REUSABLE OPENCL FPGA INFRASTRUCTURE

by

S. Alexander Chin

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2012 by S. Alexander Chin
Abstract

Reusable OpenCL FPGA Infrastructure

S. Alexander Chin

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2012

OpenCL has emerged as a standard programming model for heterogeneous systems. Recent work combining OpenCL and FPGAs has focused on high-level synthesis. Building a complete OpenCL FPGA system requires more than just high-level synthesis. This work introduces a reusable OpenCL infrastructure for FPGAs that complements previous work and specifically targets a key architectural element - the memory interface. An Aggregating Memory Controller that aims to maximize bandwidth to external, large, high-latency, high-bandwidth memories and a template Processing Array with soft-processor and hand-coded hardware elements are designed, simulated, and implemented on an FPGA. Two micro-benchmarks were run on both the soft-processor elements and the hand-coded hardware elements to exercise the Aggregating Memory Controller. The micro-benchmarks were simulated as well as implemented in a hardware prototype. Memory bandwidth results for the system show that the external memory interface can be saturated and the high-latency can be effectively hidden using the Aggregating Memory Controller.
Acknowledgements

I would first like to thank my supervisor, Prof. Paul Chow, for his guidance throughout the past two and a half years. I am very grateful for the freedom given to me during this time to explore on my own, learning and growing as a researcher; as Ms. Frizzle says, “Take chances! Make mistakes! Get messy!” I have gotten messy and have learnt so much.

Thank you to everyone in our research group. All of you have, in numerous ways, supported me throughout my endeavours with your knowledge, experience, and friendship. I would like to especially thank Arun for imparting upon me sound advice whenever needed; Vince and Jasmina for their support when things were not the best.

My friends and new found friendships, you have all, in your own individual ways, helped to keep my life in balance. Thank you.

Thank you Mark and Bill. If it weren’t for the both of you in undergrad, I may not have made the decision to pursue further education.

Clara, your companionship and love throughout has been truly wonderful. Thank you for your patience, understanding, and care during all of the busy times.

Lastly, I would like to thank my family; my parents, Sharon and Stephen; and my siblings, Rebecca and Nicholas. I know that your unyielding love and support will continue with me throughout all of my endeavours.
## Contents

1 Introduction .......................................................... 1
   1.1 Contributions .................................................. 2
   1.2 Thesis Organization .......................................... 3

2 Programming Model ...................................................... 4
   2.1 OpenCL .......................................................... 4
      2.1.1 OpenCL Sub-Models ....................................... 4
      2.1.2 OpenCL FPGA Model ...................................... 9

3 Background ............................................................... 14
   3.1 Memory Access Models ......................................... 14
      3.1.1 Caching ..................................................... 14
      3.1.2 Streaming .................................................. 15
      3.1.3 Multi-Ported Memory Controllers ....................... 16
      3.1.4 Scalar versus Vector Memory Access .................. 16
   3.2 Previous OpenCL-FPGA Work .................................... 16
      3.2.1 FCUDA ...................................................... 17
      3.2.2 SOpenCL ................................................... 17
      3.2.3 OpenRCL .................................................. 18
      3.2.4 FSM SYS Builder .......................................... 18
      3.2.5 O4F ........................................................ 19
3.2.6 Relationship to Previous Work ............................................. 19

4 System Architecture ......................................................... 21

4.1 Overview ........................................................................... 21

4.2 Xilinx MIG SDRAM Controller ........................................... 21

4.3 Aggregating Memory Controller ......................................... 23

4.3.1 Key Modules .................................................................. 24

4.3.2 Split Datapath Design ..................................................... 27

4.3.3 Unified Datapath Design .................................................. 29

4.3.4 Scalability ..................................................................... 31

4.4 Processing Array ............................................................. 31

4.4.1 Soft-Processor Elements ............................................... 34

4.4.2 Hardware Cores ............................................................ 35

4.5 Device Controller ............................................................. 35

5 Methodology & Evaluation .................................................... 37

5.1 Hardware Platform .......................................................... 37

5.2 CAD Tool Flows ............................................................... 37

5.2.1 Hardware ...................................................................... 38

5.2.2 Simulation ..................................................................... 38

5.2.3 Soft-Processor Compilation Flow .................................... 40

5.3 Micro-Benchmarks ............................................................ 40

5.3.1 Vector Addition ............................................................ 41

5.3.2 Matrix Multiply ............................................................ 41

5.4 Performance ..................................................................... 41

5.4.1 Soft-Processor Overhead ............................................... 44

5.4.2 Effective AMC Memory Bandwidth ................................. 44

5.4.3 Unified AMC Bandwidth Improvements ......................... 48
List of Tables

5.1 Split AMC: Cycle Count ........................................... 42
5.2 Unified AMC: Cycle Count ...................................... 43
5.3 Split AMC: Effective Micro-Benchmark Bandwidth (MB/s) ........ 43
5.4 Unified AMC: Effective Micro-Benchmark Bandwidth (MB/s) .... 44
5.5 Processing Array Area ............................................ 52
# List of Figures

2.1 Heterogeneous OpenCL Platform Model .............................................. 5
2.2 Grid of threads showing work-group subdivisions and Compute Units each
with 16 Virtual Scalar Processors ......................................................... 7
2.3 OpenCL Memory Model showing Host Memory and Global, Constant,
Local, and Private Device Memory ....................................................... 7
2.4 OpenCL FPGA Model and the three major components required for a
complete OpenCL-FPGA system .............................................................. 10
3.1 Scalar- and Vector-threaded Memory Access ........................................... 17
4.1 System Overview .................................................................................. 22
4.2 Eight-Ported Bose-Nelson Sorting Network ........................................... 25
4.3 Eight-Ported Read Reorder Network ..................................................... 26
4.4 Eight-Ported Write Reorder Network .................................................... 27
4.5 Split Datapath AMC ............................................................................ 28
4.6 AMC Port ............................................................................................ 30
4.7 Unified Datapath AMC .......................................................................... 30
4.8 Generic Processing Element Interface .................................................. 33
4.9 Scalar- to Vector-threaded Memory Access Transformation ....................... 34
4.10 Soft-Processor Element ....................................................................... 35
4.11 Device Controller ............................................................................... 36
5.1 Hardware Flow .................................................. 39
5.2 Simulation Flow .................................................. 39
5.3 Split AMC: Vector Addition Effective Memory Bandwidth .... 45
5.4 Unified AMC: Vector Addition Effective Memory Bandwidth .... 46
5.5 Bandwidth Comparison of Split and Unified AMC: 512 Element Vector Addition ................................................. 48
5.6 Bandwidth Comparison of Split and Unified AMC: 1024 Element Vector Addition ................................................. 49
5.7 Bandwidth Comparison of Split and Unified AMC: 2048 Element Vector Addition ................................................. 50
Glossary

API  Application Programming Interface.

BRAM  Block Random Access Memory.

DIMM  Dual In-line Memory Module.

DMA  Direct Memory Access.

EDIF  Electronic Design Interchange Format. a vendor-neutral netlist format.

FIFO  First-In-First-Out queue.

FSB  Front-Side Bus.

GPGPU  General-Purpose computing on Graphics Processing Units.

HDL  Hardware Description Language.

HLS  High-Level Synthesis.

HyperTransport  is a standard for interconnecting computer processors.

memcpy  short form for memory copy.

MPSoc  Multi-Processor System-on-Chip.

NGC  Xilinux-specific netlist format.
PAR Place and Route.

Partial-Reconfiguration a scheme in which select regions of an FPGA are reconfigured, usually at runtime.

PCIe Peripheral Component Interconnect Express.

QPI Intel QuickPath Interconnect.

SDRAM Synchronous Dynamic Random Access Memory.

SoC System-on-Chip.

SODIMM Small-Outline Dual In-line Memory Module.

SREC is an ASCII hexadecimal text encoding for binary data.

UART Universal Asynchronous Receiver/Transmitter.
Chapter 1

Introduction

Programming models assist developers in creating high performance computing systems by forming a higher level abstraction of the target platform. Having a standard programming model across platforms allows developers to reuse prior knowledge when switching between platforms. OpenCL has emerged as a standard programming model for heterogeneous systems. Using it to program FPGAs is a logical step forward. The standard was originally developed towards CPU and GPGPU systems but recently there has been much activity combining OpenCL and FPGAs.

One main area of research has been focused on using OpenCL as a language for high-level synthesis (HLS). This is understandable, since parallelism is inherently expressed in OpenCL Kernels, as opposed to C, C++ or SystemC. However, a complete OpenCL system requires more than just HLS - a supporting infrastructure is required. The goal of this thesis is to build an infrastructure for experimenting with architectural templates for eventual use in HLS. To this end, this work targets a key architectural element, the memory system, with a specific focus on the interface to external memory.

Making FPGA computing systems easier to use requires a higher level of abstraction than programming in hardware description languages (HDLs). Whenever a higher level abstraction is created, lower level details such as clock management, external memory
management and off-chip communication become hidden. A side effect of the abstraction is that fine-grained control is taken away from the developer, which can reduce performance and increase area. New larger and faster FPGAs will be able to absorb these abstraction losses. At the same time, developers will be able to effectively harness these larger FPGA areas using these higher level abstractions.

Another effect of bringing OpenCL to FPGAs is that CPU-GPGPU-FPGA systems can be created and programmed solely using OpenCL. Developers already using OpenCL for CPU-GPGPU systems should be able to easily reuse their prior knowledge to utilize FPGAs. Having such a system opens up previously unexplored research areas for applications that benefit from all three computation units being used simultaneously on a single platform.

1.1 Contributions

This work details what is necessary to create a complete OpenCL FPGA system and focuses on an implementation of an Aggregating Memory Controller (AMC) - a new method for accessing large high-latency memories on FPGAs, similar to how access is performed on current GPGPUs. This work also shows an implementation of a Processing Array that uses either soft-processors or custom hardware cores, executing in a vector-threaded manner \[1\]. Two micro-benchmarks are implemented on the array. The main purpose of this array and the micro-benchmarks is to exercise the AMC.

The main contributions of this thesis are:

1. An OpenCL FPGA Model

2. An Aggregating Memory Controller

3. A Processing Array using soft-processor and custom hardware cores.

As part of creating the Processing Array, the Tiger “MIPS” processor \[2\], which
was originally targeted towards Altera hardware, was also reworked so that it could be synthesized for Xilinx hardware.

1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes OpenCL in relation to FPGAs and outlines three key areas of work that need to be completed for a fully functional OpenCL FPGA system. Chapter 3 outlines related external memory access methods and previous work on OpenCL on FPGAs. Chapter 4 describes the entire system architecture. Chapter 5 details the implementation methods and evaluation of the system through two micro-benchmarks. The thesis is then concluded in Chapter 6 where suggestions of future work are also given.
Chapter 2

Programming Model

2.1 OpenCL

OpenCL has emerged as a programming model for heterogeneous computing \[3\]. It is developed by the Khronos Group as a standard model for developing applications that can be implemented on devices that support the model. Furthermore, any device that supports the model can be used in conjunction with other OpenCL devices, enabling heterogeneous computing. Four important sub-models are defined by the standard: the Platform, Execution, Memory and Programming model. Each model is further discussed in the following subsections both in general OpenCL terms as well as how they interact with FPGAs as OpenCL devices. In the final subsection, an OpenCL FPGA Model is presented, which logically partitions individual system components that are necessary for a fully functional OpenCL FPGA system implementation.

2.1.1 OpenCL Sub-Models

Platform Model

An OpenCL platform is a Master-Slave system consisting of a Host and one or more OpenCL enabled devices (Figure 2.1). Typical OpenCL platforms consist of only GPGPU
slave devices connected through a PCIe bus to an x86 host processor. However, the Host and Devices need not be limited to x86 processors and GPGPUs. Embedded processors and accelerators such as FPGAs could also be used. Subsequently, the type of interconnect changes according to the host and devices.

There are two broad types of OpenCL platforms that can be created using FPGAs. The first type, \textit{Unified x86}, adds an FPGA OpenCL device to the typical CPU-GPGPU system and adds additional system heterogeneity. This fits well with current supercomputing trends and opens the possibility of using specialized hardware implemented on FPGAs in conjunction with GPGPUs and general purpose CPUs. These types of systems currently require the host to be an x86 system due to proprietary GPGPU drivers and libraries. Fortunately, there are currently systems that allow FPGAs to tightly couple with x86 processors through processor sockets \cite{4}, using the FSB as the interconnect or through the PCIe bus \cite{5, 6}.

The second type of platform, \textit{Embedded SoC}, has the host and device(s) fully embedded in an SoC fashion. These types of systems have been recently demonstrated using embedded PowerPC processors \cite{7, 8} and in the future, these types of systems will be of greater significance with large FPGA companies planning devices with FPGAs coupled with higher performance embedded hard processors \cite{9}. The respective embedded inter-
connects would be used for communication between the host processor and devices. For example, Avalon for Altera NIOS soft-processors [10] or AXI for ARM processors [11].

**Execution Model**

The main executing thread on the host issues *Kernels* and host-device memory transfers over an interconnect to the devices. The kernel-issue and memory transfer commands are stored within a work queue on the device before they are scheduled to run.

Kernels are data or task parallel processes that perform operations on the device-side memory, are written in a C-like syntax, and can be queued to execute on any OpenCL device connected to the host. The kernels themselves are processed by the *Compute Units* within the device. Each Compute Unit is a physical element that implements an array of *Virtual Scalar Processors* (VSP).

A kernel is executed over an array of threads called a *Grid*. Grids can be 1, 2, or 3 dimensional. The threads within the grid are subdivided into *work-groups*. By definition, each work-group is disjoint in terms of synchronization and data. Each work-group executes on a Compute Unit where each thread directly maps to one virtual scalar processor. This concept is illustrated in Figure 2.2 where a 2-dimensional grid of 256 threads is subdivided into size-16 work-groups. Two compute units, 0 and N, are also shown to be executing separate work-groups. Since each work-group is disjoint, each work-group can be independently scheduled to run on separate compute units.

Within the work-group, threads share access to local memory and can be synchronized. The mechanisms used to implement these requirements are usually contained within the Compute Unit architecture. Memory and synchronization are further discussed in the following two sections, Memory Model and Programming Model.
Figure 2.2: Grid of threads showing work-group subdivisions and Compute Units each with 16 Virtual Scalar Processors.

Figure 2.3: OpenCL Memory Model showing Host Memory and Global, Constant, Local, and Private Device Memory.
Memory Model

A hierarchical memory space is defined by the OpenCL standard and follows a relaxed consistency model \[3\]. In the hierarchy, explicit memory transfers need to be issued from the host system or within the kernel to transfer memory between the four device memory regions and the host memory depicted in Figure 2.3. From the Host, only Global and Constant memory can be read and written. From the Device, Global and Constant memory is accessible (R/W and R, respectively) by all threads in the grid whereas Local is only accessible by threads within a work-group and Private is limited to a single thread. This private memory can be thought of as a thread’s scratchpad memory.

The relaxed memory consistency does not guarantee that all memory visible from each thread is always consistent. Private memory is always consistent and threads in the same work-group can see consistent Local and Global memory only after a synchronization barrier. Threads in different work-groups are only synchronized at kernel completion. Therefore, overall Global memory consistency can only be guaranteed upon kernel completion.

The device implementer has the freedom to decide how this Memory model is designed. Furthermore, caching or prefetching can also be incorporated into the memory system. One interesting example of diversity is that Host memory and Global memory could actually reside on a single physical memory as demonstrated in AMD Fusion architectures \[12\].

On FPGAs, there are many options for implementing the memories. Each memory space can be implemented in block memories, distributed memories, external memories or any combination thereof. Newer FPGAs, such as the Virtex-7, support speeds up to 14.933 GB/s per commodity 64-bit DIMM. For reference, Nvidia Tesla and GeForce 500 series have 150-200 GB/s memory bandwidth on bus-widths up to 384 bits.
Programming Model

On the main host thread, operations such as host-device memcpys or kernel-launches are buffered within a virtual device work queue. This work queue can be executed in- or out-of-order, depending on implementation.

Task and Data parallelism is supported through the programming model. Data parallelism is the focus within kernels and task parallelism is achieved through the simultaneous execution of multiple kernels on a single device or multiple devices.

The programming model allows the programmer two types of synchronization. The first is as a result of the device work queue - each operation, whether kernel-launch or host-device memory transfer, is executed when dependent operations are completed. Through this mechanism, global device memory is ensured to be consistent at the boundary between dependent operations.

The second method of synchronization exists within the kernel as an explicit barrier OpenCL API call. This barrier ensures that all threads within a work-group are at the same execution point after the API call and subsequently have a consistent view of all memory regions.

2.1.2 OpenCL FPGA Model

The proposed model is a logical partitioning of components within an OpenCL FPGA system with the goal of allowing User Code to be executed on the FPGA fabric as transparently as possible. The model is depicted in Figure 2.4.

At the top, User Code is written according to the OpenCL standard. The code is compiled and linked to the OpenCL runtime library using the OpenCL Software Framework. The framework runtime communicates through the Device Interconnect, with the Device Controller and the Memory Interface on the FPGA. Host-device memory transfers are issued to the Memory Interface and kernel-launches are issued to the Device Controller.
The Device Controller launches the kernels on Compute Units. The kernels running on the Compute Units access memory through the Memory Interface and communicate with the Device Controller for synchronization and kernel completion.

From this model, there are three major components to consider and all must be implemented for a complete OpenCL system:

1. **OpenCL Software Framework**

2. **Code Translation**

3. **Device Architecture**

Each component, enumerated in Figure 2.4, is somewhat self-contained and is further discussed in the following sections.
OpenCL Software Framework

The OpenCL Software Framework resides fully on the host system and consists of two major parts: the runtime library and kernel compiler. The runtime library contains all of the function interfaces to the OpenCL API and interacts with devices connected to the host through functions such as `get_device_info()`. An implementation of these functional interfaces needs to be added to the library so that the new FPGA OpenCL device is visible to the user code. On the host, device drivers may also be needed for the interconnect and OpenCL device. Lastly a kernel compiler specific to the FPGA OpenCL device is needed. Technically the compiler is part of the software framework, but it is deferred to its own component, which is discussed in the following subsection, due to its complexity.

The framework will greatly differ based on the type of platform. As previously discussed, there are two main types of OpenCL FPGA platforms: the Embedded SoC and Unified x86. Currently, an entire framework from the ground up needs to be created for Embedded SoC platforms. For unified x86 systems, there are already OpenCL implementations for GPGPUs. The OpenCL standard has provisions for extending an implementation through plug-ins containing the compiler and drivers for new devices. Only through this plug-in method can the library be extended since most implementations for commodity GPGPUs are proprietary.

Code Translation

Code translation is the interpretation and manipulation of kernel code for the target device architecture. There have been two main choices for kernel code translation depending on the type of Compute Units present on the system: high-level synthesis (HLS) straight to hardware or compilation to soft-processors. Both of these Compute Unit options are further discussed in the subsequent Device Architecture section. LLVM [13] is commonly used as a framework for both methods since the front end interface Clang [14] supports OpenCL.
During Code translation, memory patterns can also be extracted from the kernel code to augment the memory interface. With this additional information, the memory interface can be optimized through prefetching or other compile-time memory access optimizations.

For both HLS and soft-processors, kernel specific API calls, such as the barrier, need to be implemented. These calls will depend heavily on the device architecture that is discussed in the following section.

**Device Architecture**

The Device Architecture is the part of the OpenCL FPGA model that resides physically in hardware on the FPGA. Each component is necessary for kernels to be executed on the device. This section will address some of the design options and trade-offs present in each of the hardware modules.

Current FPGA platforms provide many options for the Device Interconnect. The type of interconnect is largely dependent on the type of the targeted OpenCL platform. Interconnect options for Unified x86 OpenCL systems include Intel FSB and QPI interfaces, HyperTransport and PCIe. For embedded OpenCL systems, the SoC interconnect will usually be used to connect the host processor to the hardware OpenCL devices. To create reusable infrastructure for both types of OpenCL systems, the entire interconnect needs to be abstracted from the device architecture.

A Device Controller is needed to provide general control of the OpenCL device. It can manage the control flow of data over the interconnect, manage the work queue, schedule kernel launches on the Compute Units and interact with the Memory Interface providing host-device memory transfers. The Device Controller can also be used to provide device memory allocation to the host and in the future, handle device-to-device copies\(^1\). An embedded processor seems like the logical choice for implementing the controller.

\(^1\)Device-to-device copies are currently not supported by the OpenCL specification.
Chapter 2. Programming Model

The Memory Interface is necessary to provide the Compute Units timely access to on- or off-chip memories as well as facilitate host-device copies to and from Global and/or Constant memories.

Compute units are the heart of the OpenCL device. They are either implemented through soft-processors or custom hardware, either handwritten or generated through HLS. Using soft-processors as Compute Units creates a general architecture on which compiled kernel code can be executed. Due to the generality of soft-processors, area can be high with mediocre performance. One way to potentially improve performance and area could be to tailor soft-processors towards the entire set of the OpenCL application’s kernels, while still retaining generality of the Compute Units.

HLS has been an attractive idea for OpenCL on FPGAs due to the parallelism inherent within kernels. However, one drawback is that applications that have many kernels could have potentially high area usage, since compute units generated through HLS are unique to a given kernel. Partial-reconfiguration of the FPGA could be employed to mitigate the potentially high area cost that would be incurred in applications with many kernels. By keeping a library of post-PAR bitstreams of different compute units and runtime reconfiguring the FPGA, different kernel implementations could be swapped on-the-fly.
Chapter 3

Background

3.1 Memory Access Models

A number of memory access models exist that can provide timely access to high-latency, high-bandwidth external memories while effectively distributing bandwidth. In the following sections, brief summaries of known methods are given with focus on how these methods relate to the OpenCL model.

3.1.1 Caching

Caching is widely used in a vast number of computer architectures. Caching is well suited for single-threaded computation since there is sequential access and memory reuse - two behaviors that benefit most from caching. Caches have had previous use with soft-processors on FPGAs [15, 16].

There needs to be multiple processors or a large number of queued requests to fully utilize the external memory bandwidth. With multiple cores and multiple caches, some consistency model must also be brought forth if a shared memory model is to be used.

Caching is not primary within the scope of OpenCL and does not fit well in the kernel execution model due to the relaxed consistency of memory and general kernel design. The
processor in OpenCL is the Compute Unit and caching per Compute Unit is possible. However, the relaxed consistency of memory motivates no data sharing between workgroups (that execute on Compute Units) within a kernel. In addition, kernels are usually designed to cross the external memory boundary infrequently and to store intermediate results in shared local memory within the Compute Unit. This behavior results in low temporal locality to external memory. However, OpenCL does not restrict having a work-group access the same blocks of external memory multiple times and in these cases a cache for external memory would be beneficial. Kernels are also usually designed for spatially close threads (within the Grid) to access close memory locations. This spatial locality could be exploited through a cache.

The Aggregating Memory Controller presented in this thesis tries to exploit the spatial locality of kernels by considering multiple memory requests at a time while also allowing multiple Compute Units make memory requests, hiding access latency.

### 3.1.2 Streaming

Streaming is an easy way to maximize external memory bandwidth due to the inherent hidden access latency and sustained access. This type of access can largely be seen in processing systems where memory accesses are well coordinated between computation units. These memory accesses usually have a fixed access pattern that is known pre-runtime. On FPGAs, there is current work on streaming frameworks [17].

In OpenCL, the Compute Units are independent and no data is passed or streamed directly between units. No fixed access pattern is required by OpenCL since all memory accesses are resolved at runtime. However, access patterns can be extracted at compilation and exploited by the OpenCL kernel compiler.
3.1.3 Multi-Ported Memory Controllers

Multi-Ported Memory Controllers (MPMC) are frequently used on multimedia System-on-Chips. They arbitrate between many processing cores within the chip to provide access to external memories. Some MPMCs are sophisticated enough to handle multiple Quality-of-Service requirements to each port [18]. The distinguishing attribute between MPMCs and the Aggregating Memory Controller presented in this thesis is that MPMCs take scalar requests for bursts of SDRAM from multiple cores whereas the AMC takes a vector of requests that may or may not be from the same SDRAM burst. Scalar versus Vector memory accesses are discussed in the next section.

3.1.4 Scalar versus Vector Memory Access

Most traditional computing architectures follow a scalar memory access pattern where one data element from memory is requested per cycle. Vector memory access refers to multiple memory requests being issued within the same cycle. The requests within the vector are arbitrary and do not need to be consecutive locations within memory. Some GPGPUs, such as the ones produced by Nvidia, execute threads and access memory in this vectored fashion in groups of 32 threads.

Figure 3.1a shows a scalar-memory access from a set of threads time-sharing a single core. Figure 3.1b shows a set of threads, time-shared over multiple cores allowing a vector of threads to execute a vector of memory requests on a single cycle. Threads can diverge but they must all synchronize before memory requests so that a vector of requests can be made.

3.2 Previous OpenCL-FPGA Work

In this section, previous works for OpenCL on FPGAs are outlined. Two previous works, FCUDA [19] and SOpenCL [8], have focused on Code Translation using High-Level Syn-
thesis of OpenCL-like kernel code to hardware elements. Two others use soft-processors as the primary computation elements. With the exception of FCUDA, all other works have been published within the time frame of this thesis. Details for each of these works follow, concluded by some discussion between these works and this thesis.

### 3.2.1 FCUDA

FCUDA [19] is based around CUDA kernels that are almost identical to OpenCL kernels in syntax and semantics. They implement a C-to-C translation that generates C code annotated for Auto-pilot, a commercial HLS tool which supports both on- and off-chip memory. Their work is mostly focused around this translation step since the HLS tool is largely responsible for hardware generation.

### 3.2.2 SOpenCL

SOpenCL [8] leverages the LLVM framework [13] for template based HLS and generation of the datapath within a pre-built accelerator infrastructure. Their HLS tool creates compute units that sequentially execute threads within a work-group. The compute units are connected to the infrastructure’s memory streaming engine, which makes requests to the embedded SoC bus. Their streaming memory interface performs well if their compiler framework can extract necessary memory access patterns for data to be prefetched and
not as well with data-dependent memory access. Their memory interface can also remove redundant requests for sequential data-fetches from their scalar-threaded compute units but it was unclear how efficient the removal was. In addition, there is a potential for Read-After-Write (RAW) hazards in their system since they do not implement any RAW checking. They also implemented a custom OpenCL Software Framework, specific for their embedded PowerPC host processor, which communicated to the accelerator device through the Processor Local Bus (PLB).

3.2.3 OpenRCL

OpenRCL [7], also uses the LLVM framework [13] for code translation for compilation of kernels to assembly for hardware multi-threaded MIPS processors. They also extract the memory access pattern to schedule a large crossbar-connected on-chip global device memory. Each multi-threaded MIPS processor acts as an OpenCL Compute Unit, processes all threads within the work-group in lock-step, and accesses global memory through the memory crossbar. Shared local memory is implemented in BRAMs within each Compute Unit. Their OpenCL device is connected through ethernet to a host computer that implements a simple software OpenCL library - enough to initialize memory and issue kernels.

3.2.4 FSM SYS Builder

Cartwright et al. [20] approach OpenCL on FPGAs from the MPSoC point of view. They use their tool FSM SYS Builder to create a bus-connected hierarchy of processors with DMA engines for memory transfers from external memory to the soft-processors’ BRAMs and perform software transformations to map OpenCL threads to each soft-processor. One host soft-processor has access to external SDRAM and controls the DMA engines transferring data from external SDRAM to device BRAMs.
3.2.5 O4F

Recently, Ahmed’s MASc thesis [21] implemented an OpenCL Framework, O4F, containing middleware that connects CPU, GPU, and FPGA thereby creating a Unified x86 OpenCL system. Custom hardware cores were implemented on the FPGA and the FPGA was then integrated using the PCIe bus as interconnect. His thesis focus was concentrated on creating this middleware and details the technical challenges of integrating FPGAs into the OpenCL platform.

3.2.6 Relationship to Previous Work

This thesis specifically targets the creation of a Device Architecture for the Memory Interface and Processing Array structure that can be used in both Unified x86 and Embedded SoC OpenCL platforms.

Ahmed’s MASc thesis [21] focuses on the OpenCL Software Framework that is in the layer above the Device Architecture (Figure 2.4). The Device Architecture shown here complements Ahmed’s work.

The Aggregating Memory Controller presented in this thesis is similar to the controller in SOpenCL. However, this work differs in four aspects: it produces the minimal number of memory requests with a parallel aggregation scheme; it connects directly to the external memory interface; it allows for both scalar- and vector-threaded style accesses; and most importantly RAW hazards are eliminated.

OpenRCL [7] also uses a MIPS style soft-processor. The processor in OpenRCL has multiple hardware contexts whereas the soft-processor used in this thesis only has one. Furthermore, the soft-processors in OpenRCL execute in a scalar-threaded manner as opposed to the soft-processors presented in this thesis that execute in a lock stepped vector-threaded manner.

The Processing Array presented here aims to standardize interfaces for elements that
can, in the future, result in a unified compiler / synthesis to either soft-processor, full hardware, or hybrid processing arrays. In addition, the Processing Array presented is the only vector-threaded style OpenCL targeted processing infrastructure for FPGAs, to the author’s knowledge.
Chapter 4

System Architecture

4.1 Overview

The architecture of the implemented OpenCL-based processing system is described in this chapter and a high-level overview of the system is shown in Figure 4.1. There are four main components: the Xilinx MIG SDRAM controller, the Aggregating Memory Controller, the Processing Array and the Device Controller. The Xilinx MIG SDRAM controller provides an interface to an external DDR2 SODIMM and is generated using the Xilinx CoreGen tool [22]. The soft-processor used in the Processing Array is an open-source Verilog design originally targeted towards Altera platforms. The source of this soft-processor was adapted to use Xilinx-style on-chip RAM and special function units such as multipliers and dividers. All other components are were hand-coded SystemVerilog.

4.2 Xilinx MIG SDRAM Controller

The Xilinx MIG SDRAM Controller [23] provides a FIFO-based user interface to external DDR2 SDRAM for Xilinx FPGAs. The component is generated using the Xilinx CoreGen tool [22], through which the memory interface widths and burst access size is specified. Bursts of four or eight can be specified before generating the core.
Figure 4.1: System Overview
The user interface has two user-FIFOs, the command FIFO, the write-data FIFO, and raw read-data lines. The read or write request and corresponding address is specified through the command FIFO. The data to be written and a write-mask, which is used to specify which bytes within the write-data should be written, are queued in the write-data FIFO. Read-data is returned with an accompanying valid signal. The read-data is not buffered and needs to be used as soon as it is available.

In this design, the DDR2 interface is 64 bits wide connected to an external SODIMM and is clocked at 200MHz. Since DDR2 returns data on the rising and falling edge of the clock, the effective user-side data width is 128 bits. So, the read-data and write-data ports are both 128 bits wide. Accordingly, the byte-granularity write-mask is 16 bits wide. The burst size of the MIG is set to four, which means that every burst will contain 256 bits of data over two clock cycles.

While the MIG runs at 200MHz, the rest of the system is clocked at 100MHz. FIFOs are used for the clock crossing and the effective bandwidth of the controller is halved, since no data-width-expansion across the domains is performed. Both clocks are phase-aligned and a data-width-expansion from 128 bits to 256 bits could be performed when crossing from 200MHz to 100MHz clock-domains. However, for simplicity of the design prototype, no data-expansion was performed. The final bandwidth available to the rest of the system is 128 bits at 100MHz or 1600 MB/s.

### 4.3 Aggregating Memory Controller

The Aggregating Memory Controller (AMC) is one way to allow Compute Units to effectively utilize growing external memory bandwidth and also allows the Compute Units to be able to apply the GPGPU paradigm of coalesced memory accesses.

Given a vector of \( N \) requests of \( M \) sized words, the AMC requests the \textit{minimal} number of SDRAM bursts to fulfill the vector of \( N \) requests. This maximizes bandwidth efficiency
by eliminating redundant requests. The requested words within the SDRAM bursts are then distributed to each requesting port. The price paid for this increased efficiency is the area of the controller and added latency. However, this latency can be offset through multi-threading and queued requests as will be shown in Section 5.4.

Two designs of the AMC were implemented. Both implementations assume eight 32-bit wide request ports to a 32-bit byte-addressable address space and 256-bit data bursts over a 128-bit wide path to the SDRAM controller. The first design was implemented using a split read and write datapath approach. The second implementation unified these datapaths, increasing performance and reducing area. Performance and Area results are given in Sections 5.4 and 5.5, respectively. In both designs there is also an external DMA access port that is a direct interface to the MIG controller. This port is utilized for host-device memory transfers.

4.3.1 Key Modules

Three key modules that are used by both implementations of the AMC are discussed in this section: the Sorting Network, Read Reorder Network, and Write Reorder Network.

Sorting Network

A sorting network sorts $N$ inputs into $N$ ordered outputs using $O(n(\log n)^2)$ compare-and-swap elements in $O((\log n)^2)$ stages \cite{24, 25, 26}. An example of a size eight Bose-Nelson network \cite{27} is shown in Figure 4.2. Eight inputs ingress from the left, go through seven stages of compare-and-swap elements and egress on the right. Each compare-and-swap element is represented by a vertical bar connecting two inputs. In the hardware implementation, each stage is registered (represented by the vertical dotted lines in Figure 4.2).

Within the AMC, sorting networks are used to order the requested addresses from each access port. When the requested addresses are sorted, $N - 1$ comparators can be
used to find which addresses lie within the same SDRAM burst. These sorting stages contribute the most to the memory access latency added by the AMC.

**Read Reorder Network**

The Read Reorder Network is series of multiplexers that organize data from the SDRAM burst to each port. An eight-ported network is shown in Figure 4.3. On the left a 256-bit burst of data organized into 32-bit words is received by registers from the MIG SDRAM controller. From the registers, eight multiplexers select the correct 32-bit word to forward to the access port based on the address requested by the port.

Each burst from memory need not map directly to every port. For example, a port may only require one data word from an SDRAM burst and additional bursts may be needed to fulfill the other requests within the vector, for the rest of the ports. When the SDRAM bursts are returned, the ports themselves figure out if their data word is within the returned SDRAM burst.
Write Reorder Network

The Write Reorder Network is significantly more complex than the Read Reorder Network as it needs to avert write collisions. An eight-ported network is shown in Figure 4.4. The write addresses from the ports are compared to the word being written to. In Figure 4.4, the first element writes to word 0 and the last writes to word 7. Ideally, only one of the addresses should match per burst. The conflict cases are accounted for using a priority encoder so that only one data word is written to the SDRAM burst, per element. Additionally, a write-mask is generated for the burst depending on whether the data word within the burst has valid data.

Like the Read Reorder Network, multiple bursts may need to be generated to fully complete a request. After the SDRAM burst is populated, the burst with its corresponding write-mask is sent to the MIG SDRAM controller [23].
4.3.2 Split Datapath Design

The initial AMC design consisted of a read and write path that make requests to the FIFO controlled Xilinx MIG controller [23]. This design is depicted in Figure 4.5. Both paths operate independently of each other and require arbitration before requesting to the MIG. Additionally, the external DMA requests also need to be arbitrated.

The read path, shown in the top half of Figure 4.5, contains read ports. These ports have two internal FIFOs, which are not shown in the figure - one for requested addresses and one for read-data. The FIFOs buffer incoming read requests and forward the addresses to the sorting network. The sorted addresses are filtered to remove redundant SDRAM burst requests using $N - 1$ comparators and an $N$-input priority encoder in the Address Filter to produce a stream of minimal requests to the Request FIFO. The Request FIFO buffers requests to the Request Arbiter. Pending requests are also tracked so that the read ports receive data from the correct SDRAM data burst. Finally, the pending request address is compared to the port’s requested addresses as the requested
SDRAM burst is put through the Reorder Network and distributed to the correct Read Ports.

The Write Datapath path in the bottom half of Figure 4.5 is similar to the read datapath with the exception that the data is put through the more complex Write Reorder Network that packs data words into a full SDRAM burst. The Write Port also contains FIFOs for requests and data. A Sorting Network and Address Filter are also utilized. In this case, the Write Reorder Network also creates an appropriate write-data mask for the external memory data burst.

Finally, the read and write requests from both paths and the external DMA interface are arbitrated to the MIG controller. The two datapaths maintain order for read requests and for write requests but do not maintain the ordering between read and write requests. Since there is no guarantee of order between read and write requests, there is a potential for RAW hazards. However, the micro-benchmarks used to test both AMC designs do not
contain any RAW dependencies. So although RAW hazards can exist, none were present in the micro-benchmarks. The second design, described in the next section, solves the RAW hazards problem by combining the datapaths.

### 4.3.3 Unified Datapath Design

The unified AMC combines both read and write datapaths into a single path, saving area and preserving the order of read and write requests. All requests are now fully ordered which eliminates the possibility of having a RAW hazard. Read and Write ports are combined into *AMC Ports* shown in Figure 4.6. The AMC Port is the FIFO interface to the Processing Array. There are five internal FIFOs. The first three buffer the read and write requests to the memory controller. The Request FIFO buffers both read and write requests that get sent to the sorting network. The Read and Write pending FIFOs store the addresses of the reads and writes so that the reorder networks can select the correct data to and from each AMC port. The other two FIFOs buffer data to and from the port to the internal reorder networks.

The Unified AMC design is depicted in Figure 4.7. All requests now go to the single Sorting Network that feeds sorted addresses into the Address Filter. The Address Filter is similar to the ones used in the split datapath design, using $N - 1$ comparators and an $N$-input priority encoder to filter out redundant addresses - addresses that are within the same SDRAM burst. The main control module then takes the filtered addresses and arbitrates between the DMA requests and the internal AMC requests. The control module also stores the read burst requests into a Read Pending FIFO that tracks pending read requests to the MIG SDRAM controller. Lastly, the control module coordinates with the Write Reorder Network to write to the MIG.
Figure 4.6: AMC Port

Figure 4.7: Unified Datapath AMC
4.3.4 Scalability

Since the Reorder Networks are essentially full-crossbars, it may not make sense to scale to too many ports or fine-grained byte-access. In addition, more ports may not be beneficial to the overall processing scheme since wider processing rows may be harder to synchronize.

There are better ways to scale the memory architecture. One way is to multiplex multiple Compute Units to a single AMC until the memory bandwidth is saturated, which is how the AMC is utilized in this work.

Another is to have multiple AMCs, each with its own processing arrays, and arbitrate access to the MIG controller. This can be easily done, since each array would implement separate Compute Units that process separate work-groups producing independent requests.

Caching may also be implemented in between the MIG and AMCs either as a single cache or on a per AMC basis. These caches could be highly beneficial for access times for RAW dependencies to global memory that can arise from register spilling in soft-processors or data driven accesses to global memory.

4.4 Processing Array

The Processing Array, depicted at the bottom of Figure 4.1, consists of Processing Rows that contain Processing Elements. Within the row, processing elements can share some hardware. In the case of soft-processor elements, further discussed in this section, there is a shared instruction memory. The Processing Array was designed for vectored memory accesses and within the array, these accesses to the AMC are scheduled by the Array Manager through a set of multiplexers. The Array Manager also tracks the completion of each row as well as if the row is stalling on a memory access.

Each row within the array processes a vector of threads, one thread per physical
Processing Element (PE). After the current vector of threads is complete, a new vector of threads is started on the row. The row itself implements an OpenCL Compute Unit. For example, consider a grid of 64 threads subdivided into four work-groups and array rows of width eight. Each work-group containing 16 threads execute on a Compute Unit or row within the array. The row itself processes eight of the 16 threads at a time and will need to switch to a second set of eight threads after the first eight complete.

The entire array could also implement a single Compute Unit. However synchronization and shared local memory architectures will differ between this and the row implementation and will likely be more complicated in this configuration since the synchronization and sharing implementation will need to cross the row boundaries. However, in the context of this thesis, either partitioning of the array is valid since the micro-benchmarks created to exercise the AMC do not use work-group synchronization or shared local memory.

Design started with one row of soft-processor elements for initial testing. The array was further expanded with more rows and the addition of the array manager to allow memory request arbitration between rows through a set of multiplexers to the AMC. The interfaces to all elements within the array were kept generic, as shown in Figure 4.8. This array with standard element interfaces formed a reusable infrastructure that later allowed easy swapping of soft-processing elements to hand-coded hardware cores.

Currently, each element within the row executes in lock-step. This restriction is imposed to avoid the side-effects of thread divergence between elements. If divergent execution is allowed, some mechanism is needed to eventually re-synchronize before all threads are allowed to perform vector memory access. There is current vector-threaded architecture research [1], but this along with other array micro-architectures is not our current focus.

The main targets for HLS is a single Compute Unit and in this infrastructure, there are two main targets for HLS, a single element within the row or the row itself. Targeting
a single element is akin to swapping soft-processing elements to hand-coded hardware cores. In this method, it is possible that intra-element synchronization can become difficult due to the fine granularity of the Processing Elements.

Targeting a row could be a better option. In this case, shared local memory and thread synchronization can be implemented internal to the synthesized core. Both Scalar- or vector-threaded HLS cores can be encapsulated into a processing row. Vector-threaded cores fit directly while some transformation is required for Scalar-threaded cores. This is illustrated in Figure 4.9. A scalar stream of memory requests can simply be piped into a shift register to access memory in a vectored fashion. The downside to this method is additional memory request latency. SOpenCL [8] generates a scalar-threaded core that could be inserted in this manner. OpenRCL’s multi-context MIPS soft-processor is also scalar-threaded [7] and could be inserted.
Conversion of Scalar access to Vector access,

\[
\begin{align*}
&LD[X_0] \\
&LD[X_1] \\
&LD[X_2] \\
&\ldots
\end{align*}
\]

Figure 4.9: Scalar- to Vector-threaded Memory Access Transformation

### 4.4.1 Soft-Processor Elements

The Tiger “MIPS” soft-processor from the University of Cambridge [2] was chosen for the soft-processing element, shown in Figure 4.10. The processor has a MIPS-like ISA with a five-stage in-order pipeline. The soft-processor receives instructions from an external instruction memory that is shared by each element in the row. Only a single instruction memory per row is required, since the row is constrained to execute in lock-step. The processor’s memory bus is connected to a private BRAM memory that holds stack memory as well as static data. Also connected to the memory bus is the Global Memory Port that acts as a bridge for the processor to access global memory through the AMC. Because it is a bridge, any read access to global memory stalls the processor for the full memory access latency. To mitigate the effects of the stall, the Global Memory Port signals the Array Manager that the Global Memory Port is waiting on a memory request. The Array Manager subsequently switches row access to the next row within the Processing Array. Each row is given round-robin access to the AMC in this fashion. If all rows are waiting on a memory access, access to the AMC is given to the row that was waiting the longest. The PE Control contains a unique element ID that allows the soft-processors to calculate thread IDs. This PE Control element also allows the soft-processor to signal the array manager that all threads have been processed using the done signal.
4.4.2 Hardware Cores

Two hand-coded hardware cores were created for the two micro-benchmarks. They both use the generic element interfaces and swap directly into the array. Micro-benchmark specifics can be found in Section 5.3.

4.5 Device Controller

The device controller is minimal in implementation and is depicted in Figure 4.11. The controller consists of a Xilinx MicroBlaze soft-processor with two peripherals, a UART for communication to the Host and a custom interface to the Processing Array and Aggregating Memory Controller. The MicroBlaze is connected to a dual-ported BRAM that provides instruction and data memory through two Local Memory Buses (LMB). The UART is connected through the Processor Local Bus (PLB) and the custom interface is connected through two Fast Simplex Links (FSLs), providing bi-directional...
The firmware implements an SDRAM loader and SREC loader. It also starts and 
resets the Processing Array through the custom interface and collects cycle counters. 
All communication to the system is performed through the custom Device Interface. 
The external SDRAM is loaded through the AMC DMA port. The payloads containing 
instruction data for the Tiger “MIPS” and SDRAM initialization data are transmitted 
over the UART from the host. The SREC loader initializes the soft-processor elements’ 
instruction memories in the processing array with machine code that is also transmitted 
over the UART. Lastly, the interface reads the cycle counter that records the number of 
cycles each micro-benchmark takes to complete.
Chapter 5

Methodology & Evaluation

5.1 Hardware Platform

All experiments were run on a Xilinx XUPV5-LX110T development board using the Virtex-5 LX110T-1 FPGA. The Virtex-5 LX110T FPGA contains 17,280 slices that contains two fracturable 6-LUTs and four FFs each, 64 DSPs, and 148 36Kb BRAMs. Two external FPGA peripherals were used. The on-board 256MB SODIMM was utilized for off-chip memory and the on-board RS-232 transceiver was utilized for communications with the host PC.

5.2 CAD Tool Flows

SystemVerilog was used as the HDL for the AMC and Processing Array while the Tiger “MIPS” processor [2], Device Controller and MIG SDRAM controller [23] were described with Verilog. The primary reason for using SystemVerilog as the HDL for the AMC and Processing Array is that the language offers more flexibility for multi-dimensional arrays of signals than Verilog. This allows the AMC and Processing Array to be coded parametrically, allowing for more code reuse. The code was put through Simulation and Hardware flows to verify and prototype the system. Both flows are shown in the following
sections.

### 5.2.1 Hardware

The Xilinx 13.1 tool suite and Synopsys Synplify E-2011.03 were used in the hardware CAD flow for synthesis, mapping, place & route and bitstream generation. The flow is shown in Figure 5.1. The AMC and Processing Array, which were described in SystemVerilog, were synthesized using using Synplify that outputs a EDIF netlist. Synplify was used since the Xilinx tools do not support SystemVerilog. The MIG SDRAM controller [23], which was generated using the Xilinx CoreGen tool [22], was synthesized with ISE, which outputs an NGC netlist. The Device Controller was designed using the Xilinx EDK tool. The EDK tool allows assembly of systems with embedded soft-processors and was used to connect the MicroBlaze soft-processor with the instruction and data memory, UART device, and custom interface device that connects to the rest of the system. The output of the EDK flow is also NGC netlists. All of the netlists are then put through ISE. NGDBuild assembles all of the blackbox netlists and applies all user constraints. Then mapping, PAR, and bitstream generation takes place.

### 5.2.2 Simulation

Mentor Graphics ModelSim 6.6d was used for full system simulation. Figure 5.2 shows the entire system simulation flow. The AMC, Processing Array and MIG components are fed directly to ModelSim whereas for the Device Controller, EDK uses the SimGen tool to generate simulation libraries for ModelSim. This allows simulation of the closed-source MicroBlaze soft-processor core. Additionally, a simulation model of the external SODIMM was utilized in the system simulation. There was no simulation of the UART so a work around was necessary. A separate set of firmware for the Device Controller was created specifically for simulation that contained pre-canned system test sequences. This allowed simulation to proceed without simulating the UART.
Figure 5.1: Hardware Flow

Figure 5.2: Simulation Flow
5.2.3 Soft-Processor Compilation Flow

GCC 4.4.5 was used to compile the micro-benchmarks for the Tiger “MIPS” soft-processors. To simulate the processors, Binutils 2.20.1 was used to generate memory data files that could be read through the `$readmemh` Verilog construct that was used to initialize memory. For the hardware implementation, Binutils was also used. It was used to generate a SREC format that can be transmitted over serial to the Device Controller where the firmware decodes the SREC format and loads the on-chip BRAMs.

5.3 Micro-Benchmarks

Two micro-benchmarks, a vector addition and matrix multiplication, were implemented on the Processing Array using both the Tiger “MIPS” soft-processor [2] and custom hardware elements. The micro-benchmarks were first written in C and compiled for the soft-processors. Using the C-code as a guide, the full hand-coded hardware versions were created.

The micro-benchmarks were coded to have fully coalesced accesses to the AMC. This means that every vector memory access uses a single SDRAM burst. All data words within the SDRAM burst are used and none of the burst is wasted. This allows for maximum memory performance.

At the other end of the spectrum, worst case performance occurs when a single data word is provided by an SDRAM burst. Without an eight ported AMC or for fully uncoalesced accesses, 8 separate SDRAM bursts would be needed. Effective bandwidth is expected to degrade to one-eighth of the effective bandwidth of fully coalesced access, since both micro-benchmarks presented are memory bound.
5.3.1 Vector Addition

The vector addition uses one thread to add one element from each vector $A$ and $B$ and produces an element in vector $C$. There is a one-to-one mapping between elements and threads. A maximum of 128 custom hardware cores were able to be utilized. In this case, the multiplexer tree between the array and the AMC grew quite large (16-to-1) and was the critical path. Four vector sizes were tested: 512, 1024, 2048, and 4096. The total amount of data, $3N \times 4$ bytes, was divided by the cycle count to calculate average bandwidth. The area used is listed in Table 5.5.

5.3.2 Matrix Multiply

The matrix multiply micro-benchmark multiplies two matrices $A \times B$ to produce $C$. $A$ and $C$ are arranged in row-major order and $B$ in column-major order. The operation is divided into 8x8 tiles so that all memory accesses can be coalesced in the AMC to a full burst of eight 32-bit words. Each thread processes eight elements within a tile and each row in the array processes one 8x8 tile. For both the software-core and the custom hardware core, the maximum number of elements is restricted to 16 by the limited number of DSPs on the FPGA. Each custom hardware element uses three DSP blocks to perform a single cycle multiply-accumulate - one less than the soft-processor core. The Tiger “MIPS” uses an extra DSP since it is possible to use the full 64-bit result whereas the custom hardware core ignores the upper 32 bits. Three matrix sizes were tested: 16x16, 32x32, and 64x64. The total amount of data, $((1 + 2N) \times N^2) \times 4$ bytes, was divided by the cycle count to calculate average bandwidth. The area used is listed in Table 5.5.

5.4 Performance

Both micro-benchmarks were run on soft-processor cores and hardware cores. The total runtimes in cycles were recorded since both the soft-processor cores and hardware cores
operated at 100MHz. This was done using both implementations of the AMC. The micro-benchmark cycle counts for the split datapath AMC are shown in Table 5.1 and the unified datapath AMC in Table 5.2.

The effective memory bandwidth for each micro-benchmark was also calculated for both AMCs. This metric is a measure of how well the memory bandwidth provided by the SDRAM is utilized. An effective bandwidth that approaches the theoretical bandwidth provided by SDRAM is desirable since this means that performance has peaked and is limited by memory bandwidth. This effective memory bandwidth is calculated by dividing the total amount of data transferred over the AMC ports by the total micro-benchmark runtime. The metric does not count the total amount of memory transferred over the MIG interface but counts memory delivered to the processing elements. This means that in the case of non-fully coalesced accesses, unused parts of SDRAM bursts would not be counted.

The effective split datapath AMC bandwidth is shown in Table 5.3 and the effective unified AMC bandwidth is shown in Table 5.4. The following subsections discuss the performance characteristics of the system.

<table>
<thead>
<tr>
<th>Micro-benchmark</th>
<th>Size</th>
<th>Soft-processors</th>
<th>Hardware Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8 16</td>
<td>8 16 32 64 128</td>
</tr>
<tr>
<td>vector_add</td>
<td>512</td>
<td>4557 2303</td>
<td>2504 1376 852 572 436</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>10341 5196</td>
<td>6076 3564 2433 1461 1264</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>22192 11105</td>
<td>13604 8675 6287 5096 4647</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>44358 22199</td>
<td>27228 17337 12525 10194 9302</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>16x16</td>
<td>36331 18256</td>
<td>17132 8578 - - -</td>
</tr>
<tr>
<td></td>
<td>32x32</td>
<td>320678 160723</td>
<td>177484 89077 - - -</td>
</tr>
<tr>
<td></td>
<td>64x64</td>
<td>2560632 1281391</td>
<td>1419960 711334 - - -</td>
</tr>
</tbody>
</table>

Table 5.1: Split AMC: Cycle Count
## Chapter 5. Methodology & Evaluation

### Table 5.2: Unified AMC: Cycle Count

<table>
<thead>
<tr>
<th>Micro-benchmark</th>
<th>Size</th>
<th>Soft-processors</th>
<th>Hardware Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>vector_add</td>
<td>512</td>
<td>4597</td>
<td>2316</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>10005</td>
<td>5044</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>21043</td>
<td>10558</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>42598</td>
<td>21429</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>16x16</td>
<td>36320</td>
<td>18270</td>
</tr>
<tr>
<td></td>
<td>32x32</td>
<td>287339</td>
<td>144087</td>
</tr>
<tr>
<td></td>
<td>64x64</td>
<td>2560641</td>
<td>1281427</td>
</tr>
</tbody>
</table>

### Table 5.3: Split AMC: Effective Micro-Benchmark Bandwidth (MB/s)

<table>
<thead>
<tr>
<th>Micro-benchmark</th>
<th>Size</th>
<th>Soft-processors</th>
<th>Hardware Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>vector_add</td>
<td>512</td>
<td>134.8</td>
<td>266.8</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>118.8</td>
<td>243.6</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>116.8</td>
<td>232.8</td>
</tr>
<tr>
<td></td>
<td>4096</td>
<td>115.4</td>
<td>229.4</td>
</tr>
<tr>
<td>matrix_multiply</td>
<td>16x16</td>
<td>93.0</td>
<td>185.0</td>
</tr>
<tr>
<td></td>
<td>32x32</td>
<td>92.7</td>
<td>184.8</td>
</tr>
<tr>
<td></td>
<td>64x64</td>
<td>82.5</td>
<td>164.9</td>
</tr>
</tbody>
</table>
Chapter 5. Methodology & Evaluation

5.4.1 Soft-Processor Overhead

From Tables 5.1 and 5.2, the software overheads of the soft-processing elements is shown quite clearly when comparing them to the equivalent number of hardware cores in both micro-benchmarks. For the vector add micro-benchmark, there is between a 21-47% reduction in execution time for both AMCs. For the matrix multiplication micro-benchmark there is an overall greater reduction in execution time, between 44-53%.

5.4.2 Effective AMC Memory Bandwidth

The effective bandwidth for both AMCs for the vector add micro-benchmark for sizes 512, 1024, and 2048 are graphed in Figures 5.3 and 5.4. The largest dataset of 4096 elements was omitted since its bandwidth characteristics are quite similar to the 2048 element dataset. This is due to each vector within these datasets spanning a separate row. Each row in the SDRAM contains 8192 bytes and the 4096 and 2048 element datasets have vectors of size 16384 and 8192 bytes respectively.

The main observation from both Figure 5.3 and Figure 5.4 is that there is a large bandwidth gap between the 512 data set and the larger dataset of 2048. The significantly
Figure 5.3: Split AMC: Vector Addition Effective Memory Bandwidth
Figure 5.4: Unified AMC: Vector Addition Effective Memory Bandwidth
higher bandwidth can be attributed to all three 512 element vectors $A$, $B$ and $C$ being able to fit within the same SDRAM row, requiring no switching between rows. The 1024 dataset spans two SDRAM rows, one for both $A$ and $B$ and one for $C$ and the vectors in the 2048 dataset each span a separate SDRAM row. The matrix multiply microbenchmark also suffers a reduction in bandwidth due to row switching. The smallest matrix size, 16x16 also contains 1024 elements in matrices $A$ and $B$ and is shown to behave similar to the 1024 element vector addition in Tables 5.3 and 5.4.

Switching between rows will be unavoidable for any reasonably sized application. These results demonstrate that some knowledge of the external memory device needs to be known to mitigate the bandwidth losses and this is still true for other OpenCL platforms such as GPGPUs, where memory optimization is critical for performance. The SDRAM module that was used only supported one open row. Some SDRAMs are able to hold multiple rows open simultaneously. The ability to keep multiple rows open would reduce the delays incurred from opening and closing rows thereby increasing performance. Other ways to mitigate this problem could be through clever memory allocation schemes.

Initial examination of the split datapath AMC using a full system simulation showed that the aggregated burst requests to the MIG SDRAM controller had many interleaved reads and writes for all numbers of vector add cores as well as for matrix multiplies. The interleaving was caused due to the separate read and write datapaths in the split AMC. Since both datapaths concurrently process read and write requests, a total order of requests over time was not maintained, which allowed reads of the currently executing threads to get processed before writes from the previous set of threads. If these interleaved read and write requests are to different rows in the external memory, bandwidth is greatly reduced due to excessive SDRAM row switching. The results of the simulation, along with obvious resource reductions, lead to the unified datapath AMC.
5.4.3 Unified AMC Bandwidth Improvements

The Unified AMC brought about an overall increase in effective bandwidth by having a total ordering of all memory requests, thereby reducing unnecessary SDRAM row switching and in the process, also eliminating RAW hazards. In Figures 5.5, 5.6, and 5.7, each of the vector add micro-benchmark sizes are compared between each AMC. The maximum effective bandwidth seen overall is at 128 cores in the 512 dataset and is around 1409 MB/s in the split AMC. In the Unified AMC, the maximum effective bandwidth is around 1555 MB/s, a 10% increase, and is approaching the maximum theoretical system bandwidth of the 1600 MB/s.

The 1024 dataset saw the greatest increases in effective bandwidth using the Unified AMC, up to 51% for 128 cores. Since the vectors in the 1024 dataset span two SDRAM rows, the dataset was affected the most by the interleaved read and write requests caused by the split datapath. This can be seen in Figure 5.6 where the split datapath AMC...
Figure 5.6: Bandwidth Comparison of Split and Unified AMC: 1024 Element Vector Addition

vector add tapers off at 64 cores. In the same figure, the unified datapath AMC scales smoothly up to 128 cores.

There are little gains for the 2048 dataset shown in Figure 5.7. The increases in effective bandwidth were between 2-6%. Because each vector in the dataset spans separate rows, the inherent row switching of the micro-benchmark is the dominating factor for effective bandwidth. The small gains do however show that read and write interleaving did have some negative effects on performance in the split datapath AMC.

5.5 Area

The area occupied by the Xilinx MIG SDRAM controller [23], Aggregating Memory Controller, the various Processing Arrays, and Device Controller, are shown this section. Area for all components is listed in terms of LUTs, FFs, BRAMs and DSPs recorded
Figure 5.7: Bandwidth Comparison of Split and Unified AMC: 2048 Element Vector Addition

post-synthesis.

The Xilinx generated MIG SDRAM controller \cite{23} occupies 1669 LUTs, 2159 FFs, and 3 BRAMs. The three BRAMs within the controller are used for the user interface FIFOs. The Device Controller depicted in Figure \ref{fig:device_controller} contains 1830 LUTs, 1786 FFs, and 16 BRAMs. The majority of the area within the component is occupied by the MicroBlaze soft-processor that contains 1118 LUTs and 898 FFs. The 16 BRAMs are used as embedded memory for the soft-processor firmware.

\subsection{5.5.1 Aggregating Memory Controller}

The area for the unified datapath AMC is nearly half the size of the split datapath controller, as expected. The split datapath implementation occupies 9840 LUTs and 4510 FFs whereas the unified datapath occupies 5973 LUTs and 2628 FFs. There is approximately 60\% less area in the unified implementation.
5.5.2 Processing Array

The area occupied by each configuration of the processing array for both hardware cores and soft-processor core are listed in Table 5.5. The soft-processor elements, not surprisingly, occupy the most area. Each row of soft-processors use one BRAM for instructions and each individual element uses one BRAM for local scratchpad memory. Each soft-processor element also uses four DSP blocks to perform a 1-cycle multiply. A maximum of 16 processors could be implemented due to the lack of DSP elements on the FPGA that had a total of 64 DSPs.

The relatively simple vector add cores were the smallest. Each core individually was a little over 100 LUTs and 100 FFs. A maximum of 128 vector add cores were created. This limit was due to the increasing size of the multiplexer tree required to distribute AMC access to each Processing Row. Although 256 cores would fit on the FPGA, timing could not be closed over the critical multiplexer path. This is not a problem though since at 128 cores, it is possible to saturate the external memory bandwidth.

The matrix multiply core was also limited by the number of DSPs on the FPGA. Each core uses one less DSP than the Tiger “MIPS” processor. The processor needs four DSPs because it performs a full 32x32 to 64 bit multiplication. However, the matrix multiply core only needs three because the value is cast to 32 bits where the upper 32 bits are discarded. Each element uses half of a fracturable BRAM to store 64 intermediate values and each row uses four BRAMs. The intermediate values are generated from the tiling of the matrix multiply operation into 8x8 divisions.
## Table 5.5: Processing Array Area

<table>
<thead>
<tr>
<th>Processing Element</th>
<th>Cores</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>pe_tiger_mips</td>
<td>8</td>
<td>10312</td>
<td>4616</td>
<td>9</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>21671</td>
<td>10201</td>
<td>18</td>
<td>64</td>
</tr>
<tr>
<td>pe_vector_add</td>
<td>8</td>
<td>984</td>
<td>831</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1929</td>
<td>1878</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>3820</td>
<td>3950</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>7382</td>
<td>8062</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>14544</td>
<td>16243</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>pe_matrix_multiply</td>
<td>8</td>
<td>1084</td>
<td>495</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2125</td>
<td>1190</td>
<td>8</td>
<td>48</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusion

A model for implementing OpenCL, which is a standard model for heterogeneous systems, on FPGAs is proposed in this thesis. The model illustrated the broad spectrum of OpenCL-FPGA systems that can be created and three major components that need to be considered for OpenCL-FPGA systems: the OpenCL Software Framework, Code Translation, and Device Architecture. This thesis targeted the Device Architecture with specific focus on the external memory interface and the processing infrastructure.

An Aggregating Memory Controller was created to access external SDRAM through a FIFO controlled interface and was shown as a new way for FPGAs to access external high-latency, high-bandwidth large memories using vector-threaded compute units, similar to GPGPUs. A Processing Array was created to provide a generic interface to Processing Elements and exercise the Aggregating Memory Controller. Communication with the host machine was provided over a UART using a Device Controller containing a MicroBlaze soft-processor.

In contrast to previous work, the unified datapath AMC finds the minimal number of burst requests for a given vector of memory requests and also preserves total ordering of memory requests, eliminating RAW hazards. Additionally, previous works all use scalar memory access and a conversion from scalar to vector memory requests is suggested for
enabling them to use the AMC.

The Processing Array was used as a template for the soft-processor elements and custom hardware elements. While these elements were implemented for the purpose of exercising the AMC, they also demonstrated the gap between soft-processors and hand-coded hardware cores in the implemented Processing Array. The vector add custom core was able to approach the maximum bandwidth provided by the MIG SDRAM controller for the 512 dataset. For the larger datasets, SDRAM row switching drastically reduced external bandwidth and for the matrix multiplication micro-benchmark, scaling was limited by the number of DSP blocks on the FPGA.

In conclusion, the Aggregating Memory Controller, Processing Array and Device Controller demonstrated an initial infrastructure that is able to be a foundation for further research. Some suggestions for research exploration is given in the next section.

6.1 Future Work

The Aggregating Memory Controller was only implemented with eight 32-bit ports and a 128-bit wide external interface. The AMC was written parametrically for the number of ports and data widths except for the sorting network, which currently has to be hand-coded. Further exploration of the number of ports and datapath sizes of the AMC could be performed. The first configuration could be a 256-bit wide external interface that can be provided by performing data-expansion over the 2:1 clock-crossing between the MIG SDRAM controller [23] and the AMC. In this configuration, either 16 32-bit wide ports or 8 64-bit wide ports should be used to take advantage of the larger external input data-width.

The current Xilinx MIG SDRAM controller [23] is quite limited. Only at synthesis can you specify the burst length of either four or eight when most SDRAM chips allow switching between four or eight dynamically. Additionally, it does not support DDR
burst interrupts that can be used to interrupt unnecessary parts of the bursts for more efficient bandwidth utilization. Both of these limitations prevent potential improvements to the AMC. More flexible burst granularity could give better performance for accesses that are not fully coalesced. A more versatile SDRAM controller would be beneficial to overall system performance.

The Processing Rows in the Array can also be greatly improved. There is a wide range of exploration in the area between soft-processors and hand-coded hardware. Soft-processor accelerators are an option that can be considered. The LegUp HLS project [28] is an open-source tool that can create such accelerators that are also designed to use the Tiger “MIPS” processor [2]. This tool can be utilized to fully replace or partially accelerate the current soft-processing cores.

The design space for HLS to full vector-threaded cores that replace full Processing Rows can also be explored. Within this space, problems such as thread divergence need to be investigated.

Finally, the current host interface was designed for the purpose of small scale testing and not for full scale benchmarks. To run full OpenCL applications with an x86 processor, the Device Controller host-interconnect needs to be upgraded. Previous work by Ahmed [21], created a working OpenCL Software Framework with a PCIe host interconnect. In the future, this could be integrated with the current system to create a complete OpenCL x86-FPGA system so that full benchmarks can be run.
Bibliography


