Get Moving with Alveo

April 17, 2019

XILINX

Table of Contents

Appendix A: Additional Resources and Legal Notices

XILINX

Chapter 1

Getting Started with Alveo and Acceleration

Introduction

Xilinx FPGAs and Versal ACAP devices are uniquely suitable for low-latency acceleration of high performance algorithms and workloads. With the demise of traditional Moore's Law scaling, design-specific architectures (DSAs) are becoming the tool of choice for developers needing the optimal balance of capability, power, latency, and flexibility. But, approaching FPGA and ACAP development from a purely software background can seem daunting.

With this set of documentation and tutorials, our goal is to provide you, our customer and partner, with an easy-to-follow, guided introduction to accelerating applications with Xilinx cards. We will begin from the first principles of acceleration: understanding the fundamental architectural approaches, identifying suitable code for acceleration, and interacting with the software APIs for managing memory and interacting with the Alveo cards in an optimal way.

This set of documents is intended for use by software developers, it is **not** a low-level hardware developer's guide. The topics of RTL coding, low-level FPGA architecture, high-level synthesis optimization, and so on are covered elsewhere in other Xilinx documents. Our goal here is to get you up and running on Alveo quickly, with the confidence to approach your own acceleration goals and grow your familiarity and skill with Xilinx devices over time.

Provided Design Files

In this directory tree you will find two primary directories: **doc** and **examples**. These contain, respectively, the source to the documentation you're currently reading and the example designs that follow along with that documentation. The example designs correspond to specific sections in this guide. Every effort has been made to to keep the code samples in these documents as concise and "to the point" as possible.

XILINX

Chapter 2

Acceleration Basics

Acceleration Concepts

Rather than dive immediately into the API, especially if you don't have much experience with acceleration, let's start with a simplified metaphor to help explain what must happen in an acceleration system.

Imagine that you have been given the job of giving tours of your city. Your city may be large or small, dense or sparse, and may have specific traffic rules that you must follow. This is your **application space**. Along the way, you are tasked with teaching your wards a specific set of facts about your city, and you must hit a particular set of sights (and shops - you have to get paid, after all). This set of things you must do is your **algorithm**.

Given your application space and your algorithm, you started small: your tours fit into one car, and every year you would buy a new one that was a bit bigger, a bit faster, and so on. But, as your popularity grew, more and more people started to sign up for the tours. There's a limit to how fast your car can go (even the sports model), so how can you scale? This is the problem with **CPU acceleration**.

The answer is a tour bus! It might have a lower top speed and take much longer to load and unload than your earlier sports car idea, but you can transport many more people - much more **data** - through your algorithm. Over time, your fleet of buses can expand and you can get more and more people through the tour. This is the model for **GPU acceleration**. It scales very well - until, that is, your fuel bills start to add up, there's a traffic jam in front of the fountain, and wouldn't you know it, the World Cup is coming to town!

If only the city would allow you to build a monorail. On their dollar, with the permits pre-approved. Train after train, each one full, making all the stops along the way (and with full flexibility to bulldoze and change the route as needed). For the tour guide in our increasingly tortured metaphor, this is fantasy. But for you, this is the model for **Alveo acceleration**. FPGAs combine the parallelism of a GPU with the low-latency streaming of a domain-specific architecture for unparalleled performance.

But, as the joke goes, even a Ferrari isn't fast enough if you never learn to change gears. So, let's abandon metaphor and roll up our sleeves.

Identifying Acceleration Opportunities

In general, for acceleration systems we are trying to hit a particular performance target, whether that target is expressed in terms of overall end-to-end latency, frames per second, raw throughput, or something else.

Generally speaking, candidates for acceleration are algorithmic processing blocks that work on large chunks of data in deterministic ways.

In 1967, Gene Amdahl famously proposed what has since become known as Amdahl's Law. It expresses the potential speedup in a system:

$$S_{latency}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

In this equation, $S_{latency}$ represents the theoretical speedup of a task, s is the speedup of the portion of the algorithm that benefits from acceleration, and p represents the proportion of the execution time the task occupied pre-acceleration.

Amdahl's Law demonstrates a clear limit to the benefit of acceleration in a given application, as the portion of the task that cannot be accelerated (generally involving decision making, I/O, or other system overhead tasks) will always become the system bottleneck.

$$\lim_{s \to \infty} S_{latency}(s) = \frac{1}{1-p}$$

If Amdahl's Law is correct, however, then why do so many modern systems use either general-purpose or domain-specific acceleration? The crux lies here: modern systems are processing an ever-increasing amount of data, and Amdahl's Law *only applies to cases where the problem size is fixed*. That is to say, the limit is imposed only so long as p is a constant fraction of overall execution time.

In 1988 John Gustafson and Edwin Barsis proposed a reformulation that has since become known as Gustafson's Law. It re-casts the problem:

$$S_{latency}(s) = 1 - p + sp$$

Again, $S_{latency}$ represents the theoretical speedup of a task, s is the speedup in latency of the task benefiting from parallelism, and p is the percentage of the overall task latency of the part benefiting from improvement (prior to the application of the improvement).

Gustafson's Law re-frames the question raised by Amdahl's Law. Rather than more compute resources speeding up the execution of a single task, more compute resources allow more computation in the same amount of time.

Both of the laws frame the same thing in different ways: to accelerate an application, we are going to *parallelize* it. Through parallelization we are attempting to either process more data in the same amount of time, or the same amount of data in less time. Both approaches have mathematical limitations, but both also benefit from more resources (although to different degrees).

In general, Alveo cards are useful for accelerating algorithms with lots of "number crunching" on large data sets - things like video transcoding, financial analysis, genomics, machine learning, and other applications with large blocks of data that can be processed in parallel. It's important to approach acceleration with a firm understanding of how and when to apply external acceleration to a software algorithm. Throwing random code onto any accelerator, Alveo or otherwise, does not generally give you optimal results. This is for two reasons: first and foremost, leveraging acceleration sometimes requires restructuring a sequential algorithm to increase its parallelism. Second, while the Alveo architecture allows you to quickly process parallel data, transferring

data over PCIe and between DDR memories has an inherent *additive latency*. You can think of this as a kind of "acceleration tax" that you must pay to share data with any external accelerator.

With that in mind, we want to look for areas of code that satisfy several conditions. They should:

- Process large blocks of data in deterministic ways.
- Have well-defined data dependencies, preferably sequential or stream-based processing. Random-access should be avoided unless it can be bounded.
- Take long enough to process that the overhead of transferring the data between the CPU and the Alveo card does not dominate the accelerator run time.

Alveo Overview

Before diving into the software, let's familiarize ourselves with the capabilities of the Alveo card itself. Each Alveo card combines three essential things: a powerful FPGA or ACAP for acceleration, high-bandwidth DDR4 memory banks, and connectivity to a host server via a high-bandwidth PCIe Gen3x16 link. This link is capable of transferring approximately 16 GiB of data per second between the Alveo card and the host.

Recall that unlike a CPU, GPU, or ASIC, an FPGA is effectively a blank slate. You have a pool of very low-level logic resources like flip-flops, gates, and SRAM, but very little fixed functionality. Every interface the device has, including the PCIe and external memory links, is implemented using at least some of those resources.

To ensure that the PCIe link, system monitoring, and board health interfaces are always available to the host processor, Alveo designs are split into a **shell** and **role** conceptual model. The **shell** contains all of the static functionality: external links, configuration, clocking, etc. Your design fills the **role** portion of the model with custom logic implementing your specific algorithm(s). You can see this topology reflected in figure 2.1.

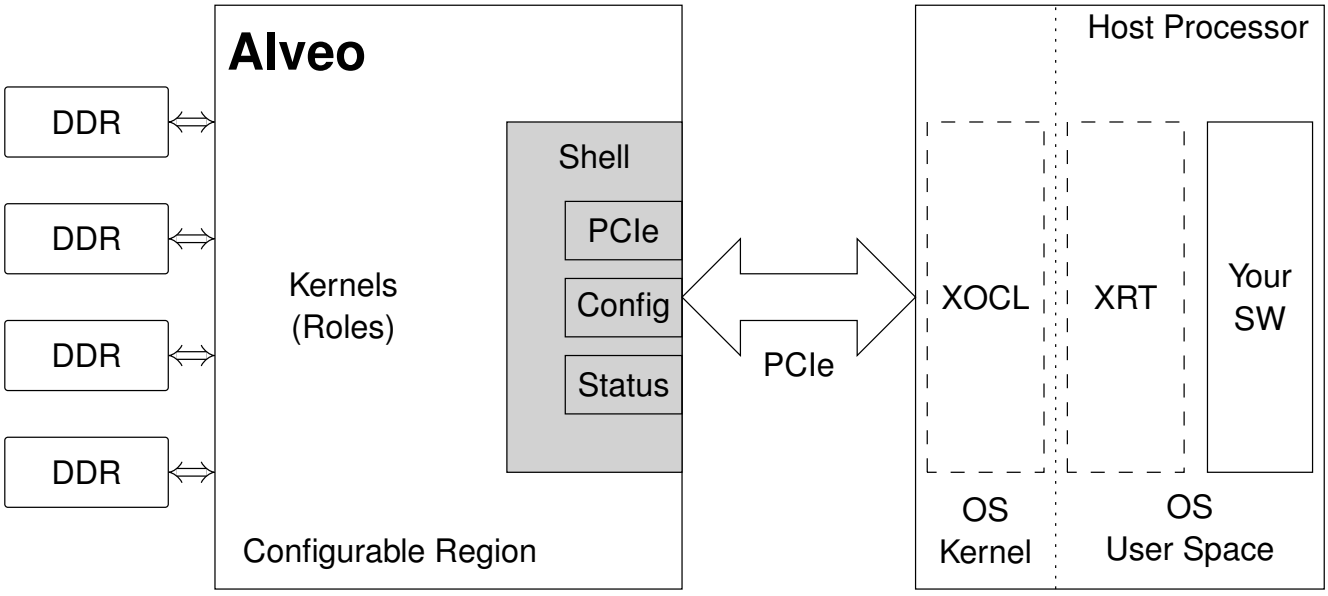

Figure 2.1: Alveo Conceptual Topology

The Alveo FPGA is further subdivided into multiple super logic regions (SLRs), which aid in the architecture of

very high-performance designs. But this is a slightly more advanced topic that will remain largely unnoticed as you take your first steps into Alveo development.

Alveo cards have multiple on-card DDR4 memories. These memories have a high bandwidth to and from the Alveo device, and in the context of OpenCL are collectively referred to as the *device global memory*. Each memory bank has a capacity of 16 GiB and operates at 2400 MHz DDR. This memory has a very high bandwidth, and your kernels can easily saturate it if desired. There is, however, a latency hit for either reading from or writing to this memory, especially if the addresses accessed are not contiguous or we are reading/writing short data beats.

The PCIe lane, on the other hand, has a decently high bandwidth but not nearly as high as the DDR memory on the Alveo card itself. In addition, the latency hit involved with transferring data over PCIe is quite high. As a general rule of thumb, transfer data over PCIe as infrequently as possible. For continuous data processing, try designing your system so that your data is transferring while the kernels are processing something else. For instance, while waiting for Alveo to process one frame of video you can be simultaneously transferring the next frame to the global memory from the CPU.

There are many more details we could cover on the FPGA architecture, the Alveo cards themselves, etc. But for introductory purposes, we have enough details to proceed. From the perspective of designing an acceleration architecture, the important points to remember are:

- Moving data across PCIe is expensive - even at Gen3x16, latency is high. For larger data transfers, bandwidth can easily become a system bottleneck.
- Bandwidth and latency between the DDR4 and the FPGA is significantly better than over PCIe, but touching external memory is still expensive in terms of overall system performance.
- Within the FPGA fabric, streaming from one operation to the next is effectively free (this is our train analogy from earlier; we'll explore it in more detail later).

Xilinx Runtime (XRT) and APIs

Although this may seem obvious, any hardware acceleration system can be broadly discussed in two parts: the hardware architecture and implementation, and the software that interacts with that hardware. For the Alveo cards, regardless of any higher-level software frameworks you may be using in your application such as FFmpeg, GStreamer, or others, the software library that fundamentally interacts with the Alveo hardware is the Xilinx Runtime (XRT).

While XRT consists of many components, its primary role can be boiled down to three simple things:

- **Programming the Alveo card kernels** and managing the life cycle of the hardware
- **Allocating memory** and migrating that memory between the host CPU and the card
- **Managing the operation of the hardware**: sequencing execution of kernels, setting kernel arguments, etc.

These three things are also, in the same order, the most to least "expensive" operations you can perform on an Alveo acceleration card. Let's examine that in more detail.

Programming the Alveo card kernels inherently takes some amount of time. Depending on the capacity of the card's FPGA, the PCIe bandwidth available to transfer the configuration image, etc., the time required

is typically in the order of dozens to hundreds of milliseconds. This is generally a “one time deal” when you launch your application, so the configuration hit can be absorbed into general setup latency, but it’s important to be aware of. There are applications where Alveo is reprogrammed multiple times during operation to provide different large kernels. If you’re planning to build such an architecture, incorporate this configuration time into your application as seamlessly as possible. It’s also important to note that while many applications can simultaneously use the Alveo card, only one image can be programmed at any point in time.

Allocating memory and moving it around is the real “meat” of XRT. Allocating and managing memory effectively is a critical skill for developing acceleration architectures. If you don’t manage memory and memory migration efficiently, you will *significantly* impact your overall application performance - and not in the way you’d hope! Fortunately XRT provides many functions to interact with memory, and we will explore those specifics later on.

Finally, XRT **manages the operation of the hardware** by setting kernel arguments and managing the kernel execution flow. Kernels can run sequentially or in parallel, from one process or many, and in blocking or non-blocking ways. The exact way your software interacts with your kernels is under your control, and we will investigate some examples of this later on.

It’s worth noting that XRT is a low-level API. For very advanced or unusual use models you may wish to interact with it directly, but most designers choose to use a higher-level API such as OpenCL, the Xilinx Media Accelerator (XMA) framework, or others. Figure 2.2 shows a top-level view of the available APIs. For this document, we will focus primarily on the OpenCL API to make this introduction more approachable. If you’ve used OpenCL before you will find it mostly similar (although Xilinx does provide some extensions for FPGA-specific tasks).

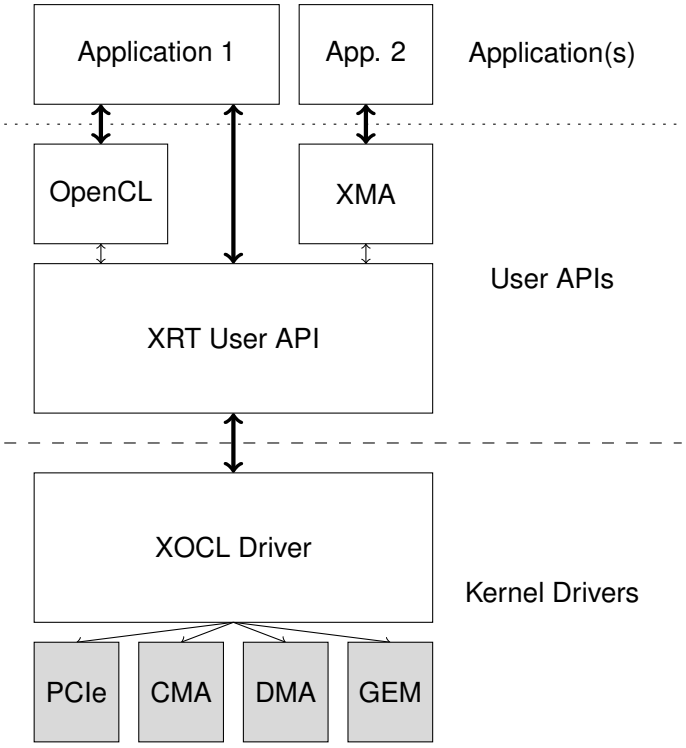

Figure 2.2: XRT Software Stack

XILINX

Chapter 3

Runtime Software Design

Memory Allocation Concepts

When you run a program on a CPU, you don't generally worry about how the underlying hardware manages memory. There are occasional issues with alignment, etc. on some processor architectures, but largely modern OSes and compilers abstract these things to a degree where they're not usually visible unless you deal with a lot of low-level driver work (or acceleration).

Fundamentally, we can think of memory as broadly having a set of six attributes. Given a pointer to a data buffer, that data pointer may be **virtual** or **physical**. The memory to which it points may be **paged** or **physically contiguous**. And, finally, from the standpoint of the processor that memory may be **cacheable** or **non-cacheable**.

Most modern operating systems make use of virtual memory. There are many reasons for doing so, but to keep this document from veering off course and morphing into a computer architecture textbook, just know that Linux, which is likely running your XRT, uses virtual memory. As a result, when you use a standard C or C++ userspace API function such as *malloc()* or *new*, you wind up with a pointer to a virtual memory address, not a physical one.

You will also likely wind up with a pointer to a block of addresses in memory that is **paged**. Nearly every modern OS (again including Linux) divides the address range into pages, typically with each page having a size of 4 KiB (although this can vary system to system). Each page is then mapped to a corresponding page in the physical memory. And before someone points out that this is an inaccurately simplified generalization, I would like to respectfully remind you that this *isn't* a computer architecture course!

There are two important things to note, though. The first is that when you allocate a buffer from the heap with standard C APIs, you're not getting a physical memory address back. And second, you're not getting a single buffer - you're getting a collection of N memory *pages*, each 4 KiB long. To put that in context, say we allocate a buffer of 6 MiB. That would give us:

$$\frac{6\,\text{MiB}}{4\,\text{KiB}} = 1536\,\text{pages}$$

If you wanted to copy that entire 6 MiB buffer from the host to your Alveo card, you would need to resolve 1536 virtual page addresses to physical memory address. You would then need to assemble these physical addresses into a *scatter gather list* to enqueue to a DMA engine with scatter-gather capability, which would then copy those pages one-by-one to their destination. It also works in reverse if you were to copy a buffer from Alveo to a virtual, paged address range in host memory. Host processors are generally quite fast, and as a result building this list doesn't take a huge amount of time. But given decently large buffers, this can contribute

to the overall system latency, so it's important to understand the implications this can have on your overall system performance.

For a simplified graphical view of such a system, refer to figure 3.1. In this example we have two buffers, creatively named A and B. Both A and B are virtual buffers. In our example we want to transfer A to Alveo, do something to it that results in an update to buffer B, and transfer it back. You can see how the virtual to physical mapping works. Within the Alveo card, accelerators operate only on physical memory addresses and data is always stored contiguously; this is primarily because this configuration generally provides the best performance.

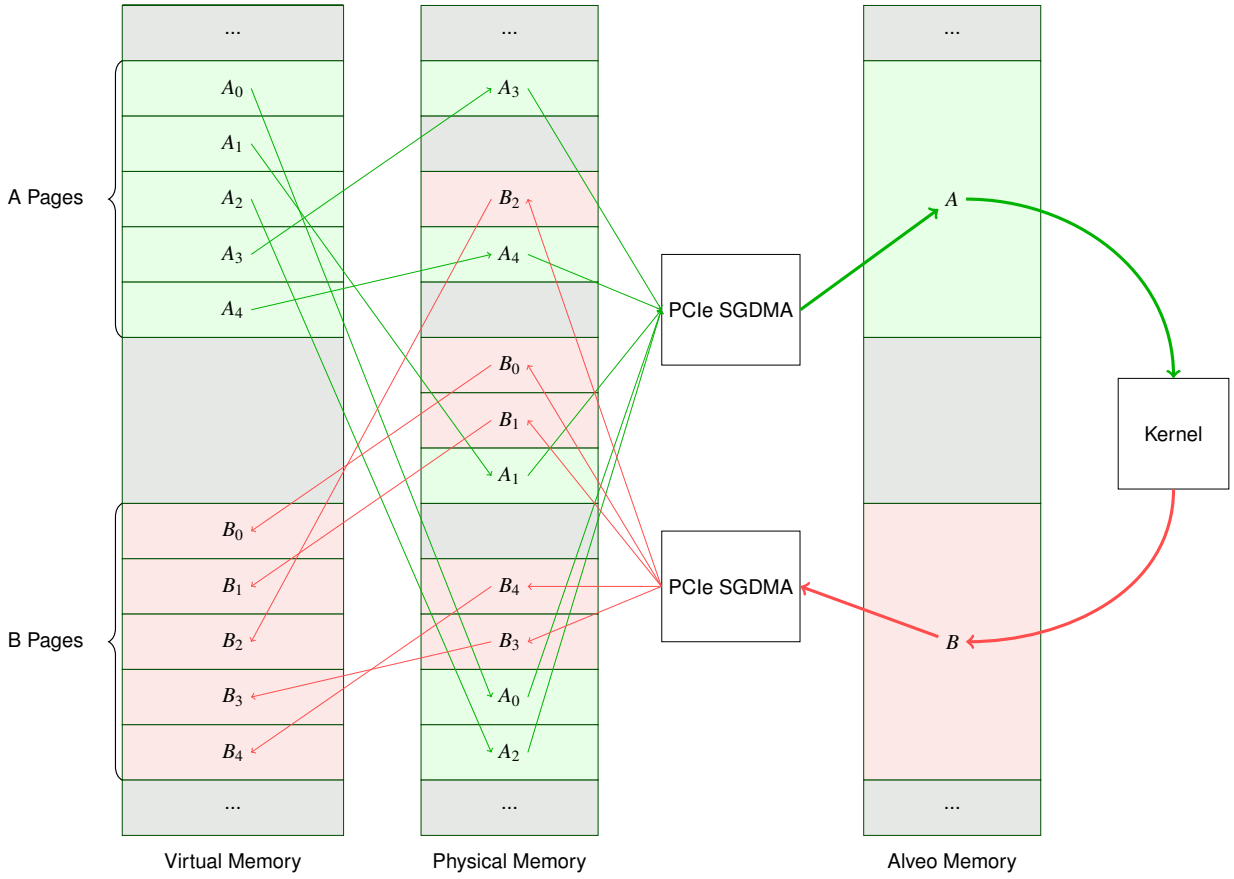

Figure 3.1: Virtual Memory Transfer to/from Alveo

As you can see, our simplified example data flow is already somewhat complex! Now, imagine that across a significantly large buffer: one spanning many megabytes (or gigabytes) with thousands and thousands of pages. You can see how building and managing those scatter gather lists and managing the page tables can become time consuming even for a fast host processor. In reality, memory is not usually quite so fragmented as our example. But because you don't usually know the physical addresses of the pages ahead of time, you must treat them as though each one is unique.

It's much easier to build a scatter gather list, though, if you know that all of the pages are **contiguous** in physical memory. That is, the data bytes are laid out sequentially, where incrementing the physical address gives you data[n+1]. In that case, you can build a scatter gather list knowing only the start address of the buffer and its size.

This is a benefit for many DMA operations, not just DMA to and from Alveo. Modern operating systems provide memory allocators (typically through kernel services) for just this purpose. In Linux this is done through the **Contiguous Memory Allocator** subsystem. This is kernel-space functionality but is exposed to users through a variety of mechanisms including *dmabuf*, the XRT API, various graphics drivers, and other things. If we allocate the previous buffer contiguously we wind up with a much simpler view of the world, as shown in figure 3.2.

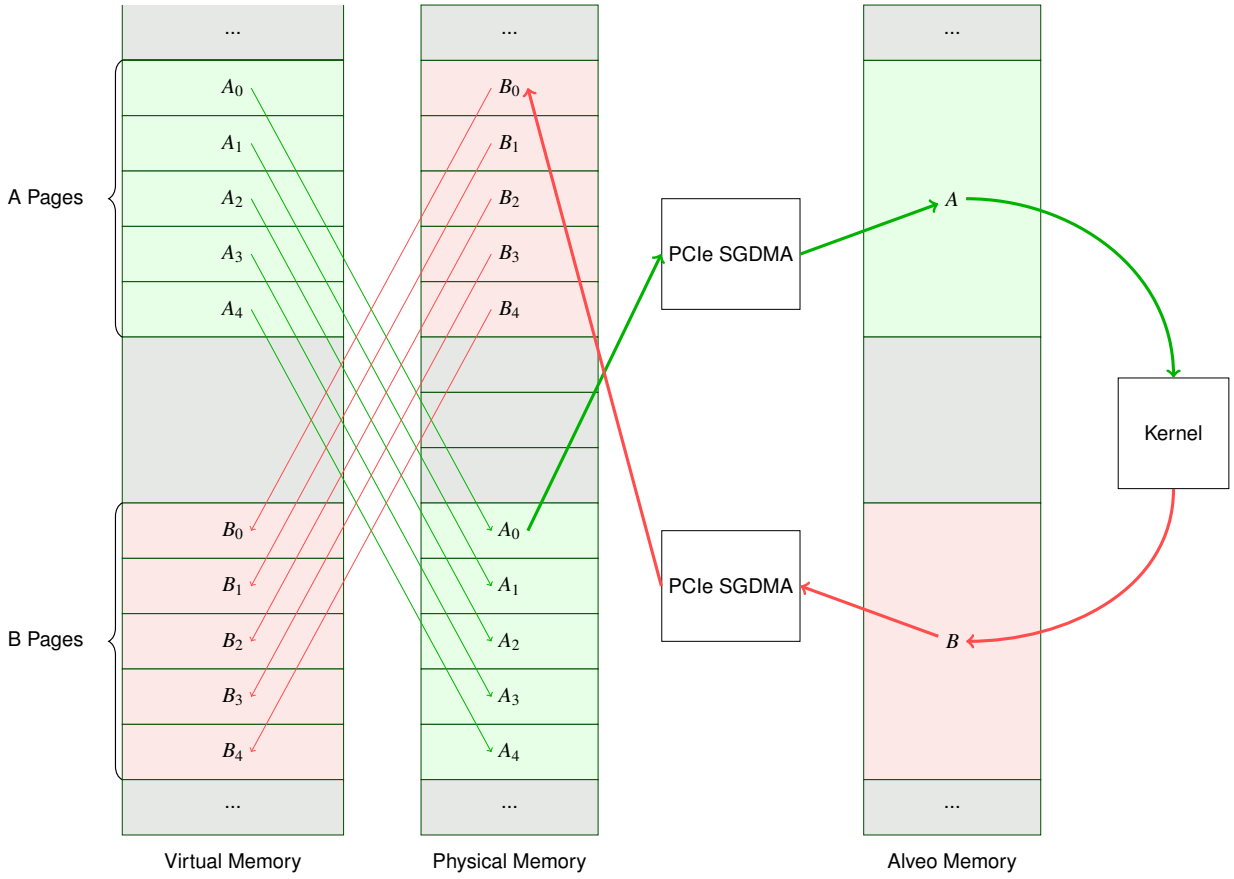

Figure 3.2: Virtual Memory Transfer to/from Alveo

You might be wondering what happens if your buffer doesn't begin on an aligned 4k page boundary. Good question! DMA engines generally require some degree of alignment, and the Alveo DMA is no different. If your allocated memory is not aligned to a page boundary *the runtime will align it for you*. Although you will incur a **memcpy()** operation, which is expensive computationally. You will also see a warning from the runtime (assuming you haven't suppressed them) because this is a problem you'll want to resolve as quickly as possible. Identifying and fixing this issue is addressed later on.

Finally, we must understand whether our memory is **cacheable** or **non-cacheable**. Because external memory access is expensive, nearly all modern processors maintain an internal cache of data with very low latency. Depending on the processor architecture, the size of the cache may vary from tens of kilobytes to many megabytes. This internal cache memory is synchronized with the external physical memory as needed. Generally this cache management is invisible to the software running on the processor - you would see the benefit of using a cache in your improved execution times, but you don't have to interact with it from a general development perspective.

However, when using DMA it's important to recognize that, absent cache coherency sharing technologies such as CCIX, you must ensure that any data the processor wants to share with the accelerator is synchronized with the external memory before it's transferred. Before starting a DMA transfer you need to ensure that data resident in the cache is **flushed** to external memory. Similarly, once data is transferred back you must ensure that data resident in the cache is **invalidated** so it will be refreshed from the external memory. These operations are very fast for x86-style processors, and are transparently handled by the runtime, but other architectures may have performance penalties for cache management. To mitigate this, the API has functionality for allocating and working with non-cacheable buffers. Do keep in mind, though, that a processor accessing data in a non-cacheable buffer will generally be much slower than just running cache management operations in the first place. This is usually used for models where the processor is sequencing buffers but not actually accessing the data they contain.

Alveo Guided Software Introduction

Whew, that's a lot of background! Let's quickly recap the important points before we jump into the guided software example:

- Acceleration is generally done on some intersection of two axes: trying to do the same task **faster** (Amdahl's Law), or trying to do more tasks in the **same time** (Gustafson's Law). Each axis has different implications on how to optimize, and how impactful that optimization will be.
- Acceleration inherently comes with an "acceleration tax." To achieve acceleration in a real system, the benefits achieved by acceleration must dominate the extra latency of the data transfers. You should pick your battles; focus on the biggest "bang for the buck" pieces of your algorithms.
- Interaction with the Alveo cards is done through XRT and higher-level API abstractions like OpenCL. Software-side optimization is done via the library, independent of the optimization of the hardware kernels.
- Memory allocation and management can have a significant effect on your overall application performance.

We will explore all of these topics in detail in the examples.

Guided Software Examples

Provided Design Files

In the installation package for this example series you will find two primary directories: **doc** and **examples**. The **doc** directory contains the source files for this document, and the **examples** directory contains all of the source files necessary to build and run the examples (with the exception of the build tools such as SDAccel, XRT, and the Alveo development shell which you must install yourself).

Under the **examples** directory, there are two primary directories: **hw_src** and **sw_src**. As the names imply, these contain the hardware and software source files for our application. The hardware sources synthesize to algorithms that run on the FPGA through the Xilinx **XOCC** compiler, and the software sources are compiled using standard GCC to run on the host processor.

We are focusing on software more than hardware in these tutorials, so we have split the source files for easy organization. In a real project any directory structures, etc. are arbitrary; you can follow the best practices of your team or organization.

Because some of the examples rely on external libraries, we are using the **CMake** build system for the software examples as it simplifies environment configuration. But on the hardware side, we're using standard **make**. This is so you can easily see the command line arguments passed to XOCC.

Hardware Design Setup

This guided introduction will introduce acceleration concepts targeting the Alveo cards. We'll start with writing host code: programming the FPGA, allocating memory, and moving memory around. Our accelerators will be very simple for these early examples. In fact, we'll likely see that the algorithms run faster on the CPU, at least at first, since our acceleration hardware is so trivial.

Building hardware designs can also be quite time consuming - it turns out that synthesizing, placing, and routing custom logic across billions of transistors with sub-nanosecond timing is just a *teeny* bit more complex than compiling to machine code. Bear with us, because the results are worth it. To avoid having to needlessly rebuild the FPGA hardware, we provide a single FPGA design with many kernel instances, which we'll mix-and-match as needed for the example designs. This guide will touch a bit on kernel optimization, but beyond basic concepts we'll mostly leave that for other documents in the Xilinx catalog.

This onboarding example comes with a pre-built hardware image targeting the Alveo U200 accelerator card with shell version **201830.1**. If you have that card and shell version, you do not need to do anything to set up the hardware and can proceed directly to the software tutorials.

If you do *not* have one or both of the above, you'll need to build the hardware design before you can run (don't worry, the example software will still work). To do that, change into the directory:

```
onboarding/examples/hw_src
```

If you have a different version of the shell installed on your board, you can skip this step. But if you are targeting a platform other than the Alveo U200, open the Makefile and change the first line to point to your platform's

.xpfm file.

Note: *Line breaks have been added to the example below for formatting. Do not add line breaks to the path in your Makefile.*

```
PLATFORM := /opt/xilinx/platforms/xilinx_u200_xdma_201830_1/
            xilinx_u200_xdma_201830_1.xpfm
```

Once you edit the Makefile, ensure that your SDAccel and XRT environments are set up properly. If you have not done so already, run the commands:

```
source /opt/Xilinx/SDx/2018.3/settings.sh
source /opt/xilinx/xrt/setup.sh
```

If your installation path for either XRT or SDAccel are different than the default locations, update the command line accordingly. Then, run the command:

```
make
```

The build process will take some time, but in the end you will have a file in this directory called **alveo_examples.xclbin**. This file contains all of the kernels we will use during this exercise. Once you have this file compiled for your combination of board and shell, you are ready to proceed to the next section.

Building the Software Designs

To avoid system-specific dependencies, for these examples we'll use CMake to build all of the test applications. All of the example-specific code resides in the **sw_src** directory. Each source file is named after the example it corresponds with: **00_load_kernels.cpp**, for example, corresponds to example 0. There are some additional "helper" files that are shared between applications and which are compiled into their own libraries.

To build the source, first ensure (as for the hardware) that XRT is set up properly in your environment. The environment variable $XILINX_XRT should point to your XRT install area. If it does not, run (assuming you have installed XRT to /opt/xilinx/xrt):

```
source /opt/xilinx/xrt/setup.sh
```

Then, with XRT configured, create and change to a build directory:

```
cd onboarding/examples
mkdir build
cd build
```

From within that build area, run `CMake` to configure the build environment and then run `make` to build all of the examples:

```
cmake ..
make
```

You will see a number of executable files corresponding to the different numbered examples along with a copy of the **alveo_examples.xclbin** file from the previous step.

Example 0: Loading an Alveo Image

Overview

For our first example, let's look at how to load images onto the Alveo card. When you power on the system, the Alveo card will initialize its shell (see figure 2.1). Recall from earlier that the shell implements connectivity with the host PC but leaves most of the logic as a blank canvas for you to build designs. Before we can use that logic in our applications, we must first configure it.

Also, recall from earlier that certain operations are inherently "expensive" in terms of latency. Configuring the FPGA is, in fact, one of the most inherently time consuming parts of the application flow. To get a feel for exactly *how* expensive, let's try loading an image.

Key Code

In this example, we initialize the OpenCL runtime API for XRT, create a command queue, and - most importantly - configure the Alveo card FPGA. This is generally a one-time operation: once the card is configured it will typically remain configured until power is removed or it is reconfigured by another application. Note that if multiple independent applications attempt to load hardware into the card, the second application will be blocked until the first one relinquishes control. Although multiple independent applications can share the *same image* running on a card.

To begin, we must include the headers as shown in listing 3.1. Note that the line numbers in the documentation correspond to the line numbers in the file **00_load_kernels.cpp**.

Listing 3.1: XRT and OpenCL Headers

```
// Xilinx OpenCL and XRT includes
#include "xcl2.hpp"

#include <CL/cl.h>
```

Of these two, only `CL/cl.h` is required. `xcl2.hpp` is a library of Xilinx-provided helper functions to wrap around some of the required initialization functions.

Once we include the appropriate headers, we need to initialize the command queue, load the binary file, and program it into the FPGA, as shown in listing 3.2. This is effectively boilerplate code you'll need to include in every program at some point.

Listing 3.2: XRT and OpenCL Headers

```
// This application will use the first Xilinx device found in the system
std::vector<cl::Device> devices = xcl::get_xil_devices();
cl::Device device               = devices[0];

cl::Context context(device);
cl::CommandQueue q(context, device);

std::string device_name    = device.getInfo<CL_DEVICE_NAME>();
std::string binaryFile     = xcl::find_binary_file(device_name, argv[1]);
```

```
cl::Program::Binaries bins = xcl::import_binary_file(binaryFile);

devices.resize(1);
cl::Program program(context, devices, bins);
```

The workflow here can be summed up as follows:

1. (Line 33-34): Discover and number the Xilinx devices in the system. In this case we can assume device 0 is the targeted card, but in a multi-accelerator card system you will have to specify them.
2. (Lines 36-37): Initialize the OpenCL context and command queues.
3. (Lines 39:41): Load the binary file targeting our Alveo board. In these examples we pass the file name on the command line, but this can be hard-coded or otherwise handled on an application-specific basis.
4. (Line 44) Program the FPGA.

Line 44 is where the programming operation is actually triggered. During the programming phase the runtime checks the current Alveo card configuration. If it is already programmed, we can return after loading the device metadata from the xclbin. But if not, let's program the device now.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./00_load_kernels alveo_examples
```

The program will output a message similar to this:

```
-- Example 0: Loading the FPGA Binary --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'

FPGA programmed, example complete!

-- Key execution times --
OpenCL Initialization : 1624.634 ms
```

Note that our FPGA took *1.6 seconds* to initialize. Be aware of this kernel load time; it includes disk I/O, PCIe latency, configuration overhead, and a number of other operations. Usually you will want to configure the FPGA during your application's startup time, or even pre-configure it. Let's run the application again with the bitstream already loaded:

```
-- Key execution times --
OpenCL Initialization :  262.374 ms
```

.26 seconds is *much* better than 1.6 seconds! We still have to read the file from disk, parse it, and ensure the xclbin loaded into the FPGA is correct, but the overall initialization time is significantly lower.

Extra Exercises

Some things to try to build on this experiment:

- Use the **xbutil** utility to query the board. Can you see which .xclbin file is loaded?
- Again using **xbutil**, which kernel(s) are present in the FPGA? Do you see a difference before and after the FPGA is programmed?

Key Takeaways

- FPGA configuration is an expensive operation. Ideally, initialize the FPGA long before it's needed. This can be done in a separate thread, during other initialization tasks in your application, or even at system boot time for dedicated systems.
- Once the FPGA is loaded, subsequent loads are substantially faster.

Now that we can load images into the FPGA, let's run something!

Example 1: Simple Memory Allocation

Overview

The FPGA image that we've loaded contains a very simple vector addition core. It takes two buffers of arbitrary length as inputs and produces a buffer of equal length as an output. As the name implies, during the process it adds them together.

Our code has not really been optimized to run well in an FPGA. It's mostly equivalent to directly putting the algorithm in listing 3.3 directly into the FPGA fabric. This isn't particularly efficient. We can process one addition operation on each tick of the clock but we're still only processing one 32-bit output at a time.

Listing 3.3: Vector Addition Algorithm

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
    for (int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

It's very important to note that at this point *there is no way this code will beat the processor*. The clock in the FPGA fabric is significantly slower than the CPU clock. This is expected, though - thinking back to our earlier example, we're only loading a single passenger into each car on the train. We also have overhead to pass the data over PCIe, set up DMA, etc. For the next few examples we'll look at how to efficiently manage the buffers for our inputs and outputs to this function. Only after that will we start to take advantage of the acceleration we can get from the Alveo card.

Key Code

This example is the first time we're going to actually run something on the FPGA, modest though it may be. In order to run something on the card there are four things that we must do:

1. Allocate and populate the buffers we'll use to send and receive data from the card.
2. Transfer those buffers between the host memory space and the Alveo global memory.
3. Run the kernel to act on those buffers.
4. Transfer the results of the kernel operation back to the host memory space so that they can be accessed via the processor.

As you can see, only one of those things actually takes place on the card. Memory management will make or break your application's performance, so let's start to take a look at that.

If you haven't done acceleration work before, you may be tempted to jump in and just use normal calls to *malloc()* or *new* to allocate your memory. In this example we'll do just that, allocating a series of buffers to transfer between the host and the Alveo card. We'll allocate four buffers: two input buffers to add together, one output buffer for the Alveo to use, and an extra buffer for a software implementation of our `vadd` function. This

allows us to see something interesting: how we allocate memory for Alveo also impacts how efficiently the processor will run.

Buffers are allocated simply, as in listing 3.4. In our case, `BUFSIZE` is 24 MiB, or $6 \times 1024 \times 1024$ values of type `uint32_t`. Any code not mentioned here is either identical or functionally equivalent to the previous examples.

Listing 3.4: Simple Buffer Allocation

```
uint32_t *a = new uint32_t[BUFSIZE];
uint32_t *b = new uint32_t[BUFSIZE];
uint32_t *c = new uint32_t[BUFSIZE];
uint32_t *d = new uint32_t[BUFSIZE];
```

This will allocate memory that is **virtual**, **paged**, and, most importantly, **non-aligned**. In particular it's this last one that is going to cause some problems, as we'll soon see.

Once we allocate the buffers and populate them with initial test vectors, the next acceleration step is to send them down to the Alveo global memory. We do that by creating OpenCL buffer objects using the flag **CL_MEM_USE_HOST_PTR**. This tells the API that rather than allocating its own buffer, we are providing our own pointers. This isn't necessarily bad, but because we haven't taken care allocating our pointers it's going to hurt our performance.

Listing 3.5 contains the code mapping our allocated buffers to OpenCL buffer objects.

Listing 3.5: Mapping OCL Buffers with Host Memory Pointers

```
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer a_to_device(context,
                       static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                 CL_MEM_USE_HOST_PTR),
                       BUFSIZE * sizeof(uint32_t),
                       a,
                       NULL);
cl::Buffer b_to_device(context,
                       static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                                 CL_MEM_USE_HOST_PTR),
                       BUFSIZE * sizeof(uint32_t),
                       b,
                       NULL);
cl::Buffer c_from_device(context,
                         static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY |
                                                   CL_MEM_USE_HOST_PTR),
                         BUFSIZE * sizeof(uint32_t),
                         c,
                         NULL);
inBufVec.push_back(a_to_device);
inBufVec.push_back(b_to_device);
outBufVec.push_back(c_from_device);
```

What we're doing here is allocating `cl::Buffer` objects, which are recognized by the API, and passing in pointers a, b, and c from our previously-allocated buffers. The additional flags `CL_MEM_READ_ONLY` and `CL_MEM_WRITE_ONLY` specify to the runtime the visibility of these buffers from the perspective of the kernel. In other words, *a* and *b* are written to the card by the host - to the kernel they are **read only**. Then, *c* is read back from the card to the host. To the kernel it is **write only**. We additionally add these buffer objects to vectors so that we can transfer multiple buffers at once (note that we're essentially adding *pointers* to the vectors, not the data buffers themselves).

Next, we can transfer the input buffers down to the Alveo card using the code in listing 3.6.

Listing 3.6: Migrating Host Memory to Alveo

```
cl::Event event_sp;
q.enqueueMigrateMemObjects(inBufVec, 0, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

In this code snippet the "main event" is the call to `enqueueMigrateMemObjects()` on line 108. We pass in our vector of buffers, the 0 indicates that this is a transfer from host to device, and we also pass in a `cl::Event` object.

This is a good time to segue briefly into synchronization. When we enqueue the transfer we're adding it to the runtime's "to-do list", if you will, but not actually waiting for it to complete. By registering a `cl::Event` object, we can then decide to wait on that event at any point in the future. In general this isn't a point where you would necessarily want to wait, but we've done this at various points throughout the code to more easily instrument it to display the time taken for various operations. This adds a small amount of overhead to the application, but again, this is a learning exercise and not an example of optimizing for maximum performance.

We now need to tell the runtime what to pass to our kernel, and we do that in listing 3.7. Recall that our argument list looked like this:

```
(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
```

In our case `a` is argument 0, `b` is argument 1, and so on.

Listing 3.7: Setting Kernel Arguments

```
krnl.setArg(0, a_to_device);
krnl.setArg(1, b_to_device);
krnl.setArg(2, c_from_device);
krnl.setArg(3, BUFSIZE);
```

Next, we add the kernel itself to the command queue so that it will begin executing. Generally speaking, you would enqueue the transfers and the kernel such that they'd execute back-to-back rather than synchronizing in between. The line of code that adds the execution of the kernel to the command queue is in listing 3.8.

Listing 3.8: Enqueue Kernel Run

```
q.enqueueTask(krnl, NULL, &event_sp);
```

If you don't want to wait at this point you can again pass in `NULL` instead of a `cl::Event` object.

And, finally, once the kernel completes we want to transfer the memory back to the host so that we can access the new values from the CPU. This is done in listing 3.9.

Listing 3.9: Transferring Data Back to the Host

```
q.enqueueMigrateMemObjects(outBufVec, CL_MIGRATE_MEM_OBJECT_HOST, NULL, &event_sp);
clWaitForEvents(1, (const cl_event *)&event_sp);
```

In this instance we *do* want to wait for synchronization. This is important; recall that when we call these enqueue functions, we're placing entries onto the command queue in a **non-blocking** manner. If we then attempt to access the buffer immediately after enqueuing the transfer, it have finished reading back in.

Excluding the FPGA configuration from example 0, the new additions in order to run the kernel are:

1. (Line 60-63): Allocate buffers in the normal way. We'll soon see that there are better ways of doing this, but this is the way many people experimenting with acceleration might do it their first time.

2. (Lines 81-102): Map the allocated buffers to `cl::Buffer` objects.
3. (Line 108): Enqueue the migration of the input buffers (*a* and *b*) to Alveo device global memory.
4. (Lines 113-116): Set the kernel arguments, both buffers and scalar values.
5. (Line 120): Run the kernel.
6. (Line 126-127): Read the results of the kernel back into CPU host memory, synchronizing on the completion of the read.

Only one synchronization is needed were this a real application. As previously, mentioned we're using several to better report on the timing of various operations in the workflow.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./01_simple_malloc alveo_examples
```

The program will output a message similar to this:

```
-- Example 1: Vector Add with Malloc() --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test with malloc()ed buffers
WARNING: unaligned host pointer '0x154f7909e010' detected,
this leads to extra memcpy
WARNING: unaligned host pointer '0x154f7789d010' detected,
this leads to extra memcpy
WARNING: unaligned host pointer '0x154f7609c010' detected,
this leads to extra memcpy

Simple malloc vadd example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  247.371 ms
Allocating memory buffer           :    0.030 ms
Populating buffer inputs           :   47.955 ms
Software VADD run                  :   35.706 ms
Map host buffers to OpenCL buffers :   64.656 ms
Memory object migration enqueue    :   24.829 ms
Set kernel arguments               :    0.009 ms
OCL Enqueue task                   :    0.064 ms
Wait for kernel to complete        :   92.118 ms
Read back computation results      :   24.887 ms
```

Note that we have some warnings about unaligned host pointers. Because we didn't take care with our allocation, none of our buffers that we're transferring to or from the Alveo card are aligned to the 4 KiB boundaries needed by the Alveo DMA engine. Because of this, we need to copy the buffer contents so they're aligned before transfer, and that operation is quite expensive.

From this point on in our examples, let's keep a close eye on these numbers. While there will be some variability on the latency run-to-run, generally speaking we are looking for deltas in each particular area. For now let's establish a baseline in table 3.1.

Table 3.1: Timing Summary - Example 1

Operation	Example 1
OCL Initialization	247.371 ms
Buffer Allocation	30 µs
Buffer Population	47.955 ms
Software VADD	35.706 ms
Buffer Mapping	64.656 ms
Write Buffers Out	24.829 ms
Set Kernel Args	9 µs
Kernel Runtime	92.118 ms
Read Buffer In	24.887 ms
$\Delta_{Alveo \rightarrow CPU}$	−418.228 ms
$\Delta_{Alveo \rightarrow CPU}$ (algorithm only)	−170.857 ms

That's certainly... not great. But are we going to give up? What, you think there must be some kind of reason Xilinx built this thing? Ok, let's see if we can do better!

Extra Exercises

Some things to try to build on this experiment:

- Vary the size of the buffers allocated. Can you derive an approximate relationship between buffer size and the timing for individual operations? Do they all scale at the same rate?
- If you remove synchronization between each step, what is the quantitative effect on the runtime?
- What happens if you remove the synchronization after the final buffer copy from Alveo back to the host?

Key Takeaways

- Once again we have to pay our FPGA configuration "tax". We will need to save at least 250 ms over the CPU to make up for it. Note that our trivial example will ***never*** beat the CPU if we're just looking at processing a single buffer!
- Simply-allocated memory isn't a good candidate for passing to accelerators, as we'll incur a memory copy to compensate. We'll investigate the impact this has in subsequent examples.
- OpenCL works on command queues. It's up to the developer how and when to synchronize, but care must be taken when reading buffers back in from the Alveo global memory to ensure synchronization before the CPU accesses the data in the buffer.

Example 2: Aligned Memory Allocation

Overview

In our last example we allocated memory simply, but as we saw the DMA engine requires that our buffers be aligned to 4 KiB pages boundaries. If the buffers are not so aligned, which they likely won't be if don't explicitly ask for it, then the runtime will copy the buffers so that their contents *are* aligned.

That's an expensive operation, but can we quantify how expensive? And how can we allocate aligned memory?

Key Code

This is a relatively short example in that we're only changing four lines vs. **Example 1**, our buffer allocation. There are various ways to allocate aligned memory but in this case we'll make use of a POSIX function, `posix_memalign()`. This change replaces our previous allocation from listing 3.4 with the code from listing 3.10. We also need to include an additional header not shown in the listing, **memory**.

Listing 3.10: Allocating Aligned Buffers

```
uint32_t *a, *b, *c, *d = NULL;
posix_memalign((void **)&a, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&b, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&c, 4096, BUFSIZE * sizeof(uint32_t));
posix_memalign((void **)&d, 4096, BUFSIZE * sizeof(uint32_t));
```

Note that for our calls to `posix_memalign()`, we're passing in our requested alignment, or 4 KiB as previously noted.

Otherwise, this is the only change to the code vs. the first example. Note that we have changed the allocation for all of the buffers, including buffer *d* which is only used by the CPU baseline VADD function. We'll see if this has any impact on the runtime peformance for both the accelerator and the CPU.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./02_aligned_malloc alveo_examples
```

The program will output a message similar to this:

```
-- Example 2: Vector Add with Aligned Allocation --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test with aligned virtual buffers

Simple malloc vadd example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  256.254 ms
Allocating memory buffer           :    0.055 ms
Populating buffer inputs           :   47.884 ms
Software VADD run                  :   35.808 ms
Map host buffers to OpenCL buffers :    9.103 ms
Memory object migration enqueue    :    6.615 ms
Set kernel arguments               :    0.014 ms
OCL Enqueue task                   :    0.116 ms
Wait for kernel to complete        :   92.110 ms
Read back computation results      :    2.479 ms
```

This seems at first glance to be much better! Let's compare these results to our results from **Example 1** to see how things have changed. Refer to table 3.2 for details, noting that we'll exclude minor run-to-run variation from the comparison to help keep things clean.

Table 3.2: Timing Summary - Example 2

Operation	Example 1	Example 2	$\Delta_{1\rightarrow2}$
OCL Initialization	247.371 ms	256.254 ms	-
Buffer Allocation	30 µs	55 µs	25 µs
Buffer Population	47.955 ms	47.884 ms	-
Software VADD	35.706 ms	35.808 ms	
Buffer Mapping	64.656 ms	9.103 ms	–55.553 ms
Write Buffers Out	24.829 ms	6.615 ms	–18.214 ms
Set Kernel Args	9 µs	14 µs	-
Kernel Runtime	92.118 ms	92.110 ms	-
Read Buffer In	24.887 ms	2.479 ms	–22.408 ms
$\Delta_{Alveo\rightarrow CPU}$	–418.228 ms	–330.889 ms	87.339 ms
$\Delta_{Alveo\rightarrow CPU}$ (algorithm only)	–170.857 ms	–74.269 ms	96.588 ms

Nice! By only changing four lines of code we've managed to shave nearly 100 ms off of our execution time. The CPU is still faster, but just by changing one minor thing about how we're allocating memory we saw huge improvement. That's really down to the memory copy that's needed for alignment; if we take a few extra microseconds to ensure the buffers are aligned when we allocate them, we can save orders of magnitude more time later when those buffers are consumed.

Also note that as expected in this use case, the software runtime is the same. We're changing the alignment of the allocated memory, but otherwise it's normal userspace memory allocation.

Extra Exercises

Some things to try to build on this experiment:

- Once again vary the size of the buffers allocated. Do the relationships that you derived in the previous example still hold true?
- Experiment with other methods of allocating aligned memory (not the OCL API). Do you see differences between the approaches, beyond minor run-to-run fluctuations?

Key Takeaways

- Unaligned memory will kill your performance. Always ensure buffers you want to share with the Alveo card are aligned.

Now we're getting somewhere! Let's try using the OpenCL API to allocate memory and see what happens.

Example 3: Memory Allocation with OpenCL

Overview

Ensuring that our allocated memory is aligned to page boundaries gave us a significant improvement over our initial configuration. There is another workflow we can use with OpenCL, though, which is to have OpenCL and XRT allocate the buffers and then map them to userspace pointers for use by the application. Let's experiment with that and see the effect it has on our timing.

Key Code

Conceptually this is a small change, but unlike **Example 2** this example is a bit more involved in terms of the required code changes. This is mostly because instead of using standard userspace memory allocation, we're going to ask the OpenCL runtime to allocate buffers for us. Once we have the buffers, we then neep to *map* them into userspace so that we can access the data they contain.

For our allocation, we change from listing 3.10 to listing 3.11.

Listing 3.11: Allocating Aligned Buffers with OpenCL

```
std::vector<cl::Memory> inBufVec, outBufVec;
cl::Buffer a_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer b_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer c_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_WRITE_ONLY |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
cl::Buffer d_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                           CL_MEM_ALLOC_HOST_PTR),
                 BUFSIZE * sizeof(uint32_t),
                 NULL,
                 NULL);
inBufVec.push_back(a_buf);
inBufVec.push_back(b_buf);
outBufVec.push_back(c_buf);
```

In this case we're allocating our OpenCL buffer objects significantly earlier in the program, and we also don't have userspace pointers yet. We can still, though, pass these buffer objects to enqueue `MigrateMemObjects()`

and other OpenCL functions. The backing storage is allocated at this point, we just don't have a userspace pointer to it.

The call to the `cl::Buffer` constructor looks very similar to what we had before. In fact, only two things have changed: we pass in the flag `CL_MEM_ALLOC_HOST_PTR` instead of `CL_MEM_USE_HOST_PTR` to tell the runtime that we want to *allocate* a buffer instead of using an *existing* buffer. We also no longer need to pass in a pointer to the user buffer (since we're allocating a new one), so we pass `NULL` instead.

We then need to *map* our OpenCL buffers to the userspace pointers to *a*, *b*, and *d* that we'll use immediately in software. There's no need to map a pointer to *c* at this time, we can do that later when we need to read from that buffer after kernel execution. We do this with the code in listing 3.12.

Listing 3.12: Mapping Allocating Aligned Buffers to Userspace Pointers

```
uint32_t *a = (uint32_t *)q.enqueueMapBuffer(a_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
uint32_t *b = (uint32_t *)q.enqueueMapBuffer(b_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
uint32_t *d = (uint32_t *)q.enqueueMapBuffer(d_buf,
                                             CL_TRUE,
                                             CL_MAP_WRITE | CL_MAP_READ,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
```

Once we perform the mapping, we can use the userspace pointers as normal to access the buffer contents. One thing to note, though, is that the OpenCL runtime does do *reference counting* of the opened buffers, so we need a corresponding call to `enqueueUnmapMemObject()` for each buffer that we map.

The execution flow through the kernel is the same, but we see something new when the time comes to migrate the input buffer back into the device. Rather than manually enqueueing a migration, we can instead just map the buffer. The OpenCL runtime will recognize that the buffer contents are currently resident in the Alveo device global memory and will take care of migrating the buffer back to the host for us. This is a coding style choice you must make, but fundamentally the code in listing 3.13 is sufficient to migrate *c* back to the host memory.

Listing 3.13: Mapping Kernel Output to a Userspace Pointer

```
uint32_t *c = (uint32_t *)q.enqueueMapBuffer(c_buf,
                                             CL_TRUE,
                                             CL_MAP_READ,
                                             0,
                                             BUFSIZE * sizeof(uint32_t));
```

Finally, as we mentioned earlier you need to *unmap* the memory objects so that they can be destroyed cleanly by the runtime. We do this at the end of the program instead of using `free()` on the buffers as before. This must be done before the command queue is finished, as in listing 3.14.

Listing 3.14: Unmapping OpenCL-Allocated Buffers

```
q.enqueueUnmapMemObject(a_buf, a);
q.enqueueUnmapMemObject(b_buf, b);
q.enqueueUnmapMemObject(c_buf, c);
q.enqueueUnmapMemObject(d_buf, d);
```

```
q.finish();
```

To summarize the key workflow for this use model, we need to:

1. (Lines 63-90): **Allocate** our buffers using the `CL_MEM_ALLOC_HOST_PTR` flag.
2. (Lines 94-108): **Map** our input buffers to userspace pointers to populate them.
3. Run the kernel as usual
4. (Lines 144-148): **Map** the output buffer(s) to migrate them back to host memory.
5. (Lines 181-185): **Unmap** all of our buffers once we're done using them so they can be destroyed properly.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./03_buffer_map alveo_examples
```

The program will output a message similar to this:

```
-- Example 3: Allocate and Map Contiguous Buffers --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test with XRT-allocated contiguous buffers

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  247.460 ms
Allocate contiguous OpenCL buffers :   30.365 ms
Map buffers to userspace pointers  :    0.222 ms
Populating buffer inputs           :   22.527 ms
Software VADD run                  :   24.852 ms
Memory object migration enqueue    :    6.739 ms
Set kernel arguments               :    0.014 ms
OCL Enqueue task                   :    0.102 ms
Wait for kernel to complete        :   92.068 ms
Read back computation results      :    2.243 ms
```

Table 3.3: Timing Summary - Example 3

Operation	Example 2	Example 3	$\Delta_{2\rightarrow3}$
OCL Initialization	256.254 ms	247.460 ms	-
Buffer Allocation	55 µs	30.365 ms	30.310 ms
Buffer Population	47.884 ms	22.527 ms	–25.357 ms
Software VADD	35.808 ms	24.852 ms	–10.956 ms
Buffer Mapping	9.103 ms	222 µs	–8.881 ms
Write Buffers Out	6.615 ms	6.739 ms	-
Set Kernel Args	14 µs	14 µs	-
Kernel Runtime	92.110 ms	92.068 ms	-
Read Buffer In	2.479 ms	2.243 ms	-
$\Delta_{Alveo\rightarrow CPU}$	–330.889 ms	–323.996 ms	–6.893 ms
$\Delta_{FPGA\rightarrow CPU}$ (algorithm only)	–74.269 ms	–76.536 ms	-

You may have expected a speedup here, but we see that rather than speeding up any particular operation, instead we've shifted the latencies in the system around. Effectively we've paid our taxes from a different bank account, but at the end of the day we can't escape them. On embedded systems with a unified memory map for the processor and the kernels we would see significant differences here, but on server-class CPUs we don't.

One thing to think about is that although pre-allocating the buffers in this way took longer, you don't generally want to allocate buffers in your application's critical path. By using this mechanism, the runtime *use* of your buffers is much faster.

You may even wonder why it's faster for the CPU to access this memory. While we haven't discussed it to this point, allocating memory via this API *pins* the virtual addresses to physical memory. This makes it more efficient for both the CPU and the DMA to access it. As with all things in engineering, though, this comes at a price - allocation time is higher and you run the risk of fragmenting the available memory if you allocate many small buffers.

In general, buffers should be allocated outside the critical path of your application and this method shifts the burden away from your high-performance sections if used correctly.

Extra Exercises

Some things to try to build on this experiment:

- Once again, vary the size of the buffers allocated. Do the relationships that you derived in the previous example still hold true?
- Experiment with other sequences for allocating memory and enqueuing transfers.
- What happens if you modify the input buffer and run the kernel a second time?

Key Takeaways

- Using the OpenCL and XRT APIs can lead to localized performance boosts, although fundamentally there's no escaping your taxes.

Our long pole is becoming our kernel runtime, though, and we can easily speed that up. Let's take a look at that in our next few examples.

Example 4: Parallelizing the Data Path

Overview

We've made some big gains in our overall system throughput, but the bottleneck is now very clearly the accelerator itself. Remember that there are two axes on which we can optimize a design: we can throw more resources at a problem to solve it faster, which is bound by **Amdahl's Law**, or we can do more fundamental operations in parallel following **Gustafson's Law**.

In this case we'll try optimizing along the axis of Amdahl's Law. Up until now our accelerator has been following basically the same algorithm as the CPU, performing one 32-bit addition on each clock tick. But, because the CPU has a significantly faster clock (and has the advantage of not having to transfer the data over PCIe) it's always been able to beat us. Now it's time to turn the tables.

Our DDR controller natively has a 512-bit wide interface internally. If we parallelize the data flow in the accelerator, that means we'll be able to process 16 array elements per clock tick instead of one. So, we should be able to get an instant 16x speed-up by just vectorizing the input.

Key Code

In this example, if we continue to focus on the host software and treat the kernel as a black box we wind up with code that's essentially identical to the code from **Example 3**. All we need to do is change the kernel name from `vadd` to `wide_vadd`.

Since the code is the same, this is a good opportunity to introduce another concept of working with memory in XRT and OpenCL. On the Alveo cards we have four memory banks available, and while it isn't necessary with these simple accelerators you may find that you wish to spread operations among them to help maximize the bandwidth available to each of the interfaces. For our simple vector addition example, just for illustration purposes we've used the topology shown in figure 3.3.

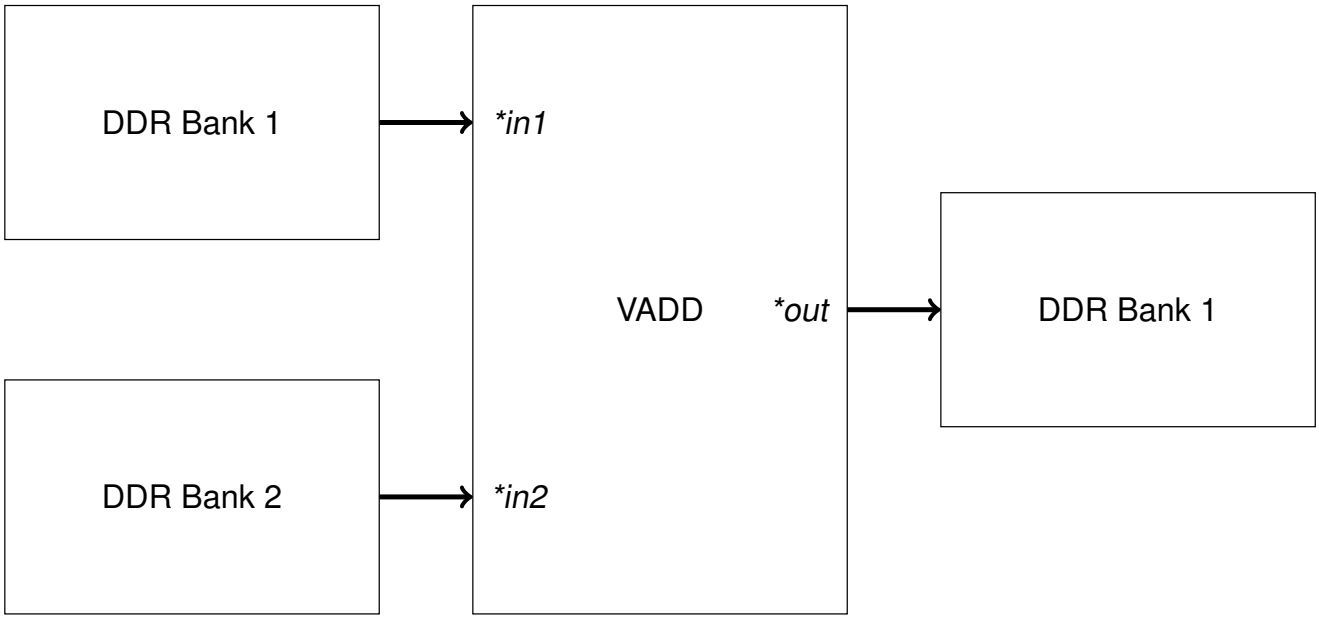

Figure 3.3: Wide VADD Memory Connectivity

What we gain by doing this is the ability to perform high-bandwidth transactions simultaneously with different external memory banks. Remember, long bursts are generally better for performance than many small reads and writes, but you can't fundamentally perform two operations on the memory at the same time.

It's easy enough to specify the hardware connectivity via command line switches, but when we want to actually transfer buffers down to the hardware we need to specify which memories to use for XRT. Xilinx accomplishes this by way of an extension to the standard OpenCL library, using a struct `cl_mem_ext_ptr_t` in combination with the buffer allocation flag `CL_MEM_EXT_PTR_XILINX`. This looks like the code in listing 3.15.

Listing 3.15: Specifying External Memory Banks with XRT

```
cl_mem_ext_ptr_t bank1_ext, bank2_ext;
bank2_ext.flags = 2 | XCL_MEM_TOPOLOGY;
bank2_ext.obj   = NULL;
bank2_ext.param = 0;
bank1_ext.flags = 1 | XCL_MEM_TOPOLOGY;
bank1_ext.obj   = NULL;
bank1_ext.param = 0;
cl::Buffer a_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank1_ext,
                 NULL);
cl::Buffer b_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_ONLY |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
                 &bank2_ext,
                 NULL);
cl::Buffer c_buf(context,
                 static_cast<cl_mem_flags>(CL_MEM_READ_WRITE |
                                           CL_MEM_EXT_PTR_XILINX),
                 BUFSIZE * sizeof(uint32_t),
```

```
                &bank1_ext,
                NULL);
```

You can see that this code is very similar to listing 3.11, with the difference being that we're passing our `cl_mem_ext_ptr_t` object to the `cl::Buffer` constructor and are using the *flags* field to specify the memory bank we need to use for that particular buffer. Note again that there is fundamentally no reason for us to do this for such a simple example, but since this example is so structurally similar to the previous one it seemed like a good time to mix it in. This can be a very useful technique for performance optimization for very heavy workloads.

Otherwise, aside from switching to the `wide_vadd` kernel the code here is exactly unchanged other than increasing the buffer size to 1 GiB. This was done to better accentuate the differences between the hardware and software.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./04_wide_vadd alveo_examples
```

The program will output a message similar to this:

```
-- Example 4: Parallelizing the Data Path --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
Running kernel test XRT-allocated buffers and wide data path:


OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  244.463 ms
Allocate contiguous OpenCL buffers :   37.903 ms
Map buffers to userspace pointers  :    0.333 ms
Populating buffer inputs           :   30.033 ms
Software VADD run                  :   21.489 ms
Memory object migration enqueue    :    4.639 ms
Set kernel arguments               :    0.012 ms
OCL Enqueue task                   :    0.090 ms
Wait for kernel to complete        :    9.003 ms
Read back computation results      :    2.197 ms
```

Table 3.4: Timing Summary - Example 3

Operation	Example 3	Example 4	$\Delta_{3\rightarrow4}$
OCL Initialization	247.460 ms	244.463 ms	-
Buffer Allocation	30.365 ms	37.903 ms	7.538 ms
Buffer Population	22.527 ms	30.033 ms	7.506 ms
Software VADD	24.852 ms	21.489 ms	–3.363 ms
Buffer Mapping	222 µs	333 µs	-
Write Buffers Out	6.739 ms	4.639 ms	-
Set Kernel Args	14 µs	12 µs	-
Kernel Runtime	92.068 ms	9.003 ms	–83.065 ms
Read Buffer In	2.243 ms	2.197 ms	-
$\Delta_{Alveo\rightarrow CPU}$	–323.996 ms	–247.892 ms	–76.104 ms
$\Delta_{FPGA\rightarrow CPU}$ (algorithm only)	–76.536 ms	5.548 ms	–82.084 ms

Mission accomplished - we've managed to beat the CPU!

There are a couple of interesting bits here, though. The first thing to notice is that, as you probably expected, the time required to transfer that data into and out of the FPGA hasn't changed. Like we disussed in earlier sections, that's effectively a fixed time based on your memory topology, amount of data, and overall system memory bandwidth utilization. You'll see some minor run-to-run variability here, especially in a virtualized environment like a cloud data center, but for the most part you can treat it like a fixed-time operation.

The more interesting one, though, is the achieved speedup. You might be surprised to see that widening the data path to 16 words per clock did not, in fact, result in a 16x speedup. The kernel itself processes 16 words per clock, but what we're seeing in the timed results is a cumulative effect of the external DDR latency. Each interface will burst in data, process it, and burst it out. But the inherent latency of going back and forth to DDR slows things down and we only actually achieve a 10x speedup over the single-word implementation.

Unfortunately, we're hitting a fundamental problem with vector addition: it's too easy. The computational complexity of vadd is $O(N)$, and a very simple $O(N)$ at that. As a result you very quickly become I/O bandwidth bound, not computation bound. With more complex algorithms, such as nested loops which are $O(N^x)$ or even computationally complex $O(N)$ algorithms like filters needing lots of calculations, we can achieve significantly higher acceleration by performing more computation inside the FPGA fabric instead of going out to the memory quite so often. The best candidate algorithms for acceleration have very little data transfer and lots of calculations.

We can also run another experiment with a larger buffer. Change the buffer size as in listing 3.16:

Listing 3.16: Larger Memory Sizes

```
#define BUFSIZE (1024 * 1024 * 256) // 256*sizeof(uint32_t) = 1 GiB
```

Rebuild and run it again, and now let's switch over to a simplified set of metrics as in table 3.5. I think you get the idea on OpenCL initialization by now, too, so we'll only compare the algorithm's performance. That doesn't

mean that this initialization time isn't important, but you should architect your application so it's a one-time cost that you pay during other initialization operations.

We will also stop tracking the buffer allocation and initialization times. Again this is something that you'd generally want to do when setting up your application. If you're routinely allocating large buffers in the critical path of your application, odds are you'd get more "bang for your buck" rethinking your architecture rather than trying to apply hardware acceleration.

Table 3.5: 1 GiB Buffer Timing Summary - Example 4

Operation	Example 4
Software VADD	820.596 ms
Buffer PCIe TX	383.907 ms
VADD Kernel	484.050 ms
Buffer PCIe RX	316.825 ms
Hardware VADD (Total)	1184.897 ms
$\Delta_{Alveo \rightarrow CPU}$	364.186 ms

Oh no! We had such a good thing going, and here we went and ruined it. With this larger buffer we're not beating the CPU anymore at all. What happened?

Looking closely at the numbers, we can see that while the hardware kernel itself is processing the data pretty efficiently, the data transfer to and from the FPGA is really eating up a lot of our time. In fact, if you were to draw out our overall execution timeline, it would look something like figure 3.4 at this point.

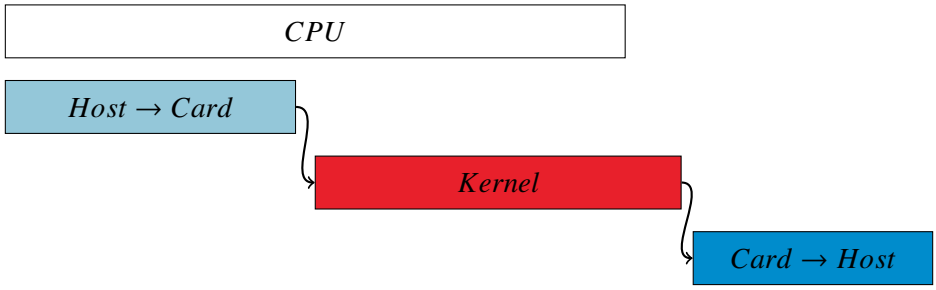

Figure 3.4: Kernel Execution Timeline

This chart is figurative, but to relative scale. Looking closely at the numbers, we see that in order to beat the CPU our entire kernel would need to run to completion in only about 120 ms. For that to work we would need to run four times faster (not likely), process four times as much data per clock, or run four accelerators in parallel. Which one should we pick? Let's leave that for the next example.

Extra Exercises

Some things to try to build on this experiment:

- Play around with the buffer sizes again. Can you find the inflection point where the CPU becomes faster?
- Try to capture the SDAccel Timeline Trace and view this for yourself. The instructions for doing so are found in SDAccel Environment Profiling and Optimization Guide (UG1207)

Key Takeaways

- Parallelizing the kernel can provide great results, but you need to take care that your data transfer doesn't start to dominate your execution time. Or if it does, that you stay under your target execution time.
- Simple computation isn't always a good candidate for acceleration, unless you're trying to free up the processor to work on other tasks. Better candidates for acceleration are complex workloads with high complexity, especially $O(N^x)$ algorithms like nested loops, etc.
- If you have to wait for all of the data to transfer before you begin processing you may have trouble hitting your overall performance target even if your kernels are highly optimized.

Now we've seen that just parallelizing the data path isn't quite enough. Let's see what we can do about those transfer times.

Example 5: Optimizing Compute and Transfer

Overview

Looking back at the previous example, once the data grew beyond a certain size the transfer of data into and out of our application started to become the bottleneck. Given that the transfer over PCIe will generally take a relatively fixed amount of time, you might be tempted to create four instances of the `wide_vadd` kernel and enqueue them all to chew through the large buffer in parallel.

This is actually the traditional GPU model - batch a bunch of data over PCIe to the high-bandwidth memory, and then process it in very wide patterns across an array of embedded processors.

In an FPGA we generally want to solve that problem differently. If we have a kernel capable of consuming a full 512 bits of data on each tick of the clock, our biggest problem isn't parallelization, it's feeding its voracious appetite. If we're bursting data to and from DDR we can very easily hit our bandwidth cap. Putting multiple cores in parallel and operating continually on the same buffer all but ensures we'll run into bandwidth contention. That will actually have the opposite of the desired effect - our cores will all run *slower*.

So what should we do? Remember that the `wide_vadd` kernel is actually a bad candidate for acceleration in the first place. The computation is so simple we can't really take advantage of the parallelism of an FPGA. Fundamentally there's not too much you can do when your algorithm boils down to $A + B$. But it does let us show a few techniques optimizing around something that's about as algorithmically simple as it gets.

To that end we've actually been a bit tricky with our hardware design for the `wide_vadd` kernel. Instead of just consuming the data raw off of the bus, we're buffering the data internally a bit using the FPGA block RAM (which is fundamentally an extremely fast SRAM) and then performing the computation. As a result we can absorb a little time between successive bursts, and we'll use that to our advantage.

Remember that we've split our interfaces across multiple DDR banks, as shown in figure 3.3). Since the PCIe bandwidth is much lower than the total bandwidth of any given DDR memory on the Alveo card, and we're transferring data to two different DDR banks, we know we'll have interleaving between the two. And so, because we can "squeeze in between the cracks" as it were, we can start processing the data *as it arrives* instead of waiting for it to completely transfer, processing it, and then transferring it back. We want to end up with something like figure 3.5.

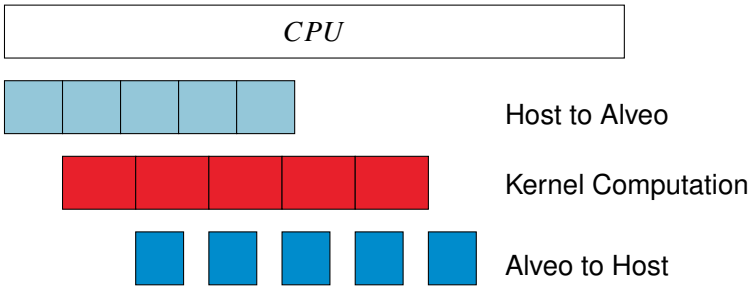

Figure 3.5: Kernel Execution Timeline (Optimized)

By subdividing the buffer in this way and choosing an optimal number of subdivisions, we can balance execution and transfer times for our application and get significantly higher throughput. Using the same hardware kernel,

let's take a look at what's required to set this up in the host code.

Key Code

The algorithmic flow of the code will be mostly the same. Before queueing up transfers, we'll loop through the buffer and subdivide it. We want to follow a few general rules, though. First, we want to try to divide the buffer on aligned boundaries to keep transfers efficient, and second we want to make sure we're not subdividing buffers when they're so small that it makes no sense. We'll define a constant **NUM_BUFS** to set our number of buffers, and then write a new function to subdivide them as in listing 3.17.

Listing 3.17: Subdividing XRT Buffers

```
int subdivide_buffer(std::vector<cl::Buffer> &divided,
                     cl::Buffer buf_in,
                     cl_mem_flags flags,
                     int num_divisions)
{
    // Get the size of the buffer
    size_t size;
    size = buf_in.getInfo<CL_MEM_SIZE>();

    if (size / num_divisions <= 4096) {
        return -1;
    }

    cl_buffer_region region;

    int err;
    region.origin = 0;
    region.size   = size / num_divisions;

    // Round region size up to nearest 4k for efficient burst behavior
    if (region.size % 4096 != 0) {
        region.size += (4096 - (region.size % 4096));
    }

    for (int i = 0; i < num_divisions; i++) {
        if (i == num_divisions - 1) {
            if ((region.origin + region.size) > size) {
                region.size = size - region.origin;
            }
        }
        cl::Buffer buf = buf_in.createSubBuffer(flags,
                                                CL_BUFFER_CREATE_TYPE_REGION,
                                                &region,
                                                &err);
        if (err != CL_SUCCESS) {
            return err;
        }
        divided.push_back(buf);
        region.origin += region.size;
    }

    return 0;
}
```

What we're doing here is looping through the buffer(s) NUM_BUFS times, calling `cl::Buffer.createSubBuffer()` for each sub-buffer we want to create. The `cl_buffer_region` struct defines the start address and size of the sub-buffer we want to create. It's important to note that sub-buffers can overlap, although in our case we're not using them in that way.

We return a vector of `cl::Buffer` objects that we can then use to enqueue multiple operations, as in listing 3.18.

Listing 3.18: Enqueueing Subdivided XRT Buffers

```
int enqueue_subbuf_vadd(cl::CommandQueue &q,
                        cl::Kernel &krnl,
                        cl::Event &event,
                        cl::Buffer a,
                        cl::Buffer b,
                        cl::Buffer c)
{
    // Get the size of the buffer
    cl::Event k_event, m_event;
    std::vector<cl::Event> krnl_events;

    static std::vector<cl::Event> tx_events, rx_events;

    std::vector<cl::Memory> c_vec;
    size_t size;
    size = a.getInfo<CL_MEM_SIZE>();

    std::vector<cl::Memory> in_vec;
    in_vec.push_back(a);
    in_vec.push_back(b);
    q.enqueueMigrateMemObjects(in_vec, 0, &tx_events, &m_event);
    krnl_events.push_back(m_event);
    tx_events.push_back(m_event);
    if (tx_events.size() > 1) {
        tx_events[0] = tx_events[1];
        tx_events.pop_back();
    }

    krnl.setArg(0, a);
    krnl.setArg(1, b);
    krnl.setArg(2, c);
    krnl.setArg(3, (uint32_t)(size / sizeof(uint32_t)));

    q.enqueueTask(krnl, &krnl_events, &k_event);
    krnl_events.push_back(k_event);
    if (rx_events.size() == 1) {
        krnl_events.push_back(rx_events[0]);
        rx_events.pop_back();
    }
    c_vec.push_back(c);
    q.enqueueMigrateMemObjects(c_vec,
                               CL_MIGRATE_MEM_OBJECT_HOST,
                               &krnl_events,
                               &event);
    rx_events.push_back(event);

    return 0;
}
```

In listing 3.18 we're doing basically the same sequence of events that we had before:

1. Enqueue migration of the buffer from the host memory to the Alveo memory.
2. Set the kernel arguments to the current buffer.
3. Enqueue the run of the kernel.
4. Enqueue a transfer back of the results.

The difference, though, is that now we're doing them in an actual queued, sequential fashion. We aren't waiting for the events to fully complete now, as we were in previous examples, because that would defeat the whole purpose of pipelining. So now we're using *event-based dependencies*. By using `cl::Event` objects we can build a chain of events that *must* complete before any subsequent chained events (non-linked events can still be scheduled at any time).

We enqueue multiple runs of the kernel and then wait for all of them to complete, which will result in much more efficient scheduling. Note that if we had built the same structure as in **Example 4** using this queuing method we'd see the same results as then, because the runtime has no way of knowing whether or not we can safely start processing before sending all of the data. As designers we have to tell the scheduler what can and can not be done.

And, finally, none of this would happen in the correct sequence if we didn't do one more very important thing: we have to specify that we can use an *out of order command queue* by passing in the flag `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE` when we create it.

The code in this example should otherwise seem familiar. We now call those functions instead of calling the API directly from `main()`, but it's otherwise unchanged.

There *is* something interesting, though, about mapping buffer C back into userspace - we don't have to work with individual sub-buffers. Because they've already been migrated back to host memory, and because when creating sub-buffers the underlying pointers don't change, we can still work with the parent even though we have children (and the parent buffer somehow even manages to sleep through the night!).

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./05_pipelined_vadd alveo_examples
```

The program will output a message similar to this:

```
-- Example 5: Pipelining Kernel Execution --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'


 -- Running kernel test with XRT-allocated contiguous buffers
    and wide VADD (16 values/clock)

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  263.001 ms
Allocate contiguous OpenCL buffers :  915.048 ms
Map buffers to userspace pointers  :    0.282 ms
Populating buffer inputs           : 1166.471 ms
Software VADD run                  : 1195.575 ms
Memory object migration enqueue    :    0.441 ms
Wait for kernels to complete       :  692.173 ms
```

Table 3.6 shows these results compared to the previous run:

Table 3.6: 1 GiB Buffer Timing Summary - Pipelined vs. Sequential

Operation	Example 4	Example 5	$\Delta_{4\rightarrow5}$
Software VADD	820.596 ms	1166.471 ms	345.875 ms
Hardware VADD (Total)	1184.897 ms	692.172 ms	–492.725 ms
$\Delta_{Alveo\rightarrow CPU}$	364.186 ms	–503.402 ms	867.588 ms

Mission accomplished *for sure* this time. Look at those margins!

There's no way this would turn around on us now, right? Let's sneak out early - I'm sure there isn't an "other shoe" that's going to drop.

Extra Exercises

Some things to try to build on this experiment:

- Play around with the buffer sizes again. Is there a similar inflection point in this exercise?

- Capture the traces again too; can you see the difference? How does the choice of the number of sub-buffers impact runtime (if it does)?

Key Takeaways

- Intelligently managing your data transfer and command queues can lead to significant speedups.

Example 6: Meet the Other Shoe

Overview

You didn't think you were getting out of here quite so fast, did you? As I said at the beginning: *vadd will never beat the processor*. It's too simple; if you don't have to transfer data and you can burn through local cache, the CPU will *always* win in the end, assuming you haven't hauled your uncle's dusty 386 Turbo out of the basement to run this test.

The results from the previous session look good - on paper. I get it, I'm in marketing. I want to call it a day. But I seem to recall also having a degree with the word "engineering" on it gathering dust somewhere, so - as a fellow engineer - let me break it to you.

Key Code

For simple algorithms, an accelerator just won't win. Let's use OpenMP to parallelize the processor loop. We include the header `omp.h`, and then apply an OpenMP pragma to the CPU code as in listing 3.19.

Listing 3.19: SParallelize the Software with OpenMP

```
void vadd_sw(uint32_t *a, uint32_t *b, uint32_t *c, uint32_t size)
{
#pragma omp parallel for
    for (int i = 0; i < size; i++) {
        c[i] = a[i] + b[i];
    }
}
```

And that's it. There are some command line flags to pass to GCC, but CMake will take care of those (assuming you have OpenMP installed), so we can directly build and run. The code for this example is otherwise identical to the code from **Example 5**.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./06_wide_processor alveo_examples
```

The program will output messages similar to this:

```
-- Example 6: VADD with OpenMP --

Loading XCLBin to program the Alveo board:

Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'


 -- Running kernel test with XRT-allocated contiguous buffers
    and wide VADD (16 values/clock), with software OpenMP

OCL-mapped contiguous buffer example complete!

--------------- Key execution times ---------------
OpenCL Initialization              :  253.898 ms
Allocate contiguous OpenCL buffers :  907.183 ms
Map buffers to userspace pointers  :    0.307 ms
Populating buffer inputs           : 1188.315 ms
Software VADD run                  :  157.226 ms
Memory object migration enqueue    :    1.429 ms
Wait for kernels to complete       :  618.231 ms
-- Example 5: Pipelining Kernel Execution --
```

Table 3.7 shows these results compared to the previous run:

Table 3.7: 1 GiB Buffer Timing Summary - Sequential vs. OpenMP

Operation	Example 5	Example 6	$\Delta_{5\rightarrow6}$
Software VADD	1166.471 ms	157.226 ms	−1009.245 ms
Hardware VADD (Total)	692.172 ms	618.231 ms	−73.94 ms
$\Delta_{Alveo\rightarrow CPU}$	−503.402 ms	461.005 ms	964.407 ms

The accelerator runtime fluctuation is primarily a result of running these tests in a virtualized cloud environment, but that's not the point of the exercise.

Extra Exercises

Some things to try to build on this experiment:

- Try to beat the processor at vector addition!
- Play with the OpenMP pragmas; how many CPU cores are needed to beat a hardware accelerator?

Key Takeaways

- I've said it before and I'll say it again: simple $O(N)$ will never win.

But despair not! Now it's time to look at something real.

Example 7: Image Resizing with OpenCV

Overview

Image processing is a great area for acceleration in an FPGA, and there are a few reasons for it. First and foremost, if you're doing any kind of pixel-level processing as the images get larger and larger, the amount of computation goes up with it. But, more importantly, they map very nicely to our train analogy from earlier.

Let's look at a very simple example: a bilateral resize algorithm. This algorithm takes an input image and scales it to a new, arbitrary resolution. The steps to do that might look something like this:

1. Read the pixels of the image from memory.
2. If necessary, convert them to the proper format. In our case we'll be looking at the default format used by the OpenCV library, BGR. But in a real system where you'd be receiving data from various streams, cameras, etc. you'd have to deal with formatting, either in software or in the accelerator (where it's basically a "free" operation, as we'll see in the next example).
3. For color images, extract each channel.
4. Use a bilateral resizing algorithm on each independent channel.
5. Recombine the channels and store back in memory.

Or, if you prefer to think of it visually, the operation would look something like figure 3.6:

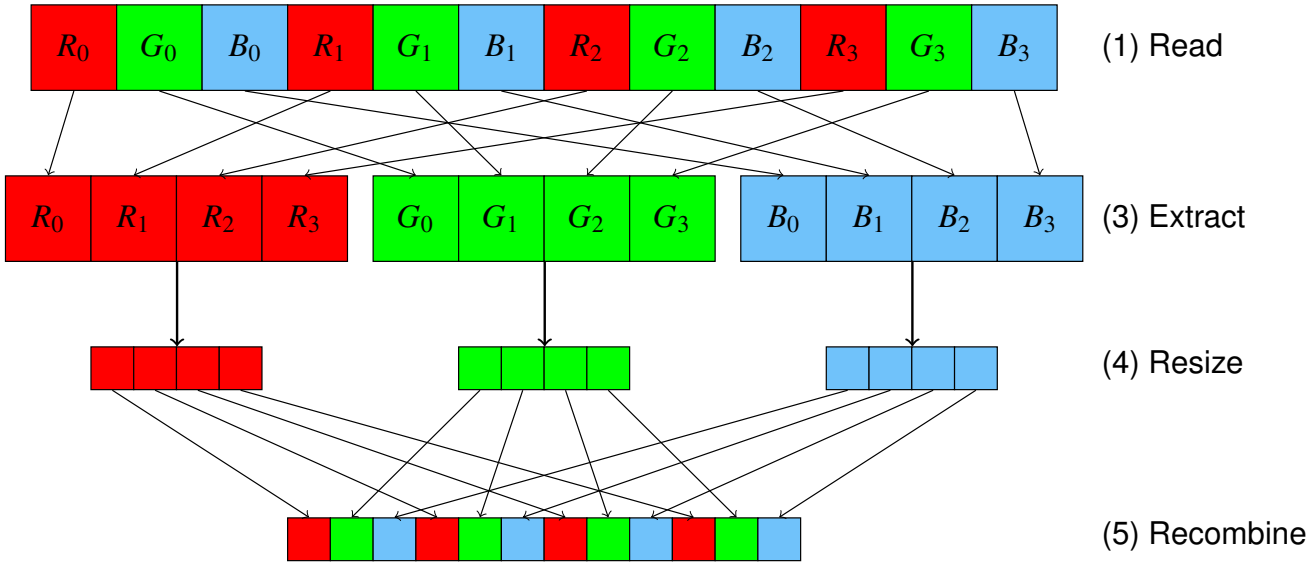

Figure 3.6: Resize Algorithm Dataflow

For the purposes of this diagram we haven't shown the optional color conversion step.

The interesting thing to note here is that these operations happen one after another *sequentially*. Remember from our discussion earlier that going back and forth to memory is expensive. If we can perform all of these calculations in the FPGA as we stream the data in from memory, then the extra latency of each individual

step or calculation will generally be extremely small. We can use that to our advantage, building custom *domain-specific architectures* that make optimal use of the hardware for the specific function(s) we're trying to implement.

You can then widen the data path to process more pixels per clock - **Amdahl's Law** - or you can process many images in parallel with multiple pipelines - **Gustafson's Law**. Ideally, you'll optimize on both axes: build a kernel that can process your algorithm as efficiently as possible, and then put as many kernels as you can take advantage of in the FPGA fabric.

Key Code

For this algorithm we'll make use of the Xilinx **xf::OpenCV** libraries. These are hardware-optimized libraries implementing many commonly-used OpenCV functions that you can directly use in your applications. We can also mix and match them with software OpenCV functions or other library calls as needed.

We can also use these libraries for image pre-and post-processing for other kernels. For example, we might want to take raw data from a camera or network stream, pre-process it, feed the results to a neural network, and then do something with the results. All of that can be done on the FPGA without needing to go back to the host memory at all, and all of these operations can be done in parallel. Imagine building a pipelined stream of functionality whose only fundamental contention is bandwidth, not register space.

In the xf::OpenCV library, you configure things such as the number of pixels to process per clock, etc. via templates. I won't go into detail here, but please refer to Xilinx OpenCV User Guide (UG1233) for more information, or the online OpenCV to xf::OpenCV tutorial available here:

SDSoC Environment Tutorial: Migrate OpenCV to xfOpenCV

Please note that this tutorial specifically refers to the Xilinx's embedded acceleration tool SDSoC, but the concepts and techniques apply equally to your Alveo board when it comes to using the OpenCV library to write your kernels.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./07_opencv_resize alveo_examples <path_to_image>
```

Because of the way we've configured the hardware in this example, your image must conform to certain requirements. Because we're processing eight pixels per clock, your input width must be a multiple of eight. And because we're bursting data on a 512-bit wide bus, your input image needs to satisfy the requirement:

$$in\_width \times out\_width \times 24 \ \% \ 512 = 0$$

If it doesn't then the program will output an error message informing you which condition was not satisfied. This is of course not a fundamental requirement of the library; we can process images of any resolution and other numbers of pixels per clock. But, for optimal performance if you can ensure the input image meets certain requirements you can process it significantly faster.

In addition to the resized images from both the hardware and software OpenCV implementations, the program will output messages similar to this:

```
-- Example 7: OpenCV Image Resize --

OpenCV conversion done! Image resized 1920x1080 to 640x360
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :    5.145 ms
OpenCL initialization            :  292.673 ms
OCL input buffer initialization  :    4.629 ms
OCL output buffer initialization :    0.171 ms
FPGA Kernel resize operation     :    4.951 ms
```

Because we're now doing something fundamentally different, we won't compare our results to the previous run. But we *can* compare the time to process our image on the CPU vs. the time it takes to process the image on the Alveo card.

For now we can notice that the Alveo card and the CPU are approximately tied. The bilinear interpolation algorithm we're using is still $O(N)$, but this time it's not quite so simple a calculation. As a result we're not as I/O bound as before; we can beat the CPU, but only just.

One interesting thing here is that we're performing a number of computations based on the *output* resolution, not the input resolution. We need to transfer in the same amount of data, but instead of converting the image to $1/3$ its input size, let's instead double the resolution and see what happens. Change the code for the example as in listing 3.20 and recompile.

Listing 3.20: Double Resolution

```
uint32_t out_width  = image.cols * 2;
uint32_t out_height = image.rows * 2;
```

Running the example again, we see something interesting:

```
-- Example 7: OpenCV Image Resize --

OpenCV conversion done! Image resized 1920x1080 to 3840x2160
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :   11.692 ms
OpenCL initialization            :  256.933 ms
OCL input buffer initialization  :    3.536 ms
OCL output buffer initialization :    7.911 ms
FPGA Kernel resize operation     :    6.844 ms
```

Let's set aside, for a moment, the buffer initializations - usually in a well-architected system you're not performing memory allocation in your application's critical path. We're now, just on the resize operation and including our data transfer, nearly 60% faster than the CPU.

While both the CPU and the Alveo card can process this image relatively quickly, imagine that you were trying to hit an overall system-level throughput goal of 60 frames per second. In that scenario you only have 16.66 ms to process each frame. If you were, for example, using the resized frame as an input into a neural network you can see that you would have almost used all of your time already.

If you're intending to pre-allocate a chain of buffers to run your pipeline then. we've now regained nearly 30% of our per-frame processing time by using an accelerator.

For a 1920x1200 input image, we get the results shown in table 3.8.

Table 3.8: Image Resize Timing Summary - Alveo vs. OpenCV

Operation	Scale Down	Scale Up
Software Resize	5.145 ms	11.692 ms
Hardware Resize	4.951 ms	6.684 ms
$\Delta_{Alveo \rightarrow CPU}$	−194 µs	−5.008 ms

And the best part: remember that color conversion step I mentioned earlier? In an FPGA that's basically a free operation - we get a few extra clock cycles of latency, but it's fundamentally another $O(N)$ operation, and a simple one at that with just a few multiplications and additions on the pixel values.

Extra Exercises

Some things to try to build on this experiment:

- Edit the host code to play with the image sizes. How does the run time change if you scale up more? Down more? Where is the crossover point where it no longer makes sense to use an accelerator?

- Add in color conversion to the FPGA accelerator (note that a hardware rebuild will be required). See if it takes longer to process.

Key Takeaways

- More computationally complex $O(N)$ operations *can* be good candidates for acceleration, although you won't see amazing gains vs. a CPU.
- Using FPGA-optimized libraries like xf::OpenCV allows you to easily trade off processing speed vs. resources, without having to re-implement common algorithms on your own. Focus on the interesting parts of your application!
- Some library optimizations you can choose can impose constraints on your design - double check the documentation for the library function(s) you choose *before* you implement the hardware.

We mentioned briefly earlier that doing additional processing in the FPGA fabric is very "cheap", time-wise. In the next section let's do an experiment to see if that's true.

Example 8: Pipelining Operations with OpenCV

Overview

We looked at a straightforward bilateral resize algorithm in the last example. While we saw that it wasn't an *amazing* candidate for acceleration, perhaps you might want to simultaneously convert a buffer to a number of different resolutions (say for different machine learning algorithms). Or, you might just want to offload it to save CPU availability for other processing during a frame window.

But, let's explore the real beauty of FPGAs: streaming. Remember that going back and forth to memory is expensive, so instead of doing that let's just send each pixel of the image along to another image processing pipeline stage without having to go back to memory by simply streaming from one operation to the next.

In this case, we want to amend our earlier sequence of events to add in a **Gaussian Filter**. This is a very common pipeline stage to remove noise in an image before an operation such as edge detection, corner detection, etc. We may even intend to add in some 2D filtering afterwards, or some other algorithm.

So, modifying our workflow from before, we now have:

1. Read the pixels of the image from memory.
2. If necessary, convert them to the proper format. In our case we'll be looking at the default format used by the OpenCV library, BGR. But in a real system where you'd be receiving data from various streams, cameras, etc. you'd have to deal with formatting, either in software or in the accelerator (where it's basically a "free" operation, as we'll see in the next example).
3. For color images, extract each channel.
4. Use a bilateral resizing algorithm on each independent channel.
5. Perform a Gaussian blur on each channel.
6. Recombine the channels and store back in memory.

So, we now have two "big" algorithms: bilateral resize and Gaussian blur. For a resized image of $w_{out} \times h_{out}$, and a square gaussian window of width k, our computation time for the entire pipeline would be roughly:

$$O(w_{out} \cdot h_{out}) + O(w_{out} \cdot h_{out} \cdot k^2)$$

For fun, let's make k relatively large without going overboard: we'll choose a 7×7 window.

Key Code

For this algorithm we'll continue to use of the Xilinx **xf::OpenCV** libraries.

As before, we'll configure the library (in the hardware, via templates in our hardware source files) to process eight pixels per clock cycle. Functionally, our hardware algorithm is now equivalent to listing 3.21 in standard OpenCV:

Listing 3.21: Example 8: Bilateral Resize and Gaussian Blur

```
cv::resize(image, resize_ocv, cv::Size(out_width, out_height), 0, 0, CV_INTER_LINEAR);
cv::GaussianBlur(resize_ocv, result_ocv, cv::Size(7, 7), 3.0f, 3.0f, cv::BORDER_CONSTANT);
```

We're resizing as in **Example 7**, and as we mentioned are applying a 7×7 window to the Gaussian blur function. We have also (arbitrarily) selected $\sigma_x = \sigma_y = 3.0$.

Running the Application

With the XRT initialized, run the application by running the following command from the build directory:

```
./08_opencv_resize alveo_examples <path_to_image>
```

As before, because of the way we've configured the hardware in this example, your image must conform to certain requirements. Because we're processing eight pixels per clock your input width must be a multiple of eight, and because we're bursting data on a 512-bit wide bus your input image needs to satisfy the requirement:

$$in\_width \times out\_width \times 24 \ \% \ 512 = 0$$

If it doesn't then the program will output an error message informing you which condition was not satisfied. And, again, this is *not* a fundamental requirement of the library; we can process images of any resolution and other numbers of pixels per clock. But for optimal performance, you can process it significantly faster if you can ensure the input image meets certain requirements.

In addition to the resized images from both the hardware and software OpenCV implementations, the program will output messages similar to this:

```
-- Example 8: OpenCV Image Resize and Blur --

OpenCV conversion done! Image resized 1920x1080 to 640x360 and blurred 7x7!
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :    7.170 ms
OpenCL initialization            :  275.349 ms
OCL input buffer initialization  :    4.347 ms
OCL output buffer initialization :    0.131 ms
FPGA Kernel resize operation     :    4.788 ms
```

In the previous example the CPU and the FPGA were pretty much tied for the small example. But while we've added a significant processing time for the CPU functions, the FPGA runtime hasn't increased much at all!

Let's now double the input size, going from a 1080p image to a 4k image. Change the code for this example, as we did with **Example 7** in listing 3.20) and recompile.

Running the example again, we see something *very* interesting:

```
-- Example 8: OpenCV Image Resize and Blur --

OOpenCV conversion done! Image resized 1920x1080 to 3840x2160 and blurred 7x7!
Starting Xilinx OpenCL implementation...
Matrix has 3 channels
Found Platform
Platform Name: Xilinx
XCLBIN File Name: alveo_examples
INFO: Importing ./alveo_examples.xclbin
Loading: './alveo_examples.xclbin'
OpenCV resize operation          :  102.977 ms
OpenCL initialization            :  250.000 ms
OCL input buffer initialization  :    3.473 ms
OCL output buffer initialization :    7.827 ms
FPGA Kernel resize operation     :    7.069 ms
```

What wizardry is this!? The CPU runtime has increased by nearly 10x, but the FPGA runtime has barely moved at all!

FPGAs are *really good* at doing things in parallel. This algorithm isn't I/O bound, it's processor bound. We can decompose it to process more data faster (**Amdahl's Law**) by calculating multiple pixels per clock, and by streaming from one operation to the next and doing more operations in parallel (**Gustafson's Law**). We can even decompose the Gaussian Blur into individual component calculations and run those in parallel (which we have done, in the xf::OpenCV library).

Now that we're bound by *computation* and not *bandwidth* we can easily see the benefits of acceleration. If we put this in terms of FPS, our x86-class CPU instance can now process 9 frames per second while our FPGA card can handle a whopping 141. And adding additional operations will continue to bog down the CPU, but so long as you don't run out of resources in the FPGA you can effectively continue this indefinitely. In fact, our kernels are still quite small compared to the resource availability on the Alveo U200 card.

To compare it to the previous example, again for a 1920x1200 input image, we get the results shown in table 3.9. The comparison column will compare the "Scale Up" results from **Example 7** with the scaled up results from this example.

Table 3.9: Image Resize and Blur Timing Summary - Alveo vs. OpenCV

Operation	Scale Down	Scale Up	$\Delta_{7\rightarrow8}$
Software Resize	7.170 ms	102.977 ms	91.285 ms
Hardware Resize	4.788 ms	7.069 ms	385 µs
$\Delta_{Alveo\rightarrow CPU}$	−2.382 ms	−95.908 ms	−90.9 ms

We hope you can see the advantage!

Extra Exercises

Some things to try to build on this experiment:

- Edit the host code to play with the image sizes. How does the run time change if you scale up more? Down more? Where is the crossover point where it no longer makes sense to use an accelerator in this case?
- Is the extra hardware latency

Key Takeaways

- Pipelining and streaming from one operation to the next in the fabric is beneficial.
- Using FPGA-optimized libraries like xf::OpenCV allows you to easily trade off processing speed vs. resources, without having to re-implement common algorithms on your own. Focus on the interesting parts of your application!
- Some library optimizations you can choose can impose constraints on your design - double check the documentation for the library function(s) you choose *before* you implement the hardware!

Summary

We hope that you've enjoyed our guided tour through the care and feeding of your Alveo care! Of course, this tutorial series can't possibly teach everything there is to know about this subject - in fact, we've barely scratched the surface. For more information on how to build kernels for Alveo we highly recommend continuing on with some of our hardware kernel focused tutorials:

SDAccel Development Environment Tutorials

SDAccel Environment Optimization Guide

For libraries:

Xilinx OpenCV User Guide (UG1233)

SDSoC Environment Tutorial: Migrate OpenCV to xfOpenCV

Of course, please make use of the Xilinx forums! The forums are frequented by a number of highly-skilled and responsive community members from both the Xilinx team and SDAccel enthusiasts worldwide.

For hardware questions regarding the Alveo card specifically:

Alveo Data Center Acceleration Cards Forum

And for the development toolchain:

SDAccel Forum

Xilinx also has forums for machine learning. Log in here for discussions on our Deep-Learning Processing Unit and its associated toolchian, the Deep Neural Network Development Kit:

DPU and DNNDK Forum

Thank you for taking the time to go through these tutorials. We sincerely hope they were helpful, and we're looking forward to working with you!

Appendix A

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.

Please Read: Important Legal Notices

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.