SDAccel Environment Optimization Guide

UG1207 (v2016.4) March 9, 2017

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/09/17	2016.4	Updated for SDx™ IDE 2016.4. Added good practices.*
11/30/16	2016.3	Initial documentation release for SDx IDE 2016.3, which includes both the SDSoC™ Environment and the SDAccel™ Environment. Due to this major change in tool architecture, this document has undergone substantial changes in structure and content since the previous release.

Table of Contents

XILINX
ALL PROGRAMMABLE™

Chapter 1

Introduction

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. 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: Application Optimization Flow

  The recommended flow for optimizing an application in the SDAccel Environment.

- Chapter 5: SDAccel Optimization Recommendations

  This chapter provides a list of recommendations on host code, data movement, and kernel development to optimize performance of an OpenCL application

- Chapter 6: Optimizing Host Code

  This chapter describes techniques and strategies on host code optimization.

- Chapter 7: Moving Data Efficiently between Kernel and Global Memory

  This chapter describes techniques and strategies on efficient data movement between the kernel and global memories.

- Chapter 8: Optimizing Kernels

  This chapter describes techniques and strategies on optimizing the logic inside a kenrel.

- Appendix A: On-Boarding Examples

  This appendix lists example categories, key concepts, and location of the examples.

- Appendix B: Additional Resources and Legal Notices

  This appendix lists additional resources for further reading. It also provides the legal copyright information for this guide.

Chapter 2

What is an FPGA?

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.

Understanding FPGA Architecture

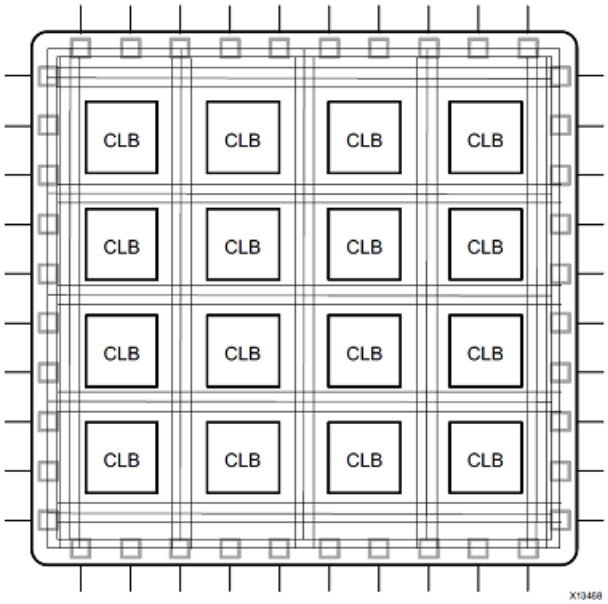
Xilinx FPGAs are heterogeneous compute platforms that include Block RAMs, DSP Slices, PCI Express support, and programmable fabric. They enable parallelism and pipelining of applications across the entire platform as all of these compute resources can be used simultaneously. SDAccel is the tool provided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

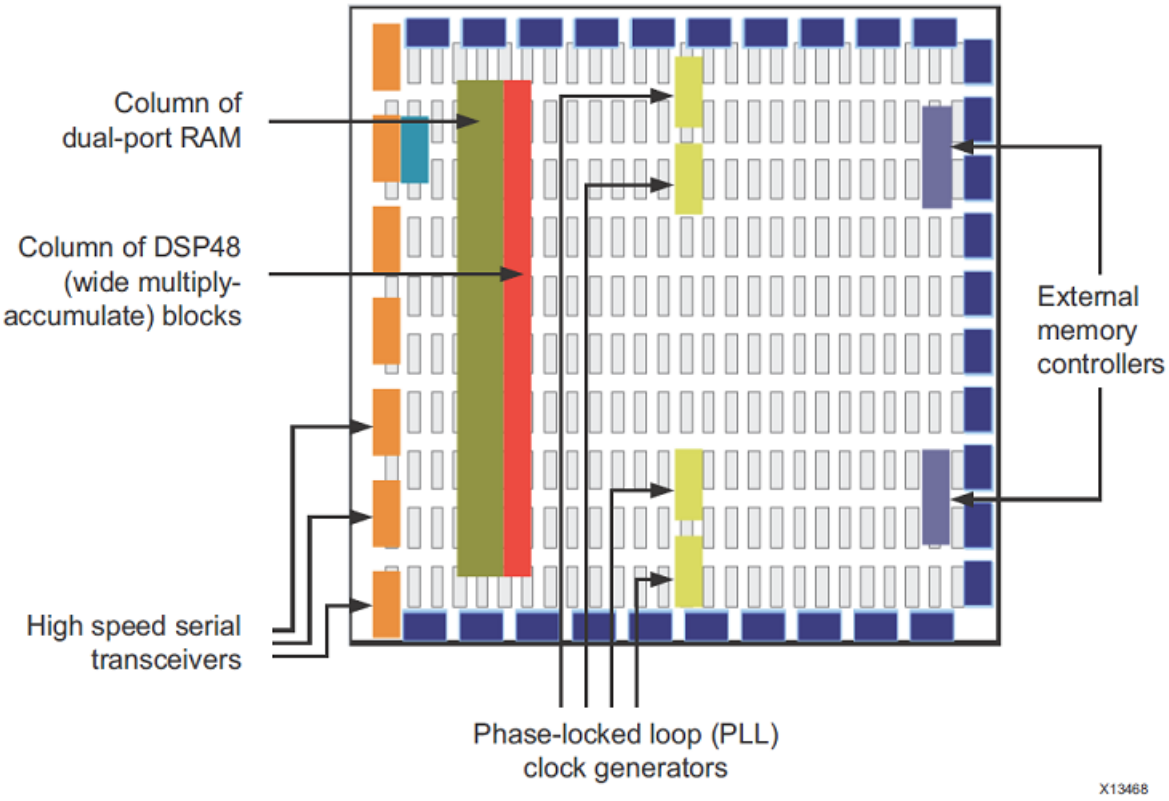
Figure 1: Basic FPGA Architecture



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: Contemporary FPGA Architecture

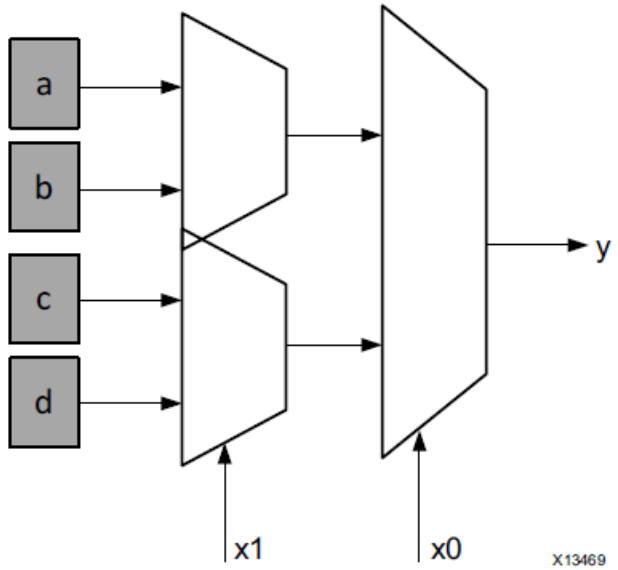


The previous figure shows the combination of these elements on a contemporary FPGA architecture. This provides the FPGA with the flexibility to implement any software algorithm running on a processor. Note that all of these elements across the entire FPGA device can be used concurrently, creating a unique 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.

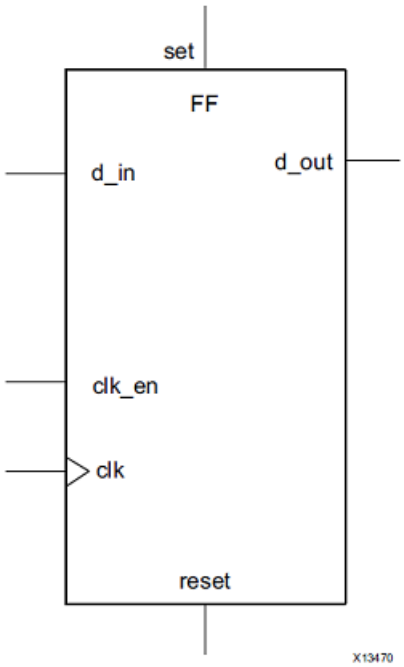
Figure 3: Functional Representation of a LUT as a Collection of Memory Cells



The hardware implementation of a LUT can be thought of as a collection of memory cells connected to a set of multiplexers. 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 4: Structure of a Flip-Flop

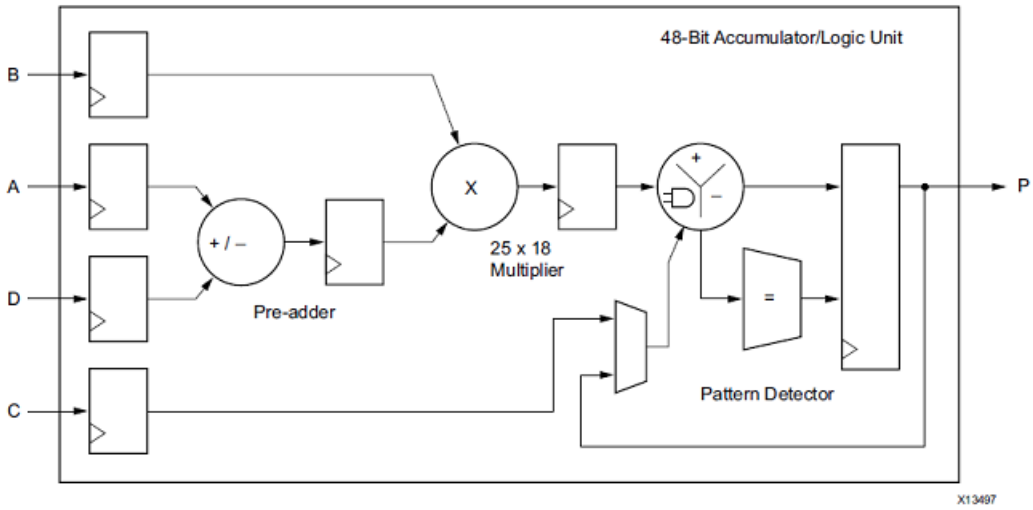


The basic structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. During normal operation, any value at the data input port is latched and passed to the output on every pulse of the clock. The purpose of the clock enable pin is to allow the flip-flop to hold 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

The most complex computational block available in a Xilinx FPGA is the DSP48 block shown below.

Figure 5: Structure of a DSP48 Block



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=Bx(A+D)+C
```

Or

```
P +=Bx(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

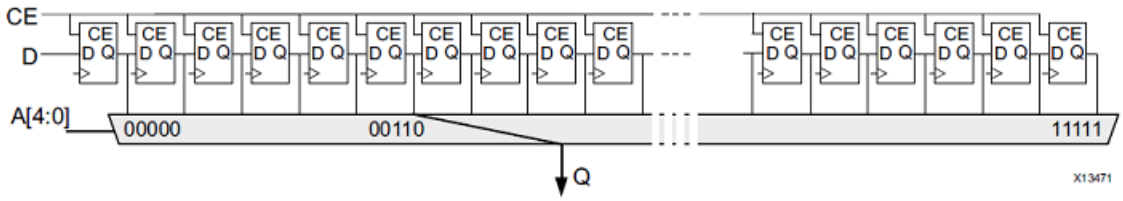
The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs, and shift registers.

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 6: Structure of an Addressable Shift Register



The shift register is a chain of registers connected to each other. 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.

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, 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.

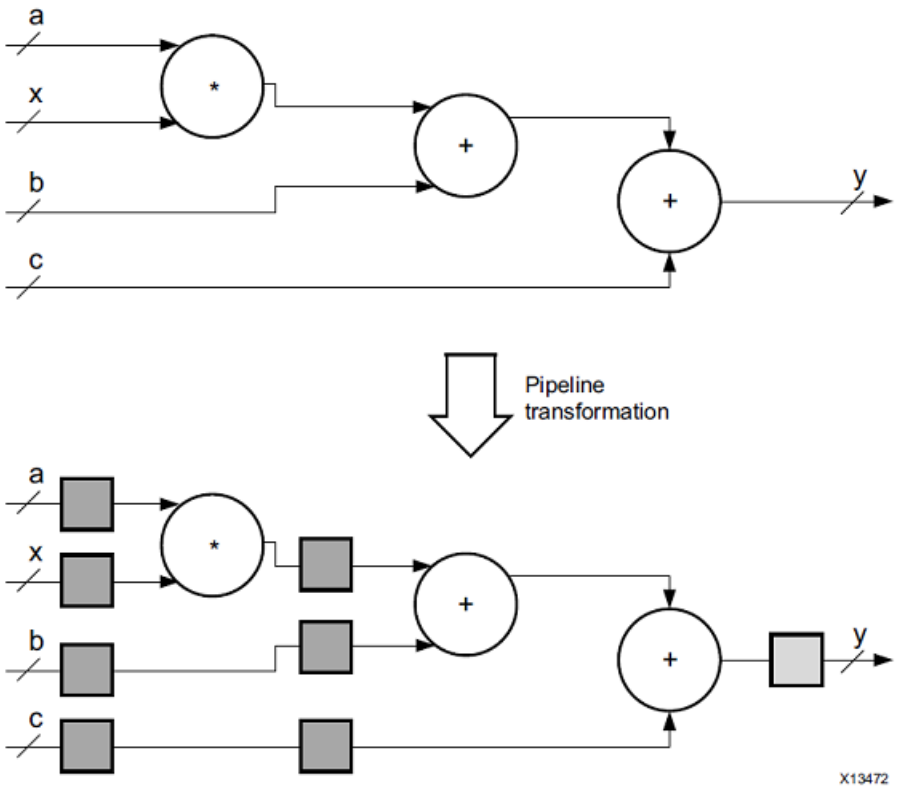
Pipelining

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 7: FPGA Implementation of a Compute Function

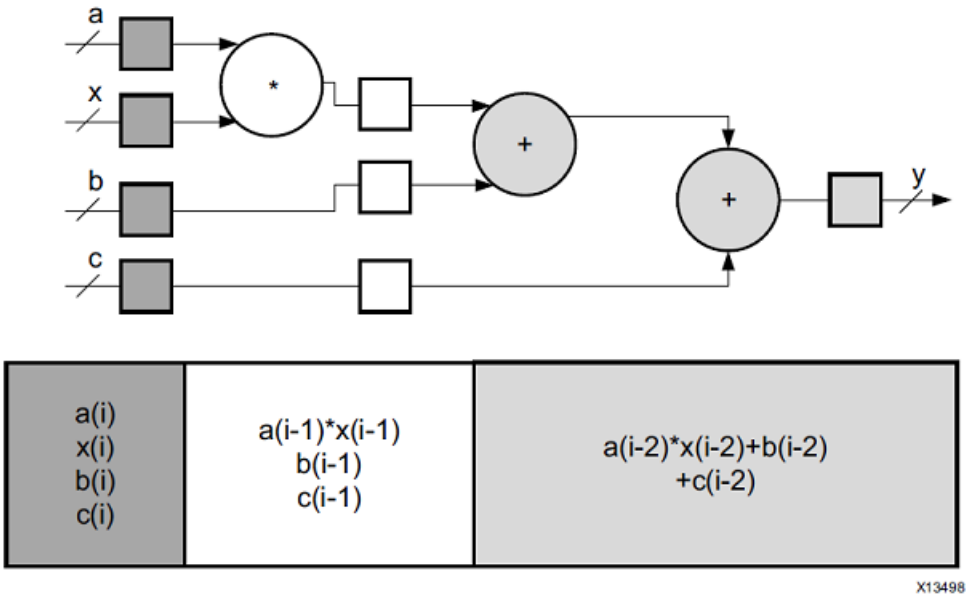


The preceding figure shows this compute structure and the effects of pipelining. It shows two implementations of the example function. The top implementation is the data path required to compute the result y without pipelining. This implementation behaves similarly to the corresponding software function 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 8: Pipelined Architecture



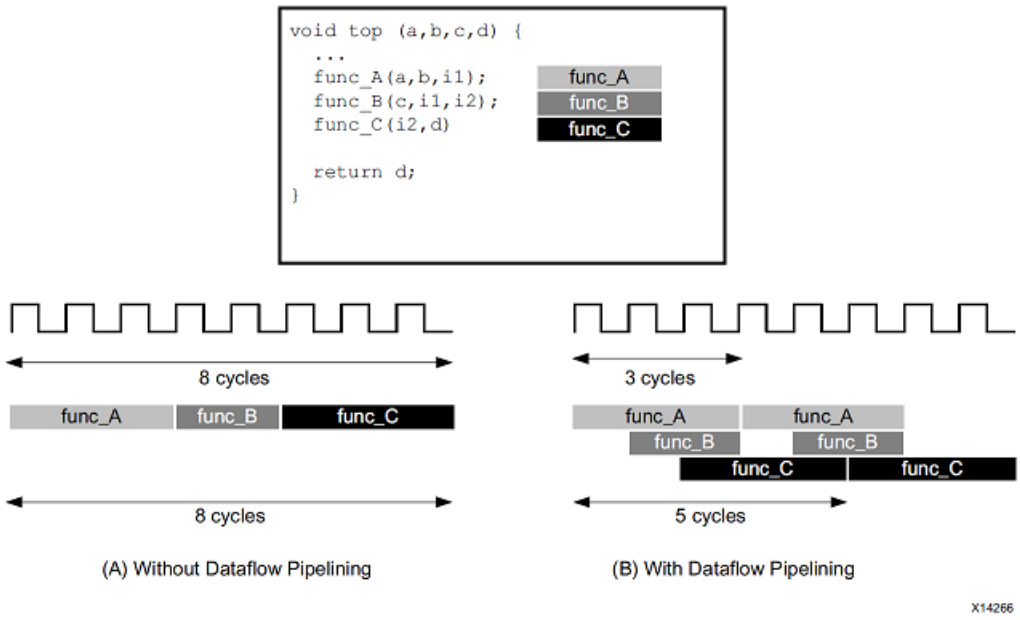
The preceding figure shows a pipelined architecture in which raw data (dark gray), semi-computed data (white), and final data (light gray) exist simultaneously, and each stage result is captured in its own set of registers. Thus, although the latency for such computation is in multiple cycles, a new 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.

The SDAccel™ compiler extracts this level of parallelism by evaluating the interactions between different functions and loops of a program based on their inputs and outputs. The SDAccel compiler can also extract this level of parallelism for functions and loops within a loop.

Figure 9: Dataflow Optimization



The preceding figure 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).

Chapter 3

What is OpenCL?

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. 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.

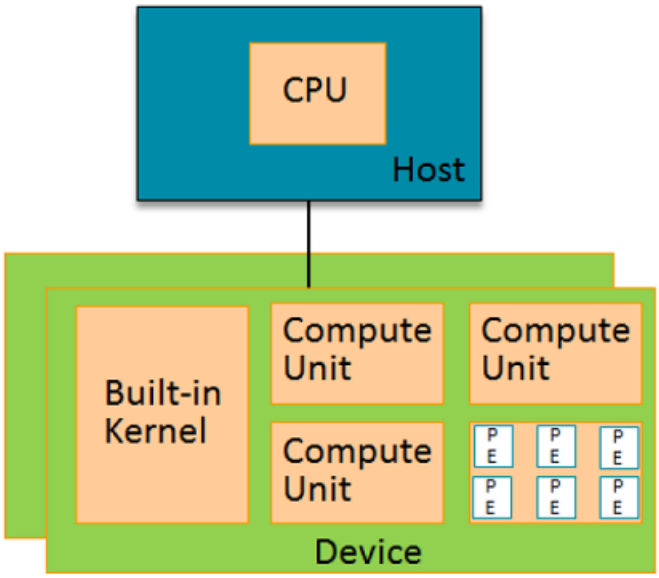
OpenCL provides a programming language and runtime API to support the development of close-to-the-metal software which is both efficient and portable. Additionally, OpenCL provides low-level hardware abstractions that allow OpenCL implementations to expose many details of underlying hardware. These low-level 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 10: OpenCL Platform Model



The preceding figure shows a conceptual view of the OpenCL platform model. 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.

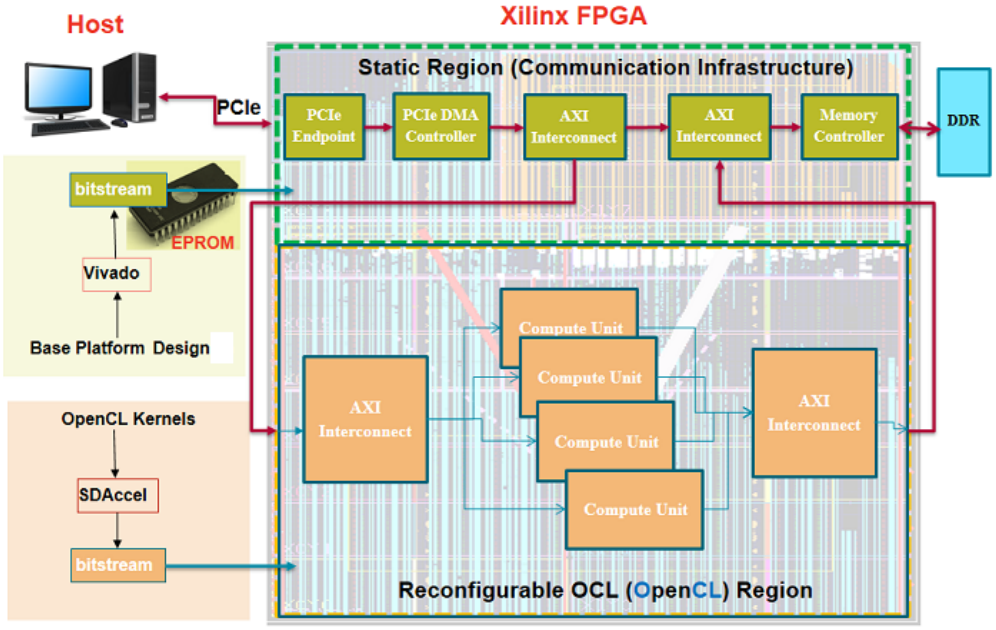
An OpenCL device for an FPGA is not limited by the constraints of a CPU/GPU device. By taking advantage of the fact that the FPGA starts off as a blank computational canvas, the user can decide the level of device customization that is appropriate to support a single application or a class of applications. 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 block diagram below shows an example SDAccel base platform design that includes these components.

Figure 11: Xilinx FPGA

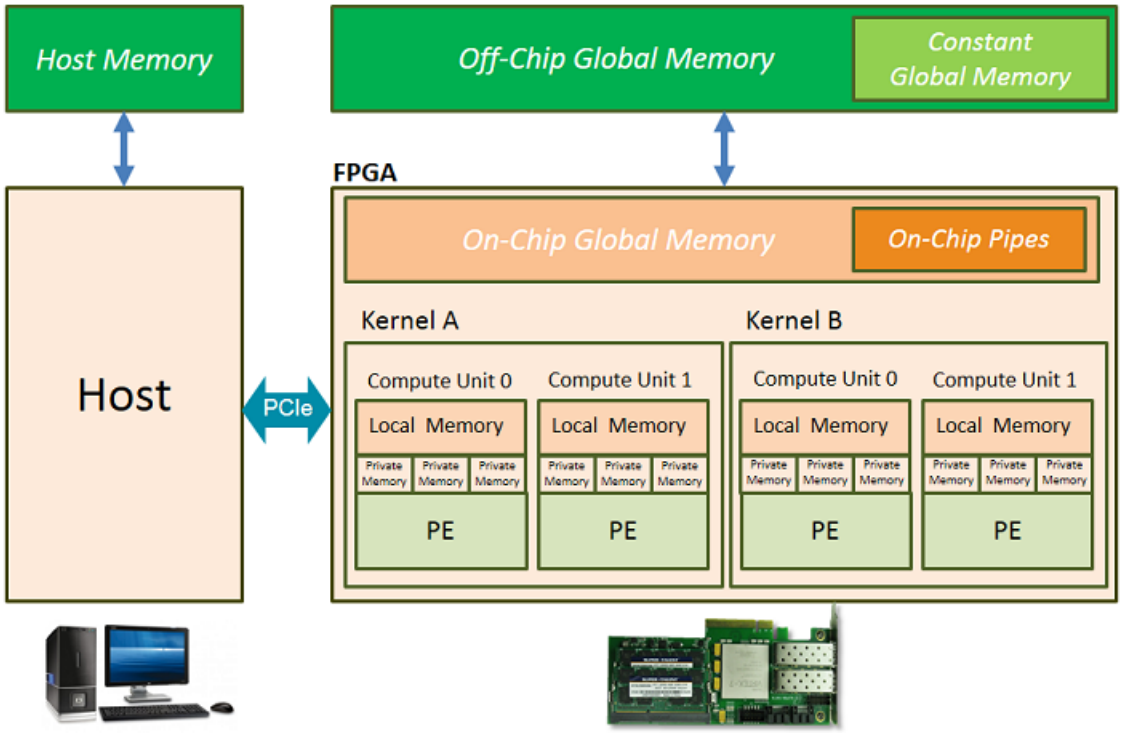


The creation of Xilinx FPGA based OpenCL devices requires FPGA design expertise and is beyond the scope of SDAccel itself. Devices for SDAccel are created using the Xilinx Vivado® design suite for FPGA designers. SDAccel provides pre-defined devices as well as allows users to augment the tool with third 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 12: OpenCL Memory Model



The OpenCL memory model defines the behavior and hierarchy of memory that can be used by OpenCL applications. 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 (and only) 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.

NOTE: *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:

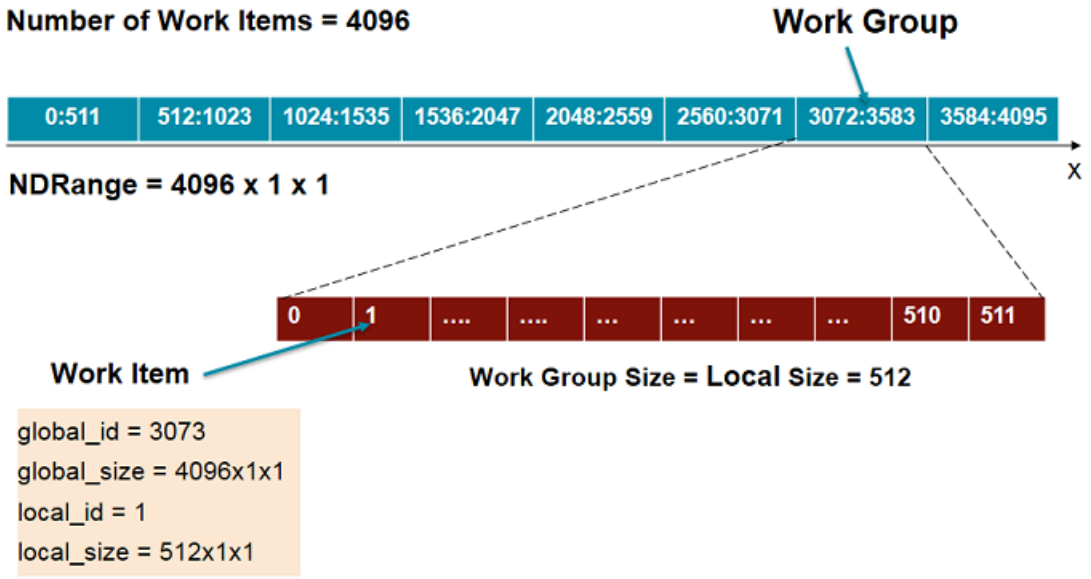
```
__kernel __attribute__ ((reqd_work_group_size(256, 1, 1)))
```

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 13: One-Dimensional Work Size



The preceding figure 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;
 or (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.

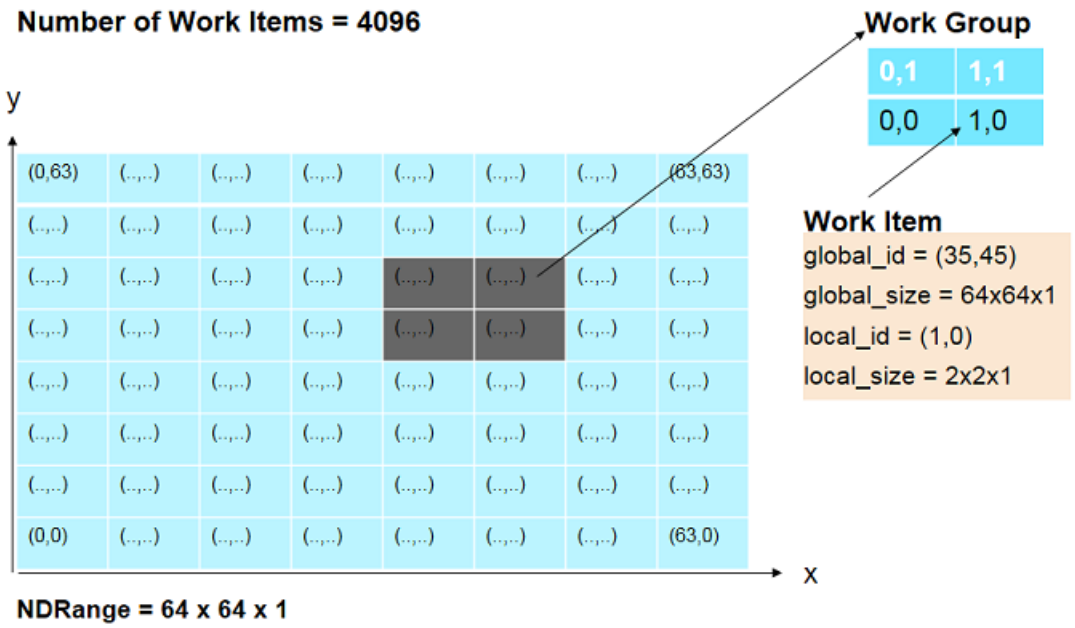
Two-Dimensional NDRange

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().

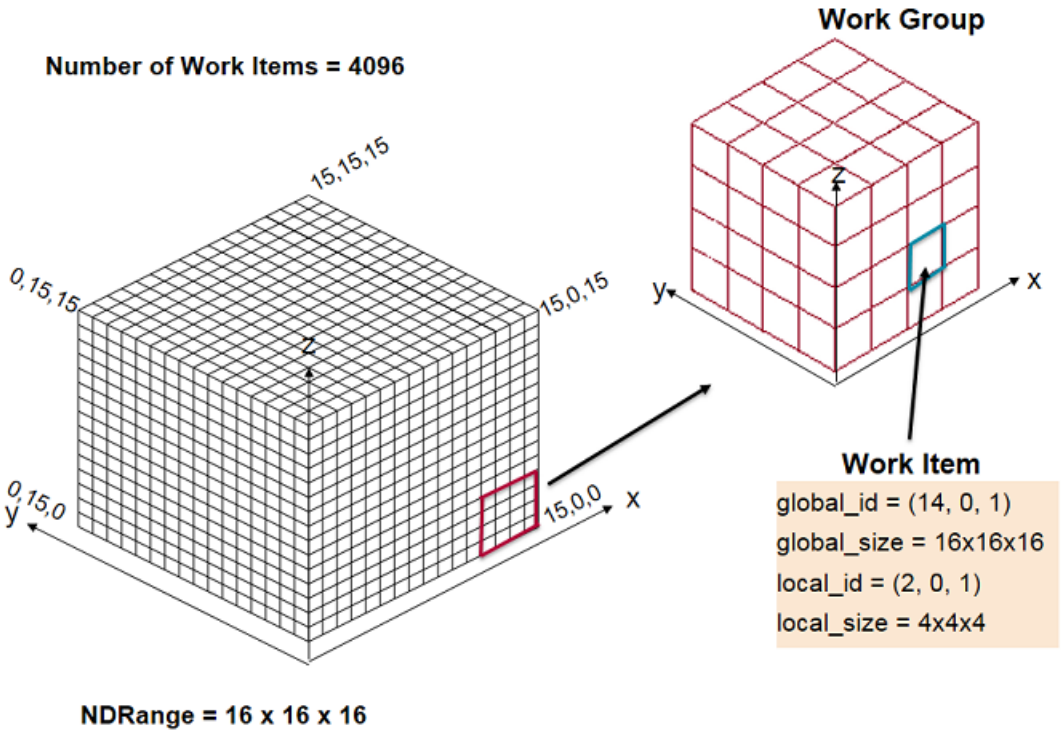
Figure 14: Two-Dimensional NDRange



The preceding figure 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 15: Three-Dimensional Work Size

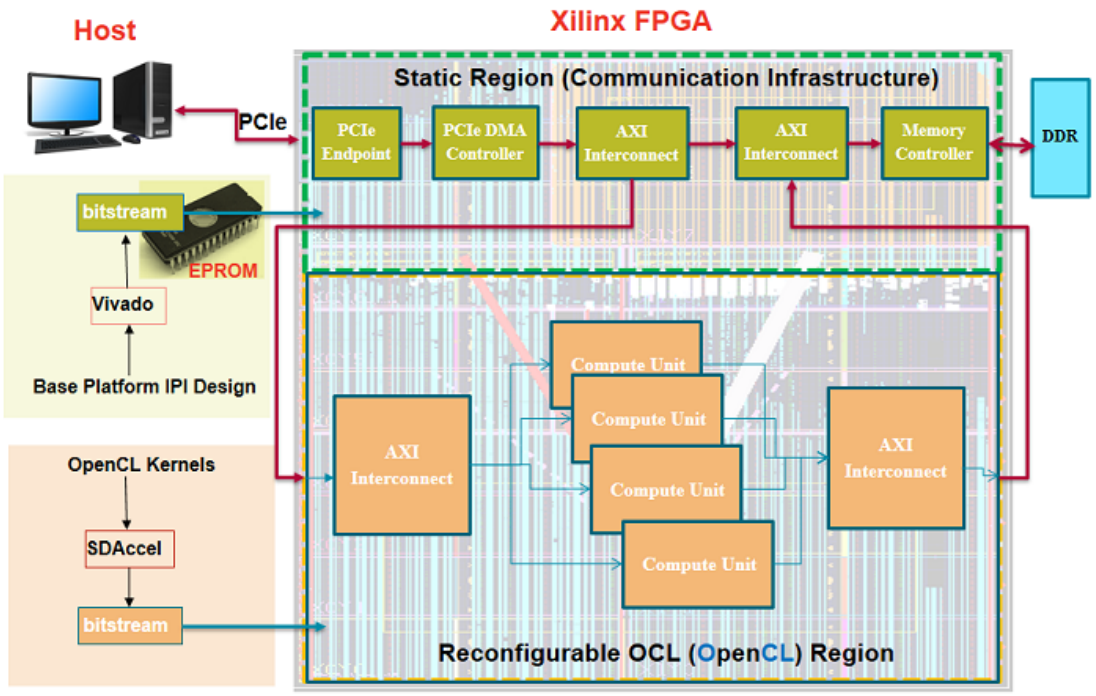


The concept of work size can be extended to a three-dimensional space. The preceding figure illustrates this work size as a three-dimensional cube of size 16x16x16. While the total number of work items is again 406, the work space is now defined across three different dimensions. This works well for 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 16: OpenCL Region Example



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);
}
```

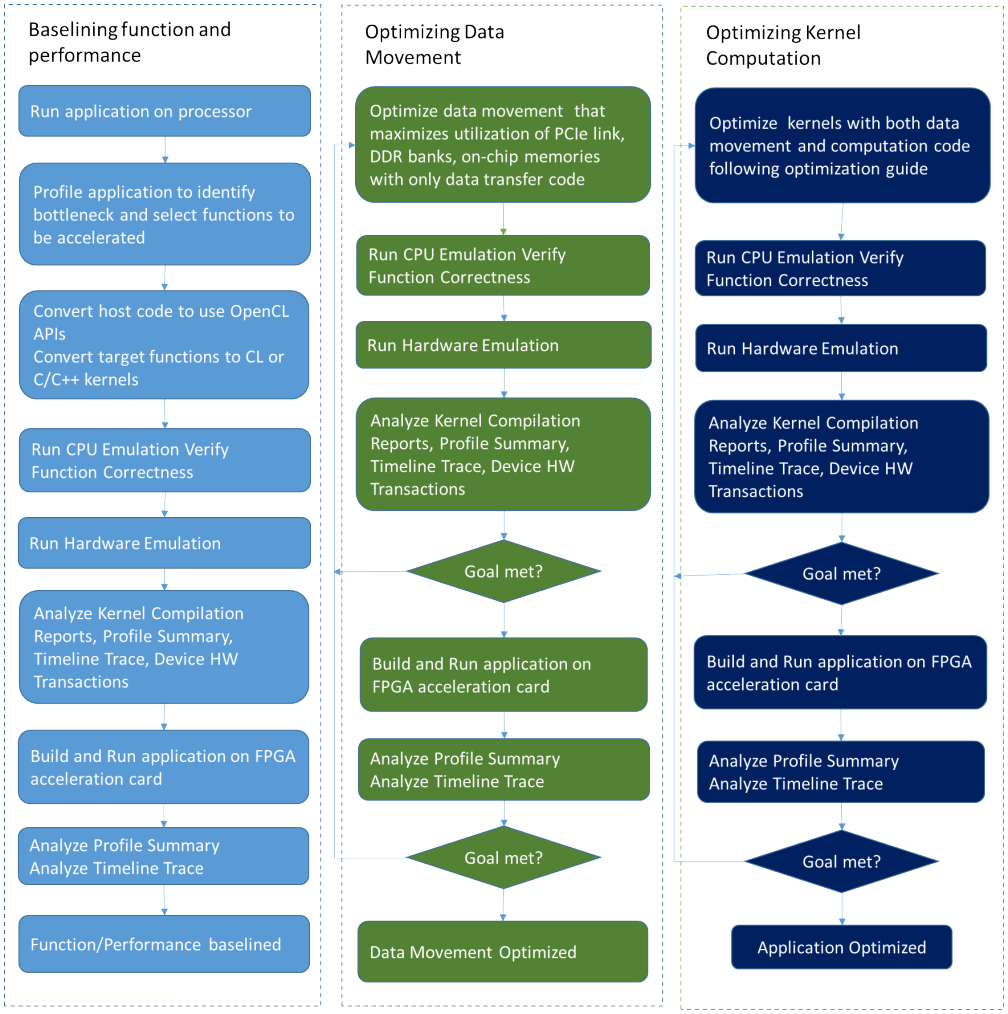
There are a number of FPGA resources leveraged by SDAccel to perform the functionality in this kernel. This includes the following:

- **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 Example and Loop Pipelining Example 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.

Application Optimization Flow

The SDAccel™ Environment is a complete software development environment for creating, compiling, and optimizing OpenCL™ applications to be accelerated on Xilinx® FPGAs. The figure below shows the recommended flow for optimizing an application in the SDAccel Environment.

Figure 17: SDAccel Recommended Flow



Baselining Functionalities and Performance

It is very important to understand the performance of your application before you start any optimization effort. This is achieved by baselining the application in terms of functionalities and performance.

The first step is to find out the bottlenecks of the current application running on your existing platform. The most effective way is to run the application with profiling tools like `valgrind`/`callgrind` and GNU `gprof`. The profiling data generated by these tools show the call graph with the number of calls to all functions and their execution time. The functions that consume the most execution time are good candidates to be offloaded and accelerated onto FPGAs.

Once the target functions are selected, convert them to OpenCL™ CL kernels or C/C++ kernels without any optimization. The application code calling these kernels will also need to be converted to use OpenCL APIs for data movement and task scheduling. Keep everything as simple as possible and minimize changes to the existing code in this step so you can quickly generate a working design on the FPGA and get the baselined performance and resource number.

Next, run CPU and hardware emulation to verify the function correctness and generate profiling data on the host code and the kernels. Analyze the kernel compilation reports, profile summary, timeline trace, and device hardware transactions to understand the baselined performance estimate such as timing, interval, and latency and resource utilization such as DSP, BRAM.

The last step in baselining is to build and run the application on an FPGA acceleration card. Analyze the reports from the system compilation and the profiling data from application execution to see the actual performance and resource utilization.

Save all the reports during the baselining so that you can be reference and compare during optimization exercise.

Optimizing Data Movement

In the OpenCL™ programming model, all data are transferred from the host memory to the global memory on the device first and then from the global memory to the kernel for computation. The computation results are written back from the kernel to the global memory and lastly from the global memory to the host memory. How data can be efficiently moved around in this programming model is a key factor for determining strategies for kernel computation optimization, so it is recommended to optimize the data movement in your application before taking on optimizing the computation.

During data movement optimization, it is important to isolate data transfer code from computation code because inefficiency in computation may cause stalls in data movement. Xilinx recommends that you modify the host code and kernels with data transfer code only for this optimization step. The goal is to maximize the system level data throughput by maximizing PCIe bandwidth utilization and DDR bandwidth utilization. It usually takes multiple iterations of running CPU emulation, hardware emulation, as well as execution on FPGAs to achieve the goal.

Optimizing Kernel Computation

One of the key benefits of FPGA is that you can create custom logic for your specific application. The goal of kernel computation optimization is to create processing logic that can consume all the data as soon as they arrive at kernel interfaces. The key metric during this step is the initiation interval (II). This is generally achieved by expanding the processing code to match the data path with techniques such as function pipelining, loop unrolling, array partitioning, dataflowing, etc. The SDAccel Environment produces various compilation reports and profiling data during hardware emulation and system run to assist your optimization effort. Please refer to "Application Report and Profiling" chapter in SDAccel Environment User Guide (UG1023) for details on the compilation and profiling report.

XILINX
ALL PROGRAMMABLE™

Chapter 5

SDAccel Optimization Recommendations

This chapter provides a list of recommendations on host code, data movement, and kernel development to optimize performance of an OpenCL™ application. Some recommendations will be explained in details in later chapters. This should by no means be considered an exhaustive list, but instead a starting point for ideas to consider or investigate further.

Using clEnqueueMigrateMemObjects to Transfer Data

The `clEnqueueMigrateMemObjects` command migrates memory objects explicitly performed ahead of the dependent commands. This allows the application to preemptively change the association of a memory object, through regular command queue scheduling, in order to prepare for another upcoming command. This also permits an application to overlap the placement of memory objects with other unrelated operations before these memory objects are needed potentially hiding transfer latencies.

Choosing Optimal Workgroup Size

Workgroup is a concept exclusive to OpenCL™ and should be exploited. It is recommended to define an explicit workgroup size when declaring a kernel. This gives the compiler more flexibility to optimize the size of the kernel. In addition define the workgroup size as large as the application allows, which will minimize the overhead associated with kernel startup.

Isolating Data Transfer and 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.

Maximizing Utilization of Global Memory Bandwidth

Using Burst Data Transfers

Transferring data in bursts hides the memory access latency as well as improves bandwidth utilization and efficiency of the memory controller. It is recommended to infer burst transfers from successive requests of data from consecutive address locations. Please refer to the Inferring Burst Transfer from/to Global Memory chapter for more details. Using of async_work_group_copy() (OpenCL™ C kernels) or memcpy() (C/C++ kernels) should be avoided because they cannot be pipelined with the computation in the current SDAccel™ Environment.

Using Full User Data Width of Memory Controller

The user data width between the kernel and the memory controller can be configured by the SDAccel™ compiler based on the data types of the kernel arguments. To maximize the data throughput, it is recommended to choose data types that will map to the full data width on the memory controller. The memory controller in all supported acceleration cards supports 512-bit user interface, which can be mapped to OpenCL™ vector data types such as `int16` or C/C++ arbitrary precision data type `ap_int<512>`.

Using Multiple DDR Banks

Acceleration cards supported in SDAccel™ Environment provide 1, 2 or 4 DDR banks and up to 80GB/s raw DDR bandwidth. For kernels moving large amount of data between the FPGA and the DDR, it's recommended to direct the SDAccel compiler and runtime library to use multiple DDR banks. Please refer to Utilizing Multiple DDR Banks chapter for more details.

Using On-chip Memories

Acceleration platforms supported in SDAccel™ environment can have as much as 10MB on-chip memories that can be used as on-chip global memories, pipes, local and private memories. Using these resources effectively can greatly improve the efficiency and performance of your applications.

Using Optimized Built-in Math Functions from HLS MATH Library

OpenCL™ Specification provides a wealth of math built-in functions. All math built-in functions with the `native_` prefix are mapped to one or more native device instructions and will typically have better performance compared to the corresponding functions (without the `native_` prefix). The accuracy and in some cases the input ranges of these functions is implementation-defined. In SDAccel™ environment these `native_` built-in functions use the equivalent functions in Vivado® HLS Math library, which are already optimized for Xilinx® FPGAs in terms of area and performance. It's recommended to use `native_` built-in functions if the accuracy meets the application requirement.

Exploring Fixed Point Arithmetic

Some applications use floating point computation only because they are optimized for other hardware architecture. As explained in "Deep Learning with INT8 Optimization on Xilinx Devices", using fixed point arithmetic for applications like deep learning can save the power efficiency and area significantly while keeping the same level of accuracy. It is recommended to explore fixed point arithmetic for your application before committing to using floating point operations

Avoiding Complex Structs or Classes for Kernel Arguments

Kernel arguments are mapped onto hardware interfaces between the host code and the FPGA. Complex structs or classes can lead to very complex hardware interfaces due to memory layout and data packing differences. This may introduce potential issues that are very difficult to debug in a complex system. It's recommended to use simple structs for kernel arguments that can be always packed to 32-bit boundary. Please refer to the "Custom Data Type Example" in "kernel_to_gmem" category on Xilinx On-boarding Example GitHub on the recommended way for using structs.

Chapter 6

Optimizing Host Code

Using clEnqueueMigrateMemObjects to Transfer Data

OpenCL™ provides a number of APIs for transferring data between the host and the device. Typically, data movement APIs such as `clEnqueueWriteBuffer` and `clEnqueueReadBuffer` implicitly migrate memory objects to the device after they are enqueued. They do not guarantee when the data are transferred. This makes it difficult for the host application to overlap the placements of the memory objects onto the device with the computation carried out by kernels.

OpenCL 1.2 introduced a new API, `clEnqueueMigrateMemObjects`, with which memory migration can be explicitly performed ahead of the dependent commands. This allows the application to preemptively change the association of a memory object, through regular command queue scheduling, in order to prepare for another upcoming command. This also permits an application to overlap the placement of memory objects with other unrelated operations before these memory objects are needed, potentially hiding transfer latencies. Once the event associated by `clEnqueueMigrateMemObjects` has been marked CL_COMPLETE, the memory objects specified in `mem_objects` have been successfully migrated to the device associated with `command_queue`.

The `clEnqueueMigrateMemObjects` API can also be used to direct the initial placement of a memory object after creation, possibly avoiding the initial overhead of instantiating the object on the first enqueued command to use it.

Another advantage of `clEnqueueMigrateMemObjects` is that it can migrate multiple memory objects in a single API call. This reduces the overhead of scheduling and calling functions for transferring data for more than one memory object.

Below is the code snippiet showing the usage of `clEnqueueMigrateMemObjects` from Vector Multiplication for XPR Device example in the host category from Xilinx On-boarding Example GitHub.

```
int err = clEnqueueMigrateMemObjects(
                        world.command_queue,
                        1,
                        &d_mul_c,
                        CL_MIGRATE_MEM_OBJECT_HOST,
                        0,
                        NULL,
                        NULL);
```

Overlapping Data Transfers with Kernel Computation

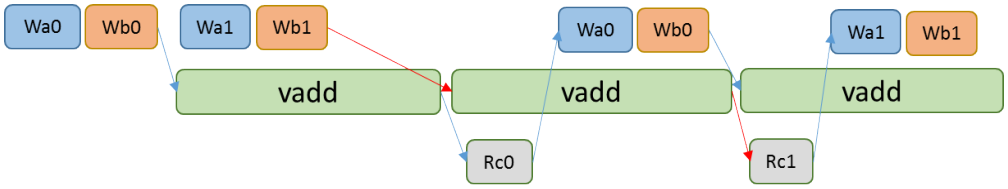
Applications like database analytics have much larger data set than the available memory on the acceleration device. They require the complete data to be transferred and processed in blocks. Techniques that overlap the data transfers with the computation are critical to achieve high performance for these applications.

Below is the vector add kernel from the OpenCL Overlap Data Transfers with Kernel Computation Example in the host category from Xilinx On-boarding Example GitHub.

```
kernel __attribute__((reqd_work_group_size(1, 1, 1)))
void vadd(global int* c,
          global const int* a,
          global const int* b,
          const int offset,
          const int elements)
{
    int end = offset + elements;
    vadd_loop: for (int x=offset; x<end; ++x) {
        c[x] = a[x] + b[x];
    }
}
```

There are four tasks to be performed in the host application for this example: write buffer a (Wa), write buffer b (Wb), execute vadd kernel, and read buffer c (Rc). The asynchronous nature of OpenCL data transfer and kernel execution APIs allows overlap of data transfers and kernel execution as illustrated in the figure below. In this example, double buffering is used for all buffers so that the compute unit can process one set of buffers while the host can operate on the other set of buffers. The OpenCL event object provides an easy way to set up complex operation dependencies and synchronize host threads and device operations. The arrows in the figure below show how event triggering can be set up to achieve optimal performance.

Figure 18: Event Triggering Set Up



The host code snippet below enqueues the four tasks in a loop. It also sets up event synchronization between different tasks to ensure that data dependencies are met for each task. The double buffering is set up by passing different memory objects values to `clEnqueueMigrateMemObjects` API. The event synchronization is achieved by having each API to wait for other event as well as trigger its own event when the API completes.

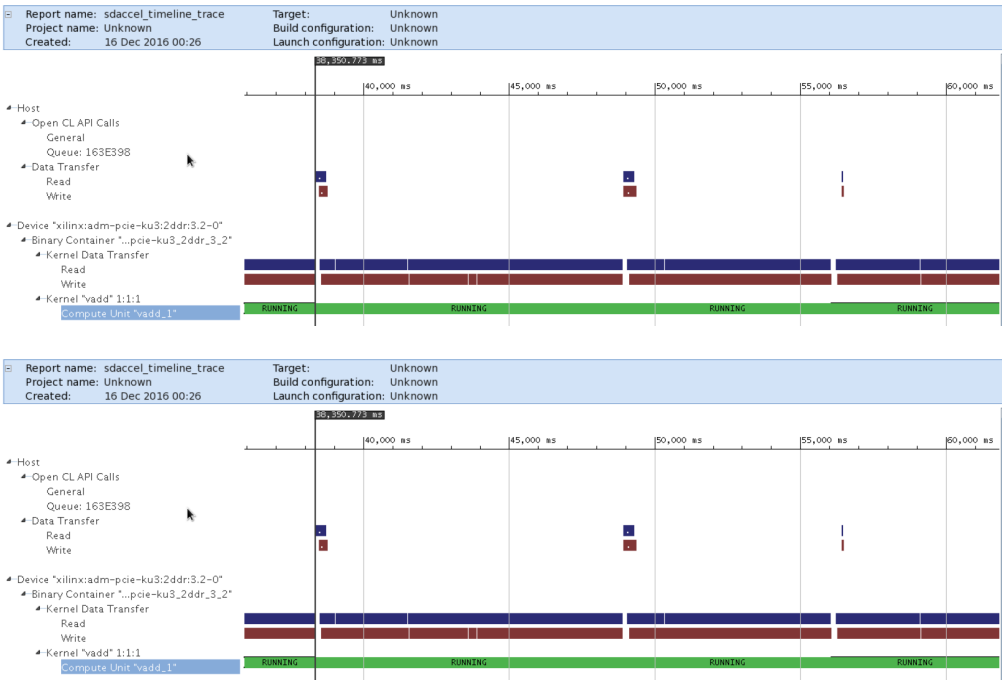
```
  for (size_t iteration_idx = 0; iteration_idx < num_iterations;
iteration_idx++) {
    int flag = iteration_idx % 2;

    if (iteration_idx >= 2) {
        clWaitForEvents(1, &map_events[flag]);
```

```
        OCL_CHECK(clReleaseMemObject(buffer_a[flag]));
        OCL_CHECK(clReleaseMemObject(buffer_b[flag]));
        OCL_CHECK(clReleaseMemObject(buffer_c[flag]));
        OCL_CHECK(clReleaseEvent(read_events[flag]));
        OCL_CHECK(clReleaseEvent(kernel_events[flag]));
    }

    buffer_a[flag] = clCreateBuffer(world.context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &A[iteration_idx * elements_per_iteration],
NULL);
    buffer_b[flag] = clCreateBuffer(world.context,
            CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &B[iteration_idx * elements_per_iteration],
NULL);
    buffer_c[flag] = clCreateBuffer(world.context,
            CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
           bytes_per_iteration, &device_result[iteration_idx *
elements_per_iteration], NULL);
    array<cl_event, 2> write_events;
    printf("Enqueueing Migrate Mem Object (Host to Device) calls\n");
    // These calls are asynchronous with respect to the main thread because
we
    // are passing the CL_FALSE as the third parameter. Because we are
passing
    // the events from the previous kernel call into the wait list, it will
wait
    // for the previous operations to complete before continuing
    OCL_CHECK(clEnqueueMigrateMemObjects(
        world.command_queue, 1, &buffer_a[iteration_idx % 2],
        0 /* flags, 0 means from host */,
        0, NULL,
        &write_events[0]));
    set_callback(write_events[0], "ooo_queue");

    OCL_CHECK(clEnqueueMigrateMemObjects(
        world.command_queue, 1, &buffer_b[iteration_idx % 2],
        0 /* flags, 0 means from host */,
        0, NULL,
        &write_events[1]));
    set_callback(write_events[1], "ooo_queue");

    xcl_set_kernel_arg(kernel, 0, sizeof(cl_mem), &buffer_c[iteration_idx %
2]);
    xcl_set_kernel_arg(kernel, 1, sizeof(cl_mem), &buffer_a[iteration_idx %
2]);
    xcl_set_kernel_arg(kernel, 2, sizeof(cl_mem), &buffer_b[iteration_idx %
2]);
    xcl_set_kernel_arg(kernel, 3, sizeof(int), &elements_per_iteration);

    printf("Enqueueing NDRange kernel.\n");
    // This event needs to wait for the write buffer operations to complete
    // before executing. We are sending the write_events into its wait list
to
    // ensure that the order of operations is correct.
    OCL_CHECK(clEnqueueNDRangeKernel(world.command_queue, kernel, 1,
```

```
nullptr,
                                        &global, &local, 2 ,
write_events.data(),
                                        &kernel_events[flag]));
    set_callback(kernel_events[flag], "ooo_queue");

    printf("Enqueueing Migrate Mem Object (Device to Host) calls\n");
    // This operation only needs to wait for the kernel call. This call will
    // potentially overlap the next kernel call as well as the next read
    // operations
    OCL_CHECK( clEnqueueMigrateMemObjects(world.command_queue, 1,
&buffer_c[iteration_idx % 2],
                CL_MIGRATE_MEM_OBJECT_HOST, 1, &kernel_events[flag],
&read_events[flag]));

    set_callback(read_events[flag], "ooo_queue");
    clEnqueueMapBuffer(world.command_queue, buffer_c[flag], CL_FALSE,
CL_MAP_READ, 0,
            bytes_per_iteration, 1, &read_events[flag], &map_events[flag],
0);
    set_callback(map_events[flag], "ooo_queue");

    OCL_CHECK(clReleaseEvent(write_events[0]));
    OCL_CHECK(clReleaseEvent(write_events[1]));
  }
```

The Application Timeline view below clearly shows that the data transfer time is completely hidden, while the compute unit vadd_1 is running constantly.

Figure 19: Data Transfer Time Hidden in Application Timeline View



Reducing Overhead of Kernel Enqueing

The OpenCL™ execution model supports data parallel and task parallel programming models. Kernels are usually enqueued by the OpenCL Runtime multiple times and then scheduled to be executed on the device. You must send the command to start the kernel in one of two ways:

- Implicitly, using clEnqueueNDRange API for the data parallel case
- Explicitly, using clEnqueueTask for the task parallel case

The dispatching process is executed on the host processor and the actual commands and kernel arguments need to be sent to the FPGA via PCIe link. In the current OpenCL Runtime Library of SDAccel Environment, the overhead of dispatching the command and arguments to the FPGA is between 30us and 60us, depending the number of arguments on the kernel. You can reduce the impact of this overhead by minimizing the number of times the kernel needs to be executed.

For the data parallel case, Xilinx recommends that you carefully choose the global and local work sizes for your host code and kernel so that the global work size is a small multiple of the local work size. Ideally, the global work size is the same as the local work size as shown in the code snippet below:

```
size_t global = 1;
size_t local = 1;
clEnqueueNDRangeKernel(world.command_queue, kernel, 1, nullptr,
                       &global, &local, 2, write_events.data(),
                       &kernel_events[0]));
```

For the task parallel case, Xilinx recommends that you minimize the call to clEnqueueTask. Ideally, you should finish all the work load in a single call to clEnqueueTask.

Using Multiple Compute Units

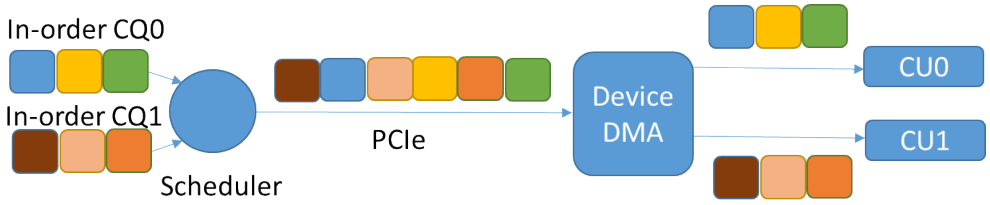
Depending on available resources on the target device, multiple compute units of the same kernel or different kernels can be created and run in parallel to improve the system processing time and throughput.

An application can use multiple compute units in the target device by createing multiple in-order command queues or a single out-of-order command queue.

Multiple In-Order Command Queues

The following figure shows an example with two in-order command queues, CQ0 and CQ1. The scheduler dispatches commands from each queue in order, but commands from CQ0 and CQ1 can be pulled out by the scheduler in any order. You must manage synchronization between CQ0 and CQ1 if required.

Figure 20: Example with Two In-Order Command Queues



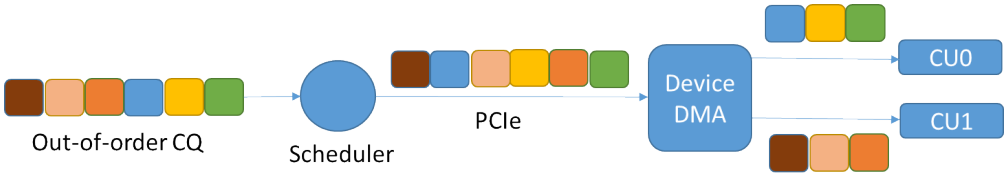
Below is the code snippet from the Concurrent Kernel Execution Example in host category from Xilinx On-boarding Example GitHub that sets up multiple in-order command queues and enqueues commands into each queue:

```
cl_command_queue ordered_queue1 = clCreateCommandQueue(
      world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err)

cl_command_queue ordered_queue2 = clCreateCommandQueue(
world.context, world.device_id, CL_QUEUE_PROFILING_ENABLE, &err);

clEnqueueNDRangeKernel(ordered_queue1, kernel_mscale, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[0]));

clEnqueueNDRangeKernel(ordered_queue1, kernel_madd, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[1]);

clEnqueueNDRangeKernel(ordered_queue2, kernel_mmult, 1, offset,
                       global, local, 0, nullptr,
                       &kernel_events[2]);
```

Single Out-of-Order Command Queue

The figure below shows an example with a single out-of-order command queue CQ. The scheduler can dispatch commands from CQ in any order. You must set up event dependencies and synchronizations explicitly if required.

Figure 21: Example with Single Out-of-Order Command Queue



Below is the code snippet from the Concurrent Kernel Execution Example from Xilinx On-boarding Example GitHub that sets up single out-of-order command queue and enqueues commands into the queue:

```
cl_command_queue ooo_queue = clCreateCommandQueue(
      world.context, world.device_id,
      CL_QUEUE_PROFILING_ENABLE | CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE,
&err);

clEnqueueNDRangeKernel(ooo_queue, kernel_mscale, 1, offset, global,
                       local, 0, nullptr, &ooo_events[0]);

clEnqueueNDRangeKernel(ooo_queue, kernel_madd, 1, offset, global,
                      local, 1,
                      &ooo_events[0], // Event from previous call
                      &ooo_events[1]);

clEnqueueNDRangeKernel(ooo_queue, kernel_mmult, 1, offset, global,
                       local, 0,
                       nullptr, // Does not depend on previous call
                       &ooo_events[2])
```

The Application Timeline view below shows that the compute unit `mmult_1` is running in parallel with the compute units `mscale_1` and `madd_1`, using both multiple in-order queues and single out-of-order queue methods.

Figure 22: Application Timeline View Showing mult_1 Running with mscale_1 and madd_1

Chapter 7

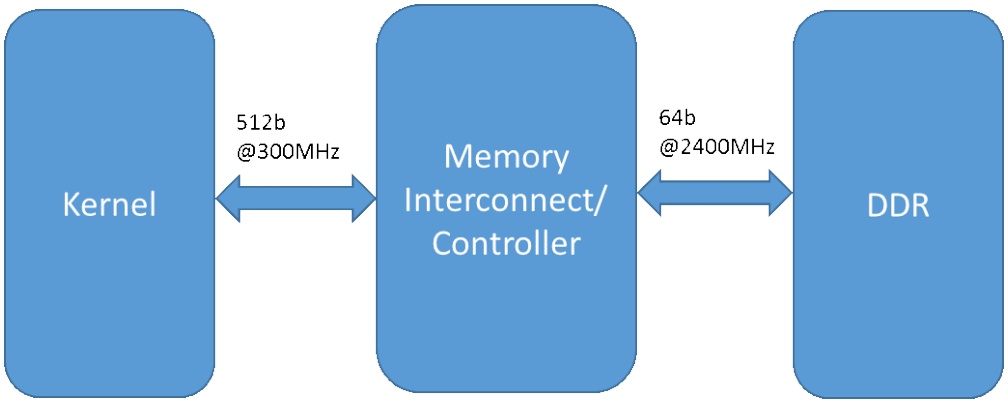
Moving Data Efficiently between Kernel and Global Memory

Efficient data movement between the kernel running in the FPGA and external global memory is critical to the performance of acceleration applications. There is an inherent latency overhead to read and write data from external DDR SDRAM. A well-designed kernel minimizes this latency impact and maximizes the usage of the available data bandwidth provided by the acceleration platform.

Choosing Optimal Data Width

SDAccel™ Environment includes a variety of FPGA acceleration cards with different DDR memory configurations. The figure below shows the data path between a kernel and one of 4 DDR channels on the XIL-ACCEL-RD-KU115 card. Each DDR channel provides 20GB/s raw DDR bandwidth with 80GB/s total for the entire card.

Figure 23: Data Path between a Kernel and XIL-ACCEL-RD-KU115 Card



The width of the data path between the kernel and the memory interconnect/controller can configured by the SDAccel compiler as 32, 64, 128, 256, and 512 bits depending on the kernel argument types. For applications that require maximum data bandwidth between the kernel and DDR memory it is recommended that global pointers are defined explicitly as 512-bit data types.

Defining 512-bit Interface in OpenCL C Kernel

OpenCL C specification defines vector data types that can have up to 16 elements of the same basic C data type. Kernel arguments defined as `int16`, `uint16`, and `float16` are automatically packed by the SDAccel compiler as 512-bit interfaces during synthesis.

Below is the code snippet from the Wide Memory Read/Write OpenCL C Example in kernel_to_gmem category on Xilinx On-boarding Example GitHub. It defines all global pointers in the kernel argument list as `uint16`. A 512-bit AXI4 memory mapped interface will be generated for these global pointers after compilation.

```
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd(
        const __global uint16 *in1, // Read-Only Vector 1
        const __global uint16 *in2, // Read-Only Vector 2
        __global uint16 *out,       // Output Result
        int size                   // Size in integer
        )
{

}
```

Defining 512-bit Interface in C/C++ Kernel

HLS provides C++ arbitrary precision template classes `ap_int<>` and `ap_uint<>` that can be used to define integers with any number of bits. Below is the code snippet from the Wide Memory Read/Write C++ Example in kernel_to_gmem category on Xilinx On-boarding Example GitHub. It defines all global pointer in the kernel argument list as `ap_uint<512>`. A 512-bit AXI4 memory mapped interface will be generated for these global pointers after compilation.

```
void vadd(
        const ap_uint<512> *in1, // Read-Only Vector 1
        const ap_uint<512> *in2, // Read-Only Vector 2
        ap_uint<512> *out,       // Output Result
        int size               // Size in integer
        )
{

}
```

Inferring Burst Transfer from/to Global Memory

The most common global memories used on Xilinx® acceleration cards are DDR3 and DDR4 SDRAMs. They are most efficient when operated in burst mode. In addition there are overheads associated with switching between DDR read and write. Xilinx recommendes that you transfer large amount of data in a single burst to achieve the best efficiency of the memory controller and keep the compute unit inside the FPGA device busy all the time.

The memory layout of data objects is a key factor to consider for improving the data transfer efficiency. Considering a 4x4 matrix "a" example, conceptually it is a two dimensional array as shown in the matrix logical layout in the Figure below. In C/C++ programming, arrays are physically stored in row-major order that all data within a row are stored in consecutive locations followed by the data within the next row as shown in the matrix physical layout below. The implication is that if your algorithm reads the data column-wise, the burst transfer will not happen as it reads from discrete location each time. This can generally be optimized by either transposing your data in the host code or caching multiple columns of data in the kernel.

Figure 24: Memory Layout Matrix

Matrix Logical Layout

row/col	0	1	2	3
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]
3	a[3][0]	a[3][1]	a[3][2]	a[3][3]

Matrix Physical Layout

addr	value
0	a[0][0]
1	a[0][1]
2	a[0][2]
3	a[0][3]
4	a[1][0]
5	a[1][1]
6	a[1][2]
7	a[1][3]
8	a[2][0]
9	a[2][1]
10	a[2][2]
11	a[2][3]
12	a[3][0]
13	a[3][1]
14	a[3][2]
15	a[3][3]

This chapter discusses the most common data access patterns and presents guidelines and examples on how to infer burst transfers for these data access patterns as well as how to analyze the profiling data to confirm that.

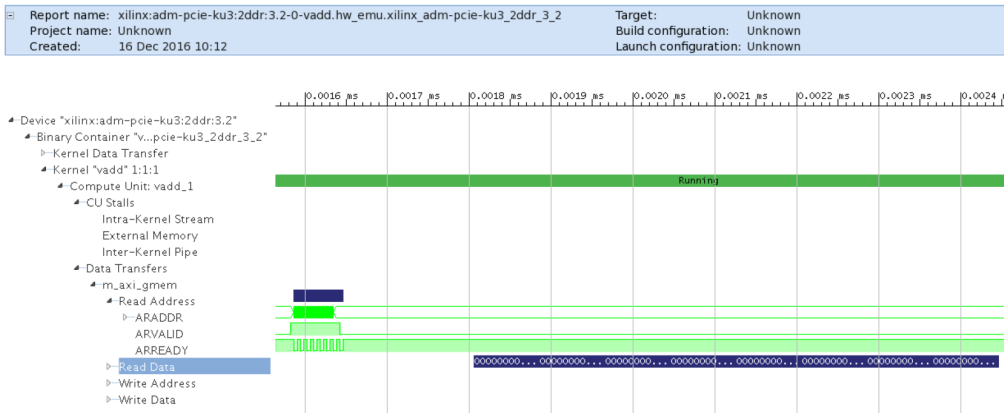
One Dimensional Read and Write

Below is the code snippet from the Wide Memory Read/Write Example on Xilinx On-boarding Example GitHub that shows the recommended coding style for automatically inferring burst read for one dimensional vectors. A local memory `v1_local` is used for buffering the data from a single burst. The entire input vector is read in multiple bursts. The choice of `LOCAL_MEM_SIZE` depends on the specific application and available on-chip memory on the target FPGA.

```
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vadd(
        const __global uint16 *in1, // Read-Only Vector 1
        const __global uint16 *in2, // Read-Only Vector 2
        __global uint16 *out,       // Output Result
        int size                   // Size in integer
        )
{
    local uint16 v1_local[LOCAL_MEM_SIZE];    // Local memory to store
vector1
    int size_in16 = (size-1) / VECTOR_SIZE + 1;
    ...
    for(int i = 0; i < size_in16;  i += LOCAL_MEM_SIZE)
    {
    ...
        int chunk_size = LOCAL_MEM_SIZE;
        //boundary checks
        if ((i + LOCAL_MEM_SIZE) > size_in16)
            chunk_size = size_in16 - i;

        v1_rd: __attribute__((xcl_pipeline_loop))
        for (int j = 0 ; j <  chunk_size; j++){
            v1_local[j] = in1 [i + j];
        }
    ...
    }
}
```

The Device Hardware Transaction View below shows that multiple read bursts are sent at the kernel start and all read data come back continuously after the memory read latency.

Figure 25: Device Hardware Transaction View Showing Multiple Read Bursts



Two Dimensional Read and Write

Below is the code snippet from the Row of 2D Array Using AXI4-Master Interface in "kernel_to_gmem" category on Xilinx On-boarding Example GitHub that shows the recommended coding style for inferring burst read and write of two dimensional arrays. Note that the data from the global memory `inx` is read consecutively into the local memory `buffer_in` in the inner loop. Also note that the inner loop is pipelined using the `xcl_pipeline_loop` attribute.

```
#define NUM_ROWS    64
#define WORD_PER_ROW 64

void read_data(__global int* inx, int* buffer_in)
{
    for(int i = 0; i < NUM_ROWS; ++i) {
        __attribute__((xcl_pipeline_loop))
        for (int j = 0; j < WORD_PER_ROW; ++j) {
            buffer_in[WORD_PER_ROW*i+j] = inx[WORD_PER_ROW*i+j];
        }
    }
}
```

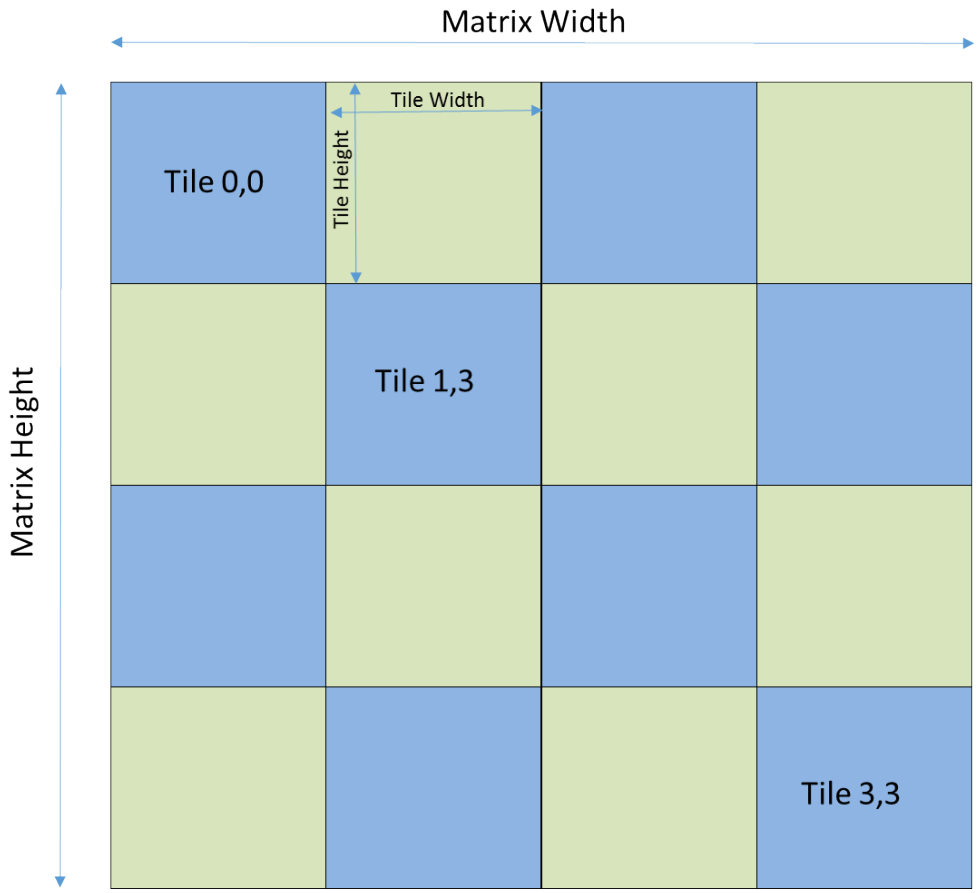
The Device Hardware Transaction view below shows that the burst read and write requests are sent out from the kernel as indicated by the **Read Address** and **Write Address** channels. It also shows that the read and write data are transferred continuously between the kernel and the global memory after the requests have been serviced.

Figure 26: Device Hardware Transaction View Shows Burst Read and Write Requests

Tiled Read and Write

Some applications like very large matrix multiplications process large amount of data that cannot all be brought into the local memory at once for fast access. It is more practical and efficient to divide the data into tiles and read/compute/write one tile of data at a time. The figure below shows a square matrix divided into 16 tiles with equal size.

Figure 27: Matrix Divided into 16 Tiles



Below is the code snippet from the Window of 2D Array Example Using AXI4-Master Interface Example in "kernel_to_gmem" category on Xilinx On-boarding Example GitHub that shows the recommended coding style for inferring burst read and write of tiles of data from a two dimensional array (e.g. matrix). Note that the data from the global memory `inx` is read consecutively into the local memory `tile` in the innermost loop. Also note that the innermost loop is pipelined using the `xcl_pipeline_loop` attribute.

```
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void read_data(__global int *inx) {
    int tile[TILE_HEIGHT][TILE_WIDTH];
    rd_loop_i: for(int i = 0; i < TILE_PER_COLUMN; ++i) {
        rd_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {
            rd_buf_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
                __attribute__((xcl_pipeline_loop))
                rd_buf_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
                    // should burst TILE_WIDTH in WORD beat
                    tile[m][n] =
inx[TILE_HEIGHT*TILE_PER_ROW*TILE_WIDTH*i+TILE_PER_ROW*TILE_WIDTH*m+TILE_WIDTH*j+n];
                }
            }
            rd_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
                __attribute__((xcl_pipeline_loop))
                rd_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
                    write_pipe_block(inFifo, &tile[m][n]);
                }
            }
        }
    }
```

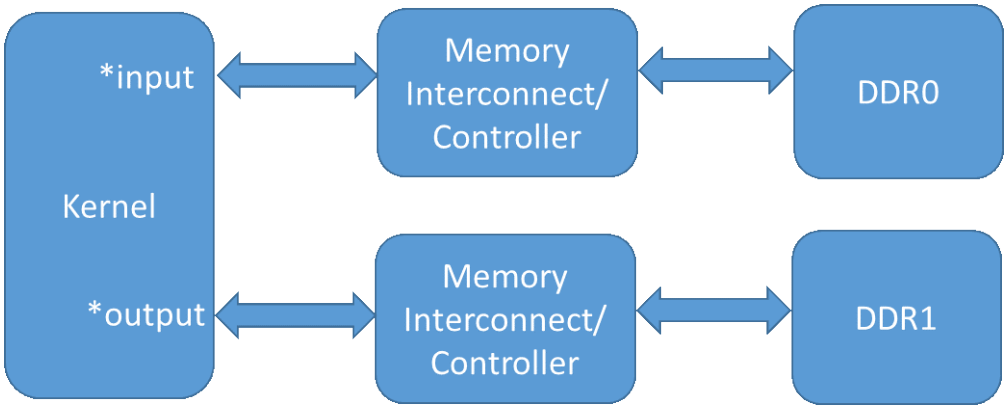
Using Multiple DDR Banks

For applications requiring very high bandwidth to the global memory, devices with multiple DDR banks can be targeted so that kernels can access all available memory banks simultaneously. For example, SDAccel includes a platform, xilinx:xil-accel-rd-ku115:4ddr-xpr:3.3, that supports four DDR banks.

In order to take advantage of multiple DDR banks, users need to assign CL memory buffers to different banks in the host code as well as configure XCL binary file to match the bank assignment in xocc command line.

The block diagram shows the "Global Memory Two Banks Example" in "kernel_to_gmem" category on Xilinx On-boarding Example GitHub that connects the input pointer to DDR bank 0 and output pointer to DDR bank 1.

Figure 28: Global Memory Two Banks Example



Assigning DDR Bank in Host Code

Bank assignment in host code is supported by Xilinx® vendor extension. The following code snippet shows the header file required as well as assigning input and output buffers to DDR bank 0 and bank 1 respectively:

```
#include <CL/cl_ext.h>
…
int main(int argc, char** argv)
{
…
    cl_mem_ext_ptr_t inExt, outExt;  // Declaring two extensions for both
buffers
    inExt.flags  = XCL_MEM_DDR_BANK0; // Specify Bank0 Memory for input
memory
    outExt.flags = XCL_MEM_DDR_BANK1; // Specify Bank1 Memory for output
Memory
    inExt.obj = 0   ; outExt.obj = 0; // Setting Obj and Param to Zero
    inExt.param = 0 ; outExt.param = 0;

    int err;
    //Allocate Buffer in Bank0 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_inImage = clCreateBuffer(world.context, CL_MEM_READ_ONLY
| CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &inExt, &err);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
    //Allocate Buffer in Bank1 of Global Memory for Input Image using
Xilinx Extension
    cl_mem buffer_outImage = clCreateBuffer(world.context,
CL_MEM_WRITE_ONLY | CL_MEM_EXT_PTR_XILINX,
            image_size_bytes, &outExt, NULL);
    if (err != CL_SUCCESS){
        std::cout << "Error: Failed to allocate device Memory" << std::endl;
        return EXIT_FAILURE;
    }
…
}
```

`cl_mem_ext_ptr_t` is a `struct` as defined below:

```
typedef struct{
    unsigned flags;
    void *obj;
    void *param;
  } cl_mem_ext_ptr_t;
```

- Valid values for `flags` are XCL_MEM_DDR_BANK0, XCL_MEM_DDR_BANK1, XCL_MEM_DDR_BANK2, and XCL_MEM_DDR_BANK3.
- `obj` is the pointer to the associated host memory allocated for the CL memory buffer only if `CL_MEM_USE_HOST_PTR` flag is passed to `clCreateBuffer` API, otherwise set it to NULL.
- `param` is reserved for future use. Always assign it to 0 or NULL.

Assigning DDR Bank in Kernel Code

A kernel needs to have multiple AXI4 MM interfaces before it can be connected to multiple DDR banks.

- For OpenCL C kernel, **--max_memory_ports** option is required to generate one AXI MM interface for each global pointer on the kernel argument. The AXI MM interface name is based on the order of the global pointers on the argument list. In this example, the first global pointer `input` is assigned an AXI MM name `M_AXI_GMEM0` and the second global pointer `output` is assigned a name `M_AXI_GMEM1`. The AXI MM interface names are needed in the **map_connect** option discussed below.
- For C/C++ kernel, multiple AXI4 interfaces are generated by specifying different "bundle" names in the interface pragma for different global pointers. Below is code snippet from the Global Memory Two Banks C Example that assigns the `input` pointer to the bundle `gmem0` and the `output` pointer to the bundle `gmem1`. Note that the bundle name can be any valid C string and the AXI4 interface name generated will be `M_AXI_<bundle_name>`. For this example, the input pointer will have AXI4 interface name as `M_AXI_gmem0` and the output pointer will have `M_AXI_gmem1`.

  ```
  #pragma HLS INTERFACE m_axi port=input  offset=slave bundle=gmem0
  #pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem1
  ```

Once multiple AXI4 interfaces are created, they are connected to DDR banks using xocc **map_connect** option. The **map_connect** option value is in the format of `add.kernel.<kernel_instance_name>.<AXI_IF_name>.core.OCL_REGION_0.<DDR_bank_name>` where fields in <> need to be replaced with the actual names from the design. Valid DDR bank names for the **map_connect** option are M00_AXI, M01_AXI, M02_AXI, and M03_AXI for DDR banks 0, 1, 2, 3 respectively.

Below is the command line option that connects the input pointer (`M_AXI_GMEM0`) to DDR bank 0 and the output pointer (`M_AXI_GMEM1`) to DDR bank 1 in the Global Memory Two Banks Example.

```
xocc --max_memory_ports apply_watermark
--xp
misc:map_connect=add.kernel.apply_watermark_1.M_AXI_GMEM0.core.OCL_REGION_0.M00_AXI
--xp
misc:map_connect=add.kernel.apply_watermark_1.M_AXI_GMEM1.core.OCL_REGION_0.M01_AXI
```

Figure 29: Device Hardware Transaction View Transactions on DDR Bank

Chapter 8

Optimizing Kernels

Unrolling Loops

Loop unrolling is the first optimization technique available in the SDAccel compiler. The purpose of the loop unroll optimization is to expose concurrency to the compiler. This is an official attribute in the OpenCL 2.0 specification.

For example, starting with the code:

```
#define LENGTH 64
__kernel void __attribute__ ((reqd_work_group_size(1, 1, 1)))
krnl_vmult(
     __global int* a,
     __global int* b,
     __global int* c)
{
    local int bufa[LENGTH];
    local int bufb[LENGTH];
    local int bufc[LENGTH];

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

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

    for(int i = 0; i < LENGTH; i++) {
        c[i] = bufc[i];
    }
    return;
}
```

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. 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.

Below is the latency and area estimate after running xocc with **--report estimate** option with the baseline design without any loop unroll attribute. Note that in this particular example, the actual computation only uses one 32-bit integer multiplier (4 DSPs) and takes 64 cycles to complete.

```
Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -----------  --------------  ---------  --------  ----------
krnl_vmult_1  krnl_vmult   krnl_vmult   536             535        535       535

Area Information
Compute Unit  Kernel Name  Module Name  FF    LUT   DSP  BRAM
------------  -----------  -----------  ----  ----  ---  ----
krnl_vmult_1  krnl_vmult   krnl_vmult   1428  1681  4    5
--------------------------------------------------------------------------------
```

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++) {
    bufc[i] = bufa[i] * bufb[i];
}
```

The code above tells SDAccel to unroll the loop by a factor of two. Conceptually the compiler transforms the loop above to the code below:

```
for(int i = 0; i < LENGTH; i+=2) {
    bufc[i] = bufa[i] * bufb[i];
    bufc[i+1] = bufa[i+1] * bufb[i+1];
}
```

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. Below is the latency and area estimate for the unroll factor of 2. Note that 2x DSPs are used for the compute unit and the computation is reduced by 32 clock cycles.

```
Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -----------  --------------  ---------  --------  ----------
krnl_vmult_1  krnl_vmult   krnl_vmult   504             503        503       503

Area Information
Compute Unit  Kernel Name  Module Name  FF    LUT   DSP  BRAM
------------  -----------  -----------  ----  ----  ---  ----
krnl_vmult_1  krnl_vmult   krnl_vmult   1433  1694  8    8
--------------------------------------------------------------------------------
```

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++) {
    bufc[i] = bufa[i] * bufb[i];
}
```

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

Pipelining Loops

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**. By default, the SDAccel™ compiler automatically applies this attribute on the innermost loop with trip count more than 64 or its parent loop when its trip count is less than or equal 64.

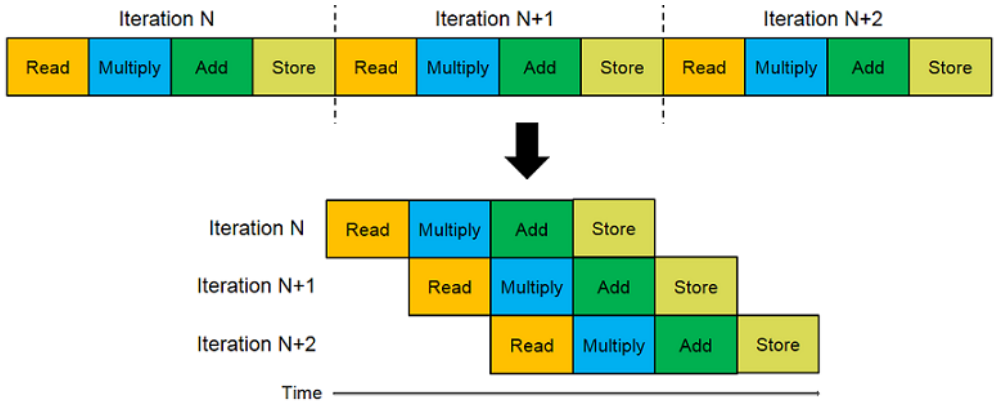
To understand the effect of loop pipelining on performance, consider the following code example:

```
__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;

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

During the kernel compilation, the compiler automatically adds **xcl_pipelie_loop** attribute to `LOOP_I`, which conceptually looks like the code snippet below:

```
__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))
    LOOP_I: 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. The figure below 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 Application Timeline 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.



Pipelining Workitems

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**. The following kernel is an example where work pipelining can be applied:

```
__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];
}
```

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.

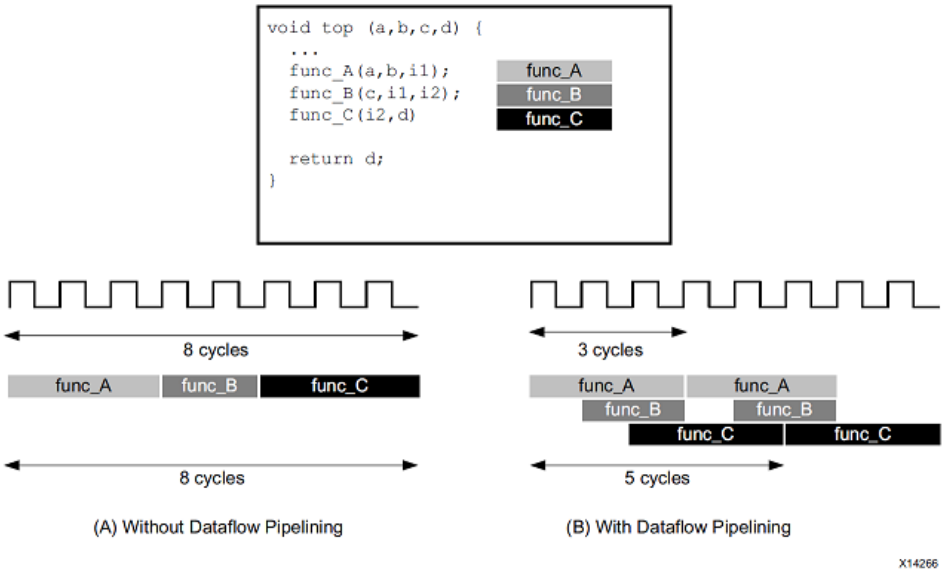
The work item pipeline attribute can be added to the code as follows:

```
__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 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);

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

Enabling Concurrent Processing with DATAFLOW

DATAFLOW expresses parallelism at a coarse-grain level. It allows the SDAccel compiler to schedule multiple sequential loops and functions concurrently to achieve higher throughput and lower latency. The figure below 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`. With DATAFLOW enabled, the SDAccel compiler can 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. With DATAFLOW optimization, the interval is reduced to only three clock cycles. The tasks shown in this example are functions, but dataflow optimization can be applied to any combinations of functions and loops.

Figure 31: DATAFLOW Optimization



Enabling DATAFLOW on OpenCL C Kernels

DATAFLOW is supported in OpenCL™ C kernels with a Xilinx vendor extension to the OpenCL specification. The attribute `xcl_dataflow` can be added to a kernel to enable concurrent scheduling of sub-functions and loops within the kernel function.

Below is a functon dataflow example in dataflow category on Xilinx On-boarding Example GitHub. The top level kernel `adder` consists of three sub-fucntions with `xcl_dataflow` attribute applied to the kernel definition.

```
__kernel
__attribute__ ((reqd_work_group_size(1, 1, 1)))
__attribute__ ((xcl_dataflow))
void adder(__global int *in, __global int *out, int inc, int size)
{
    int buffer_in[BUFFER_SIZE];
    int buffer_out[BUFFER_SIZE];

    read_input(in,buffer_in,size);
    compute_add(buffer_in,buffer_out,inc,size);
    write_result(out,buffer_out,size);
}
```

Currently there are some limitations on `xcl_dataflow` usage in the current version of the SDAccel compiler:

- The `xcl_dataflow` attribute can only be applied to kernel function defintion to enable function level DATAFLOW. Loop level DATAFLOW is not supported.
- Kernel function must have `reqd_work_group_size(1, 1, 1)` attribute. OpenCL workitem built-in functions such as `get_global_size()`, `get_local_id` can not be used in the kernel with the `xcl_dataflow` attribute.
- SDAccel™ compiler uses FIFOs for data channels between processes by default when DATAFLOW is enabled. The default FIFO depth is the same as the array size. The `--xp "param:compiler.xclDataflowFifoDepth=depth_value"` option can be passed to xocc command line to change the default FIFO depth.

Enabling DATAFLOW in C/C++ Kernels

Below is a loop DATAFLOW example in dataflow category on Xilinx On-boarding Example GitHub. The top level function `adder` consists of three loops with `dataflow` pragma. The three loops are automatically pipelined by the tool for the maximum throughput from the individual loop.

```
void adder(unsigned int *in, unsigned int *out, int inc, int size)
{
    hls::stream<unsigned int> inStream;
    hls::stream<unsigned int> outStream;
#pragma HLS STREAM variable=inStream  depth=32
#pragma HLS STREAM variable=outStream depth=32

#pragma HLS dataflow

    mem_rd: for (int i = 0 ; i < size ; i++){
#pragma HLS LOOP_TRIPCOUNT min=4096 max=4096
        inStream << in[i];
    }

    execute: for (int j = 0 ; j < size ; j++){
#pragma HLS LOOP_TRIPCOUNT min=4096 max=4096
        outStream << (inStream.read() + inc);
    }
    mem_wr: for (int k = 0 ; k < size ; k++) {
#pragma HLS LOOP_TRIPCOUNT min=4096 max=4096
        out[k] = outStream.read();
    }
}
```

Below is the latency estimate when the kernel is compiled with the dataflow pragma removed:

```
Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name  Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -----------  --------------  ---------  --------  ----------
adder_1       adder        adder        12309           12308      12308     12308
```

Below is the latency estimate when the kernel is compiled with the dataflow pragma:

```
Latency Information (clock cycles)
Compute Unit  Kernel Name  Module Name          Start Interval  Best Case  Avg Case  Worst Case
------------  -----------  -------------------  --------------  ---------  --------  ----------
adder_1       adder        Loop_mem_rd_proc235  4105            4105       4105      4105
adder_1       adder        Loop_execute_proc    4098            4098       4098      4098
adder_1       adder        Loop_mem_wr_proc     4104            4104       4104      4104
adder_1       adder        adder                4106            4112       4112      4112
```

As shown in the latency estimate report, the SDAccel generates a separate process for each loop with ~4000 clock cycle latency each. The scheduler is able to schedule three processes concurrently, so the latency of the top level module `adder` is also ~4000 clock cycles. This reduces the overall latency of the kernel top level to 1/3 of the case without DATAFLOW enabled.

Reducing Kernel to Kernel Communication Latency with On-Chip Global Memories

When a global memory buffer used for inter-kernel communication does not need to be visible to the host processor, the SDAccel environment enables you to move the buffer out of DDR based memory and into the FPGA logic. This optimization is called on-chip global memory buffers and is part of the OpenCL 2.0 specification.

The on-chip global memory buffer optimization makes use of the block memory instances embedded in the FPGA logic to create a memory buffer that is only visible to the kernels accessing the buffer. The following code example illustrates the usage model for global memory buffers that is suitable for the on-chip global memory buffer optimization.

```
// 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_state(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 based memory. Because 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. The SDAccel environment 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 on the memories are that all kernels with access to the on-chip memory are executed in the FPGA logic and that the memory has at least 4096 bytes. For example, in an array of `int` data type, the minimum array size needs to be 1024.

IMPORTANT: *On-chip global memory optimization is an automatic optimization in the SDAccel environment that is applied to memory buffers that are only accessed by kernels executed in the FPGA logic.*

Reducing Kernel to Kernel Communication Latency with OpenCL Pipes

The OpenCL 2.0 specification introduces a new memory object called pipe. A pipe stores data organized as a FIFO. Pipe objects can only be accessed using built-in functions that read from and write to a pipe. Pipe objects are not accessible from the host. Pipes can be used to stream data from one kernel to another inside the FPGA device without having to use the external memory, which greatly improves the overall system latency.

In the SDAccel development environment, pipes must be statically defined outside of all kernel functions. Dynamic pipe allocation using the `OpenCL 2.x clCreatePipe` API is not currently supported. The depth of a pipe must be specified by using the **xcl_reqd_pipe_depth** attribute in the pipe declaration. The valid depth values are 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768.

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
```

A given pipe, can have one and only one producer and consumer in different kernels.

Pipes can be accessed using standard OpenCL `read_pipe()` and `write_pipe()` built-in functions in non-blocking mode or using Xilinx® extended `read_pipe_block()` and `write_pipe_block()` functions in blocking mode. The status of pipes can be queried using OpenCL `get_pipe_num_packets()` and `get_pipe_max_packets()` built-in functions. Please see The OpenCL C Specification, Version 2.0 from Khronos Group for more details on these built-in functions.

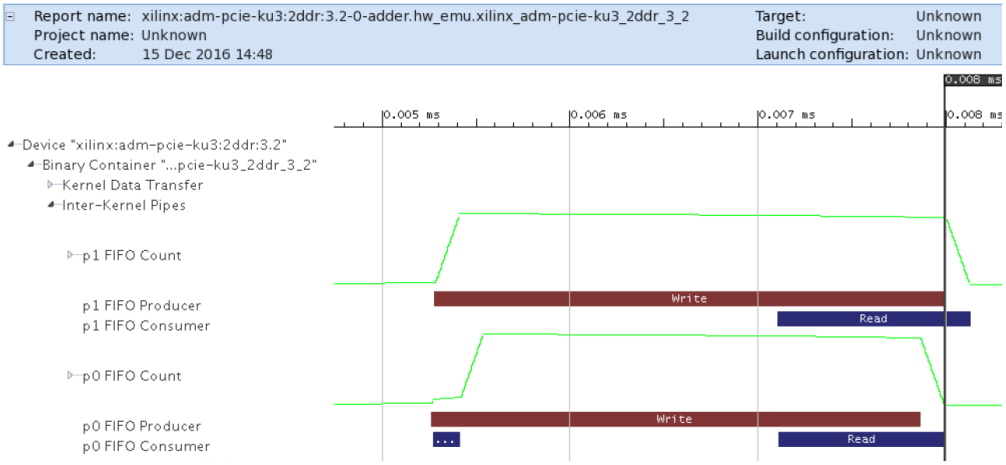
The following are the function signatures for the currently supported pipe functions, where gentype indicates the built-in OpenCL C scalar integer or floating-point data types.

```
int read_pipe_block (pipe gentype p, gentype *ptr)
int write_pipe_block (pipe gentype p, const gentype *ptr)
```

The following is the "Blocking Pipes Example" from Xilinx On-boarding Example GitHub that use pipes to pass data from one processing stage to the next using `blocking read_pipe_block()` and `write_pipe_block()` functions:

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(32)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(32)));
// Input Stage Kernel : Read Data from Global Memory and write into Pipe P0
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void input_stage(__global int *input, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_rd: for (int i = 0 ; i < size ; i++)
    {
        //blocking Write command to pipe P0
        write_pipe_block(p0, &input[i]);
    }
}
// Adder Stage Kernel: Read Input data from Pipe P0 and write the result
// into Pipe P1
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void adder_stage(int inc, int size)
{
    __attribute__((xcl_pipeline_loop))
    execute: for(int i = 0 ; i < size ;  i++)
    {
        int input_data, output_data;
        //blocking read command to Pipe P0
        read_pipe_block(p0, &input_data);
        output_data = input_data + inc;
        //blocking write command to Pipe P1
        write_pipe_block(p1, &output_data);
    }
}
// Output Stage Kernel: Read result from Pipe P1 and write the result to
Global
// Memory
kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void output_stage(__global int *output, int size)
{
    __attribute__((xcl_pipeline_loop))
    mem_wr: for (int i = 0 ; i < size ; i++)
    {
        //blocking read command to Pipe P1
        read_pipe_block(p1, &output[i]);
    }
}
```

The device traceline view shows the detailed activities and stalls on the OpenCL pipes after hardware emulation is run. This info can be used to choose the correct FIFO sizes to achieve the optimal application area and performance.



Improving Kernel Frequency

One of the key advantages of an FPGA is its flexibility and capacity to create customized design specifically for your algorithm to improve throughput and save power. The downside of creating custom logic is that the design will need to go through traditional FPGA design flow and meet the target frequency each time the design is built. The guidelines below will generally help you improve achievable kernel frequency:

- Partition design into sub-functions with reasonable size for each function.
- Pipeline each sub-function to get best II.
- Create top function with sub-function calls only.
- Optimize the top function with DATAFLOW attribute/pragma.
- Keep loop body small in each sub-function.
- Watch out for warnings on high fanout net in the kernel and recode the design accordingly.
- Use pipeline and array partition attributes/pragma precisely to have a good balance of performance and design size.

XILINX
ALL PROGRAMMABLE™

Appendix A

On-Boarding Examples

To help users quickly get started with the SDAccel Environment, Xilinx On-boarding Example GitHub hosts many examples to demonstrate good design practices, coding guidelines, design pattern for common applications, and most importantly optimization techniques to maximize application performance. The on-boarding examples are divided into several main categories. Each category has various key concepts that are illustrated by individual examples in both OpenCL C and C/C++ when applicable. All examples include Makefile for running software emulation, hardware emulation, and running on hardware and a `README.md` file that explains the example in details.

XILINX
ALL PROGRAMMABLE

Appendix B

Additional Resources and Legal Notices

Xilinx Resources

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

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. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDAccel Environment User Guide* (UG1023)
3. *SDAccel Environment Optimization Guide* (UG1207)
4. *SDAccel Environment Tutorial: Introduction* (UG1021)
5. *SDAccel Environment Platform Development Guide* (UG1164)
6. SDAccel Development Environment web page
7. Vivado® Design Suite Documentation
8. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
9. *Vivado Design Suite User Guide: High level Synthesis* (UG902)
10. Khronos Group web page: Documentation for the OpenCL standard
11. Alpha Data web page: Documentation for the ADM-PCIE-7V3 Card
12. Pico Computing web page: Documentation for the M-505-K325T card and the EX400 Card

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.