local int linebuf1[WIDTH];
  local int linebuf2[WIDTH];
  local int result[WIDTH];

  for (int line = 0; line < HEIGHT; line++) {
    // Fetch Lines
    if (line == 0) {
      async_work_group_copy(linebuf0, input, WIDTH, 0);
      async_work_group_copy(linebuf1, input, WIDTH, 0);
      async_work_group_copy(linebuf2, input + WIDTH, WIDTH, 0);
    }
    else if (line < HEIGHT-1) {
      if (line % 3 == 0)
        async_work_group_copy(linebuf0, input + (line+1)*WIDTH, WIDTH, 0);
      else if (line % 3 == 1)
        async_work_group_copy(linebuf1, input + (line+1)*WIDTH, WIDTH, 0);
      else if (line % 3 == 2)
        async_work_group_copy(linebuf2, input + (line+1)*WIDTH, WIDTH, 0);
    }
    barrier(CLK_LOCAL_MEM_FENCE);

    async_work_group_copy(output + line*WIDTH, result, WIDTH, 0);
    barrier(CLK_LOCAL_MEM_FENCE);
  } // for line
}
```

This test kernel contains the same data transfers as the median filter kernel. However, it contains no computation of median values. The contents of the result buffer is simply copied to the output pointer without any processing or computation. While clearly this kernel is not functionally correct, this type of test kernel addresses the two questions mentioned above:

1. What is the maximum throughput of this design? Creating a test kernel that only contains the data transfers allows us to isolate the reading and writing of data to/from the kernel.
   a. Since data transfers are typically bottlenecks in a kernel, emulating this kernel or running it on a board will tell you the maximum performance expected given the selected data transfers.
   b. If this test kernel does not meet your performance requirements, then you need to improve on either the amount of data or the method that you read and write the data.

2. When do you stop optimizing? Once you have achieved the desired throughput and performance with this test kernel, you can compare it to the performance of the actual kernel.

   a. If the actual kernel has much lower performance than the test kernel, then optimizations need to be performed to improve on the computation time. For the median filter, this involved pipelining or unrolling two of the loops.

   b. Once the test kernel and actual kernel have similar performance, then the computation time of the kernel has been optimized.

The execution time for the test kernel shown above is approximately 6.0 msec to operate on a 512x512 image. Almost exactly the same execution time of the fully optimized design. That means that the computation time of the kernel was fully optimized. Any further improvements to the kernel would need to come from optimizing the memory accesses.

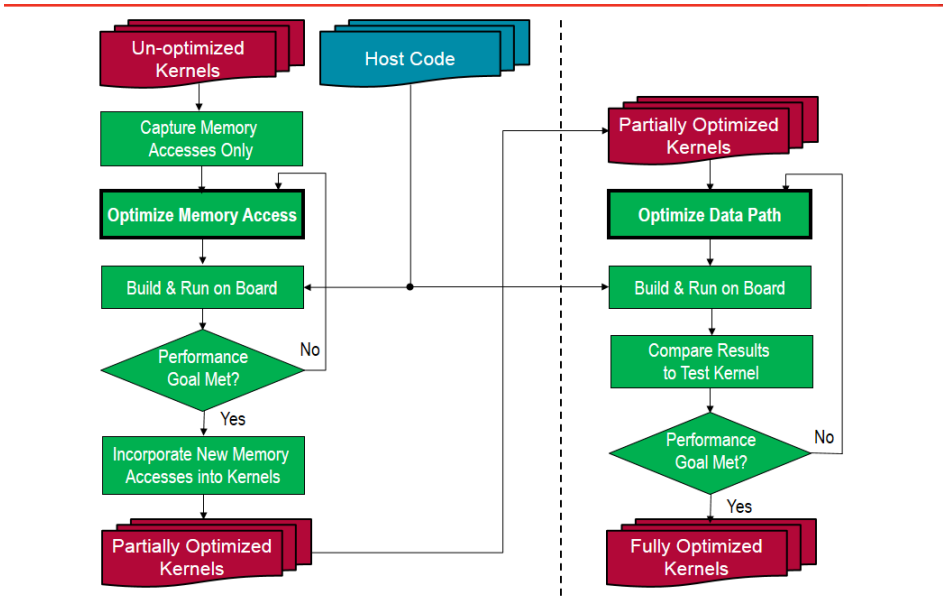

Figure 6-10: **Flow Diagram of One Methodology to Optimize Kernels**

Figure 6-10 shows a flow diagram for one suggested method of optimizing the performance of kernels. This follows the procedure taken to optimize the median filter. The memory accesses for the original, un-optimized kernel were extracted and a test kernel was created. The memory accesses were then optimized using line buffers and burst transfers with `async_work_group_copy`. If this extraction is not possible, then the memory access optimizations can certainly be performed on the original kernel.

Once the desired performance was met, then if needed, these new data transfers were incorporated back into the median filter kernel and a partially optimized kernel was created. The computation or data path was then optimized using pipelining and unrolling until the desired performance goal was met. The result was a fully optimized kernel. Note that the same host code was used throughout the process, including using the test kernel and all variations of the median filter kernel.

Performance Checklist

Overview

The goal of this chapter is to create a checklist of items to consider when evaluating the performance of your SDAccel design. This should by no means be considered an exhaustive list, but instead a starting point for ideas to consider or investigate further.

- **Verify functional correctness**: First and foremost before you begin performance optimizations, be certain that you have correctly captured the desired functionality of your algorithm. In SDAccel, this can be verified using CPU emulation. You can create a "testbench" by running a corresponding function in host software and comparing the results with the kernel.
- **Use profiling summary**: This is a great starting point to tell you the overall performance health of your system. The summary tables can tell you whether or not performance goals have been met as well as clues on where to begin making improvements.
    - For continued functional correctness, first make sure the total data transferred for both write and read is correct by viewing the results in the Top Data Transfer: Kernels and Global Memory table. Some applications have very specific data transfer amounts (e.g., an HD video frame).
    - Next, view other important metrics such as number of transfers, average size or bytes, transfer rate, and bandwidth utilization. These metrics can be used to analyze the proficiency of the memory accesses and the data paths in the system.
- **Use timeline trace**: Trace tells you details of the timeline progression of your emulation or run on a board. The timestamped events provide insight into potential bottlenecks or dependencies that may have executed your system in an un-optimized manner. While the profiling summary provides useful aggregated results, trace shows you the relative timing of important events in the system. In SDAccel, host and device level events are plotted on the same timeline.
- **Use burst data transfers**: This enables a kernel to transfer as much data as is required and allows kernels to read data before it is needed by the computation. Large bursts also optimize the performance of the memory controller. Burst transfers can be defined using async_work_group_copy in OpenCL C, memcpy in HLS C/C++, or inferred from successive requests of data from consecutive address locations. Both the optimized

Smith-Waterman kernel (described in Optimized Design, page 42) and the partially and fully optimized median filter kernels (described in Partially Optimized Design, page 48 and Fully Optimized Design, page 52) used this method to optimize their performance.

- **Isolate data transfer or computation**: Take advantage of the fact that FPGAs are re-programmable and iterate to verify the performance of multiple test kernels. One methodology to better understand the performance of your kernels is to isolate either the data transfers or the computation. Separating out the two will help you to better understand where to begin your optimizations. See Data Transfer Analysis, page 53 for more details.
- **Use local and private memories**: This complements the burst data transfer concept. Local and private memories can be used repetitively as scratch pads, and both reads and writes can be accomplished in a single clock cycle. FPGAs provide a wealth of BlockRAMs distributed throughout the chip. Use them as much as possible.
- **Use on-chip global memories**: This also complements the burst data transfer concept by allowing data to be shared between multiple kernels and compute units. See On-Chip Global Memories, page 70 for more details.
- **Use on-chip pipes**: This allows data to be streamed between multiple kernels and compute units. See On-Chip Pipes, page 72 for more details.
- **Use workgroups**: This is a concept exclusive to OpenCL and should be exploited. Workgroups allow the system to schedule tasks in parallel and thus, can enable better overall performance. See Multiple Compute Units, page 36 for how this was used to improve the performance of the matrix adder.
- **Use multiple memory ports**: This setting increases the memory bandwidth available to a kernel by increasing the number of connections to memory attached to a kernel. See Multiple Memory Ports per Kernel, page 73 for more information.
- **Use the entire port width**: Take advantage of the entire 512-bit memory controller interface. This can be accomplished using a combination of vectors and Tcl settings. See Vectorization, page 60 and Adjustable Memory Port Data Width, page 74 for more information.
- **Unroll loops**: Loops that are unrolled are performed simultaneously rather than in an iterative, sequential fashion. This is appropriate for small to medium sized loops such as the color channel loop described in Chapter 6, Putting It All Together. See Loop Unrolling, page 63 for further details.
- **Use pipelining**: This can be either loop or work item pipelining and exposes pipelining capabilities to Vivado HLS. This is appropriate for larger loops such as the x for loop described in Chapter 6, Putting It All Together. See Loop Pipelining, page 65 and Work Item Pipelining, page 67 for more information.

Tool Flow Suggestions

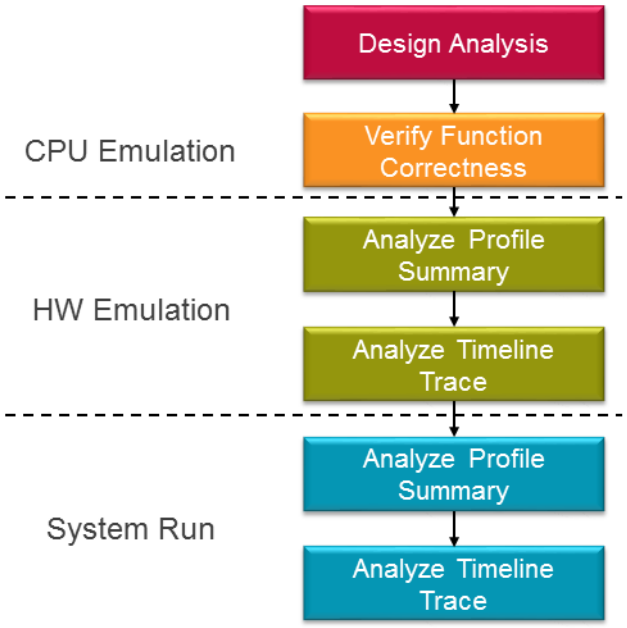

Figure 7-1: **Flow Diagram Using All Three Flows To Optimize Kernel Performance**

Figure 7-1 shows a possible flow diagram using all three compilation flows in SDAccel to analyze and optimize kernel performance. First, verify functional correctness using CPU emulation. This is the fastest flow and can enable fast verification of functionality. One method of doing this is to add a 'testbench' in your host code. A testbench provides a corresponding software function that performs the exact same functionality as the kernel, runs both, and compares the results.

Next, use hardware (HW) emulation to evaluate performance. Check the profile summary for an overview of the run, then view the timeline trace for details. A few items should be noted when running HW emulation. First, consider using a smaller, representative data set size to minimize emulation runtime. Second, note that the profile results are estimates of actual performance. Profiling results for HW emulation are very good at providing comparative results across multiple kernel revisions and solutions.

Finally, the same performance evaluation can then be done for a system run on a board. Again, check the profile summary for overview metrics of your run. If needed, then view the timeline trace for details of timing and performance.

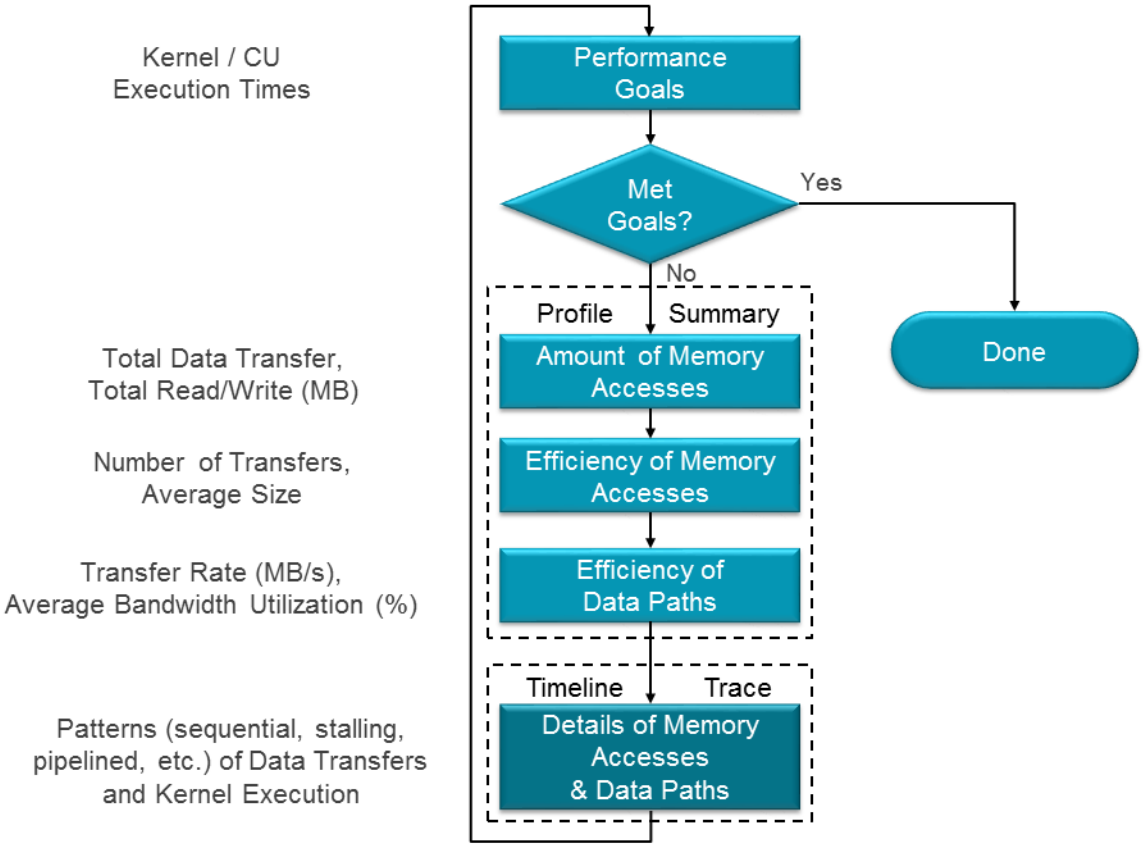

Figure 7-2: **Flow Diagram of Kernel Performance Optimizations Within System Runs**

Once you start running on a board, Figure 7-2 shows a simplified flow diagram to achieve kernel performance optimizations. At the highest level, iterations are accomplished until performance goal(s) have been met. Within each iteration, profiling results are analyzed to verify amounts and efficiencies of both memory accesses and data paths. These different results are listed on the left-hand side of the above figure and can all be found in the SDAccel profile summary and timeline trace.

First, memory accesses are analyzed. If the amount of data (in MB) is correct, then the efficiency is analyzed. This efficiency is reported as such values as number of transfers and average size. Second, the efficiency of the kernel data paths or computation is evaluated using such metrics as transfer rate and average bandwidth utilization. Finally, the details of the timeline trace are viewed to see if further information can be extracted from the run. After subsequent updates are made to the kernel(s), these steps are repeated until the previously stated goals have been met.

Improving Data Path Performance

Overview

There are a number of different methods to improve the performance of an OpenCL kernel. This appendix introduces an important subset of these methods which specifically address data path performance. To achieve this, there are a number of key constructs and attributes for the OpenCL kernel language. This appendix describes a few of these, demonstrates how they are used, and shows their effects on performance using the integrated profiling capabilities in SDAccel.

Vectorization

Vectorization creates a wide computation data path inside the kernel. Vectorization is created by changing the data type of the data to be processed. For example, the following "unvectorized" code performs 256 loop iterations and reads each element of `a` and `b` separately.

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd( __global int* a, __global int* b, __global int* c) {
  int i;
  for (i=0; i< 256; i++) {
    c[i] = a[i] + b[i];
  }
}
```

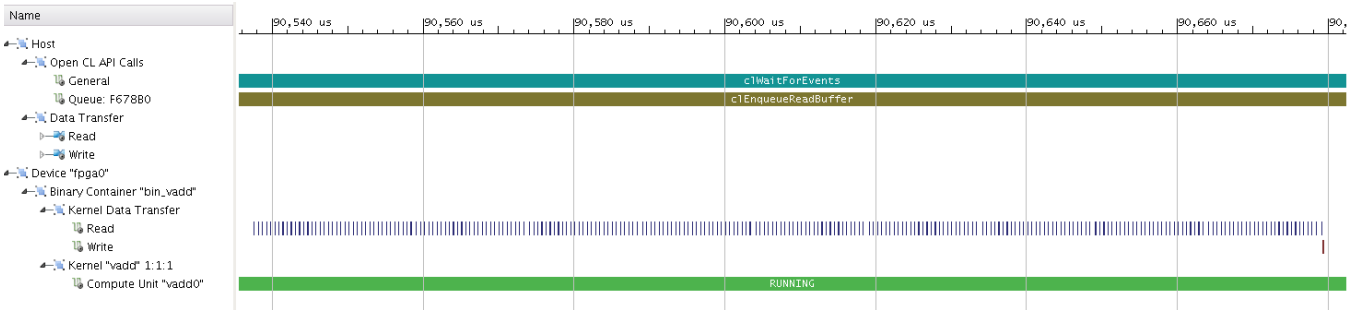

Figure A-1: **Timeline Trace of Unvectorized Vector Adder with 256 Loop Iterations**

Figure A-1 shows the timeline trace of the unvectorized vector adder. There are 256 loop iterations that are completed separately. Each transaction only obtains one 32-bit integer value, and there are 512 total read transactions. While the implementation of the kernel is able to group the write into a single transaction containing all 256 integers, the timeline above is not ideal performance for this design.

The following vectorized code reads 16 words from global buffer `a` and `b` at a time:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd( __global int16* a, __global int16* b, __global int16* c) {
  int i;
  for (i=0; i< 256/16; i++){
    c[i] = a[i] + b[i];
  }
}
```

The vector is signified by the `int16` pointer type for both inputs as well as the output `c`. This results in 16 integer values being read or written at a time. This value of 16 was chosen for a reason: 16 x 32bits = 512 bits, the width of the memory word on the Alpha Data Virtex-7 board. Also, because 16 additions are performed in parallel, the loop iterates only 16 times, as opposed to 256.

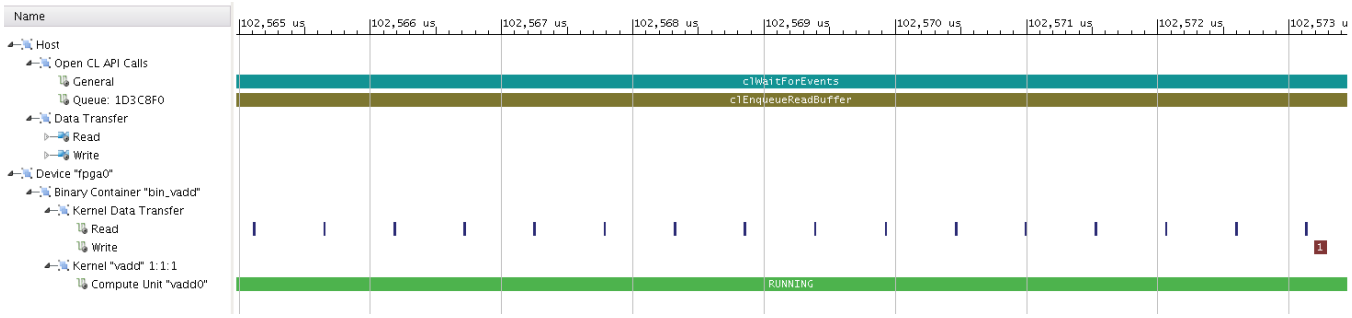

Figure A-2: **Timeline Trace of Vectorized Vector Adder with 16 Loop Iterations**

Figure A-2 shows the timeline trace of the vectorized vector adder design. There are now only 16 loop iterations since each iteration computes 16 output values. This results in a total of 32 read transactions and a single write transaction performed at the end to write the entire vector to DDR. Since the memory interface is 512 bits, a burst size of one can read or write 16 32-bit values in one word.

The profiling summary confirmed that the performance significantly improved: the kernel execution time went from 8.33 msec for the un-vectorized version to 0.08 msec for the vectorized kernel. This is an improvement of 104x.

Note that vectorization on the host is not necessary. In other words, the host code does not have to be re-written if vectors are used in the compute units on the device. The host creates and buffers `a` and `b` as regular 256 element integer arrays. The benefits of vectorization can only be determined by either running hardware emulation or running on a board and viewing the profiling results.

Table A-1: **Vector Types Supported by SDAccel**

Data Type	N=2	N=3	N=4	N=8	N=16
Character	char2	char3	char4	char8	char16
Unsigned Character	uchar2	uchar3	uchar4	uchar8	uchar16
Short	short2	short3	short4	short8	short16
Unsigned Short	ushort2	ushort3	ushort4	ushort8	ushort16
Integer	int2	int3	int4	int8	int16
Unsigned Integer	uint2	uint3	uint4	uint8	uint16
Half	half2	half3	half4	half8	half16
Float	float2	float3	float4	float8	float16

Table A-1 lists all of the vector types supported by SDAccel. These are all defined in the OpenCL 2.0 specification [Ref 1], and SDAccel support is verified using conformance testing [Ref 4]. The data types are the same used by traditional software programming, while the value of N is the number of values in the vector.

Loop Unrolling

Loop unrolling is an important optimization technique available in SDAccel. The purpose of the loop unroll optimization is to expose concurrency to the compiler and take advantage of the parallelism inherent to an FPGA. This is an official attribute in the OpenCL 2.0 specification. For example, consider the following vector multiplier kernel:

```
#define LENGTH 64

__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vmult(__global const int* a, __global const int* b, __global int* c) {
  local int bufa[LENGTH];
  local int bufb[LENGTH];
  local int bufc[LENGTH];
  int tid = get_global_id(0);

  async_work_group_copy(bufa, a, LENGTH, 0);
  async_work_group_copy(bufb, b, LENGTH, 0);

  for (int i=0; i < LENGTH; i++) {
    int idx = tid*LENGTH + i;
    bufc[idx] = bufa[idx] * bufb[idx];
  }

  async_work_group_copy(c, bufc, LENGTH, 0);
}
```

This kernel multiplies two integer vectors, `a` and `b`. The length of the vectors is 64. Since we want to isolate the performance of the for loop, we first read the two vectors into local memories using calls to `async_work_group_copy`. Also, a third local memory is used to store the output vector, `c`, so all data in the for loop uses local memories. Once the loop is completed, the entire output vector is written back to DDR.

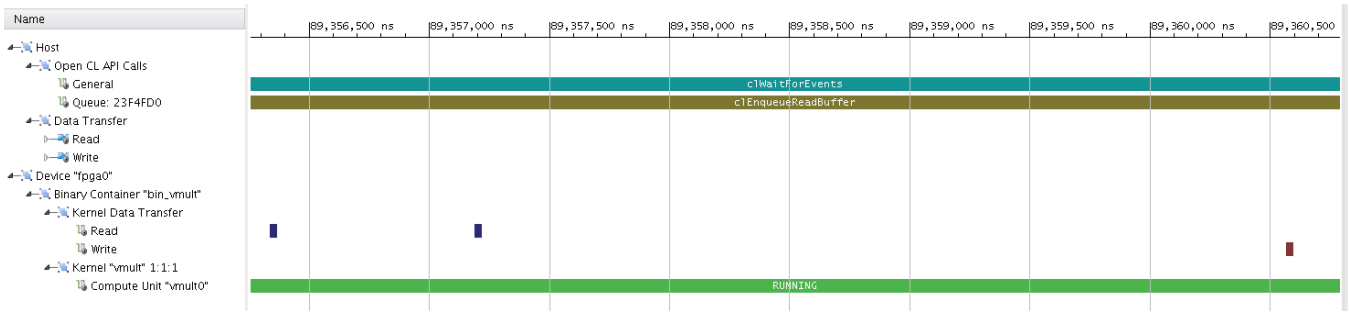

Figure A-3: **Timeline Trace for Sequential Vector Multiplier**

Figure A-3 shows the timeline trace for the sequential vector multiplier. The two read and one write data transfers are evident in the timeline. To demonstrate loop unrolling, the metric of concern is the time in between the second read and the write transfer. This is the processing time of the for loop.

The performance of the vector multiplier can be improved by using the `opencl_unroll_hint` attribute with an unroll factor of 2:

```
__attribute__((opencl_unroll_hint(2)))
for (int i=0; i < LENGTH; i++) {
  int idx = tid*LENGTH + i;
  bufc[idx] = bufa[idx] * bufb[idx];
}
```

The code above tells SDAccel to unroll the loop by a factor of two. This results in `LENGTH`/2 or 32 `loop` iterations for the compute unit to complete the operation. By enabling SDAccel to reduce the loop iteration count, the programmer has exposed more concurrency to the compiler. This newly exposed concurrency reduces latency and improves performance, but also consumes more FPGA fabric resources.

Another variety of this attribute is to unroll the loop completely. The syntax for the fully unrolled version of the vector multiplier example is as shown below:

```
__attribute__((opencl_unroll_hint))
for (int i=0; i < LENGTH; i++) {
  int idx = tid*LENGTH + i;
  bufc[idx] = bufa[idx] * bufb[idx];
}
```

In this fully unrolled design, all of the possible concurrency in the loop nest is exposed to the compiler. SDAccel analyzes the data and control dependencies of the unrolled loop nest and automatically parallelizes all operations that can be executed concurrently.

Due to resource constraints, note that full unrolling is appropriate for `for` loops of small or medium length. Large `for` loops may require too many resources to implement on the FPGA device. For larger loops, it is recommended to use loop pipeline (see the next section).

In general, it is recommended to use the `report_estimate` command and understand how Vivado HLS compiles the kernel code before building a complex system.

Table A-2: **Summary of Performance Results Comparing Different Vector Multiplier Kernels**

Vector Multiplier Kernel	Total Loop Time (usec)
Sequential	3.96
Partially Unrolled	2.68
Fully Unrolled	1.27

Table A-2 summarizes results showing the impact of loop unrolling on the performance of the vector multiplier kernel. The total loop time (in usec) is measured using the timeline trace panel in SDAccel and is the time between the completion of the last read data transfer to the start of the write data transfer. This metric was chosen since the data transfers are the same for all three versions of the kernel. These values show how the loop unrolling can impact the overall performance of a kernel execution.

Loop Pipelining

Although loop unrolling exposes concurrency, it does not address the issue of keeping all elements in a kernel data path busy at all times. This is necessary for maximizing kernel throughput and performance. Even in an unrolled case, loop control dependencies can lead to sequential behavior. The sequential behavior of operations results in idle hardware and a loss of performance.

Xilinx addresses this issue by introducing a vendor extension on top of the OpenCL 2.0 specification for loop pipelining. The Xilinx attribute for loop pipelining is `xcl_pipeline_loop`.

In order to understand the effect of loop pipelining on performance, consider the following code example:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vaccum(__global const int* a, __global const int* b, __global int* result) {
  int tmp = 0;

  for (int i=0; i < 32; i++) {
    tmp += a[i] * b[i];
  }
  result[0] = tmp;
}
```

This kernel code has no attributes and is executed sequentially per the order of operations stated in the kernel code. Although the execution is functionally correct, the implementation is not maximizing performance because the read, multiply, add, and store operations are not always busy. The pipeline attribute serves as a command to the SDAccel compiler to maximize performance and minimize the idle time of any stage in the generated logic.

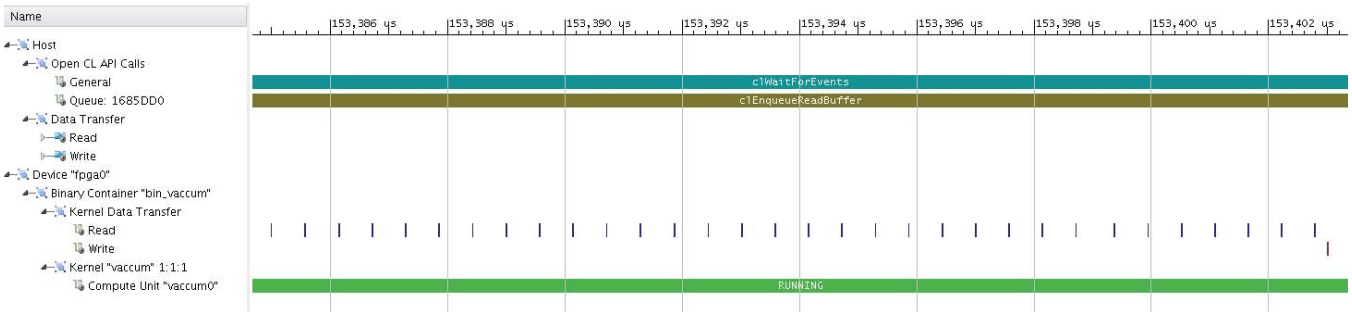

Figure A-4: **Timeline Trace of Sequential Vector Accumulator**

Figure A-4 shows the timeline trace for the sequential vector accumulator. The read data transfers are performed two at a time, one to read the next value of `a` and another for `b`. There is also approximately 100 nsec in between each read to perform the multiplication and addition. Since all of these operations can be pipelined, this is clearly an inefficient kernel design. Most of the design is idle at any point in time.

We can address these concerns by adding pipelining. Example code with an added loop pipeline attribute looks like the following:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vaccum(__global const int* a, __global const int* b, __global int* result) {
  int tmp = 0;

  __attribute__((xcl_pipeline_loop))
  for (int i=0; i < 32; i++) {
    tmp += a[i] * b[i];
  }
  result[0] = tmp;
}
```

Adding this attribute exposes the pipeline nature of the design to the compiler. In turn, the compiler then adds appropriate pipelining to improve the performance of the design.

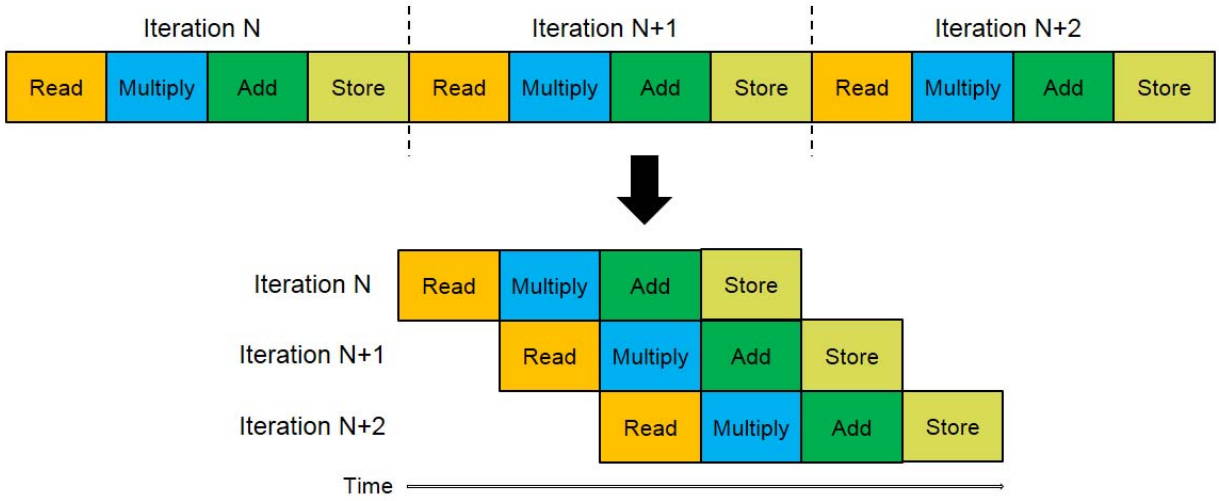

Figure A-5: **Timing Diagram of Loop Pipelining**

Figure A-5 shows a timing diagram of the vector accumulator before and after exposing loop pipelining. The diagram on top is sequential in nature and was confirmed with the timeline trace for the un-pipelined kernel. The diagram below shows the improved timing of the pipelined version. Notice how the different operations are kept busy throughout the loop iterations. Similar to loop unrolling, this exploits the vast hardware resources available on an FPGA.

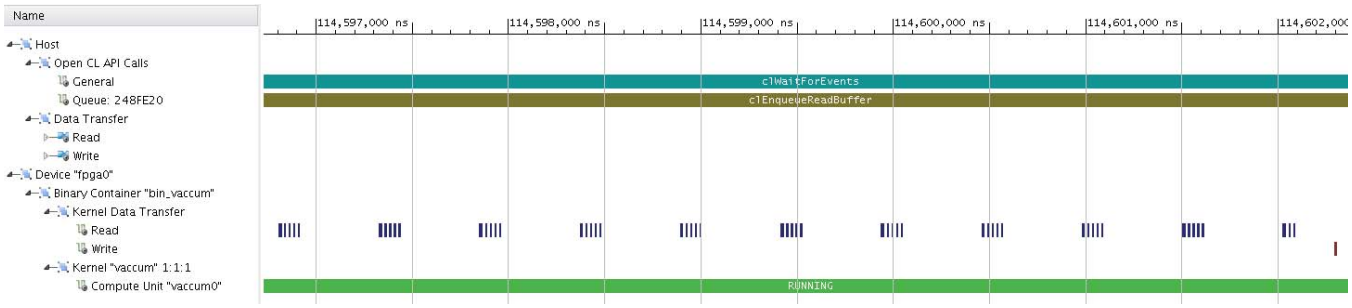

Figure A-6: **Timeline Trace of Pipelined Vector Accumulator Kernel**

Figure A-6 shows the timeline trace of the pipelined vector accumulator kernel. Soon after the kernel starts, it requests six values (three for `a` and three for `b`). This is a limit on the queue size in the kernel. However, even with this limit, the kernel is able to pipeline the operations and perform the multiplication and addition at the same time that the next values are being read. By adding loop pipelining to this example, the execution time of the kernel is decreased by 19% without any other code modification or additional hardware.

Work Item Pipelining

Work item pipelining is the extension of loop pipelining to the kernel work group. The syntax for the attribute for this optimization is `xcl_pipeline_workitems`. An example where work pipelining can be applied is the following kernel:

```
__kernel __attribute__ ((reqd_work_group_size(8, 8, 1)))
void madd(__global int* a, __global int* b, __global int* output)
{
  int rank = get_local_size(0);
  __local unsigned int bufa[64];
  __local unsigned int bufb[64];

  int x = get_local_id(0);
  int y = get_local_id(1);
  bufa[x*rank + y] = a[x*rank + y];
  bufb[x*rank + y] = b[x*rank + y];
  barrier(CLK_LOCAL_MEM_FENCE);

  int index = get_local_id(1)*rank + get_local_id(0);
  output[index] = bufa[index] + bufb[index];
}
```

In order to handle the `reqd_work_group_size` attribute, SDAccel automatically inserts a loop nest to handle the multi-dimensional characteristics of the ND range. For this example, the local work size is specified as (8, 8, 1). As a result of the loop nest added by SDAccel, the execution profile of this code is the same as that of an un-pipelined loop.

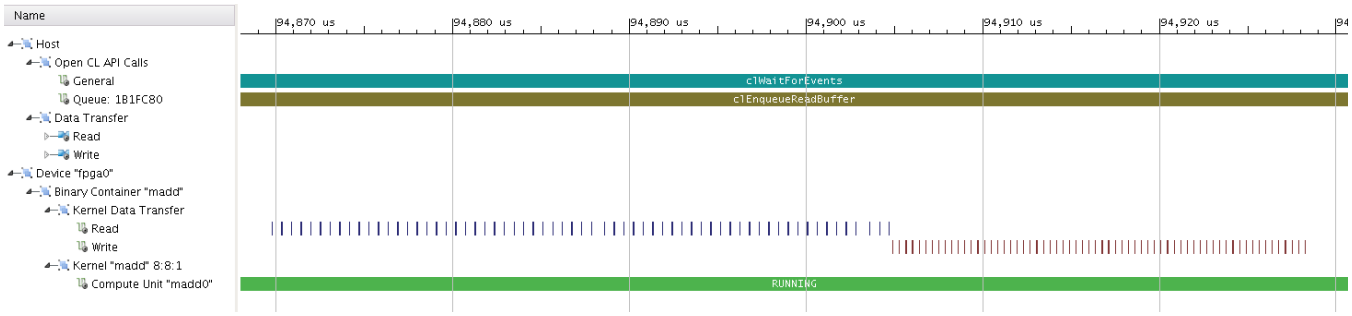

Figure A-7: **Timeline Trace of Sequential Matrix Adder Kernel**

Figure A-7 shows the timeline trace of the un-optimized or sequential matrix adder. Two read data transfers are performed at a time, one to request the next value of `a` and another for `b`. There is then a gap of 80 nsec before the next set of reads begin. The largest number of outstanding data transfers is two. The kernel is also written in such a way that the writes are not performed until the reads have completed. This is due to the `barrier(CLK_LOCAL_MEM_FENCE)` added in the kernel.

The work item pipeline attribute can be added to the code as follows:

```
__kernel __attribute__ ((reqd_work_group_size(8, 8, 1)))
void madd(__global int* a, __global int* b, __global int* output)
{
  int rank = get_local_size(0);
  __local unsigned int bufa[64];
  __local unsigned int bufb[64];

  __attribute__((xcl_pipeline_workitems)) {
    int ix = get_local_id(1)*rank + get_local_id(0);
    bufa[ix] = a[ix];
    bufb[ix] = b[ix];
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  __attribute__((xcl_pipeline_workitems)) {
    int ix = get_local_id(1)*rank + get_local_id(0);
    output[ix] = bufa[ix] + bufb[ix];
  }
}
```

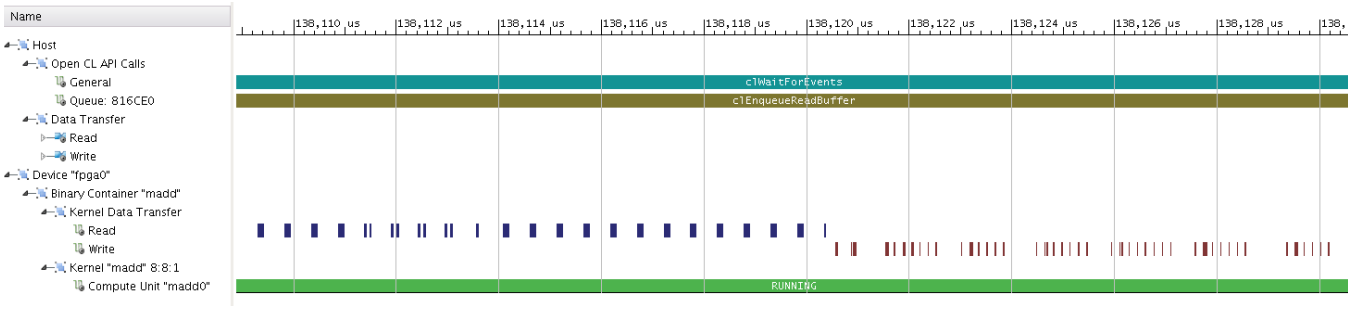

Figure A-8: **Timeline Trace of Pipelined Matrix Adder Kernel**

Figure A-8 shows the timeline trace of the pipelined matrix adder. The maximum number of outstanding data transfers is now six, which is again limited by the queue in the kernel. However, the pipelining is evident since multiple operations are being performed simultaneously by the compute unit. Similar to loop pipelining, the resulting hardware keeps all operations as busy as possible, which in turn maximizes performance. Adding this work item pipelining improves the execution time of the matrix adder kernel from 0.11 msec to 0.07 msec.

Improving Memory Efficiency

Overview

There are a number of different methods to improve the performance of an OpenCL kernel. This appendix introduces another important subset of these methods which specifically address memory efficiency. To improve memory access efficiency, there are a number of memory types, key constructs, and burst access capabilities available in the OpenCL kernel language.

Figure 3-2, page 23 shows the OpenCL memory model used by the Xilinx SDAccel tool. There are a multitude of memory types, and it is important to understand their usage and benefits. This includes the following types: host memory, off-chip global memory, on-chip global memory, local memory, and private memory. The examples in Chapters 4-6 describe many use cases for off-chip global and local memories. This appendix describes on-chip global memories, including a special type called on-chip pipes.

There are also some key constructs and burst access capabilities that can significantly improve performance. This appendix also describes a few of these, demonstrates how they are used, and shows their effects on performance.

On-Chip Global Memories

One memory architectural optimization available in SDAccel utilizes global memories that are used to pass data between kernels. In cases where the global memory buffer used for inter-kernel communication does not need to be visible to the host processor, SDAccel enables you to move the buffer out of DDR-based memory and into the BlockRAMs available on the FPGA. This optimization is called on-chip global memories and is part of the OpenCL 2.0 specification.

On-chip global memories are very useful because they can be accessed by all the kernels. SDAccel generates an AXI bus in the programmable region and connects the global on-chip memory to only the kernels which access it, thus saving on resources. Note that this bus is narrow, and thus does not always achieve as high bandwidth as local memory. However, the cross-kernel accessibility is key.

The following code example illustrates a usage model for global memory buffers:

```
// Global memory buffers used to transfer data between kernels
// Contents of the memory do not need to be accessed by host processor
global int g_var0[1024];
global int g_var1[1024];

// Kernel reads data from global memory buffer written by the host processor
// Kernel writes data into global buffer consumed by another kernel
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void input_stage(global int *input) {
    __attribute__((xcl_pipeline_workitems)) {
      g_var0[get_local_id(0)] = input[get_local_id(0)];
    }
}

// Kernel computes a result based on data from the input_stage kernel
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void adder_stage(int inc) {
    __attribute__ ((xcl_pipeline_workitems)) {
      int input_data, output_data;
      input_data = g_var0[get_local_id(0)];
      output_data = input_data + inc;
      g_var1[get_local_id(0)] = output_data;
    }
}

// Kernel writes the results computed by the adder_stage to
// a global memory buffer that is read by the host processor
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void output_stage(global int *output) {
    __attribute__ ((xcl_pipeline_workitems)) {
      output[get_local_id(0)] = g_var1[get_local_id(0)];
    }
}
```

In the code example above, the `input_stage` kernel reads the contents of global memory buffer input and writes them into global memory buffer `g_var0`. The contents of buffer `g_var0` are used in a computation by the `adder_stage` kernel and stored into buffer `g_var1`. The contents of `g_var1` are then read by the `output_stage` kernel and stored into the output global memory buffer. Although both `g_var0` and `g_var1` are declared as global memories, the host processor only needs to have access to the input and output buffers. Therefore, for this application to run correctly the host processor must only be involved in setting up the input and output buffers in DDR memory.

Since buffers `g_var0` and `g_var1` are only used for inter-kernel communication, the accesses to these buffers can be removed from the system-level memory bandwidth requirements. SDAccel automatically analyzes this kind of coding style to infer that both `g_var0` and `g_var1` can be implemented as on-chip memory buffers.

The only requirements in SDAccel are that all kernels with access to the on-chip global memory are executed in the FPGA logic and that the memory has at least 1,000 entries.

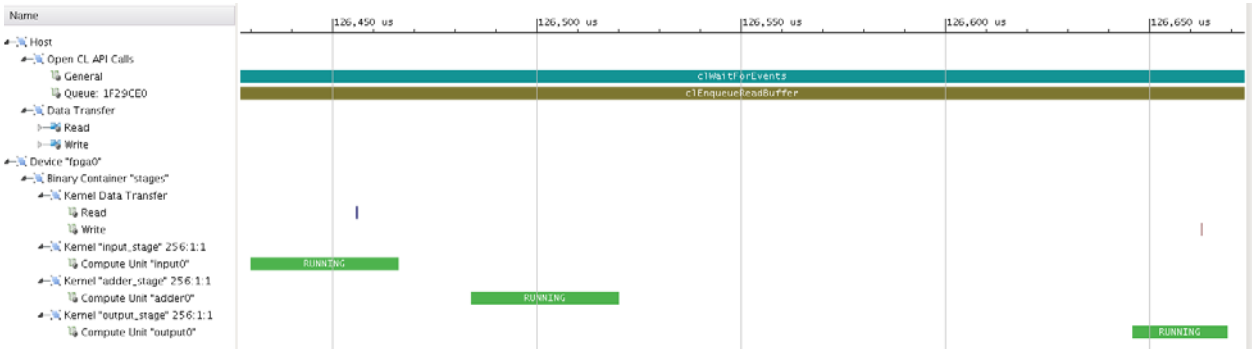

Figure B-1: **Timeline Trace of Global Memory Test Kernels**

Figure B-1 shows the timeline trace of the global memory test kernels. All three kernels are executed in the following order: `input_stage`, `adder_stage`, then `output_stage`. Note that the device profiling reports data transfers to off-chip global memory only, and hence, the accesses to the on-chip global memories are not shown.

From a performance standpoint, these on-chip global memories provide much lower latency than accessing data from the DDR or off-chip global memory. Therefore, these memories are excellent for applications requiring heavy data reuse amongst multiple kernels. You can create a scratchpad with on-chip global memories, re-use the data as often as is required, then write the results to the host-accessible DDR.

On-Chip Pipes

Another type of global memory that allows two kernels to communicate with each other is called a pipe. A pipe is essentially a FIFO and allows data streaming between kernels. It is specified with a maximum depth and stores data in a first-in, first-out order.

Consider the following sample code:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(512)));

// Stage 1
kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void input_stage(__global int *input) {
  write_pipe(p0, &input[get_local_id(0)]);
}

// Stage 2
kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void adder_stage(int inc) {
  int input_data, output_data;
  read_pipe(p0, &input_data);
  output_data = input_data + inc;
  write_pipe(p1, &output_data);
}
```

```
// Stage 3
kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
void output_stage(__global int *output) {
  read_pipe(p1, &output[get_local_id(0)]);
}
```

There are two pipes specified in the above code, `p0` and `p1`, which transfer data between the three kernels. Each has a depth of 512 values. Note that the functions `write_pipe()` and `read_pipe()` are used to write to and read from the pipes, respectively. The stage 2 kernel named `adder_stage` modifies the values in the pipe by adding inc to each value. The input to `input_stage` and the output of `output_stage` are in off-chip global memory.

Pipes are excellent for transferring data between kernels when order needs to be preserved and random access is not required. Note that pipes are implemented with AXI Stream, and their activity is currently not observable using the current profiling infrastructure.

Multiple Memory Ports per Kernel

The default behavior of SDAccel is to generate functionally correct FPGA compute units that consume the least amount of FPGA resources. This behavior produces the most area efficient implementation, but might not always achieve the desired performance requirements. The area efficient behavior of SDAccel is demonstrated by the following example:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd( __global int* a, __global int* b, __global int* c) {
  int i;
  for (i=0; I < 256; i++) {
    c[i] = a[i] + b[i];
  }
}
```

The code above is a vector adder which reads 256 values from `a` and `b`, adds them together, and writes the result to `c` (see Vectorization, page 60 for one method of improving the performance of this kernel). All 256 results are then stored into the appropriate locations in DDR dictated by `c`. Given the kernel code above, the hardware implementation generated by SDAccel has a single port to global memory through which all accesses to buffers `a`, `b`, and `c` are time-multiplexed. During execution, the single port forces sequential access to off-chip global memory to fetch individual elements from all three buffers. For designs with multiple data ports like the vector adder, this can lead to inefficiencies.

One way of increasing the memory bandwidth available to a kernel is to increase the number of physical connections to memory that are attached to a kernel. Proper implementation of this optimization requires knowledge of both the application and the target compute platform. Therefore, SDAccel requires user specification to increase the number of physical memory ports on a kernel. The SDAccel command to increase the number of physical memory ports available to the kernel is:

```
set_property max_memory_ports true [get_kernels <kernel name>]
```

The `max_memory_ports` property tells SDAccel to generate one physical memory interface for every global memory buffer declared as arguments to the kernel. This command is only valid for kernels that have been placed into binaries that will be executed in the FPGA logic. There is no effect on kernels executing in a processor.

Note that for HLS C/C++ files, you can also specify maximum memory ports using pragmas in the kernel source file:

```
void vadd(int * a, int * b, int * c) {
  #pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem0
  #pragma HLS INTERFACE m_axi port=b offset=slave bundle=gmem1
  #pragma HLS INTERFACE m_axi port=c offset=slave bundle=gmem2
  …
}
```

In the example code above, three different values are specified for the bundle: `gmem0`, `gmem1`, and `gmem2`. This creates three separate AXI interfaces for accessing the three pointers. To achieve the best performance for your kernel, it is highly recommended to do this.

Adjustable Memory Port Data Width

In addition to increasing the number of memory ports available to a kernel, you have the ability to change the bit width of the memory port. The benefit of modifying the bit width of the memory interface depends on the computation in the kernel. Consider the following kernel:

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd( __global int16* a, __global int16* b, __global int16* c) {
  int i;
  for (i=0; i< 256/16; i++){
    c[i] = a[i] + b[i];
  }
}
```

Where `a`, `b`, and `c` are of type `int16`. This is a 16 element vector data type created from the fundamental `c` data type int (see Vectorization, page 60 for more information on vector data types). SDAccel uses the fundamental data type when determining the default bit width of a memory interface. In the case above, the memory interfaces have a bit width of 32 bits. Therefore, the kernel requires 16 memory transactions to read enough data to complete the vector. You can override the default behavior of SDAccel with the following kernel property command:

```
set_property memory_port_data_width <bit width> [get_kernels <kernel name>]
```

The bit widths currently supported by SDAccel are 32, 64, 128, 256, and 512 bits. In cases where your defined bit width does not match the bit width declared in the kernel source

code, SDAccel handles all data width conversions between the physical interface and the data type in the kernel source code. This optimization is only supported for kernels mapped for execution in the FPGA logic.

Burst Transfers from Off-Chip Global Memory

Some kernels require large amounts of data and can potentially only start processing once a certain amount of data is available. Burst transfers enable a kernel to transfer as much data as is required and allow kernels to read data before it is needed by the computation. Large bursts also optimize the performance of the memory controller.

Consider the following vector multiplier example:

```
#define LENGTH 64

__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vmult(__global const int* a, __global const int* b, __global int* c) {
  for (int i=0; i < LENGTH; i++)
    c[i] = a[i] * b[i];
  }
}
```

In this example, every iteration of the loop performs two reads and one write. Figure A-1, page 61 shows the timeline trace for this vector adder with 256 loop iterations. The memory controller receives 3 requests per loop iteration (2 reads, 1 write), none of which are bursts. While the kernel creates a burst for the write transfer, this still results in under-utilization of the off-chip memory bandwidth.

To perform a burst read from off-chip memory, we can use local memory to buffer the incoming and outgoing data. Consider the following modified kernel that takes advantage of burst reads and writes:

```
#define LENGTH 64

__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vmult(__global const int* a, __global const int* b, __global int* c) {
  local int bufa[LENGTH];
  local int bufb[LENGTH];
  local int bufc[LENGTH];

  async_work_group_copy(bufa, a, LENGTH, 0);
  async_work_group_copy(bufb, b, LENGTH, 0);

  for (int i=0; i < LENGTH; i++) {
    bufc[i] = bufa[i] * bufb[i];
  }

  async_work_group_copy(c, bufc, LENGTH, 0);
}
```

The `async_work_group_copy` command performs a burst data transfer between two specified locations in memory. In the kernel above, the kernel first performs read accesses by copying data from `a` to `bufa` and from `b` to `bufb`. These are transfers between off-chip global memory and local memories. In a physical sense, this copies data from DDR to BlockRAMs on the FPGA. The amount of data is specified by `LENGTH`. At the end of processing, a similar burst transfer is performed. This time, however, it copies data from `bufc` to `c`. In a physical sense, this copies data from BlockRAMs to DDR.

Timeline trace can confirm this activity. Figure A-3, page 63 shows a similar timeline trace for this vector multiplier design. Trace essentially shows 3 data transfers: 2 read and 1 write. The time in between is the time it takes to perform the computation in the for loop.

Since the `for` loop now only reads and writes from local memories, the computation is now decoupled from the off-chip global memory accesses. Since local memories enable one cycle reads and writes, this optimization usually leads to significant performance improvement. The overhead is the time to transfer data between global and local memories. However, since these transfers utilize large burst sizes, this overhead is minimized.

In the vector multiplier code above, there is no need to pipeline the entire kernel using the `xcl_pipeline_workitems` attribute. The burst reads and writes are already pipelined automatically. Individually, the for loop can be pipelined by using the `xcl_pipeline_loop` attribute just before the for loop (see Loop Pipelining, page 65 for more information).

Burst transfers can also be performed in HLS C/C++ using the `memcpy` command.

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

For a glossary of technical terms used in Xilinx documentation, see the Xilinx Glossary.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

1. *Khronos OpenCL Working Group, "The OpenCL Specification", Version 2.0, October 17, 2014* (khronos.org/registry/cl/sdk/2.0/docs/man/xhtml)
2. *Ulrich Drepper, "What Every Programmer Should Know About Memory", November 21, 2007* (cs.bgu.ac.il/~os142/wiki.files/drepper-2007.pdf)
3. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
4. *Khronos Conformance* (khronos.org/conformance)
5. *Smith-Waterman Algorithm* (cs.stanford.edu/people/eroberts/courses/soco/projects/computers-and-the-hgp/smith_waterman.html)
6. *Median Filter* (wikipedia.org/wiki/Median_filter)
7. *ADM-PCIE-7V3 board* (xilinx.com/products/boards-and-kits/1-4i8a6z.html)

Please Read: Important Legal Notices