UT-OCL: AN OPENCL FRAMEWORK FOR EMBEDDED SYSTEMS USING XILINX
FPGAs

by

Vincent Mirian

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2016 by Vincent Mirian

Abstract

UT-OCL: An OpenCL Framework for Embedded Systems Using Xilinx FPGAs

Vincent Mirian

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2016

The number of heterogeneous components on a System-on-Chip (SoC) has continued to increase. Software developers leverage these heterogeneous systems by using high-level languages to enable the execution of applications. For the application to execute correctly, hardware support for features and constructs of the programming model need to be incorporated into the system.

OpenCL is a standard that enables the control and execution of kernels on heterogeneous systems. The standard garnered much interest in the FPGA community when two major FPGA vendors released CAD tools with a modified design flow to support the constructs and features of the standard. Unfortunately, this environment is closed and cannot be modified by the user, making the features and constructs of the standard difficult to explore.

The purpose of this work is to present UT-OCL, an open-source OpenCL framework for embedded systems on Xilinx FPGAs, and use UT-OCL to explore system architecture and device architecture features. By open-sourcing this framework, users can experiment with all aspects of OpenCL, primarily targeting FPGAs, including testing possible modifications to the standard as well as exploring the underlying computing architecture. The framework can also be used for a fair comparison between hardware accelerators (also known as devices in the OpenCL standard), since the environment and the testbenches are constant, leaving the devices as the only variable in the system.

This dissertation shows that the UT-OCL framework enables the exploration of a mechanism to efficiently transfer data between the host and device memory, a fair comparison for two versions of a CRC application and shows the trade-offs between resource utilization and

performance for a device using a network-on-chip paradigm. In addition, by using the framework, the dissertation explores six approaches implementing Shared Virtual Memory (SVM), a feature in the OpenCL specification that enables the host and device to share the same address space. Finally, this dissertation presents the first published implementation of a pipe that is compliant to the OpenCL specification.

I would like to dedicate this thesis to my beloved parents, Fabian and Marie-Juliette Mirian, to my lovely sister, Stéphanie Mirian, for their support throughout my education and most importantly my godmother, Bernadette Caradant, for the inspiration to continue my education.

Acknowledgements

First and foremost, I would like to thank Professor Paul Chow for his invaluable guidance and advice throughout the years. Thank you for supporting this project and making it as fun and exciting as it was stimulating and enlightening. I am grateful for the opportunity to learn so much from you not only from your extensive experience on technical matters, but on matters of professionalism and conduct as well. I am very grateful for the significant knowledge, guidance and patience provided by Paul Chow. Without Paul Chow, this project would definitely not be possible!

I would also like to thank my committee members, Natalie Enright-Jerger and Andreas Moshovos. Throughout this journey, they have provided a great deal of wisdom. A sincere thank you to Professor Enright-Jerger for providing insight on methodologies and strategies for a successful research, and to Professor Moshovos for his guidance on the technical matters and most importantly his anecdotes and insight on life.

Furthermore, I would like to thank everyone in the Chow group for their support and feedback over the years. It was great working with all of you and thank you for making this experience so enjoyable.

To the many friends that I have made throughout my graduate studies at the University of Toronto, thank you for great memories and priceless moments... I am grateful to call you my friends!

To my many friends outside of graduate studies, you have given me sound advice and reminded me that the world is a beautiful place... thank you for the reminder!

To the faculty members in the Electrical and Computer Engineering (ECE) department, thank you for your help and motivation. I am indebted to be part of this department and rub elbows with the elites in their fields. Most importantly, your actions inspired me to continuously

reach for higher standards... Thank you!

I would also like to acknowledge the financial support, as well as the equipment and software donations, that I have received from the following organizations that have made this research possible: the Canadian Microelectronics Corporation, the Natural Sciences and Engineering Research Council, the University of Toronto and Xilinx.

Contents

1	Introduction	1
1.1	Programming a Heterogeneous System-on-Chip	1
1.2	Motivation	2
1.3	Contributions	4
1.4	Thesis Organization	5
2	UT-OCL: An OpenCL Framework for Embedded Systems using FPGAs	6
2.1	OpenCL Overview	6
2.2	Limitations of OpenCL in embedded systems using FPGAs	8
2.3	The platform model	9
2.3.1	Host subsystem	9
2.3.2	Device subsystem	11
2.4	A Custom Device Example	14
2.5	The Execution model	16
2.6	The Memory model	17
2.7	The Programming model	19
2.8	Compiling host applications	20
2.9	Example flow for compiling and executing kernels	20
2.10	Debugging	23
2.11	Related Work	26
2.12	Experiments	28
2.12.1	Architectural changes applied to the host subsystem	28

2.12.2	CRC application	32
2.12.3	Architectural changes applied to a device	33
2.13	Conclusion	39
3	Shared Virtual Memory (SVM) in the OpenCL Standard	41
3.1	Details of the OpenCL Memory Model	42
3.2	Shared Virtual Memory (SVM)	45
3.3	Memory Management in UT-OCL	47
3.4	Implementing the Fine-Grained System	
	SVM Type in UT-OCL	49
3.4.1	The Components	50
3.4.2	The Approaches	51
3.5	Related Work	53
3.6	Results	54
3.6.1	Evaluating the proposed approaches	55
3.6.2	Evaluating UT-OCL with SVM and without SVM support	64
3.6.3	Resource Utilization	67
3.7	Conclusion	68
4	Pipes in the OpenCL Standard	69
4.1	Background	70
4.2	Proposed Hardware Implementation (Pipe-hw)	71
4.3	Pipe Software Driver	76
4.4	Related Work	77
4.5	Other pipe implementations	78
4.5.1	Software pipe implementations	78
4.5.2	Using off-the-shelf IPs	79
4.6	Results	80
4.6.1	Performance	80
4.6.2	Resource Utilization	85
4.7	Conclusion	86

5	Conclusions and Future Work	88
5.1	Summary	88
5.1.1	The UT-OCL Framework	89
5.1.2	Shared Virtual Memory	89
5.1.3	Pipe	90
5.2	Other Ph.D.-related Publications	90
5.3	Future Work	91
5.4	Recommendation for the OpenCL Standard	92
	Bibliography	93

List of Tables

2.1	Application's runtime for each topology relative to AXI-S	37
2.2	Average barrier performance for each topology relative to AXI-S	37
2.3	Resource utilization summary	39
3.1	Average runtime (cycles) per element access for the two scenarios.	58
3.2	Hit rate of the proposed approaches for 16 threads accessing 32768 and 131072 elements	59
3.3	Host-TLB-miss-runtime product of Intr+IOMMU, Intr+TLB_MGMT and PTE+IOMMU for: 16 threads accessing 32768 and 131072 elements	61
3.4	Resource Utilization	66
4.1	Resource utilization of the hardware kernel instances	82
4.2	Normalized Runtime of barrier-sw	84
4.3	Resource Utilization of the hardware IPs	85

List of Figures

2.1	Details of the OpenCL framework	7
2.2	Block diagram of the UT-OCL's hardware system	10
2.3	Kernel file structure	12
2.4	Block diagram of the OpenCL device used in the experiments	15
2.5	C code transformation example	22
2.6	Hardware system developed for debugging	24
2.7	Runtime of the Datamover core with the stream driver normalized to the runtime with iomem driver	30
2.8	Runtime of the Datamover core with the modified stream driver normalized to the runtime with iomem driver	31
2.9	Runtime of crc-sw normalized to crc-hw	33
2.10	Block diagram of the OpenCL device used in the experiments	34
2.11	Block diagram of an interconnect implementation using the special interconnect framework	35
3.1	Details of the OpenCL framework	43
3.2	Address space of the host and device in OpenCL 1.2	43
3.3	Address space of the host and device in OpenCL 2.0	44
3.4	Address space of the host and device using the Fine-Grained system SVM type .	46
3.5	Diagram of the UT-OCL framework	48
3.6	Virtual memory addressing scheme in the Linux kernel	48
3.7	Modifications applied to the hardware system for the proposed approaches	51

3.8	Runtime of the proposed approaches for 16 threads accessing 131072 elements for each pattern	56
3.9	Runtime in cycles (logscale) of the proposed approaches executing the linear pattern for 1, 2, 4, 8 and 16 threads assessing 131072 elements	62
3.10	Runtime in cycles (logscale) of the proposed approaches executing the page pat- tern for 1, 2, 4, 8 and 16 threads assessing 131072 elements	63
3.11	Runtime in cycles (logscale) of the proposed approaches executing the random pattern for 1, 2, 4, 8 and 16 threads assessing 131072 elements	63
3.12	Runtime using UT-OCL and UT-OCL+SVM	65
4.1	Executing kernel application with pipe support	70
4.2	Executing kernel application without pipe support	70
4.3	Pipe hardware implementation	72
4.4	Steps for writing to a pipe in parallel-to-serial buffer mode	73
4.5	Steps for using the pipe as a work group barrier	75
4.6	Gaussian Filter Application	81
4.7	Synthetic Network Application Runtime	83

Chapter 1

Introduction

As more features and functionality are integrated into today's commodity devices, to satisfy size constraints and power requirements, many of these features and functionalities are implemented by interconnecting various hardware components within a single chip, composing a System-on-Chip (SoC). With various types of computing components being integrated into these systems, they are becoming significantly more heterogeneous compared to SoCs in the previous decades.

Software developers leverage these systems by using high-level languages to enable the execution of applications. However, for these application to execute correctly, the underlying system requires hardware support for features and constructs of the programming paradigm. For example, programming with a multithreaded paradigm in a shared memory system requires hardware support for atomic operations to ensure correctness.

1.1 Programming a Heterogeneous System-on-Chip

Software developers can implement their application using different programming paradigms, where the paradigm highlights the manner to program the application. There can be standards that define a paradigm. For example, the message passing paradigm leverages a communication protocol for programming parallel computers, and the Message Passing Interface (MPI) [1] is an open standard that defines this paradigm. Until the release of OpenCL in 2009 [2], there were no standards that defined the control and execution of applications in a heterogeneous system.

OpenCL has garnered much interest in the FPGA community because it is a language based on an open standard that can run on many different heterogeneous platforms. In the FPGA community, Altera introduced an OpenCL compiler in 2013 [3] that truly raised the FPGA design abstraction to a much higher level than hardware description languages with the significant advantage that the code can be easily migrated between different kinds of computing platforms. Xilinx has announced its OpenCL compiler for FPGAs [4] in 2014, which raises the commercial interest in OpenCL for FPGAs even more because of the potential for portability between FPGA platforms.

Targeting FPGAs from OpenCL has its unique challenges because FPGAs are not simply processors using a typical software design flow. The FPGA architecture is much different than the common platforms (e.g. CPUs, GPUs) targeted by OpenCL implementations. For example, FPGA vendors have recently introduced programmable systems-on-chip (SoCs), where an SoC is coupled with FPGA fabric creating a ready-to-use platform for an embedded environment [5]. Furthermore, the flexibility of FPGAs, the long compile times and the potential for partial reconfiguration [6] are some of the challenges that FPGAs introduce, leaving much opportunity for OpenCL to adapt to this type of platform. Currently, there is no open-source solution for exploring an OpenCL-compatible framework, investigating platform designs and analyzing other challenges for an embedded system implementable on an FPGA. This dissertation presents UT-OCL, an OpenCL framework for embedded systems using Xilinx FPGAs. UT-OCL is an open-source framework that can be used to experiment with all aspects of OpenCL, primarily targeting FPGAs, including testing possible modifications to the standard as well as exploring the underlying computing architecture.

1.2 Motivation

Compared to common practices of programming for embedded systems for FPGAs, the higher level of abstraction provided by OpenCL should make application development for FPGAs much easier for software engineers. For example, during the testing/verification phase of the custom accelerator (called *devices* in the OpenCL platform model), minor changes to the source code are required for the user to compare the output of the device under test with that of other

functioning devices.

During testing, the developer can perform more extensive tests with greater ease since test cases are software. And when evaluating multiple devices using an open-source framework like UT-OCL, the environment and the testbenches are constant, leaving the devices as the only variable in the system. Therefore, the evaluation and the comparison between multiple devices are fair and easy to setup.

Furthermore, by using an OpenCL-compatible framework as a basis for building complex heterogeneous embedded systems, an application programmer can start the implementation using an OpenCL framework in a workstation environment, where the tools for debugging and development/prototyping are easier to use and more readily available. Once the programmer is satisfied that the application is functionally correct on the workstation, the application can be migrated to the embedded platform with more confidence that the application will work. With good high-level synthesis support, some of the code could be turned into hardware devices. The goal is to do most of the development in the friendlier workstation environment and make the *embedded* design more of a porting exercise. To support such a development process, the embedded system needs to provide the architectural support needed to model the workstation environment so that any changes required to do the migration are minimized. By adding support in the architecture, the programming abstraction is at a higher level, which reduces the time to develop in these systems.

While OpenCL defines a particular programming model, there is still lots of opportunity to explore the implementation of the hardware and software that supports the programming model as well as the programming model itself. This is especially important given that the standard continues to evolve and will. For example, in the OpenCL 2.0 specification [7], streaming capability amongst kernels is now present. It will be important to study the best ways to support streaming, also known as Pipes. To do this research, it is necessary to have a completely accessible framework that allows experimentation on both the hardware architecture and the software. Furthermore, when using an open-source framework, comparison between related studies using the framework would be easier and more fair. Hence, the motivation for the UT-OCL framework and architectural exploration at the system level.

1.3 Contributions

This dissertation has three significant contributions.

- The first contribution is the development of an open-source OpenCL framework for embedded systems using Xilinx FPGAs. FPGA vendors provide OpenCL frameworks, however these frameworks are closed restricting the user from modifying the OpenCL implementation and the underlining hardware system. With an open-source OpenCL framework, such as UT-OCL, the OpenCL implementation and hardware system is exposed to the user with the possibility of modifying various aspects of the framework. This contribution extends the capability of the OpenCL frameworks offered by FPGA vendors and presents a more versatile tool for the research community in the exploration of OpenCL within the context of FPGAs. This contribution is presented in Chapter 2 and encapsulates the content from my published works [8] [9].
- The second contribution is the architectural exploration at the system level for *Shared Virtual Memory (SVM)*, a feature in the OpenCL specification. This contribution is the study of six approaches implementing SVM in a two-domain system on an FPGA. Through this study, the trade-offs between the approaches are analyzed to suggest an efficient implementation of SVM that is better suited for OpenCL applications where the execution toggles between the host domain and the device domain. The observations from this study provide insight on the suitability of the FPGA architecture for the implementation of SVM support. In addition, these observations highlight advantages and limitations for a platform, where resource and architecture constraints are present, to guide future implementations of SVM on an FPGA. This contribution is presented in Chapter 3 and encapsulates the content from my published work found in [10] as well as additional results.
- The third contribution is the architectural exploration at the system level for a *pipe* object, a construct in the OpenCL specification. A pipe object enables kernel-to-kernel communication, which is used in streaming applications where FPGAs perform well, thus making their implementation critical for FPGAs to remain competitive as a computing

platform. This contribution focuses on an efficient implementation of the features that conforms a pipe object to the OpenCL standard in a challenging environment where there are a fixed amount of resource types available, such as in an FPGA. This contribution is presented in Chapter 4 and is the first published [11] implementation of this type of object that is compliant to the OpenCL specification.

In addition to these contributions, during my Doctor of Philosophy degree, exploration with coherent memory hierarchies on FPGAs was conducted. The content of this work was published in [12], [13] and [14], however it is not present in this dissertation as the exploration was not conducted with the aid of the UT-OCL framework. Nonetheless, the contributions of these works as well as other work on coherent memory hierarchies on FPGAs [15] [16] can be further explored within the UT-OCL framework to provide insight on coherent memory hierarchies within the context of OpenCL.

During the exploration of the second contribution, a security flaw in the Xilinx design flow was discovered. This security flaw was exploited to modify the Memory Management Unit (MMU) of the MicroBlaze microprocessor [17], a secure IP from the Xilinx IP library. Details of the methodology was published in [18].

1.4 Thesis Organization

The remainder of this dissertation has four chapters. Chapter 2 presents the various components of the UT-OCL framework by mapping these components to the OpenCL standard. The chapter also highlights the ease of using this framework for evaluating and comparing system characteristics for various applications. Shared Virtual Memory (SVM), a feature in the OpenCL standard, is architecturally explored at the system-level in Chapter 3. Chapter 4 demonstrates the first published implementation of a pipe object that is compliant to the OpenCL specification. The dissertation ends with concluding remarks and notes on future work in Chapter 5.

Chapter 2

UT-OCL: An OpenCL Framework for Embedded Systems using FPGAs

This chapter presents UT-OCL, an OpenCL framework for embedded systems using FPGAs. The framework is composed of a hardware system and its necessary software counterparts, which together form an embedded Linux system augmented to run OpenCL applications within a single FPGA. The framework contains debugging tools and simple hooks that allow for custom devices to be easily integrated in the hardware system. UT-OCL's OpenCL implementation is compliant with OpenCL 2.0 [7].

This chapter also presents an analysis of the OpenCL specification for an FPGA platform, and describes the challenges with implementing an OpenCL framework for embedded systems on FPGAs. The work presented in this chapter is based on my published works [8] and [9].

2.1 OpenCL Overview

For this thesis, it is only required to define the concepts of the OpenCL framework rather than provide the technical details. The technical details of the OpenCL framework can be found in the OpenCL specification [7].

OpenCL is best described using four models: the Platform Model, the Execution Model, the memory model and the programming model. The lower part of Figure 2.1 shows the Platform Model of the OpenCL framework. The *Host* is connected to one or more Devices. A Device is

Figure 2.1: Details of the OpenCL framework

composed of one or more *Compute Units*, and the Compute Units are composed of one or more *Processing Elements*. For the remainder of this thesis, a device is analogous to a hardware accelerator in an embedded system. In the example given in Figure 2.1, *Device B* has two compute units, each with three processing elements.

The Execution Model of an OpenCL application has two parts. One part, known as the Host Application, executes on the Host. It is responsible for most of the runtime control of the OpenCL application. The other part is called the Kernel, which executes on the Processing Elements.

Traditionally, when developing with OpenCL, the Host application is executed on a CPU running an Operating System (OS), whereas the Kernels executed on the Processing Elements do not have OS support. In an OpenCL framework, the Host application uses a compiler (JIT, interpreter, etc..) and an OpenCL implementation to execute the source on the Host, and the Kernel is compiled and executes on the Device using a driver.

An OpenCL implementation implements the functions defined by the OpenCL specification, including the functions for managing the devices, such as querying for device information and queueing device tasks. Device tasks are queued in a *command queue*, where a scheduler assigns the tasks to the device. In addition, the OpenCL specification includes definitions for built-in functions that the device should be able to support, such as work-group synchronization using a barrier.

A *work-group* is a virtual N -dimensional indexed space defined for the execution of the kernel. One kernel instance is executed for each point in this indexed space, where each kernel instance is executed by a *work-item*.

There are four memory types in the memory model:

1. *global memory*, which has read/write access from the host and work-items;
2. *constant memory*, which is read-only memory from the work-items, and, is initialized by the host;
3. *local memory*, which is memory accessible by a work-group; and,
4. *private memory*, which is only accessible by a work-item.

Under the OpenCL programming model, computation can be done in data parallel, task parallel, or a hybrid of these two models.

The UT-OCL framework described in this chapter provides architectural support for the platform model and makes it easy for the user to integrate custom hardware devices. The framework also provides the source code for the execution, memory and programming model with *hooks* for the user to add support for their custom device.

In the remainder of this thesis, the term *OpenCL framework* refers to the OpenCL API, compiler (JIT, interpreter, etc..) and driver allowing the OpenCL application to execute on the platform.

2.2 Limitations of OpenCL in embedded systems using FPGAs

In OpenCL, the constructs used in device management are strongly influenced by GPUs and SIMD-like architectures. These types of devices are commonly found in heterogeneous work-station machines, but are not common in embedded systems. Typically, in embedded systems, devices are auxiliary peripherals that aid in computational tasks as hardware accelerators that are memory-mapped, which facilitates hardware/software interactions.

In OpenCL, the platform model of the device is compute units. When the device is an FPGA, or a region of an FPGA, there need not be any defined compute unit architecture that

is the device. Instead, the FPGA is really a *blank* device that can be made specific by the application developer. This property is not considered in the OpenCL standard and is one reason that FPGAs require different treatment in the OpenCL model. In addition, when developing an OpenCL implementation for hardware accelerators using the current specification, all devices on FPGAs are limited to the *custom* device type, which have very limited constructs compared to other type of devices (e.g. GPUs). For example, custom devices should only support built-in functions, and cannot be programmed via OpenCL C. However, by enabling custom devices to be programmed using OpenCL C, high-level synthesis tools can use the constructs in OpenCL C to generate better hardware in terms of power, performance and resource utilization. Moreover, by providing additional constructs and properties, the full capabilities of FPGAs can be leveraged, including that of partial reconfiguration [6]. For example, FPGA compile times are significant, so it is not realistic to do the compilation for an FPGA-based device at run time. Another approach is required. In addition, some infrastructure for configuring the FPGA at runtime is required.

2.3 The platform model

Figure 2.2 shows a block diagram of the hardware system of the UT-OCL framework. Much like the standard OpenCL platform model, the hardware system is composed of two subsystems: a subsystem executing the host application (Host subsystem), and another subsystem executing the kernels (Device subsystem). The hardware system is implemented on the ML605 development platform [19]. The remainder of this section will describe details of these subsystems.

2.3.1 Host subsystem

The vast majority of OpenCL implementations run on CPUs running an Operating System (OS), thus providing the host application with OS support. To allow for a host application to be easily ported to the UT-OCL framework, which is targeted for embedded systems on FPGAs, the host subsystem should be capable of running an OS.

PetaLinux is a tool suite to customize, build and deploy Embedded Linux solutions on Xilinx FPGAs. To build this subsystem, the instructions in the *PetaLinux SDK User Guide*:

Figure 2.2: Block diagram of the UT-OCL's hardware system

Board Bringup Guide [20] were followed. System peripherals that are only used during the OS boot process (e.g. the GPIO, FLASH controller, debug module and ethernet controller) are not shown in Figure 2.2. The Host is implemented using a MicroBlaze microprocessor [17], and connects to the Device SubSystem Manager using an AXI Stream interconnect.

The host subsystem communicates with the device subsystem using the stream port. When the MicroBlaze is configured to use the Memory Management Unit (MMU), the instructions to access the stream ports are privileged, meaning these instructions can only be executed in the kernel space of the operating system. A device driver, in the form of a Loadable Kernel Module (LKM) was implemented to allow a user process running on the Linux OS to execute these instructions. When using a LKM, the device is represented by a file. In the implementation of the stream LKM, the stream ports are represented by a file, and, the functions, that overload the read and write functions, execute the corresponding stream instruction. From hereafter, this LKM will be referred to as the *stream driver*.

A good OpenCL implementation requires the host subsystem to send to and receive from the device subsystem concurrently. Hence, the driver is implemented with two virtual buffers,

one for each direction of the stream port, and two kernel threads managing these buffers. By having two threads, the communication between the host subsystem and the device subsystem can occur concurrently, however OS overhead for managing these threads is introduced.

In the Linux Operating System, memory accesses use virtual memory and paging. Unfortunately, the device subsystem does not have access to the memory management mechanism in Linux. However, in an OpenCL applications with the execution of kernels, the host application and the kernels typically share data. Therefore, it is necessary for the addressing scheme between the host and device subsystems to be compatible, so they can reference the same data. This issue and the proposed solution is further described in Section 2.6.

2.3.2 Device subsystem

The device subsystem is composed of four major components: the Device SubSystem Manager (*DevSubSys*), the Kernel Database Manager (*KernelDB*), Pipe components (*Pipe*) and a Device Manager (*DevManager*).

The major components within the Device subsystem communicate with each other using a message passing paradigm. Each major component is associated with a mailbox containing a 1024 32bit-width buffer, and a mutex component containing a mutex variable. The mailbox and the mutex components are memory-mapped and their addresses are known by their senders. (Note: The mailbox and the mutex components are omitted from Figure 2.2.)

There can be at most one sender to a major component at any given time. As a result, the sender must acquire the mutex lock prior to sending a message and must release the mutex lock once the sender has completed the message transfer. All messages have a fixed-sized message header containing the packet type and length, and a variable-sized message body. The messages are also designed to utilize the smallest memory footprint (number of bytes) possible.

The Device SubSystem Manager (*DevSubSys*)

The DevSubSys is implemented using a MicroBlaze microprocessor [17]. It is the communication portal for control and requests between the Host and the devices in the system. In addition, it is responsible for setting up the Pipes by assigning their pipe packet width and maximum pipe depth, routing messages from the Host to the appropriate DevManager, and sending requests

Figure 2.3: Kernel file structure

to the KernelDB.

The Kernel Database (*KernelDB*)

The KernelDB is also implemented using a MicroBlaze microprocessor [17]. It is responsible for accessing kernel information. The kernel information for individual kernels is stored in a file that the KernelDB accesses using the xilfatfs library [21]. The xilfatfs library provides support for the KernelDB to access a FAT16 file system, which is the file system required for configuring the FPGA at startup. Choosing a FAT16 file system with the xilfatfs library preserves the feature of configuring the FPGA at startup whilst being able to read a kernel from a file within the framework.

Each device is associated to a directory labelled with its unique device ID. Each file within the directory corresponds to a kernel executable by the device. These files are required to be in binary format with the structure shown in Figure 2.3. Each kernel file contains four entries: *kernel name*, *kernel attributes*, *kernel arguments* and *kernel body*. The first byte of the *kernel name* entry corresponds to the length of the kernel name and the following bytes correspond to the kernel name. The first byte of the *kernel attribute* entry contains the number of attributes that follow. Each attribute has two fields: size, corresponding to the size in bytes of the attribute's value, and its value. Similarly, the *kernel arguments* entry has the same fields and organization as the *kernel attribute* entry, where the value field corresponds to the default

value for the argument. The first four bytes of the *kernel body* entry corresponds to the number of bytes of the kernel body, the remaining bytes is the kernel body data, which corresponds to the raw binary needed by a processing element for kernel execution. For example, for a CPU type device, the kernel body data refers to the instructions to execute.

OpenCL defines two kinds of platform profiles: a full profile and a reduced-functionality embedded profile. A full profile platform must provide an online compiler for all its devices. An embedded platform may provide an online compiler, but is not required to do so.

Given that the framework is targeted for embedded systems, where the host's processing power is an order of magnitude less than current desktop computers and storage is scarce, the OpenCL implementation will follow the Embedded Profile. In other words, the kernels must be compiled on a separate system prior to execution in the framework.

For dynamically loadable kernels, the file containing kernel information must reside in a Compact Flash device. The Compact Flash device was selected because: 1) the memory is non-volatile, so the kernels do not have to be loaded in memory after every boot, 2) it is the storage medium for configuring the FPGA at startup using the ML605 development board, and 3) large memory capacities for a Compact Flash can be acquired at a relatively low-cost.

Furthermore, a reference design demonstrating Partial Reconfiguration (PR) on development boards, like the board used in this dissertation, uses the Compact Flash as storage for the bitstream for reconfigurable regions. PR is a technology where a region of an FPGA can be re-configured while the remaining region remains static [6]. Although this feature is currently not implemented in the framework, future plans include the support for PR where the kernel can dynamically reconfigure regions of the FPGA. The majority of this infrastructure is currently in place.

Pipe component (*Pipe*)

The pipe components are custom peripherals representing the physical implementation of the pipe object in the OpenCL implementation. This section presents a high-level overview of the pipe object, a thorough description on pipes can be found in Chapter 4. The pipe components are first-in-first-out (FIFO) structures with a maximum packet size of four bytes (32 bits), and a maximum pipe depth of 1024. For larger configurations of pipe objects, multiple Pipes would

need to be used.

There are four pipe components in the Device subsystem, and like the mailbox and mutex components, these components are memory-mapped. Their addresses are preset and known by the DevSubSys, which manages their availability.

The Device Manager (DevManager)

The DevManager is also implemented using a MicroBlaze microprocessor [17]. One DevManager is required per device in the system. It is responsible for managing and interfacing to the device. For example, the DevManager receives the kernel arguments set by *clSetKernelArg* and the work dimensions specified in the *clEnqueueNDRangeKernel*, as well as the kernel body from the KernelDB.

A template of the DevManager source code is provided in the framework. The user is required to fill-out a table providing the device attributes. These devices attributes correspond to the *cl_device_info* definitions in Table 4.3 of the OpenCL specification [7], which is used by the DevManager to respond to the query requests (*clGetDeviceInfo*).

In addition to filling-out a table, the user is responsible for implementing the logic for mapping a work-item to a processing element. The user is responsible for connecting the compute units of the device to the DevManager. An example of this connection is described in Section 2.4.

The DevManager needs to notify the DevSubsys when a kernel has completed its execution on the device. As a result, the device is required to notify the DevManager when the kernel has completed, and the user needs to add logic to support this feature. An example of this feature is described in Section 2.4.

2.4 A Custom Device Example

As mentioned, the DevManager is implemented using a MicroBlaze, which has a local memory bus (LMB) [22], a data port with an AXI4 interface and 16 stream ports. In this example, a compute unit is connected to a stream port. A consequence of using stream ports to connect the compute units to the DevManager is that the device is limited to a maximum of 16 compute

Figure 2.4: Block diagram of the OpenCL device used in the experiments

units. This should be noted by the device designer if this port is used to connect its device to the DevManager.

Figure 2.4 shows the block diagram of a custom device. The device consists of a single compute unit with two processing elements. When using this device design, a work-group maps to a compute unit and a work items maps to a processing element. The processing elements are chosen to be MicroBlaze microprocessors [17] (*PEBlaze*), since it is software programmable making the development much easier. However, if desired, hardware engines can substitute the *PEBlaze*.

Each *PEBlaze* has its own private memory, which contains its instructions and data. In addition to processing elements, the device has a local memory accessible by both *PEBlazes* represented by the (*Local Mem*) component, and a *Mutex* component implemented using the XPS Mutex Core [23] from Xilinx. These components will aid in implementing the OpenCL work-group synchronization built-in function, and the contents of these components are initialized by the DevManager.

The private memory of both *PEBlazes* is initialized to a custom bootloader that retrieves the kernel body data (processor instructions), the kernel arguments, work dimension information and work-item information from the DevManager in the correct order. There is also a connection from the compute units to global memory, and the details of this connection is described in Section 2.6.

A custom router was developed to connect the compute unit to the DevManager. The router has a single master port and can have one or more slave ports. The master port can send to one or more slave ports, and slave ports can only send to the master port. The master port is connected to the DevManager and the slave ports are connected to the PEBlazes. For the custom device shown in Figure 2.4, there are two slave ports, one for each PEBlaze.

Upon completion of the kernel on a PEBlaze, the PEBlaze sends a notification through the slave port. The DevManager configures the router to receive from each slave port in a round-robin fashion. When both notifications are received by the DevManager, it sends a notification to the DevSubSys stating that the kernel execution has completed on the device, at which time, the PEBlazes return their execution to the bootloader.

There are other alternatives in connecting the compute units to the DevManager depending on how the compute units and processing elements are defined. For example, for this custom device, each MicroBlaze could have been connected to an individual stream port at the expense of implementing a different algorithm in the DevManager for mapping multiple stream ports to a single compute unit, but omit the development of the custom router. However, the implementation of the custom router is better suited for the target FPGA architecture than the alternative approach of having each compute unit connect to a stream port of the DevManager.

As mentioned, the MicroBlaze does have other ports for communication with external peripherals. As a result, the communication between the DevManager and the device did not have to use the stream port of the MicroBlaze. The user could have replicated the infrastructure of the message paradigm used by the four major components of the device subsystem, which uses the AXI port.

2.5 The Execution model

In UT-OCL's OpenCL implementation, the command queue objects are implemented in software on the Host subsystem, and the task state's follow that in Section 3.2 of the OpenCL specification. To select a command to execute from the multiple command queues associated with a device object, a software scheduler is implemented. The scheduler uses a round-robin priority scheme to select amongst the multiple command queues from the device. The sched-

uler implements in-order execution with support for synchronization commands as described in Section 3.2 of the OpenCL specifications.

The scheduler is implemented in a separate software thread that is created on the first call to *clCreateCommandQueueWithProperties* on a given device. With the scheduler implemented in software, it is possible to quickly experiment with different scheduling algorithms, profile their performance and accelerate computationally heavy tasks using hardware on the FPGA. Support for device command queues is the responsibility of the device designer, however a generic device command queue is provided for use by devices.

On the execution of the *clEnqueueNDRangeKernel* command, the Host MicroBlaze sends the kernel arguments, work dimension and the kernel name to the DevSubSys. The DevSubSys sends the kernel arguments and work dimension to the DevManager of the device that will execute the kernel, and sends the kernel name to the KernelDB. The KernelDB sends the kernel body data to the DevManager of the device that will execute the kernel.

The OpenCL specification provides a profiling interface for the tasks in the command queue. A naive implementation of the profile interface is using the system calls provided by the OS that manage time. In the Linux kernel, the notion of time is represented using a timer peripheral that interrupts the kernel (similar to a watchdog timer). However, when using the non-blocking stream instruction from the MicroBlaze processor, status bits are set and need to be read immediately after a stream instruction is executed. As a result, interrupts must be disabled to prevent scheduling of other instructions after the stream instruction, increasing the inaccuracy of the time management in the kernel used by the profiling interface.

Therefore, a 64-bit timer peripheral was added to the system to implement the profiling interface. The timer peripheral is implemented using an AXI4-lite interface. Two read operations are required to read the 64-bit timer. A software mechanism is used to check for roll-overs.

2.6 The Memory model

The host application runs on the Linux Operating System, which uses virtual memory and paging to access memory. Unfortunately, the device subsystem does not have access to the memory management mechanism in Linux. However, when kernels are executed in OpenCL

application, the host application and the kernels typically share data. Therefore, it is necessary for the addressing scheme between the host and device subsystems to be compatible, so they can reference the same data.

To solve this issue, the physical memory is partitioned into two equal sized partitions. The first partition (Linux partition) is used and managed by the Linux OS. The second partition (shared partition) is accessed by the host application and the kernels. Both parts access the same data in the shared partition by using physical addresses. By using this approach, both subsystems, thus the host application and the kernels, will use the same addressing scheme to access data, thus solving the issue.

To further manage the shared partition, a dynamic memory allocator was implemented in software. The function signatures of the dynamic memory allocation for the shared partition are identical to those used in the C standard library (*malloc*, *free*, etc..), with the exception that they also require the allocation list as a parameter. The algorithm allocates to continuous memory and does not permit fragmentation.

For the host application to access the physical addresses of the shared partition, another LKM was implemented. In the implementation of the LKM, the shared partition is modelled by a file, the memory address is modelled by the file cursor, and reading from and writing to the shared partition overloads the functions for reading from and writing to a file. Therefore, to read from and write to the shared partition, the read and write functions are simply called on the file descriptor representing the device. From hereafter, this LKM will be referred to as the *iomem driver*.

The host application and the Linux OS are executed on the host MicroBlaze. As per the instructions in [20], the Host MicroBlaze has its data and instruction caches enabled to increase the performance of the OS running on the processor. Since there is no coherency scheme between the two subsystems, the host MicroBlaze was modified to only cache the address range from the Linux partition. As a consequence of this modification, only the memory requests for the Linux partition are sent to the memory bus, so an AXI-to-AXI connector [24] (the dotted line in Figure 2.2) was added between the peripheral bus (*Peripheral Bus* in Figure 2.2) and the global memory bus (*Global Memory Bus* in Figure 2.2), permitting the host MicroBlaze to access the contents in the shared partition. The port connected to the peripheral bus does not

have a mechanism for memory bursting as does the port connected to the memory bus. Thus, with the new changes, accessing continuous addresses through the peripheral bus requires more time to complete compared to accessing continuous addresses through the memory bus. In the experiments found in Section 2.12, this current setup is compared to a setup with a core that performs burst accesses to the shared partition using the stream driver.

In conformance to the OpenCL memory model, the shared partition represents the global memory, hence the connection from device to global memory bus in Figure 2.4. And when using this custom device as a reference, the Local Mem component represents the local memory of a work-group and the PEBLaze’s private memory represents the private memory of a work-item. The Host memory is represented by the Linux Partition.

Chapter 3 enables shared virtual memory (SVM) that allows the host and device to access the same virtual address space. With SVM, the complete physical memory is allocated to the Linux OS, and the time needed to transfer the data to and from the Linux partition to the shared partition is no longer suffered. Moreover, with SVM, security risks with kernels accessing another kernel’s memory space is managed and supported by the OS.

2.7 The Programming model

In the OpenCL programming model, the host program queries the hardware system for available platforms using the *clGetPlatformIDs* function. This function returns a list of available platform objects (*cl_platform_id*), and is generally the first OpenCL API call within a host application. By using these platform objects, the host program can identify the devices in the hardware system that can be supported by the OpenCL implementations.

In UT-OCL’s OpenCL implementation, the *clGetPlatformIDs* function initializes the stream driver and iomem driver if they are not already initialized. The platform object in the implementation is a singleton object, where a single instance of this object is present in the system to reduce the memory footprint. The platform object contains the device file names for the stream and iomem drivers, a pointer to the allocation list used by the dynamic memory allocator for the shared partition and a list of memory objects (*cl_mem*) created in the system.

In the OpenCL memory model, the global memory is accessible by both the host and

device. As a result, memory objects of type *buffer* created using *clCreateBuffer* must use the *CL_MEM_COPY_HOST_PTR* flag to copy the data from the host memory (Linux Partition) to the global memory (shared partition). When a memory object of type *buffer* is an argument to the kernel, the implementation retrieves the physical memory address in the shared partition corresponding to the memory object and sends it to the kernel. In addition, the OpenCL implementation maps the four Pipes found in the device subsystem to the memory object of type *pipe* created using *clCreatePipe*. Hence the list of memory objects stored in the platform object is used to identify memory objects that are used as kernel arguments, so the correct memory reference is passed to the kernel, and to keep track of available Pipes.

All objects within the implementation have a reference count variable, which represents the number of references to this object, and a mutex variable guarding the reference count variable. Retaining an object increments the reference count, and releasing an object decrements the reference count. When the variable reaches zero, the memory allocated for the object is freed. When releasing the last command queue object associated with a device object, the scheduler thread is destroyed to reduce CPU utilization. Object implementations conform to the OpenCL class diagram found on the OpenCL 2.0 Reference Card [25].

2.8 Compiling host applications

To compile the host application, a GNU compiler is required. The compiler can be acquired by installing the Petalinux Enviroment [26]. The application also needs to link UT-OCL's OpenCL implementation, as well as the user libraries for accessing the LKMs and the dynamic memory allocator of the shared partition.

2.9 Example flow for compiling and executing kernels

This section describes the flow for compiling and executing kernels on the custom device introduced in Section 2.4. Given that the device is a custom device, the custom device should only support built-in functions, and cannot be programmed via OpenCL C.

For the MicroBlaze processor, the CLang [27] front end compiler paired with LLVM [28] can compile OpenCL C kernels into MicroBlaze executables. Unfortunately, LLVM does not

support MicroBlaze architectures with advanced features like hardware multipliers or dividers. Therefore, using this compiler would not be feasible for all configurations of MicroBlaze. However, in the early stages of debugging and testing this device, this compiler was used to generate kernels for the device. Moreover, it is important to note that the OpenCL specification states that custom devices cannot be programmed via OpenCL C.

In the UT-OCL framework, where the devices are custom due to the versatility of the FPGA, disallowing OpenCL C code on a custom device limits the expansion of OpenCL to conform to the FPGA platform. From a research perspective, enabling the execution of OpenCL C code on custom devices on a platform like FPGAs would allow the use of the Standard Portable Intermediate Representation (SPIR), containing OpenCL C extensions, in high-level synthesis. The HLS tool can exploit these extensions to generate better hardware. Such extensions can also be used in the generation of coarse grain reconfigurable computing elements, or in allowing for architectural exploration within the custom device. Hence, such a restriction should be relaxed due to the programming and architectural versatility of an FPGA.

Nonetheless, in the end, a Python script was developed to transform the kernel written with OpenCL-C extensions to C code. An example OpenCL-C extensions to C code transformation is shown in Figure 2.5. The script reads the kernel's argument list and removes any OpenCL-C extensions from the kernel's body (line 1 of Figure 2.5a to line 4 of Figure 2.5b). Then, the script creates a main function that reads the arguments in-order from the stream port, as well as work information and work-item attributes (lines 12 to 26 of Figure 2.5b). Finally, the script adds an instruction to write to the stream port at the end of the main function (lines 28 of Figure 2.5b). This instruction notifies the DevManager that the kernel execution is finished. The script also has a built-in memory allocator for the Local Mem component that translates all references to the local memory in the kernel (line 2 of Figure 2.5a) to the correct memory address of the Local Mem component on the device (line 5 of Figure 2.5b).

Once the transformation is complete, a GNU compiler acquired from the Embedded System Edition of the Xilinx Tool Suite is used to compile the kernel. Built-in kernel functions, a custom linker script and C-Run-Time libraries have been developed and are linked into the kernel during compilation.

After the compilation, an ELF-to-binary tool reads the kernel source code and the ELF file

```

1  __kernel void kernel_example (__global float* input, __global float* output){
2  __local int tempArray[16];
3  //work
4  }

```

(a) OpenCL Kernel Example

```

1  #include <fsl.h>
2
3  #define LOCAL_MEM_ADDR 0x60008000
4  void kernel_example (float* input, float* output){
5      int *tempArray = (int *) LOCAL_MEM_ADDR;
6      //work
7  }
8
9  #define READ_STREAM_INSTRUCTION(x) fslget(x, 0, FSL_DEFAULT)
10 #define WRITE_STREAM_INSTRUCTION(x) fslput(x, 0, FSL_DEFAULT)
11 void main (){
12     int work_dimension, global_work_offset[3], global_work_size[3],
13         local_work_offset[3];
14     float *input;
15     float *output;
16     READ_STREAM_INSTRUCTION(input)
17     READ_STREAM_INSTRUCTION(output)
18     READ_STREAM_INSTRUCTION(work_dimension)
19     READ_STREAM_INSTRUCTION(global_work_offset[0])
20     READ_STREAM_INSTRUCTION(global_work_offset[1])
21     READ_STREAM_INSTRUCTION(global_work_offset[2])
22     READ_STREAM_INSTRUCTION(global_work_size[0])
23     READ_STREAM_INSTRUCTION(global_work_size[1])
24     READ_STREAM_INSTRUCTION(global_work_size[2])
25     READ_STREAM_INSTRUCTION(local_work_size[0])
26     READ_STREAM_INSTRUCTION(local_work_size[1])
27     READ_STREAM_INSTRUCTION(local_work_size[2])
28     kernel_example(input, output);
29     int notification = 0xFFFFFFFF;
30     WRITE_STREAM_INSTRUCTION(notification)
31 }

```

(b) C kernel transformation

Figure 2.5: C code transformation example

to generate a binary file that has the format from Figure 2.3. The tool reads the kernel’s ELF file and extracts the data and instructions needed for the execution of the kernel. The user is responsible for implementing the communication protocol between the DevManager and the PEBlazes that includes transferring the work-item attributes and other device dependent data.

As mentioned, the kernels must be compiled on a separate system prior to execution in UT-OCL’s framework. This is especially important for FPGAs supporting PR, since vendor CAD tools require lots of processing power, memory and time for generating a bitstream from HDL.

2.10 Debugging

Our UT-OCL framework provides tools for debugging the software and hardware system. To debug all software parts running on the host subsystem, the Petalinux tool suite provides a machine emulation tool using QEMU [29]. The emulation environment disables any devices for which there is no model available. For this reason it is not possible to use QEMU to test your own custom device. However, instructions on integrating a model for custom devices can be found in [30].

In addition to a machine emulator, the GNU compiler provided within the Petalinux tool suite supports integration with the GNU Project Debugger (GDB) [31]. As a result, the user can debug the host application with a familiar tool used to debug software on desktop workstations. By using a familiar tool like GDB, less time is required for porting an OpenCL application to the UT-OCL framework.

On the ML605 development board, there are a few I/O peripherals that enable the hardware system to communicate with the outside world. Amongst these peripherals: the LCD screen, buttons, LEDs, DVI and USB peripheral are not currently used by the hardware system. The LCD screen cannot display sufficient debug information on a single screen for it to be useful to the user. Even with the LCD screen coupled with buttons to allow more virtual screens, this debugging interface is cumbersome.

Amongst these peripherals, the DVI and USB host port are the better options. However the DVI requires an additional monitor to display the debug information, and the USB host

Figure 2.6: Hardware system developed for debugging

port does not have API support for the ML605. As a result, the RS-232 UART peripheral is selected because most development boards have this peripheral and it is well supported by the vendor tools with available IP cores and software support for the MicroBlaze.

Figure 2.6 shows a block diagram of the debugging hardware system. Only two hardware modifications are needed to create the debugging hardware system from the original hardware system described in Section 2.3. The first modification is to connect the DevSubSys, the KernelDB and the DevManager to the RS-232 IP core (labelled with *Added Bus* in Figure 2.6). The second modification is to add a Mutex core with a single variable to the hardware system. The mutex variable allows exclusive access to the RS-232 by the host MicroBlaze, DevSubSys, the KernelDB and the DevManager. By sharing the RS-232 IP core in the debugging hardware system, the host MicroBlaze no longer runs a Linux Operating System, since the RS-232 IP core is used as the console for the Linux Operating System. As a result, all applications executed by the host MicroBlaze must run as bare metal.

To display debug information, a debug library is implemented that also supports the hardware changes discussed above. The library displays the name of the component the debug information originates from, followed by memory content in hexadecimal format using a print function. The print function implemented is smaller in size compared to the *printf* function in the standard I/O library (stdio), because the UT-OCL hardware system is targeted for

embedded systems where memory is scarce.

With the lack of the OS support on the host MicroBlaze, the OpenCL application can no longer execute on it within the debugging hardware system. In the end, the behavior of the host MicroBlaze when executing the OpenCL application with OS support needs to be captured. When executing an OpenCL application, the host MicroBlaze is responsible for initializing the global memory and sending messages to the DevSubSys. As a result, the stream LKM was modified to log the stream instructions executed by the host MicroBlaze into a file. This log file can then be used by a script that generates a bare metal application replicating the same stream instructions from the log file.

The debug system also has the feature to dump and restore the data in the shared partition. When using the hardware system, the KernelDB is able to read the contents of the shared partition and store it into a file on the Compact Flash device prior to sending the kernel data body to the device. During the boot-up of the debugging hardware system, the file containing the contents of the shared partition is restored. This feature is feasible since large storage for the compact flash is available at low cost.

In the current design of the debugging hardware system, the user can retrieve debug information from the custom device through the DevManager. However, it is left to the user to implement a communication protocol between the DevManager and the device representing the number of transactions containing debug information prior to sending the kernel completion notification.

However, if the user would like to have a deeper view inside the custom device, the user is required to connect the device to the RS-232 UART IP peripheral and the Mutex core, and then implement support for displaying the debug information in the correct format. Future work consists of implementing a hardware abstraction layer that implements the functionality in the debug library mentioned earlier for generic use by custom devices.

Another method for debugging the hardware system provided by the UT-OCL framework is hardware simulation. The Embedded System Edition of the Xilinx Tool Suite can produce a simulation environment for debugging the hardware system. The simulation framework is not ideal since the global memory is large and simulating large memories would be quite slow. However, this method is recommended to test the custom device's hardware functionality if a

large global memory size is not needed.

The debugging hardware system was used to debug the communication amongst the four major components in the device subsystem in the early stages of development. When debugging an OpenCL application, the scope of the issue can be isolated by executing the application on the debugging hardware system as well as the hardware system. For example, if an issue is observed when the OpenCL application executes on the hardware system as well as the debugging hardware system, the scope of the issue is limited to the communication between the host and device or to an issue in the device subsystem. If the communication protocol and messages sent from the host to the device are correct, then the issue would reside in the device subsystem of the hardware system. Therefore, the use of the debugging hardware system aids in isolating an observed issue during the execution of an OpenCL application.

2.11 Related Work

Portable Computing Language (pocl) [32] is an open source implementation of the OpenCL standard that can be easily adapted for new targets and devices, both for homogeneous CPU and heterogeneous GPUs/accelerators. In contrast to pocl, UT-OCL is targeted for embedded systems using Xilinx FPGAs.

While there have been several efforts exploring the use of OpenCL as a description language for high-level synthesis, that is not the goal of this work. The remainder of this section describes prior work that has the goal of developing platforms that can execute OpenCL applications in the context of FPGAs.

Tomiya [33] presents SMYLE OpenCL: A Programming Framework for Embedded Many-core SoCs. In their framework, they analyze the host source program to identify the type and the size of the various OpenCL objects. Then, they statically reserve memory space for the objects in shared memory. As a consequence, they must statically map the kernels onto the devices in the system. In UT-OCL's framework, the kernels can be mapped to the devices dynamically. Furthermore, memory for the objects is also allocated dynamically. Such features make the framework more versatile during runtime, where the device can execute more than a single kernel.

Similar to SMYLE OpenCL, Ma et al. [34] present a design flow that analyzes the source code of an OpenCL application and generates the hardware platform, as well as the corresponding executable running on the platform. In their design flow, OpenCL constructs are mapped to components from a predefined system model [35]. For example, the work-items are transformed into “HybridThreads” (Hthreads) [36], so they can execute in their SoC designed for FPGAs.

Contrary to Ma et al. [34], the design of the UT-OCL hardware system is independent of the source code of an OpenCL application. Moreover, the host application is executed on an OS. As mentioned, enabling the host application to execute on an OS provides the user with abstractions available when developing with OpenCL in a workstation environment. This makes the porting of an OpenCL application to the framework much easier.

As shown with Ma et al. [34], an OpenCL application exhibits properties of a threaded paradigm. For example, a work-item can be modelled by a thread. Some works merge a software threaded paradigm onto reconfigurable platforms [37] [38] [39]. Applications running on these systems have the context of a thread, which is managed and scheduled by the OS.

In an OpenCL framework, a work-item is not an entity managed by the part of the OpenCL application running on the host, but an entity managed by the device. Therefore, unlike the threaded systems for reconfigurable platforms [37] [38] [39], the UT-OCL framework manages and schedules the work-items outside the scope of the OS, more specifically in the device subsystem.

In the work presented by Ahmed et al. [40], the FPGA platform was incorporated into an OpenCL framework to enable the execution of kernels on an FPGA. The host application is executed on a CPU running the Linux OS, and the kernels are executed on the FPGA. In UT-OCL’s framework, the host application is executed on a processor that resides in the FPGA, bringing the OpenCL framework into embedded systems. As a result, the communication infrastructure between the host and the device differ within these two systems. In Ahmed et al., the communication of the host and the device is performed off-chip through Peripheral Component Interconnect (PCI) Express, whereas, in UT-OCL’s framework, the communication of the host and the device is done on-chip. More fundamentally, the amount of available memory differ on these systems. The abundance of memory on the desktop system removes the worry of how much memory is consumed, whereas in the embedded system, there are memory limitations,

creating an additional constraint for the developer.

In contrast to the work presented thus far, some work [41] [42] [4] extend the OpenCL framework to a high-level-synthesis tool. Such an application of the OpenCL framework is not the intent of this work, but the addition of high-level synthesis into the framework for generating custom accelerators would significantly improve the means for creating the custom devices and, as mentioned, a prospective enhancement to the framework.

2.12 Experiments

This chapter describes the many features of the UT-OCL framework as well as highlights research potential when using this open-source framework. Hence, the practicality of the framework is demonstrated by evaluating architectural changes applied to the hardware system, the performance of an application performing a cyclic redundancy check (CRC) and the impact of various interconnect implementations in a device.

All experiments have been executed using the ML605 development board with designs targeting a 100 MHz system clock. The runtimes were recorded using the profile interface in UT-OCL. To account for OS overhead and noise such as context switching and scheduling, ten runs of the same experiment were executed and the average of these runs was taken.

2.12.1 Architectural changes applied to the host subsystem

As mentioned in Section 2.6, the host subsystem is designed with an iomem driver to access the physical address of the shared partition. Such a design enables the addressing scheme between the host and device subsystems to be compatible, so they can reference the same data. A consequence of using this design is that the host processor is unable to perform burst accesses to the shared partition.

The Datamover core from Xilinx [43] is a core that can be configured to perform burst memory accesses. It has three bi-directional stream ports: a port that controls the read transaction, a port that controls the write transaction and a port for transferring the read and write data. It also has a port with an AXI4 interface that supports memory bursts. When using this core to perform a read burst, the read operation is initialized through the read control port, and the

core performs the read burst and the data is received from the read data port. Similarly, when using this core to perform a write burst, the write operation is initialized through the write control port, and the core performs the write burst using the data received through the write data port. To communicate with the Datamover core, the host MicroBlaze uses the stream driver. This section compares the trade-offs of accessing the shared partition using the iomem driver to the Datamover core from Xilinx [43] accessed by the stream driver.

For the experiment, a host application was implemented using the read (*clEnqueueReadBuffer*) and write (*clEnqueueWriteBuffer*) functions from the OpenCL implementation. These functions are designed to copy data between the Linux partition and the shared partition. The read and write functions were executed with five different data sizes (64KB, 256KB, 1MB, 4MB and 16MB) and three different maximum burst lengths (16, 64 and 256). The runtimes using the stream driver with the Datamover core normalized to the runtimes using the iomem driver for the read and write operations are shown in Figure 2.7. In Figure 2.7, labels *Stream-16*, *Stream-64* and *Stream-256* refer to experiments with the stream driver with a burst length of 16, 64 and 256 respectively.

From Figure 2.7, for both the read and write operations, the runtime using the datamover core with the stream driver is larger than the runtime using the iomem driver. Across all experiments performing the read operation (Figure 2.7a), the runtime with the Datamover core and the stream driver averages 1.4 times the runtime with the iomem driver, and for the write operation (Figure 2.7b), the runtime with the Datamover core and the stream driver averages 3.0 times the runtime with the iomem driver. The additional runtime is a result of the overhead of the virtual buffers and threads in the stream driver. Although the virtual buffers and threads in the stream driver are needed for the host and device subsystem to communicate concurrently, such a requirement is not needed for transferring data between the shared partition and the Linux partition.

As a result, the stream driver was modified to bypass the virtual buffers and access the stream ports directly for the ports connected to the Datamover core. Figure 2.8 shows the runtime using the modified stream driver with the Datamover core normalized to the runtimes using the iomem driver for the read and write operations. In Figure 2.8, labels *Stream-Direct-16*, *Stream-Direct-64* and *Stream-Direct-256* refer to experiments with the modified stream driver

(a) Read operation

(b) Write operation

Figure 2.7: Runtime of the Datamover core with the stream driver normalized to the runtime with iomem driver

(a) Read operation

(b) Write operation

Figure 2.8: Runtime of the Datamover core with the modified stream driver normalized to the runtime with iomem driver

that accesses the stream ports directly with a burst length of 16, 64 and 256 respectively.

From Figure 2.8, it is observed that the modified stream driver with the Datamover core performs faster than the iomem driver. The average runtime for the read operation with modified stream driver and the Datamover core is 0.8 times the runtime with the iomem driver, shown in Figure 2.8a. Similarly, the average runtime for the write operation with modified stream driver and the Datamover core is 0.8 times the runtime with the iomem driver, shown in Figure 2.8b. In conclusion, when using the data from Figure 2.7 and 2.8, the virtual buffer and the threads in the stream driver represent 43% of the runtime for the read operation and 73% of the runtime for the write operation.

In addition, for all experiments in Figures 2.7 and 2.8, the burst length does not have an impact on the runtime. The reason being that the host processor is not sending data to the Datamover core quick enough to leverage the bandwidth of the memory controller.

2.12.2 CRC application

This Section explores the effects of architectural changes on a CRC application. A CRC application is selected, since it is commonly used in telecommunication in an embedded environment. For this experiment, the CRC implementation from MI Bench [44], a commercially representative embedded benchmark suite, is used. The core computation from the benchmark is extracted to create an OpenCL kernel.

The kernel was implemented in software (*crc-sw*) and in hardware (*crc-hw*). The software version of the kernel is executed on a device composed of a single MicroBlaze. The hardware version of the kernel was created using the Vivado High Level Synthesis (HLS) Tool [45]. By comparing these two implementations, the performance of the kernel using a general purpose processor and using custom hardware can be evaluated. The kernels were executed with five different input sizes: 64KB, 256KB, 1MB, 4MB and 16MB. The runtime of *crc-sw* normalized to *crc-hw* is shown in Figure 2.9.

For all input sizes, *crc-hw* performs faster than *crc-sw*. The performance benefit is a result of parallelism extracted from the kernel by Vivado HLS. In addition, the HLS Tool was able to collapse many instructions from the software implementation that execute in many cycles on a processor into a single cycle with the aid of the Vivado HLS scheduler.

Figure 2.9: Runtime of crc-sw normalized to crc-hw

The normalized runtime shown in Figure 2.9 starts to saturate with input sizes 4M and 16M. This saturation is due to the memory controller being unable to satisfy the number of memory requests by both kernel implementations, making the memory controller a bottleneck. In the end, with regards to the application’s performance, the results show that the hardware implementation of the CRC application performs faster than the software implementation

A benefit of using UT-OCL to compare two implementations of a kernel is that the majority of the infrastructure for evaluation is provided. In addition, UT-OCL provides a consistent environment where only the device changes to fairly evaluate these implementations. Furthermore, UT-OCL provides simple hooks for the user to easily integrate a kernel implementation into the framework.

2.12.3 Architectural changes applied to a device

With UT-OCL, it is possible to experiment with all aspects of the architecture using OpenCL programs as the driving inputs. This section describes the study where a number of topologies for interconnecting components within a device are evaluated.

The device under evaluation is shown in Figure 2.10. The device is composed of the same components described in Section 2.4, and has a similar architecture to the device in Figure 2.4. In contrast to the device in Figure 2.4, the device under evaluation has eight PEBlazes, and

Figure 2.10: Block diagram of the OpenCL device used in the experiments

an *Interconnect* interconnecting the PEBlazes, Local Mem, Mutex and connection to global memory. The Interconnect component is the element of study in this Section. It is implemented using Xilinx’s AXI Interconnect [46] and a special interconnect framework supporting a network-on-chip (NoC) paradigm as its transport layer.

Xilinx’s AXI Interconnect [46] can be configured to use a shared crossbar or a full crossbar. In the shared crossbar configuration, at most one master can communicate with one slave at any given time. The full-crossbar configuration allows for more master-slave communications to occur in parallel. Two AXI Interconnects are used, one AXI Interconnect configured to use a shared crossbar (AXI-S) and the other interconnect configured to use a full crossbar (AXI-F). AXI-S is used as the baseline interconnect for comparison.

The special interconnect framework supports interconnect implementations with a network-on-chip paradigm. This framework makes it easy to change the component that defines the actual interconnection network (transport layer). This is currently done statically, i.e., before the FPGA is synthesized. Figure 2.11 shows a block diagram of an interconnect implementation using this framework. Components connecting to the special interconnect use an AXI4-lite or AXI4 interface [47] at each port. The AMBA-AXI protocol [47] is converted to a routing protocol using the *Protocol Converter*, which interfaces to the *network-on-chip implementation* (enclosed in the box with a dotted perimeter in Figure 2.11). In contrast to the AMBA-AXI protocol, the special interconnect framework does not require a response from a slave for master component write requests. Therefore, master components continue their execution after a write

Figure 2.11: Block diagram of an interconnect implementation using the special interconnect framework

request.

The Interconnect of the device is connected to nine masters (eight PEBLazes and the DevManager), and three slaves (global memory, Local Mem and Mutex), resulting in a total of twelve components. The NoC implementations used in the Interconnect were generated using CONNECT [48]. The following topologies were used in these NoC implementations: a 12-node bidirectional ring (Bi-Ring), a 4x3 mesh, a 6x2 mesh, a 4x3 torus, a 6x2 torus and a 16-node butterfly. These are the minimal configurations needed for each topology to connect the twelve components.

The routers in these implementations use a Separable Input-First Round Robin allocation scheme, Simple Input Queued router type and a transmit (XON/XOFF) flow control [49]. The buffers connecting these routers are 16 entries deep. These configurations have been chosen because they use the least amount of FPGA resources [48] for the target FPGA and satisfy the network requirements. The data field of a flit is 33-bits wide, 32-bits for the address and data field and one bit for the operation type (read or write). To avoid back-pressure from the AXI interconnects (AXI-S and AXI-F), AXI-S and AXI-F are configured with read and write FIFOs on each master and slave ports to buffer memory transactions, making them comparable with the NoC implementations. For the remainder of this section, the interconnects using a NoC paradigm will be referred to as “NoC Interconnects”.

OpenCL Applications

Memory in embedded systems is scarce. As a result the private memory of the PEBlazes, which hold the kernel instructions, is not very large. Therefore, although there is an extensive list of open-source OpenCL benchmark suites, the kernels found in these suites are too large to fit in a PEBlaze's private memory. Moreover, the open-source benchmark suites for embedded systems were not written in OpenCL or the source is designed for sequential execution.

As a result, parallel implementations of common benchmarks were adapted into the framework. The remainder of this section will describe the benchmarks used to showcase the observations discussed in Section 2.12.3.

The OpenCL applications are:

1. The game of life (gol): calculates the evolution of an initial state.
2. The Jacobi application (jacobi): calculates the average of each element in a matrix using itself and the elements to the top, bottom, left and right. The average is not calculated for the elements on the perimeter of the matrix. The elements in the matrix are floating-point numbers.
3. The matrix multiplication application (mm-int): calculates the product of two matrices, where the elements of the matrices are integer numbers.
4. The palindrome application (palindrome): computes partial results aiding the host application to decide whether the input string has the properties of a palindrome.
5. The integrate application (integrate): computes partial results aiding the host application to compute the integral over a vector of integer numbers.
6. The matrix multiplication application (mm-float): performs the same calculations as mm-int, with the exception that the elements of the matrices are floating-point numbers.

For all applications, the calculation is divided evenly amongst the PEBlazes. For the given application, the matrix input size is 32x32, the input string has 8192 characters, and the vector size is 1024. For gol and jacobi, the application runs for 64 units of time, where a barrier is performed between each time unit.

Table 2.1: Application’s runtime for each topology relative to AXI-S

	AXI-S	AXI-F	Bi-Ring	Mesh 4x3	Mesh 6x2	Torus 4x3	Torus 6x2	Butterfly
gol	1	0.72	0.74	0.74	0.74	0.74	0.74	0.73
jacobi	1	0.88	0.90	0.89	0.89	0.88	0.89	0.89
mm-int	1	0.96	0.91	0.91	0.92	0.92	0.91	0.91
palindrome	1	0.88	0.93	0.92	0.92	0.90	0.90	0.92
integrate	1	0.99	0.99	0.99	0.99	0.99	0.99	0.99
mm-float	1	1.05	1.06	1.04	1.04	1.05	1.04	1.04

Table 2.2: Average barrier performance for each topology relative to AXI-S

	AXI-S	AXI-F	Bi-Ring	Mesh 4x3	Mesh 6x2	Torus 4x3	Torus 6x2	Butterfly
gol	1	0.53	0.36	0.43	0.40	0.34	0.39	0.39
jacobi	1	0.45	0.51	0.55	0.55	0.49	0.53	0.53

Observations about the Interconnects

During the execution of the applications, most of the communication occurs between the PEBlazes and the global memory. The behavior of this traffic pattern can be observed by comparing the performance of each application over the various interconnect implementations.

Table 2.1 shows the performance of the applications for the baseline interconnect (AXI-S) relative to each interconnect implementation. On average, the AXI-F and the NoC Interconnects perform better than the AXI-S. The improvement in performance results from the fact that when the PEBlazes communicate with multiple slave components, the AXI-F and the NoC interconnects allow for more master-slave communications to occur in parallel.

When comparing the relative performance amongst the NoC Interconnects, the performance is topology independent. The reason is the effect of the special interconnect framework not stalling the master components during a write request, allowing the PEBlazes to continue their execution after a write request is submitted and queueing multiple memory requests within the buffers interconnecting the routers. For these interconnects, the network is flooded with memory requests, which in turn, saturates the memory controller with requests, making the memory controller a bottleneck.

Another observation from Table 2.1 is that the AXI-F and the NoC Interconnects perform worse than the baseline interconnect for the mm-float application. The mm-float application has much more computation than the mm-int application, since the PEBlazes do not have dedicated hardware to perform floating-point operations. Instead, software is used to implement the

floating-point operations, which increases the runtime of the applications. Although mm-float has the same number of memory requests to global memory compared to mm-int, the intervals between memory requests are larger and suffer the latency of the communication between the PEBLaze and the global memory due to the multiple routers that are traversed. Similarly, AXI-F has more latency between the PEBLazes and the global memory compared to AXI-S because AXI-F supports the full AXI4 protocol [47] with more stages to satisfy additional functionality, although these functionalities are not needed in this case. The NoC Interconnects have more latency between the PEBLazes and the global memory since multiple routers are traversed.

During barrier executions, the communication is between the PEBLazes and the three slave components (global memory, Local Mem and Mutex). Intuitively, the interconnects that support multiple master-slave communication pairs occurring in parallel should perform better than AXI-S, which does not support multiple master-slave communication pairs occurring in parallel. Assessing an interconnect’s capability of handling multiple master-slave communication pairs occurring in parallel can be done by comparing its barrier performance with other interconnect implementations.

Table 2.2 shows the average barrier performance of gol and jacobi for each interconnect implementation relative to AXI-S. From this table, it is confirmed that the AXI-F and the NoC interconnects perform better than the AXI-S as expected.

In the end, interconnects that do not stall the master components after a write request is submitted and handle multiple master-slave communication pairs occurring in parallel provide better performance for the applications running on this device.

Resource Utilization

The interconnects only use Look-up Tables (LUTs) and Flip-Flops (FFs) in their implementations. Therefore, it is sufficient to compare the utilization of these resources. Table 2.3 shows the resource utilization of the various interconnect implementations, and their relative utilization compared to AXI-S.

From Table 2.1, the AXI-F and NoC Interconnects behave similarly with respect to performance and that for all applications but mm-float there is a performance improvement. However, Table 2.3 shows that there are different resource costs associated with each interconnect type.

Table 2.3: Resource utilization summary

	AXI-S	AXI-F	Bi-Ring	Mesh 4x3	Mesh 6x2	Torus 4x3	Torus 6x2	Butterfly
LUT	32427	34083	41530	45478	44638	52795	52192	41487
FF	28726	29270	31665	33861	33599	35722	35711	31275
LUT Utilization Ratio	1	1.05	1.28	1.40	1.38	1.63	1.61	1.28
FF Utilization Ratio	1	1.02	1.10	1.18	1.17	1.24	1.24	1.09

For example, AXI-F provides about 28% improvement in performance for 5% more LUTs and 2% more flip flops compared to AXI-S, whereas the Torus interconnects require over 60% more LUTs and 24% more flip flops. The additional resources utilized in the NoC Interconnects compared to the AXI interconnects (AXI-S and AXI-F) implement the routing logic in the routers and the buffers interconnecting the routers. Therefore, the choice of interconnect should be made based on the resources used since the performance differences are minimal. In conclusion, the AXI-F is the best interconnect implementation for the applications executing on the device.

2.13 Conclusion

Recently, there has been significant effort by FPGA vendors to program FPGAs using OpenCL. However, targeting FPGAs with OpenCL presents new and unique challenges, since FPGAs differ from common platforms (e.g. CPUs, GPUs) targeted by OpenCL implementations. In contrast to these other platforms, FPGAs suffer from long compile times. In addition, its architecture allows for different configurations of processing elements and for the potential of partial reconfiguration.

FPGA vendors have also recently developed FPGA platforms [5] [50] with an integrated SoC to easily build embedded systems using FPGAs. However, support for integrating and managing custom accelerators, devices in the OpenCL model, is significantly lacking. This chapter presented UT-OCL an open-source OpenCL framework for embedded systems on FPGAs. UT-OCL is composed of a hardware system and its software counterparts that can execute OpenCL applications compliant with the 2.0 specifications. The framework also contains debugging tools to aid the user when developing with the framework. The chapter also highlights

steps on how to prepare an OpenCL application and integrate a device (hardware accelerator) into the hardware system.

By having an OpenCL framework for embedded system, an application programmer can develop using an OpenCL framework in a workstation environment, where the tools for development and prototyping are more readily available. Once the programmer is satisfied that the application is functionally correct on the workstation, the application can be migrated to the embedded platform with more confidence that the application will work. Thus, making the integration into UT-OCL essentially a porting exercise.

By making the framework open-source, continuing effort on adapting and improving OpenCL for FPGAs can be performed, including testing possible modifications to the standard. This is very important as the standard continues to evolve, which is the primary motivation for developing this framework.

To demonstrate the practicality of the framework, architectural changes applied to the hardware system and to a CRC application have been evaluated. In the hardware system, the overhead of virtual buffers and threads in the stream driver were quantified, a burst mechanism that increases the performance of data transfer between the Linux partition and the shared partition was developed. For a CRC application, it was shown that a commercial HLS tool can be applied to a kernel to create custom hardware and easily be integrated into the UT-OCL framework. By using UT-OCL as an evaluation environment, future CRC implementations can be compared fairly the implementations presented in Section 2.12.2.

In addition, UT-OCL was used to perform an initial investigation of interconnect implementations using a network-on-chip paradigm versus the conventional crossbar implementation found in most FPGA vendor interconnect solutions for a custom device. For the custom device presented in Section 2.12.3, data that could be used to evaluate the trade-offs between the application's performance and its cost in term of resource utilization was provided. Furthermore, although the PEBLazes were able to proceed with their execution after write requests, the global memory became a bottleneck, highlighting an opportunity for further architectural exploration.

In the end, the release of UT-OCL into the research community permits the exploration of a broad range of research topics, as well as fuel other prospective research topics in the context of OpenCL in an embedded environment using FPGAs.

Chapter 3

Shared Virtual Memory (SVM) in the OpenCL Standard

As the OpenCL standard continues to evolve, new features are added. For example, in OpenCL 2.0 [7], a new feature known as *Shared Virtual Memory (SVM)* has been introduced. With the introduction of SVM into the standard, OpenCL developers can write code with extensive use of pointer-linked data structures, like linked-lists or trees, that are shared between the host and the device side of an OpenCL application. In the previous version of OpenCL, version 1.2, there is no guarantee that the pointer assigned on the host side can be used to access data by the kernels on the device side and vice-versa. Thus, the pointers cannot be shared between the two sides. This is an artifact of a separate address space for the host and device side that is addressed by OpenCL 2.0 SVM.

In the OpenCL memory model, the device may require access to other memory types that are typically not accessed using virtual addresses. This model is similar to a hardware accelerator in an embedded system accessing memory-mapped peripherals using their physical addresses. Accessing these peripherals using virtual addresses, where an address translation mechanism is required, will cause the system to suffer from unneeded overhead [51]. Hence a challenge for implementing OpenCL 2.0 SVM in embedded systems is to enable the devices to address both virtual memory and physical memory.

FPGAs designed for embedded systems are equipped with processors and programmable

logic on the same die. Within these systems, when the processor is running an Operating System with virtual memory support, the processor references off-chip memory using a virtual address. However, the peripherals in the programmable logic access off-chip memory using a physical address. Without a unified mechanism for translating addresses, the processor needs to marshal the data, which is subject to runtime overhead, so the peripherals can address the data correctly. By extending the mechanism for address translation to the peripherals, the peripheral can access data without the overhead of marshalling or moving the data, which would also facilitate the implementation of the OpenCL 2.0 SVM within these systems.

In this Chapter, different approaches for implementing shared virtual memory in UT-OCL, an OpenCL framework for embedded systems on FPGAs, are introduced. The approaches satisfy the challenge of accessing both virtual and physical memory. The objective is to find the approach that is best suited for a given application and system. Furthermore, this best approach was compared with an approach that implements data sharing between the host and the devices using an OpenCL implementation conforming to OpenCL 1.2.

This Chapter continues with a brief overview of the OpenCL memory model in Section 3.1 with some background on SVM given in Section 3.2. Section 3.3 describes the hardware system and the mechanisms for translating virtual addresses to physical addresses in the UT-OCL framework. Section 3.4 presents the modifications applied to the framework to enable shared virtual memory. In Section 3.5, the approaches are contrasted with related work. Section 3.6 evaluates the performance of the proposed approaches, and continues with a comparison of using an OpenCL implementation with SVM support and without SVM support. The Chapter ends with concluding remarks in Section 3.7.

3.1 Details of the OpenCL Memory Model

Figure 3.1 shows the details of the UT-OCL framework, where the platform model, the execution model and the programming model are described in Section 2.1. When using SVM, programming in OpenCL can be substantially simplified. Details on this simplification are described in Section 3.2.

There are two memory types in the memory model affected by the introduction of SVM

Figure 3.1: Details of the OpenCL framework

Figure 3.2: Address space of the host and device in OpenCL 1.2

Figure 3.3: Address space of the host and device in OpenCL 2.0

in OpenCL. The first is the *Host Memory*, which is memory that is managed and accessed by the host using the host address space. The second is *Global Memory*, which is memory that is also managed by the host. It is accessed by the host using the host address space or by the device using the device global address space. Both these memory types can store OpenCL memory objects such as buffers or images. These memory objects represent regions in the global memory.

Figure 3.2 shows an example of the addresses for memory objects in the device global address space and the host address space when using OpenCL 1.2 (without SVM support). The addresses for the memory objects in the different address spaces can differ as illustrated in Figure 3.2. For example, a memory object in the device address space can occupy address range 0x5000 to 0x5100, and for the host to access the same memory region, the mapped memory object in the host address space can occupy address range 0xC880 to 0xC980. The host can map and unmap the address of a memory object into its address space using the map and unmap commands. The map command is a request to give ownership of the memory object to the host, so the host can modify the memory object. The unmap command releases the ownership. Figure 3.2 also illustrates other host data (e.g. heap, stack) in the host address space as well as an unmapped memory object.

With the introduction of SVM, the device and host share the address space. As a result,

the host and device can access data using the same address. The effects of SVM are illustrated in Figure 3.3. In contrast to Figure 3.2, an SVM memory object has the same address range in both the host address space and device global address space. For example, in Figure 3.3, the SVM memory object has address range 0x3000 to 0x3400 in the device address space and the host address space. When a memory object is created using SVM, it is referred to as a SVM memory object. In addition, for compatibility with OpenCL 1.2, OpenCL 2.0 also allows for mapped memory objects as shown in Figure 3.3.

3.2 Shared Virtual Memory (SVM)

Shared Virtual Memory (SVM) enables the host and device portions of the OpenCL applications to seamlessly share pointers. It is realized by extending the address space of the global memory region into the host memory region and using a single address space for both memory regions.

In addition to a shared address space, SVM provides features that simplify programming in OpenCL. SVM enables SVM memory objects without the need to explicitly create memory objects using the OpenCL API. As a result, the developer does not need to create memory objects in the OpenCL application. SVM also provides *map-free access*, where the host does not need to use the map/unmap command to access the memory object. These features enable legacy C/C++ programs to be easily integrated into OpenCL and managed by the OpenCL memory resources on the host.

There are two characteristics of SVM support that aid in defining the different types of SVM. The first characteristic relates to memory allocation. If memory is allocated explicitly using OpenCL API functions, then it is referred to as *buffer allocation*, where the term *buffer* refers to the basic memory object type in OpenCL. If the memory is allocated using Operating System functions (e.g. *malloc* or *new*), then it is referred to as *system allocation*. The second characteristic relates to the sharing granularity. If the sharing granularity is the region of memory representing a memory object, then it is referred to as *Coarse-Grained*. If the sharing granularity is individual memory locations, where memory locations are analogous to bytes of memory objects, then it is referred to as *Fine-Grained*. Using these characteristics, the three types of SVM are defined.

Figure 3.4: Address space of the host and device using the Fine-Grained system SVM type

The first SVM type is Coarse-Grained buffer SVM. Within this SVM type, sharing occurs at the granularity of regions of OpenCL SVM memory objects. The host can update SVM memory objects using map and unmap commands, much like OpenCL 1.2, but the address range of SVM memory objects is the same in the host address space and device address space as illustrated in Figure 3.3. To ensure memory consistency between the host and the device, the host uses synchronization commands.

The second SVM type is Fine-Grained buffer SVM. This SVM type exploits the map-free access feature by allowing the host and the device to concurrently make modifications to adjacent bytes of memory objects. For this SVM type, no specific side has ownership of the memory object, and both the host and the device can concurrently access the memory object. The memory objects are created by buffer allocation.

The third SVM type is Fine-Grained system SVM. Within this SVM type, sharing occurs at individual memory locations anywhere within the host memory. Essentially sharing the entire host address space provided by an operating system without creating an SVM buffer for it. Figure 3.4 illustrates the host address space and the device address space for this SVM type. In contrast to the other SVM types, the entire host address space is accessible through the global memory address space, including other host data (e.g. stack, heap) and the OS environment.

For fine-grained sharing, where the host and device access the memory locations concurrently, the SVM implementation can provide optional atomic operations enabling light-weight

synchronization. In contrast to coarse-grained sharing, the atomic operations provide tighter synchronization as they are executed by the host or the device without enqueueing device commands. Therefore, in addition to sharing the address space, OpenCL 2.0 SVM also defines the memory consistency for SVM allocation.

Due to the features enabled by the different SVM types, the relationship between the SVM types is hierarchical and can be viewed as: Coarse-Grained buffer SVM is a subset of Fine-Grained buffer SVM which is a subset of Fine-Grained system SVM. For an OpenCL implementation to be compliant with the SVM feature, it requires at a minimum to support the Coarse-Grained buffer SVM type. The SVM type supported by a device is achieved by probing for the device information. The work presented in this Chapter satisfies the requirements for the Fine-Grained system SVM type, thus also capable of satisfying the other SVM types. Although, the current implementation does not support atomic operations, future work consists of implementing this feature.

In the end, SVM eliminates the need to marshal or move data between the host and devices by sharing the address space of the host and the devices, in turn simplifying programming in OpenCL. SVM also introduces new features that may require dedicated support from the hardware, operating system, or device driver. The Chapter continues with a description on the changes applied to the UT-OCL framework to support the Fine-Grained system SVM type.

3.3 Memory Management in UT-OCL

UT-OCL contains a hardware system designed for the ML605 development board, which has a single off-chip memory attached to the FPGA. A diagram of the framework including the architecture of the hardware system is illustrated in Figure 3.5. The host is implemented using a MicroBlaze processor that runs Linux, and the devices are typically custom hardware accelerators that do not have OS support.

The host memory and global memory reside in the off-chip memory. From the perspective of the host, both memory regions are accessed using their virtual address defined in the Linux kernel, and, from the perspective of the device, the global memory region is accessed using its physical address.

Figure 3.5: Diagram of the UT-OCL framework

Figure 3.6: Virtual memory addressing scheme in the Linux kernel

In Linux, the kernel is responsible for managing the address space in the system. Memory allocated by the application maps to the application's space, which in turn maps to the physical address. Figure 3.6 shows an example of this mapping. In the example of Figure 3.6, the application's virtual address range 0x1800 to 0x18FF maps to physical address range 0x8300 to 0x83FF.

The virtual memory, analogous to the host memory, is partitioned into pages. When allocating memory using system functions (e.g. *malloc* or *new*), the virtual address space is contiguous, however the physical address space may not be contiguous. The translation between the virtual

memory space to the physical memory space is performed with the help of two mechanisms in the UT-OCL framework.

The first mechanism is a software algorithm in the Linux kernel. The algorithm uses the virtual address, the application's page table and the page translation table to compute the physical memory address. The page translation table structure implemented for the MicroBlaze processor has two levels as shown in Figure 3.6.

The second mechanism is the Memory Management Unit (MMU) hardware component in the MicroBlaze processor. The MMU enables translation from the virtual address to the physical address. It also provides control by protecting pages from unauthorized access. The MMU contains a two-level Translation Look-Aside Buffer (TLB) for the instruction-addresses and data-addresses that can be accessed through a unified interface. The TLB is essentially a cache for address translation and can store 64 translations.

The architecture of the hardware system as well as the mechanisms to translate virtual addresses to physical addresses within UT-OCL are modified to implement the Fine-Grained system SVM type.

3.4 Implementing the Fine-Grained System

SVM Type in UT-OCL

The objective of implementing the Fine-Grained system SVM type in UT-OCL is for the device to access the host memory using virtual addresses. As mentioned in Section 3.3, the mechanisms for translating the virtual address to the physical address are contained within the scope of the host, and not within the scope of the device.

A naive approach is to have the host translate the address prior to sending it to the device. Such an approach is incomplete since allocated memory will have a contiguous virtual address space that may not be mapped to a contiguous physical address space. A more feasible approach is to enable the mechanisms for virtual to physical address translation to be accessible by the device. To enable virtual to physical address translation by the device, an existing component was modified and three hardware components were implemented.

3.4.1 The Components

The MicroBlaze processor was modified to have its MMU accessible by the devices, essentially making it an Input/Output Memory Management Unit (IOMMU). To complete this task, the TLB was modified to be multi-ported and MMU logic was duplicated for access by the device. The original TLB implementation used a dual-ported BRAM, where a single port writes to the BRAM at any given cycle. The modified TLB is implemented as a 1-Write/3-Read multi-ported memory. In addition to transforming the MicroBlaze's MMU into an IOMMU, the logic for the MMU was isolated to create a standalone MMU for use in the system.

Three components were implemented: the Address Checker, the IOMMU Engine and the PTE Engine. The Address Checker checks the address of the memory request from the device and is used to enable the device to address both virtual memory and physical memory. All the SVM approaches introduced in this section use this component. If the Address Checker finds that the address is in the range of the global memory, it forwards the memory request to the global memory region. Otherwise, it will send the address to the *IOMMU Engine* or *PTE Engine* for translation.

The second component is the *IOMMU Engine*. This engine is able to send address translation requests to the IOMMU of the MicroBlaze processor and the standalone MMU component. It can also raise interrupts and interface to the *PTE Engine*. Its functionality depends on the approach used.

The *PTE Engine* is the third component and is responsible for computing the physical address using the page translation table. It is the hardware implementation of the software algorithm for translating the virtual address to the physical address described in Section 3.3. It is connected to the off-chip memory, so it can access the page translation table located in the host memory. The host is required to setup the PTE Engine by sending the base address of the OpenCL application's page table.

A version of the PTE Engine was created using Vivado High Level Synthesis (HLS) [45]. The solution required a minimum of 36 cycles for computing the physical address. A hand-written HDL version was created requiring a minimum of 15 cycles. The hand-written HDL version is used in the experiments. The Address Checker and IOMMU Engine have also been implemented

Figure 3.7: Modifications applied to the hardware system for the proposed approaches

using hand-written HDL. The remainder of this Section will present the six different approaches for implementing the Fine-Grained system SVM type in UT-OCL.

3.4.2 The Approaches

The first approach uses interrupts to notify the host to perform an address translation. In this approach, when the *IOMMU Engine* receives the virtual address from the Address Checker, it will interrupt the host. Then, the host reads the virtual address from the IOMMU Engine, computes the physical address using the software algorithm in the Linux kernel, and services the interrupt by writing the physical address associated with the virtual address to the IOMMU Engine. Figure 3.7 illustrates the changes to UT-OCL's hardware system for implementing this approach. The changes are labelled with a 1. Compared to the base hardware system, this approach requires an additional interrupt line, the IOMMU Engine and Address Checker components. Hereafter, this approach will be referred to as *Intr*.

The second approach uses the IOMMU. Similar to *Intr*, the *IOMMU Engine* receives the virtual address from the Address Checker. When the *IOMMU Engine* receives the virtual address from the Address Checker, it will query the IOMMU for an address translation. If the

translation cannot be satisfied, the IOMMU Engine will interrupt the host. Figure 3.7 illustrates the changes to UT-OCL's hardware system for implementing this approach, the changes are labelled with a 2. Compared to the hardware system implementing *Intr*, the MicroBlaze is replaced with the MicroBlaze version with an IOMMU, and the IOMMU Engine is connected to the IOMMU. Hereafter, this approach will be referred to as *Intr+IOMMU*.

The third approach extends *Intr+IOMMU* by reserving a section of the TLB for use by the device. For this approach, the Linux kernel was modified to use 32 entries of the TLB, and the remaining 32 entries are reserved for the device. A TLB manager was implemented in software for the TLB entries reserved for the device. The manager uses a round robin replacement policy and is invoked when an interrupt occurs by the IOMMU Engine. The hardware system is identical to that of *Intr+IOMMU*, and the changes are labelled with a 3 in Figure 3.7. This approach is referred to as *Intr+TLB_MGMT*.

The fourth approach is referred to as *PTE+IOMMU* and does not use interrupts to notify the host to perform an address translation. Instead, when the IOMMU is unable to satisfy an address translation, the IOMMU Engine will send the virtual address to the PTE Engine to compute the physical address. Figure 3.7 illustrates the changes to UT-OCL's hardware system for implementing this approach, the changes are labelled with a 4. Compared to the hardware system implementing *Intr*, the MicroBlaze is replaced with the MicroBlaze version with an IOMMU, and the IOMMU Engine is connected to the IOMMU and the PTE Engine.

The fifth approach solely uses the PTE Engine to perform address translation. For this approach, the Address Checker will send the virtual address to the PTE Engine. Then, the PTE Engine will compute the physical address and return this value to the Address Checker. Hereafter, this approach will be referred to as *PTE*. Changes to the hardware system are labelled with a 5 in Figure 3.7.

The sixth approach extends the fifth approach by providing a standalone MMU to store address translations. In this approach, the Address Checker sends the virtual address to the IOMMU Engine. The IOMMU Engine queries the MMU component for an address translation. If the translation cannot be satisfied, then the IOMMU Engine will send the virtual address to the PTE Engine. The PTE Engine will compute the address translation and send the physical address to the IOMMU Engine. The IOMMU Engine will update the MMU component using

a round robin replacement policy prior to sending the physical address to the Address Checker. This approach will be referred to as *PTE+MMU*, and the changes to the hardware system are labelled with a *6* in Figure 3.7.

In addition to creating hardware components for the realization of SVM in UT-OCL, the OpenCL implementation was updated to support SVM. Moreover, the caches on the host are disabled when SVM is used in an OpenCL application, since there is no cache coherency mechanism between the host and device. Future work consists of creating a cache-coherency mechanism between the host and device in hopes of increasing performance. However, emerging SoCs platforms [52] incorporate a cache-coherency mechanism between the SoC (host) and FPGA fabric (devices) that can be leveraged for future implementations of the UT-OCL framework.

3.5 Related Work

Advanced Micro Devices (AMD) and Intel are vendors that provide OpenCL SVM support for desktop environments that require hardware support from an IOMMU. AMD's IOMMU [53] uses AMD's Graphical Aperture Remapping Table (GART) technology. GART is a translation table in hardware. It only translates memory addresses within a specified window. The window selection functionality is similar to the Address Checker in the approaches.

Intel provides a generalized IOMMU architecture under the Virtualization Technology for Directed I/O (VT-d) specification [54]. It translates the device's memory address to the host's memory address using a page translation table as does the PTE Engine in the approaches (PTE+IOMMU, PTE and PTE+MMU). It also uses interrupts to update address translations similar to Intr, Intr+IOMMU and Intr+TLB_MGMT. The goal of this work was to extend the global address space into the host address space. With VT-d the reciprocal is also possible, where the host address space is extended into the global address space, permitting the host to access device memory where the device is running an operating system with virtual memory management.

There are works that evaluate virtual memory accesses using IOMMUs in environments using servers or desktop machines [55] [51]. There also exists work where tasks on the FPGA use an MMU to access virtual memory in a desktop machine [56]. Compared to these works,

the scope of this work is focused on virtual memory access in embedded systems on FPGAs.

In the context of embedded systems, there is work that uses an IOMMU to solely manage protection for application tasks [57] [58]. And, there is work that uses a software virtual memory manager in an MMU-less system [59]. However, the focus is in sharing the virtual address space in multiple domains of an embedded system running an Operating System. The remainder of this Section will present the work within this context.

Lange et al. [60] have altered the memory layout in the Linux kernel to relate the virtual to the physical address by an offset. There are limitations to this method that they later address [61]. In this later work, the authors create an MMU that is accessible by the processor and the device. The MMU is updated by the processor using an interrupt and a page translation mechanism in software. The devices use a hardware version of the page translation table. This MMU implementation is similar to PTE+IOMMU. Although these works are studying a shared-memory system, ours uses a shared-memory system augmented to run OpenCL.

Meenderinck et al. [62] use the MicroBlaze processor to implement a composable virtual memory scheme. In this scheme, the second level of the TLB is modified to represent page validations for an application. When an application is swapped, the first-level of the TLB is flushed, and the second-level of the TLB is replaced with the TLB of the application being swapped-in. The system's memory layout is also altered. Meenderinck's work focuses on restoring the application's page layout in memory to increase its performance. In this work, the page layout does not use a composable scheme, but uses the page layout mechanism embedded in the Linux kernel.

To date, there is no work that delivers an IOMMU with a page translation table mechanism in hardware that performs address translation on virtual addresses from an operating system in the context of OpenCL for embedded systems using FPGAs. This capability is explored in Section 3.6.

3.6 Results

This section evaluates the trade-offs between the proposed approaches and between a version of the UT-OCL framework with Fine-Grained System SVM (*UT-OCL+SVM*) and without SVM

support (*UT-OCL*). To perform the experiments, the device in Section 2.4 was modified to support 16 MicroBlazes, which is the maximum number of allowed slave ports for the custom router. A device composed of MicroBlazes was chosen, since the OpenCL kernel compiler for a MicroBlaze-based device is provided in the UT-OCL framework. These MicroBlazes run concurrently and not in lockstep.

In the following experiments, the page size is 4 KB. The runtime is measured using the profiling interface in UT-OCL, and calculated using the average of ten runs to account for variations in operating system overhead (e.g. context switching and scheduling).

3.6.1 Evaluating the proposed approaches

To evaluate the proposed approaches, three patterns stressing the MMU and IOMMU were used. These patterns were selected since their behavior is found in real-world applications. The first pattern accesses elements of the heap sequentially. This pattern is referred to as *linear*. The second pattern accesses 1024 elements within a page for all pages in the heap. The page is selected randomly and accesses within the page are done sequentially. This pattern is referred to as *page*. The third pattern randomly accesses elements of the heap. The location of the element in the heap is computed in software using a pseudo random number generator. This pattern is referred to as *random*. An element of the heap is four bytes in width.

A synthetic benchmark for each pattern is created. Each benchmark accesses 32768, 65536 and 131072 elements, where the respective heap size is equivalent to the number of element accesses. These heap sizes were selected to observe the behavior of the proposed approaches in two polar scenarios: 1) when there are sufficient TLB slots to cache all address translations of the pages composing the heap, and 2) when there are insufficient TLB slots to cache all address translations of the pages composing the heap. The benchmarks are performed with 1, 2, 4, 8 and 16 threads. For each execution, the number of hits, the number of IOMMU or standalone MMU requests (depending on the approach), the number of host TLB misses during the execution of the benchmark (kernel) and the runtime of the benchmark (kernel) are collected.

Figure 3.8: Runtime of the proposed approaches for 16 threads accessing 131072 elements for each pattern

Runtime of the proposed approaches

Each approach has a different mechanism for computing the address translation or has different constraints imposed on its mechanism. By comparing the runtime of each approach, the performance of these mechanisms can be evaluated. A higher runtime signifies that the computation for address translation requires more clock cycles to complete. Figure 3.8 shows the runtime of the approaches for 16 threads accessing 131072 elements for each pattern. The execution of 16 threads represents a realistic scenario of an OpenCL kernel. In Figure 3.8, the runtimes of Intr and Intr+IOMMU are clipped at the maximum value of the y-axis so that the other values remain visible. By setting the range of the y-axis from 0 to 1×10^{10} , the Intr and Intr+IOMMU runtimes relative to the remaining approaches, as well as the runtimes within the remaining approaches can be clearly observed.

Within each pattern, Intr and Intr+IOMMU have the longest runtime. These long runtimes are a result of useful TLB entries being overwritten. During the execution of the kernel, the work performed by the host, including the interrupt service routine, accesses the virtual memory addresses that populate the TLB, causing the host to replace the entries in the TLB. Hence,

the reason for long runtimes with Intr+IOMMU and Intr. The performance of Intr+IOMMU is also affected by collisions of multiple accesses to the same location of the TLB memory. When a collision occurs, the request to the IOMMU re-executes.

When comparing Intr+TLB_MGMT with Intr and Intr+IOMMU, the runtime is significantly reduced. Intr's and Intr+IOMMU's runtime are approximately 39 times, 44 times and 36 times the runtime of Intr+TLB_MGMT for the linear, page and random patterns respectively. This significant reduction in runtime is due to TLB slots reserved for the device in Intr+TLB_MGMT not being evicted by the host application. As a result, the device can perform address translations by accessing the IOMMU, which has fewer cycles of latency than with the interrupt service routine.

PTE is the hardware implementation of the software algorithm for address translation in Intr. Compared to PTE, Intr is approximately 18 times, 22 times and 22 times the runtime of PTE for the linear, page and random patterns respectively. These results show that the hardware implementation of the software algorithm for translating a virtual address to a physical address is more efficient than the pure software algorithm using interrupts.

PTE+IOMMU has a longer runtime than PTE. The IOMMU in PTE+IOMMU does not provide any benefit in PTE+IOMMU for the same reasons that the IOMMU does not provide any benefit in Intr+IOMMU. The useful TLB entries are overwritten by the host. TLB misses in the IOMMU cost 67 cycles, two cycles for initializing the search, 64 cycles to traverse the TLB entries and 1 cycle for the response. This overhead is added to each memory request in PTE+IOMMU, hence the longer runtime with PTE+IOMMU than PTE.

To evaluate the overhead of using the IOMMU, the PTE and PTE+IOMMU are used. Although a comparison between Intr and Intr+IOMMU should be sufficient, a system with interrupts enabled increases the variance of the runtime. Hence, PTE and PTE+IOMMU are used for a more accurate result. When comparing the runtimes of PTE and PTE+IOMMU, the overhead of a TLB miss in the IOMMU is roughly 41% for all patterns.

PTE's runtime is approximately 2.2, 2.0 and 1.9 times that of PTE+MMU for the linear, page and random patterns respectively. The presence of the standalone MMU increases the performance of the address translation, since the PTE Engine does not need to read from off-chip memory to calculate the physical address, but can retrieve the physical address from the

Table 3.1: Average runtime (cycles) per element access for the two scenarios.

Approach	pattern					
	linear		page		random	
	# of elements		# of elements		# of elements	
	1x	2x	1x	2x	1x	2x
Intr+TLB_MGMT (cycles/element access)	999	1030	1014	1068	1257	1277
PTE+MMU (cycles/element access)	962	960	1004	1023	1100	1103

MMU.

In the end, PTE+MMU has the shortest runtime for all patterns. However, Intr+TLB_MGMT's runtime is fairly close to PTE+MMU's runtime. These approaches differ in their TLB configuration, which impacts their performance results. Intr+TLB_MGMT has 32 TLB slots reserved for the device and PTE+MMU has 64 TLB slots reserved for the device. For a fairer comparison between these approaches, the average runtime per element access is calculated for two scenarios. The first is when the number of pages that store the elements in the heap is equivalent to the number of TLB slots reserved for the device (1x). The second is when the number of pages that store the elements in the heap is equivalent to twice the number of TLB slots reserved for the device (2x). For Intr+TLB_MGMT, these scenarios use 32768 and 65536 elements respectively. And for PTE+MMU, these scenarios use 65536 and 131072 elements respectively.

Table 3.1 shows the average runtime per element access for the two scenarios. For all patterns in both scenarios, the PTE+MMU has a lower average runtime per element access compared to Intr+TLB_MGMT. In conclusion, PTE+MMU has the shortest runtime for all patterns, thus is the best approach in terms of performance.

The Hit Rate of the Approaches

The hit rate provides insight on the effectiveness of the approach. A higher hit rate signifies that address translations are performed by a simple lookup in the IOMMU or standalone MMU, avoiding the overhead of computing the address translation. To calculate the hit rate for each execution, depending on the approach, the number of hits is divided by the number of IOMMU or standalone MMU requests. Table 3.2 shows the hit rate of the proposed approaches for: 16

Table 3.2: Hit rate of the proposed approaches for 16 threads accessing 32768 and 131072 elements

Approach	linear		page		random	
	32768	131072	32768	131072	32768	131072
Intr+IOMMU	0.0	0.0	0.0	0.0	0.0	0.0
Intr+TLB_MGMT	0.99	0.99	0.99	0.99	0.99	0.28
PTE+IOMMU	0.0	0.0	0.0	0.0	0.0	0.0
PTE+MMU	0.99	0.99	0.99	0.99	0.99	0.94

threads accessing 32768 and 131072 elements. The execution of 16 threads represents a realistic scenario of an OpenCL kernel. With the patterns accessing 32768 and 131072 elements, the relation between the size of the heap and the number of slots in the TLB can be assessed. Intr and PTE are not shown in Table 3.2 because these approaches do not have an IOMMU or a standalone MMU component in the system. The results show that Intr+IOMMU and PTE+IOMMU have no hits. Intr+IOMMU and PTE+IOMMU have no hits because useful TLB entries in the IOMMU were overwritten by the host application.

For the linear and page patterns, Intr+TLB_MGMT and PTE+MMU have a hit rate of 0.99 because one TLB miss occurs for every 1024 elements assessed. The page size is 4 KB, which can hold 1024 elements. For these patterns, when a TLB miss occurs in Intr+TLB_MGMT and PTE+MMU, an entry for the page associated with the address of the TLB is inserted into the IOMMU or standalone MMU, and all future accesses to this page occur immediately after the insertion of this entry, resulting in hits.

For the Intr+TLB_MGMT, the hit rate of the random patterns has a drop from 0.99 to 0.28 as the number of elements rise from 32768 to 131072. In Intr+TLB_MGMT, the number of TLB slots reserved in the IOMMU for the device can hold 32 address translations of 4KB pages, resulting in a total of 32768 elements. As a result, in the experiment with 32768 elements, all address translations of the pages from the heap are cached in the IOMMU. The effect observed in this scenario is identical to that of the linear and page patterns, where a TLB miss will insert an entry in the IOMMU holding the address translation for a given page such that all future accesses to this page result in a hit.

In the experiment with 131072 elements, there are four times more pages associated with the heap than in the 32768 elements experiment. As a result, the IOMMU is unable to cache

all the address translations for the pages comprising the heap, and will suffer from thrashing, yielding the low hit rate of 0.28. But, as mentioned, Intr+TLB_MGMT has 32 TLB slots reserved for the device and the other approaches have 64 TLB slots. For a fairer comparison, the hit rate for Intr+TLB_MGMT is measured with half the number of elements compared to other approaches because it only has half the TLB entries. When $131072/2 = 65536$ elements are accessed with Intr+TLB_MGMT, a hit rate of 0.36 is measured. This is still less than the hit rate of PTE+MMU, which is 0.94. PTE+MMU's hit rate is still 2.6-fold better. In the random pattern where the element accesses are random, by having more TLB slots to cache more address translations, the likelihood of accessing an element where its address translation is cached increases. Such a behavior is observed with PTE+MMU that has 64 TLB slots and Intr+TLB_MGMT that has 32 TLB slots.

For the PTE+MMU, the hit rate of the random patterns has a drop from 0.99 to 0.94 as the number of elements rise from 32768 to 131072. In PTE+MMU, the MMU has 64 TLB slots so all the address translations for 32768 elements can be cached in the TLB. Therefore, a single TLB miss occurs for every 1024 element accesses, resulting in the high hit rate of 0.99. In the experiment with 131072 elements, there are insufficient TLB slots in the MMU of PTE+MMU to cache all the page entries composing the heap, causing the MMU to suffer from some thrashing, thus reducing the hit rate from 0.99 to 0.94.

The hit rate comparison between Intr+TLB_MGMT and PTE+MMU for experiments with 131072 elements provides some insight on the impact when the number of TLB slots are doubled. By doubling the number of TLB slots, the hit rate increases by 0.66, from 0.28 to 0.94.

In the end, Intr, Intr+IOMMU, PTE and PTE+IOMMU have no hits. Also, Intr+TLB_MGMT shows a comparable hit rate for the linear and page patterns for 32768 and 131072 elements compared to PTE+MMU. For the random pattern, the Intr+TLB_MGMT and PTE+MMU have a similar hit rate with 32768 elements, but, with 131072 elements, PTE+MMU has a significantly higher hit rate than Intr+TLB_MGMT due to twice the number of TLB slots. Thus, PTE+MMU is the most effective approach in a realistic scenario of an OpenCL kernel.

Table 3.3: Host-TLB-miss-runtime product of Intr+IOMMU, Intr+TLB_MGMT and PTE+IOMMU for: 16 threads accessing 32768 and 131072 elements

Approach	linear		page		random	
	32768	131072	32768	131072	32768	131072
Intr+IOMMU	4.18×10^{13}	1.34×10^{14}	3.37×10^{13}	2.11×10^{14}	3.23×10^{13}	6.24×10^{15}
Intr+TLB_MGMT	2.63×10^{13}	1.71×10^{13}	2.52×10^{13}	1.72×10^{13}	3.27×10^{13}	2.20×10^{13}
PTE+IOMMU	2.45×10^{13}	1.11×10^{13}	2.35×10^{13}	1.14×10^{13}	2.48×10^{13}	1.18×10^{13}

Effect On the Host Application

When using the IOMMU, the host application suffers from side-effects as the device and host share the IOMMU. For Intr+IOMMU and PTE+IOMMU, the TLB slots are shared between the host and device, and for Intr+TLB_MGMT the number of TLB slots are reduced in comparison to the other approaches using the IOMMU. This section will evaluate the effect of the host application for Intr+IOMMU, Intr+TLB_MGMT and PTE+IOMMU.

To evaluate the effect of the host application, the host-TLB-miss-runtime product is used. The host-TLB-miss-runtime product is calculated by multiplying the host TLB misses with the runtime of the kernel. This metric captures two factors to help define the effect on the host application. The first is the number of host TLB misses when the host shares the TLB slots or has a reduced number of TLB slots for the host application. The second is the runtime of the kernel. A good result for both these factors is a value closest to 0, so a smaller value for the product of the two numbers indicates a better performance. Therefore by using this host-TLB-miss-runtime product metric, a fairer comparison can be done to evaluate the effect on the host.

Table 3.3 shows the host-TLB-miss-runtime product of Intr+IOMMU, Intr+TLB_MGMT and PTE+IOMMU for: 16 threads each accessing 32768 and 131072 elements. A higher product signifies that the host application has higher TLB misses and/or kernel execution runtime. Results show that Intr+IOMMU has the highest host-TLB-miss-runtime product amongst the approaches. Intr+IOMMU's high product is a result of its long runtime compared to the other approaches as discussed in Section 3.6.1.

Compared to Intr+IOMMU, PTE+IOMMU has a smaller host-TLB-miss-runtime product despite its longer runtime. Intr+IOMMU has a higher product compared to PTE+IOMMU

Figure 3.9: Runtime in cycles (logscale) of the proposed approaches executing the linear pattern for 1, 2, 4, 8 and 16 threads assessing 131072 elements

because the host suffers from more TLB misses as the interrupt service routine (ISR) executes. A TLB miss by a device memory request triggers the ISR that computes the address translation, which triggers further TLB misses by the host. In conclusion, the approaches using a software algorithm for address translation have a higher host-TLB-miss-runtime product because the mechanism for address translation has a longer runtime or more TLB misses triggered by the interrupt service routine.

Trend in the performance as the number of threads increase

Thus far, the experiments in Sections 3.6.1 were executed with 16 threads. Although a device executing 16 threads is a realistic scenario for an OpenCL kernel, current and future devices on FPGA will support more than 16 threads. Hence, this section will observe the trend as the number of threads increase for the proposed approaches.

Figures 3.9, 3.10 and 3.11 show the runtime in cycles (logscale) of the proposed approaches for 1, 2, 4, 8 and 16 threads accessing 131072 elements for all patterns. The workload increases linearly as the number of threads increases, therefore the ideal curve on the graph is a flat

Figure 3.10: Runtime in cycles (logscale) of the proposed approaches executing the page pattern for 1, 2, 4, 8 and 16 threads assessing 131072 elements

Figure 3.11: Runtime in cycles (logscale) of the proposed approaches executing the random pattern for 1, 2, 4, 8 and 16 threads assessing 131072 elements

line. Results show that, within an approach for a given pattern, the runtime increases as more threads execute the kernel, since the address translation mechanism is shared amongst the threads, making the address translation mechanism a bottleneck. Only Intr+TLB MGMT and PTE+MMU exhibit near-ideal behaviour when going from one to two threads before hitting the bottleneck when even more threads are added.

As additional threads are added, more memory requests are executed in parallel. The address mechanisms in the approaches can only service a single memory translation at a given time. To achieve a near-ideal behaviour as additional threads are added, the implementations of the address mechanisms need to support more than one address translation concurrently.

3.6.2 Evaluating UT-OCL with SVM and without SVM support

The procedure for allocating memory, enabling host-to-device/device-to-host memory access and releasing memory is different when using UT-OCL and using UT-OCL+SVM. When allocating memory in UT-OCL, the host creates a memory object. To enable host access to the allocated memory, the host maps the memory object to its address space using the map command. To release the memory, the host unmaps the memory object using the unmap command, then releases it.

When allocating or releasing memory using UT-OCL+SVM, the host uses system calls (e.g. *malloc* or *free*). The access to the allocated memory is managed by the operating system, thus no additional commands need to be executed by the host application.

The trade-offs between UT-OCL+SVM and UT-OCL support were evaluated by profiling the time to execute: allocating memory, initializing data, executing the kernel and releasing memory. The PTE+MMU approach is used to implement UT-OCL+SVM, since it has the best runtime in Section 3.6.1. To evaluate both versions of UT-OCL, a vector addition benchmark was executed on various input sizes (1KB, 4KB, 16KB, 64KB, 256KB and 1024KB). For every input size of Figure 3.12, the bars on the left refer to runtimes using UT-OCL and the bars on the right refer to runtimes using UT-OCL+SVM.

In Figure 3.12, *Alloc Input* and *Alloc Output* refer to the runtime for allocating and enabling host access to the memory for the input array and output array respectively. *Initialize Data* refers to the runtime for initializing the input and output arrays. *Kernel Execution* refers to the

Figure 3.12: Runtime using UT-OCL and UT-OCL+SVM

runtime for executing the kernel on the device. *Release Input* and *Release Output* refer to the runtime for releasing the input array and output array respectively. The Alloc Input/Output are not clearly visible for UT-OCL+SVM because the runtimes are significantly smaller than the other portions of the overall runtime.

With UT-OCL, allocating memory and releasing memory have a constant runtime. In UT-OCL, allocating memory and releasing memory, the work performed for these tasks does not depend on the input size. These tasks are completed by using kernel drivers to perform the map and unmap commands, which have constant runtime. Whereas in UT-OCL+SVM, the runtime for allocating memory and releasing memory increase as the input size increases, because the operating system searches for unallocated (free) memory pages during allocation and for the allocated (used) memory pages during the release. Therefore, the smaller the input size, the fewer memory pages are traversed during the task of allocating memory and releasing memory.

Compared to UT-OCL+SVM, UT-OCL's Initialize Data portion is larger. When initializing a mapped memory object in UT-OCL, additional address translations are performed to access the mapped memory object. These additional address translations are the reason for the additional runtime.

Table 3.4: Resource Utilization

Approach	FF	LUT	BRAM
UT-OCL	70684	83357	99
Intr	72359 (+2.4%)	84429 (+1.3%)	99 (+0.0%)
Intr+IOMMU	72571 (+2.7%)	84148 (+0.9%)	101 (+2.0%)
PTE+IOMMU	73312 (+3.7%)	84286 (+1.1%)	101 (+2.0%)
PTE	72720 (+2.9%)	83690 (+0.4%)	99 (+0.0%)
PTE+MMU	73350 (+3.8%)	84988 (+2.0%)	100 (+1.0%)

The runtime of the Initialize Data portion for both versions of UT-OCL increases as the input size increases, which is expected as the work for initializing the data is proportional to the input size. Furthermore, for both versions of UT-OCL, the Kernel Execution portion of the runtime increases proportionally with the input size, which is also as expected since the work performed by the kernel is directly proportional to the input size.

For input size less than 1024KB, UT-OCL+SVM performs better than UT-OCL. At input size 1024KB, the runtime for UT-OCL+SVM exceeds that of UT-OCL, where the Kernel Execution is the dominant portion of the overall runtime. As the input size gets larger, the kernel performs more work including memory requests that require virtual-to-physical memory translations. At input size 1024KB, the overhead of performing these memory translations dominate within the Kernel Execution runtime. On average, across all input sizes, the overhead produced by the hardware mechanisms supporting SVM accounts for 40% of the Kernel Execution time. The lower times for Alloc Input/Output and Release Input/Output hide that cost until an input size of 1024KB.

As mentioned, in Section 3.6.1, the address translation mechanisms service a single memory translation at a given time. As the number of threads increase, which is possible in the OpenCL model, these address translation mechanisms become a bottleneck as shown in Fig. 3.12 when the input size is 1024 KB. Future work needs to address the impact of the bottleneck by improving the design of the address translation mechanism in UT-OCL+SVM to support multiple address translations concurrently.

3.6.3 Resource Utilization

Table 3.4 shows the resource utilization between the hardware systems implementing the SVM approaches and the hardware system in UT-OCL. The first column represents the approach, and Columns 2 through 4 represent the flip-flops (FF), Lookup Tables (LUT) and Block RAMs (BRAM). The relative resource utilization of the approaches to the UT-OCL hardware system are shown in parentheses. Intr+IOMMU and Intr+TLB_MGMT use the same hardware system. Therefore, the resource utilization of Intr+TLB_MGMT is not shown in Table 3.4. To implement the different approaches, support for SVM requires up to 3.8% more flip flops (FF), 2.0% more LUTs and 2.0% more BRAM compared to the baseline UT-OCL platform, which is minimal overhead compared to the baseline platform. In the end, the benefits of SVM can be achieved with minimal overhead to the system, however, as mentioned in Section 3.6.2, to achieve comparable results to a system without SVM support, parallelism needs to be added to the design of the address translation mechanisms.

PTE uses 1351 additional FFs and 739 fewer LUTs than Intr. The PTE Engine in PTE implements the software algorithm for translating a virtual address to a physical address in hardware. Therefore, in addition to the runtime benefit for using PTE over Intr mentioned in Section 3.6.1, the PTE hardware system uses fewer LUTs than the Intr hardware system.

The number of additional BRAMs are used by the IOMMU or the MMU. For Intr+IOMMU and PTE+IOMMU, the additional BRAMs are used to create the additional two read ports for the TLB in the IOMMU. For PTE+MMU, the additional BRAM is used by the TLB in the MMU.

PTE+MMU uses the most FF and LUT compared to the other approaches. However, it uses 779 more FF, 840 more LUT and 2 fewer BRAM than the Intr+IOMMU (Intr+TLB_MGMT) that has comparable runtime results in Section 3.6.1. PTE+MMU requires 2666 FF, 1631 LUT and one addition BRAM for its implementation, which is not significant resource given the abundance of resource in modern FPGAs.

3.7 Conclusion

In this Chapter, six different approaches for implementing Shared Virtual Memory (SVM) in an OpenCL framework for embedded systems on FPGAs were proposed. These approaches satisfy the Fine-Grained system SVM type, which is more than sufficient to comply with the OpenCL specification. In addition, the proposed approaches enable the device to address both virtual memory and physical memory, which is a possible configuration in embedded systems.

Using the three patterns, the results show that reserving entries of the TLB in the IOMMU increases the performance for kernel execution at the cost of the host application's performance. In addition, results also show that the hardware implementation for translating a virtual address to a physical address performs between approximately 18 and 22 times faster than the software implementation, and the hardware implementation uses fewer LUTs than the software implementation. Amongst all the proposed approaches, the approach using the hardware implementation for the address translation algorithm with a Memory Management Unit (PTE+MMU) performs the best. PTE+MMU has the shortest runtime, the highest hit rate and scales well with the addition of threads.

Furthermore, using a vector addition benchmark, the evaluation of UT-OCL with SVM and without SVM support was performed. For input sizes less than 1024KB, the presence of SVM in the OpenCL framework reduces the runtime. At input size 1024KB, the hardware mechanisms for supporting SVM increase the runtime of the kernel execution, which is dominant in the overall runtime. However, the runtime for allocating memory and enabling host-to-device/device-to-host memory access is shorter with the presence of SVM for all input sizes.

Chapter 4

Pipes in the OpenCL Standard

In the FPGA community, FPGA vendors provide OpenCL programming and runtime environments that raise the FPGA design abstraction to a much higher level than hardware description languages [63] [4], and it is the responsibility of these OpenCL environments to implement the functionality and constructs found in the open standard.

As the OpenCL standard continues to evolve, new functionality and new constructs are added. For example, in OpenCL 2.0 [7], a new construct with various modes known as a *pipe* has been introduced. With the introduction of pipes into the standard, inter-kernel communication is now feasible, implying that streaming applications, for which FPGA platforms execute well, can easily be modelled by the standard. It is worth mentioning that there was a strong influence by the FPGA community to add this construct to the standard. Despite this effort, the pipe implementations from two major FPGA vendor's do not conform to the OpenCL specification at this time.

This Chapter introduces a novel pipe implementation in hardware designed for OpenCL systems that can be implemented in Xilinx FPGAs. In addition, two software pipe implementations, as well as other designs to implement the various modes of a pipe are presented. The novel pipe implementation is the first pipe implementation for FPGAs to be published that conforms to the OpenCL specification. The main contribution of this Chapter is exploring different pipe implementations in an OpenCL framework for FPGAs.

This Chapter continues with some background on pipes given in Section 4.1. The hardware details of the novel pipe implementation are described in Section 4.2 with details of its software

Figure 4.1: Executing kernel application with pipe support

Figure 4.2: Executing kernel application without pipe support

driver found in Section 4.3. In Section 4.4, this novel pipe implementation is contrasted with related work. Section 4.5 discusses other pipe implementations. Section 4.6 analyzes the different modes for the various pipe implementations in terms of performance and resource utilization, and the chapter ends with concluding remarks and notes on future work in Section 4.7.

4.1 Background

Pipes were introduced in the OpenCL 2.0 specification [7] to enable communication amongst kernel instances during the runtime of the kernels. Figure 4.1 shows an example of two kernels that communicate using a pipe. *Kernel A* executes on *Device A*, *Kernel B* executes on *Device B*, and the pipe is modelled by a memory component. In this example, *Kernel A* can send data to *Kernel B* directly through a pipe.

Figure 4.2 shows an example of a system without pipe support. In this example, to transfer data from *Kernel A* to *Kernel B*, the host must read the data from *Device A*'s memory and write it to *Device B*'s memory. In contrast to a system with pipe support, the pipe replaces the transfer from *Device A*'s memory to *Device B*'s memory using the host. Hence, depending on the platform implementation, the presence of pipes can reduce the application runtime by reducing the time to transfer data from *Device A* to *Device B*, and freeing the host to do other tasks.

In the OpenCL specification, a pipe is much more than what is implied by its name, i.e., a pipe is not merely a FIFO that hardware designers would use to stream data from one block to another. A pipe is a memory object that stores a sequence of data structures (packets) in an ordered sequence. The packets within a pipe can be of any data type and the data type is uniform throughout the pipe. The pipe can be used as a First-In-First-Out (FIFO) data structure, but it can also be used as a conventional memory component where each packet in the pipe can be accessed directly.

Access to pipes are performed using built-in functions specified in the OpenCL specification. There are designed functions that must be executed by a single work-item or by all work-items of a single work-group simultaneously. At any one time, only one kernel instance may write to a pipe and only one kernel instance may read from a pipe. It is the responsibility of the device driver to ensure that the same kernel does not perform a read and write operation on a pipe instance. Not satisfying this constraint results in a compilation error.

In FIFO mode, the pipe's functionality is identical to that of a classic FIFO given that the work-items perform a generic read or write operation on the pipe. When the pipe is used as a conventional memory component, the work-items must allocate packets for direct access. This packet allocation is associated with a *reservation ID* that is referenced during a direct packet access. The reservation ID refers to a set of packets within the pipe that are reserved for direct access.

For a pipe implementation to conform with the OpenCL specification, it must be able to support one active reservation ID. An active reservation ID is a reservation ID where the work-items can access the packets within the reservation ID. A reservation ID obtained by a work-item or a work-group is equivalent to one active reservation.

4.2 Proposed Hardware Implementation (Pipe-hw)

It is important to remember that pipes are introduced as a programming language construct in the OpenCL specification, and their implementation can be done strictly in software. However, to leverage the benefits of an FPGAs, the constructs and behavior of OpenCL can be implemented in hardware. Figure 4.3 shows a block diagram of the proposed implementation of a

Figure 4.3: Pipe hardware implementation

pipe construct into hardware. The pipe implementation uses an AMBA AXI4-Lite interface [47] to make it easy to connect within the UT-OCL hardware system (Section 2.3) that uses mostly Xilinx IPs with this interface. The read and write ports have separate controllers to service the requests from the ports concurrently.

To instruct the pipe module to perform the built-in functions of an OpenCL pipe, the functions and their parameters are encoded in the address channel of the read and write ports. Hence, a portion of the address space is used by the implementation to encode the pipe functions.

To execute a pipe function, one cycle is required to decode the address channel. Another cycle is required to execute the function, except for reads and writes to a reservation ID, which requires two cycles. The additional cycle is used to setup the address of the read/write. And another cycle is required to send the read or write acknowledgement to conform with the AXI4 protocol.

When the pipe functions as a FIFO, it is said to be in FIFO mode. When the pipe functions as a conventional memory component, the pipe can act as a parallel-to-serial or serial-to-parallel buffer, and is said to be in parallel-to-serial or serial-to-parallel buffer mode respectively. Pipes in the latter modes can be used in network applications.

A pipe in parallel-to-serial buffer mode has more than one work-item writing to the pipe in parallel and a single work-item reading from the pipe serially, hence its name. Figure 4.4 shows the steps performed by the work-items in a group for writing to a pipe in parallel-to-serial buffer mode. The first step requires the work-group to obtain a reservation ID. Each work-

1. work-group obtains write reservation ID
2. work-group verifies that the write reservation ID is valid
3. work-items of work-group write to a unique packet within the reservation ID
4. work-group commits the write reservation ID

Figure 4.4: Steps for writing to a pipe in parallel-to-serial buffer mode

item in a work-group executes this step and obtains the same reservation ID. When obtaining a reservation ID, the work-group specifies the size in packets of the reservation ID. In the example of Figure 4.4, the size of the reservation ID is the number of work-items in the work-group.

When obtaining a write reservation ID, the implementation verifies that there are enough unused packets in the packet array that can be allocated to the reservation ID. Similarly, when obtaining a read reservation ID, the implementation verifies that there are enough committed packets in the packet array that can be allocated to the reservation ID. The packet array is the memory component that stores the pipe's packets.

The packet allocation for a reservation ID is very similar to memory allocation, which is outside the scope of this work. The majority of the hardware memory allocators either implement a software algorithm in hardware [64], or the traditional buddy system [65] with hardware improvements and/or optimizations [66]. Nonetheless, these implementations are resource heavy and will not be used in the pipe implementation. Therefore, the implementation is restricted to one read and one write reservation ID, thus simplifying the algorithm for managing the reservation IDs and allocating packets, but meeting the requirements to qualify as a pipe in an OpenCL implementation. However, a single reservation ID limits the simultaneous pipe access to a single work-item or a single work group per kernel. Hence, when more than one reservation ID is needed in an application, the application will suffer from a performance hit, because the application stalls until a reservation ID is available. A work-around is to allocate additional pipe objects within the OpenCL application to satisfy the simultaneous reservation ID requests at the cost of additional resources to instantiate the extra pipes in hardware. When allocating packets for a reservation ID, the implementation allocates the packets consecutively

in the packet array.

The reservation ID obtained from the first step may be invalid. An invalid reservation ID is defined by the pipe's implementation. In this implementation, a reservation ID may be invalid if the pipe does not have any free reservation IDs or if the number of packets requested for a reservation ID cannot be granted. A valid reservation ID is an active reservation ID in the pipe module that can be accessed by the work-items.

The second step is to verify that the reservation ID obtained from the first step is valid, and this step is present to conform with the OpenCL specification. In the implementation, access to an invalid reservation ID is ignored. The *read reservation ID manager* and *write reservation ID manager* from Figure 4.3 manage the read and write reservation IDs respectively. In this step, the implementation simply queries these components to verify the state of a reservation ID, e.g. valid or invalid. If the reservation ID is valid, then the work-item continues to step 3, otherwise the work-item will re-execute from step 1.

For the third step, each work-item of a work-group writes to a packet location in the reservation ID. In the example of Figure 4.4, the work-items write to a unique packet location. An example of a unique packet location is a location equivalent to the work item's local identifier. When reading and writing to a reservation ID, the implementation ignores access to invalid reservation IDs, as well as packets addressed outside the set of packets in a reservation ID. In addition, when the read reservation ID is obtained, the implementation ignores reads that do not use a reservation ID. Similarly, when the write reservation ID is obtained, the implementation ignores writes that do not use a reservation ID. Ignoring these reads and writes simplifies packet management and does not affect common use cases involving reservation IDs. Furthermore, in this implementation, if the reservation ID was obtained by a single work-item, then the work-item's ID is stored in the reservation ID manager and used to grant future accesses.

In the OpenCL specification, when writing to a pipe, the user must verify if the write was successful. As a result, the pipe implementation stores a bit representing the function's completion status. The status is either successful or unsuccessful. The bit is stored in the *write status array* or *read status array* shown in Figure 4.3.

The read and write status arrays are implemented using LUT RAM and can store 256 entries, occupying one SLICE in the Xilinx architecture. The array is indexed using a parameter in

1. work-group obtains a valid write reservation ID
2. work-group commits the write reservation ID
3. work-group obtains a valid read reservation ID
4. work-group commits the read reservation ID

Figure 4.5: Steps for using the pipe as a work group barrier

the address channel. Given that one work-item maps to one entry in the array, the current implementation can manage 256 unique work-items to read from a pipe and another 256 work-items to write to a pipe, for a total of 512 unique work-items. The arrangement of the work-items is flexible (e.g. four work-groups of 64 work-items or one work-group of 256 work-items), however the work-items must be assigned a unique id ranging between 0 and 255.

The fourth step consists of committing the reservation ID. When all work-items in a work-group execute this step, the reservation ID is committed. When committing a write reservation ID, the packets from the reservation ID can be accessed by reading from the pipe. In addition, when committing a write reservation ID, the set of packets within the reservation ID is added to the packet array while maintaining their sequential order. Also, when committing a read reservation ID, the packets from the reservation ID are removed from the pipe, and its storage is available to be written to by the pipe.

The purpose of the hardware implementation is to provide services for the built-in functions of an OpenCL pipe. Other than these services, the implementation can allow work-group synchronization performed by the *work_group_barrier* function in the OpenCL specification. For work-group synchronization to function with the implementation, a single instance of the implementation must be reserved for this service, and the packet array must be empty. Work-group synchronization will function for work-groups with no more than 256 work-items by using the steps shown in Figure 4.5. Barrier implementations require all threads to enter the barrier before any thread can request to exit the barrier. The barrier mechanism is imposed at step 3, where the work-group stalls when obtaining a read reservation ID until all the work-items have committed the write reservation ID from step 2. In contrast to barrier implementations,

work-items committing the write reservation ID is analogous to threads entering the barrier, and work-items obtaining a read reservation ID is analogous to threads exiting the barrier.

In the pipe implementation, the depth of the packet array and width of the packet can be configured during run-time. The implementation is memory mapped and written in HDL. In the end, the pipe implementation is the only known implementation to date that conforms to the OpenCL specification. It is developed as a peripheral so it can be used as a construct in FPGA OpenCL implementations.

Hereafter, the pipe implementation will be referred to as *pipe-hw*, where the packet array can be implemented with BRAM or LUTRAM. The pipe implementation using BRAM as the packet array is referred to as *pipe-hw-bram*, and the pipe implementation using LUTRAM as the packet array is referred to as *pipe-hw-lutram*.

4.3 Pipe Software Driver

This section describes the implementation of the software driver used to interact with pipe-hw.

In the driver implementation, a pipe is defined as a pointer to an object with 17 attributes. Each attribute corresponds to the encoded address format of the functions supported by pipe-hw. These addresses must be initialized prior to using the pipe object in the kernel implementation. When targeting the 32-bit MicroBlaze processor the driver implementation uses 17% fewer instructions, where on average, the built-in functions use 63% fewer instructions compared to a driver implementation where the address encoding is computed within its respective built-in function. To put this space savings into perspective, within the space saved, 21 pipe objects can be stored. Moreover, such an implementation is well-suited for embedded systems where space is scarce.

A reservation ID is an object with two attributes: a pipe attribute, corresponding to the pipe associated with the reservation ID, and a status attribute, that contains the integer representation of the reservation ID and the reservation ID type (read or write).

When initializing the addresses in the pipe object, for functions where the work-item's ID is needed, the driver uses the result from the *get_global_linear_id* built-in function. The *get_global_linear_id* function linearises the multi-dimensional space that represents a kernel in-

stance and assigns a unique ID to the work-items. In the OpenCL specification, there is no built-in function to retrieve the number of work-items in a multi-dimensional work group. A function similar to `get_global_linear_id` is recommended to be added to the OpenCL specification, where the multi-dimensional space representing the work items is linearised to compute the total number of work items in a work group. Such a function was implemented within the driver to retrieve the number of work-items in a work-group, and it has the following signature: *get_local_linear_size()*.

4.4 Related Work

To date, there is no prior work done that qualifies as a pipe implementation in the context of OpenCL for FPGAs. This lack of pipe implementations in the context of OpenCL for FPGAs is not surprising, since the OpenCL standard was only adapted recently by the FPGA community.

There are two works that are the most similar to pipes in the realm of FPGAs. The first is *channels* in Altera's OpenCL SDK, and the second is *pipes* from Xilinx. Altera's channels do not qualify as OpenCL pipes since they do not implement the built-in functions from the OpenCL specification.

To enable the use of Altera's channels, the user must insert a directive, also known as a pragma, into the code. To use the channels, the user must use function calls that are dependent on Altera's API, where the function signatures differ from the OpenCL specification. In Altera's API, the user is unable to obtain the status of reading from or writing to a channel, where it is feasible in the OpenCL specification. Furthermore, from the perspective of the kernel code, a channel is declared as a global variable, whereas from the perspective of the kernel code satisfying the OpenCL specification, a pipe object is passed as a parameter to the kernel.

In the Altera SDK, the compiler extracts the variables that should be treated as a channel and infers a FIFO interconnecting the kernels. The width of the channel is computed depending on the assigned type and its depth is pre-computed in a performance analysis stage in the CAD flow. The user can overwrite these values, but the channel's width and depth cannot be changed during run-time as required by the OpenCL specification. If the width or depth of a channel needs to be changed, then the FPGA would need to be reconfigured, which requires additional

time. In the proposed implementation, the depth and width of the pipe can change during run-time without the need to reconfigure the FPGA, saving the overhead cost of reconfiguring the FPGA.

While there are differences in the syntax, the major difference between Altera's channels and pipe objects is with regards to compliance to the OpenCL specification. Altera channels lack support for reservation IDs. Since Altera's channels do not support reservation IDs, they cannot be used as a conventional memory component. Therefore, Altera channels cannot be used as a parallel-to-serial buffer and perform the steps in Figure 4.4, which are commonly found in networking applications.

Xilinx's pipe implementation supported in their SDAccel tool [67] has identical characteristics to the Altera channels. Although Xilinx labels their implementation as a pipe, their implementation does not satisfy the OpenCL specifications. Xilinx's pipes are implemented with on-chip memory and the AXI Stream interface.

In the end, the proposed implementation, pipe-hw, is the only implementation of a pipe object that conforms to the OpenCL specification and maintains portability. In addition, the Altera channels and the Xilinx pipes implement limited functionality of an OpenCL pipe.

4.5 Other pipe implementations

In this Section, other approaches to build the OpenCL pipe functionality that can be compared to the pipe-hw implementation are described. There are two versions implemented strictly in software and others that use some off-the-shelf IP blocks.

4.5.1 Software pipe implementations

In addition to pipe-hw, two software implementations were created. The software implementations model a FIFO using a head pointer, a tail pointer and a counter [68], protected by a mutex variable so the implementation is thread-safe. The mutex variables are managed by the Mutex IP from Xilinx [23]. To implement the mechanism to obtain or commit a reservation ID, a variable is used to indicate the status of the reservation ID, also protected by a mutex.

There are two versions of the software implementation, a version that uses the off-chip

memory as storage (*pipe-sw-ddr*), and another version that uses the on-chip memory as storage (*pipe-sw-bram*).

4.5.2 Using off-the-shelf IPs

To support the FIFO mode, there is an option to use the custom FIFO implementations created using Xilinx’s IP wizard. However, Xilinx also provides a Mailbox IP [69] where its implementation uses a FIFO. The Mailbox IP has a software API facilitating communication to the IP using a processor. Furthermore, the Mailbox IP is similar to Altera’s channel hardware implementation, however the Mailbox IP is optimized for the Xilinx architecture, which is the target architecture in these experiments. Therefore, the Mailbox IP is chosen over the use of Altera channels to compare with the pipe-hw in FIFO mode.

The Mailbox IP has been modified to allow a unidirectional communication, thus using one FIFO to be comparable to pipe-hw and the software implementations (*pipe-sw-ddr* and *pipe-sw-bram*). In contrast to pipe-hw, the Mailbox has two AXI4-Lite interfaces, one reading from the FIFO and another for writing to the FIFO, which is the same interface as the pipe-hw. The Mailbox requires one cycle to read from or write to the FIFO. The Mailbox IP is used in the experiments to evaluate the effects of having an additional access port as well as a reduced cycle time to read from and write to the FIFO on performance.

For the barrier mode, there is an extensive amount of published work for barrier implementations on FPGAs [70] [71] [72]. However, none of these works make the implementation available to the public and the lack of implementation detail presented in the publications make it difficult to reproduce the implementations accurately. Furthermore, the implementations may be dependent on the network topology, which would result in significant architecture changes in the hardware system. Therefore, pipe-hw is compared with a centralized barrier software implementation (*barrier-sw*) using a counter, two flags and a mutex variable [73]. The mutex variable will be managed by the same Mutex IP from the software implementations.

4.6 Results

In this section, the performance and resource utilization of the hardware pipe implementation, *pipe-hw*, is compared with the other implementations presented in Section 4.5. For each experiment, the application code is constant and only the pipe implementation changes. All experiments have been executed using the ML605 development board with designs targeting a 100 MHz system clock.

In the system, the off-chip memory is the DDR, and the on-chip memory is the BRAM. For the software implementations, the implementation using off-chip memory is referred to as *pipe-sw-ddr*, and the implementation using BRAM as on-chip memory is referred to as *pipe-sw-bram*.

The runtimes were recorded using the profile interface in the UT-OCL framework. To account for Operating System overhead (context switching and scheduling), ten runs of the same experiment were executed and the average of these runs were taken.

4.6.1 Performance

This section will present the performance results of the pipe implementations in the different modes.

FIFO mode

In this section, *pipe-hw-bram* acting as a FIFO is compared with the Mailbox implementation described in Section 4.5. For this experiment, a version of a two-dimensional Gaussian filter [74] is implemented. The application was created with two kernel instances, where the work of a kernel instance computes the filter along one dimension. The filter was applied to square images with a single dimension being: 512, 1024, 2048 and 4096.

The application was implemented in software and in hardware. The software version of the application is executed on two devices, each consisting of a single MicroBlaze processor. The hardware version of the application is executed on a device containing the kernel as a built-in function. The devices with built-in functions were created using the Vivado High Level Synthesis (HLS) Tool [45]. The application follows the model in Figure 4.1.

Figure 4.6: Gaussian Filter Application

Both versions of the application were executed using the Mailbox implementation and pipe-hw-bram. Figure 4.6 shows the runtime of the application using the pipe-hw-bram normalized to the runtime of the application using the Mailbox. *SW App* denotes the software application and *HW App* denotes the hardware application.

To put the benefits of pipes into perspective, the execution of the application using the models from Figure 4.1 and Figure 4.2 are compared. For the software version of the application, copying the data from one device to another requires an average of 2.4 times the runtime of pipe-hw-bram. And, for the hardware version, an average of 3.6 times the runtime of pipe-hw-bram is required. These runtimes would be in addition to the runtime of the kernel. It is clearly evident that the implementation of pipes into the system does provide performance benefits compared to a system without pipes.

When executing the software application with pipe-hw-bram, a constant runtime overhead averaging 8% more than the Mailbox implementation, shown in red in Figure 4.6, is observed. The runtime overhead is constant and independent of the image dimension, implying that the runtime overhead is proportional to a constant amount of work performed by the application. When using pipe-hw-bram, the additional work performed by the application is found in the

Table 4.1: Resource utilization of the hardware kernel instances

Hardware IPs	FF	LUT
Mailbox	6503	6292
Pipe	6380	6021

additional two cycles to execute a function in hardware as well as the additional instructions needed by the MicroBlaze to write to and read from the pipe. There are 2.7 times more instructions needed for reading from and writing to pipe-hw-bram compared to reading from and writing to the Mailbox. The work performed to initialize the pipe object becomes fairly insignificant at these image sizes.

For the hardware application, when comparing pipe-hw-bram with the Mailbox implementation, results show that the runtime of the application increases as the image dimension increases. Such a behaviour is the result of contention for the read port of the AXI4-Lite interface that is shared amongst the two kernel instances. This result also demonstrates the benefit of the additional port in the Mailbox IP.

Table 4.1 shows the number of flip-flops (FF) and Lookup Tables (LUT) used to implement the kernel instance in hardware using Vivado HLS. These results were achieved by adding the hardware kernel instances to a base system and comparing the resource utilization published in the post-placement report with that of the base system's. The application using the Mailbox IP uses more FF and LUT than pipe-hw-bram. These additional resources are used to implement the logic for reading from and writing to a Mailbox in hardware.

For the kernel instances implemented in hardware, there are an additional two cycles for writing to and reading from the pipe compared to writing to and reading from the Mailbox. These cycles are used to translate the function's completion status read from the pipe to the function's return value specified by the OpenCL standard. So, although the software implementation for reading from and writing to the pipe component has more instructions compared to writing to and reading from the Mailbox, the implementation maps well to hardware as the majority of these instructions can be performed in parallel, where the parallelism is extracted by Vivado HLS.

In the end, when comparing the Mailbox implementation to pipe-hw-bram, an additional interface does reduce the contention for the read port, which is assigned to many functions of

Figure 4.7: Synthetic Network Application Runtime

the built-in function for a pipe. Moreover, the functions in the driver that read from and write to a pipe does have significant overhead, but maps well to hardware when using hardware kernel instances.

Parallel-to-serial buffer mode

In this section, the performance of the pipe-hw-bram, pipe-sw-ddr and pipe-sw-bram in parallel-to-serial buffer mode are evaluated. For this experiment, a synthetic application that models a network application is created. The application consists of two kernel instances, which follows the model in Figure 4.1. The first kernel instance reads packets from off-chip memory, computes its parity and writes the result to the pipe. The second kernel instance reads from the pipe and writes to a dummy buffer as it would write to a network controller.

The first kernel instance is executed on a device with 16 MicroBlazes and the second kernel instance is executed on a device with one MicroBlaze. The work items from the first kernel instance write to the pipe in parallel, and the work item in the second kernel instance reads from the pipe sequentially. The application is executed on messages with 4096 packets using six different sizes for the reservation ID: 32, 64, 128, 256, 512 and 1024. The runtimes of the

Table 4.2: Normalized Runtime of barrier-sw

Number of processors	2	4	6	8	10	12	14	16
Normalized Runtime	4.6	14.1	20.0	25.9	31.8	37.7	43.6	49.5

software pipe implementations are normalized to the runtimes of the pipe-hw-bram and are shown using the bars in Figure 4.7. The runtime, in units of cycles, for pipe-hw-bram is also shown in Figure 4.7 using a dotted line.

From Figure 4.7, the results show that both software implementations are significantly slower than pipe-hw-bram. Thus making these software implementations impractical for modern day applications. The results also show that the reservation size does impact the application's runtime. The rapid increase in runtime from reservation size 512 to 1024 is a victim of workload imbalance. In the experiment with the reservation size of 1024, the reservation size is equal to the pipe's size. In this scenario, the kernel instances wait for one-another to complete, reducing parallelism in the application. When using pipes, it is important to allocate a reservation size that enables the kernel instances to progress in execution whilst absorbing the overhead of obtaining and committing a reservation ID. For the experiments executed, a reservation size of 512 does yield the best result.

Work-group synchronization mode

To evaluate the performance of a work-group synchronization, the barrier algorithm from Figure 4.5 and barrier-sw discussed in Section 4.5 are executed in software. These algorithms were executed on a device with 16 MicroBlaze processors with work groups containing an even number of work items ranging between two and 16. The runtimes of barrier-sw normalized to the implementation using pipe-hw-lutram are shown in Table 4.2.

From Table 4.2, the results show that as the number of processors increase the runtime of barrier-sw also increases. The increase in runtime is a result of the spin-lock mechanism used in the implementation. Such a mechanism increases network traffic and makes the Mutex IP a bottleneck. In addition, when using pipe-hw, the numerous steps from barrier-sw are performed concurrently in hardware by pipe-hw. For example, incrementing the counter atomically and comparing it with the total number of work items in a work group requires six transactions

Table 4.3: Resource Utilization of the hardware IPs

Implementation	FF	LUT	BRAM
Base System	38998	42332	69
Mailbox	39468 (+1.2%)	42601 (+0.6%)	70 (+1.4%)
pipe-hw-bram	39677 (+1.7%)	43235 (+2.1%)	70 (+1.4%)
pipe-sw-ddr	39158 (+0.4%)	42414 (+0.2%)	69 (+0.0%)
pipe-sw-bram	39478 (+1.2%)	42738 (+1.0%)	70 (+1.4%)
barrier-sw	39460 (+1.2%)	42712 (+0.9%)	69 (+0.0%)
pipe-hw-lutram	39670 (+1.7%)	43111 (+1.8%)	69 (+0.0%)

over the network when using barrier-sw. These steps are performed in a single transaction over the network when pipe hw commits a reservation ID.¹

4.6.2 Resource Utilization

To calculate the resource utilization of the individual IPs, seven systems were run through the placement stage of the Xilinx CAD tool. The first system contained the minimal components of UT-OCL’s hardware system required to execute the Host Application. This system is referred to as the base system. The other systems contained the base system with additional IPs needed for a particular pipe implementation. Table 4.3 shows the absolute resource utilization of these systems as well as the difference between the systems containing IPs and the base system shown in parenthesis. The first column represents the implementation, and Columns 2 through 4 represent the additional flip-flops (FF), Lookup Tables (LUT) and Block RAMs (BRAM) needed by each implementation. For all the implementations, the increase in FF and LUT utilization is less than 1% of the FPGA device used in these experiments.

¹During the execution of barrier-sw, a limitation was found with the AXI interconnect where a priority inversion is possible and happens when requesters get backed off and rearbitrated at precisely the right timing to lead to starvation. In the barrier implementation using a mutex variable, if the processor that acquired the lock is starved, then the barrier implementation will find itself in dead-lock. Xilinx was contacted in regards to this limitation, and Xilinx has confirmed this limitation. There is no public information about this limitation.

I fixed this limitation by implementing a mechanism that forces the arbiter to service a master’s request if it was not serviced within a finite number of grants. It has been tested on all AXI interconnect versions until 2.00.a, and the patch can be found at: www.eecg.toronto.edu/~pc/downloads/UT-OCL/.

FIFO functionality

Compared to the Mailbox IP, *pipe-hw-bram* uses more FF and LUT. These additional resources are used to implement the additional functionality provided by a pipe. If a FIFO is the only functionality needed within the application, the Mailbox implementation would be better suited.

Parallel-to-serial buffer

The least amount of FF and LUT is used by *pipe-sw-ddr*. In this implementation, the Mutex IP is the only additional IP used on the FPGA. Although the implementation uses the least amount of resources, as shown in Figure 4.7, its performance is significantly slower than the other implementations and impractical for most applications.

Amongst the implementations that use on-chip storage, the *pipe-sw-bram* has 199 fewer FF and 497 LUT. There is resource savings when using the *pipe-sw-bram*, however the additional resources used by *pipe-hw-bram* is insignificant when compared to the abundance of resources in modern FPGAs, and definitely worth the investment for the performance gain.

Work-group synchronization

When using *pipe-hw* as a work-group barrier, the content in the packet array is not used, implying that the size of the reservation ID can be arbitrary. Therefore, the packet array can be small and implemented using LUTRAM. By using *pipe-lutram*, the pipe implementation becomes comparable to *barrier-sw* that also uses a LUTRAM as storage.

The implementation with the *pipe-hw-lutram* uses 210 more FF and 399 more LUT than the *barrier-sw* implementation. However, similar to the parallel-to-serial buffer discussion, the absolute amount of additional resources used is insignificant when compared to the abundance of resources in modern FPGAs, and definitely worth the investment for the performance gain.

4.7 Conclusion

In this Chapter, *pipe-hw*, a novel hardware implementation of a pipe object for use in Xilinx FPGAs, was presented. *Pipe-hw* is also the only pipe implementation, to date, that conforms

to the OpenCL specification. In addition to pipe-hw, two software implementations of a pipe and some other hardware implementations using off-the-shelf IP blocks were also presented.

When comparing pipe-hw to the Mailbox, pipe-hw suffers from software overhead introduced by the software driver. Although software overhead is present, when the software driver is synthesized into hardware, there is only a two cycle overhead for the generic read and write functions. The results show that separate ports for reading from and writing to a FIFO would be beneficial as the work sizes increase. Hence, future work includes implementing the additional port.

When using the pipe as a parallel-to-serial buffer, pipe-hw performs better than the software implementations. Results showed that the reservation size does affect the application's performance. When the pipe functions as a barrier, steps from barrier-sw are absorbed in the hardware functionality of pipe-hw when obtaining and committing a reservation ID thus improving performance. For both these functions, the pipe-hw implementation uses more resources than the other implementations, but the absolute amount of additional resources used is insignificant when compared to the abundance of resources in modern FPGAs, and definitely worth the investment for the performance gain.

Therefore, in the end, it is better to use the Mailbox implementation if only the FIFO mode is desired. However for the other modes, the pipe-hw is a better choice than the other implementations.

Chapter 5

Conclusions and Future Work

As more custom components are being integrated into System-on-chips (SoCs), there is a higher degree of heterogeneity within these systems. Software developers use a high-level language to facilitate the programming of these systems. These high-level languages are essential to increase the use of FPGAs by software developers in today's market.

OpenCL is a standard that enables the control and execution of kernels on heterogeneous systems. Similar to many programming standards, the standard requires hardware support from the underlying system to implement its features and constructs. In this dissertation, UT-OCL, an OpenCL framework for embedded systems using Xilinx FPGAs, is presented, and, by using UT-OCL, Shared Virtual Memory (SVM) as well as the pipe construct from the OpenCL standard were explored.

5.1 Summary

There are three significant contributions presented in this dissertation. The first contribution is the development of UT-OCL, an open-source OpenCL framework for embedded systems using Xilinx FPGAs. The second contribution is the architectural exploration at the system level for *Shared Virtual Memory (SVM)*. And, the third contribution is the architectural exploration at the system level for a *pipe* object.

5.1.1 The UT-OCL Framework

The UT-OCL framework is composed of a hardware system and its necessary software counterparts, which together form an embedded Linux system augmented to run OpenCL applications within a single FPGA. The framework contains debugging tools and simple hooks that allow for custom devices to be easily integrated in the hardware system. With this framework, the user can experiment with all aspects of OpenCL primarily targeting FPGAs, including testing possible modifications to the standard as well as exploring the underlying computing architecture. In addition, when evaluating multiple devices using an open-source framework like UT-OCL, the environment and the testbenches are constant, leaving the devices as the only variable in the system. Therefore, the evaluation and the comparison between multiple devices are fair and easy to setup.

In Chapter 2, the architecture of UT-OCL’s hardware system and three devices were explored. Using the UT-OCL framework, the mechanism for transferring data between the host memory and device memory was explored. Results showed that, for the use of the Datamover cores to be beneficial in the system, direct access to the stream port by the MicroBlaze is necessary (Section 2.12.1). In addition, the ease of comparing two versions of a CRC application fairly was demonstrated (Section 2.12.2), and, a study of the trade-offs between resource utilization and performance for a device using a network-on-chip paradigm was presented (Section 2.12.3). The content from this Chapter is found in my published works [8] and [9].

5.1.2 Shared Virtual Memory

Shared Virtual Memory (SVM) is a feature in the OpenCL standard, where the device and host share the same address space. In Chapter 3, using the UT-OCL framework, six different approaches for implementing SVM were explored. Amongst all the proposed approaches, the approach using the hardware implementation for the address translation algorithm with a Memory Management Unit (PTE+MMU) performs the best. This Chapter encapsulates the content from my published work found in [10] as well as additional results.

5.1.3 Pipe

In the OpenCL standard, a pipe is a memory object composed of packets. The pipe is a storage unit where it can be used as a First-In-First-Out (FIFO) data structure, as well as a conventional memory component where each packet in the pipe can be accessed directly. Given these different use cases, the pipe can have different modes.

In Chapter 4, amongst other implementations, pipe-hw was presented. Pipe-hw is a novel hardware implementation of a pipe object designed for Xilinx FPGAs. Pipe-hw is also the only pipe implementation, to date, that conforms to the OpenCL specification. Results showed that the reservation size of a reservation ID relative to the pipe’s depth does affect the application’s performance. In addition, when the pipe functions as a barrier, the hardware absorbs many steps from the barrier-sw implementation making the pipe more efficient for this use case. In the end, it is better to use the Mailbox implementation if only the FIFO mode is desired. However for the other modes, the pipe-hw is a better choice than the other implementations. The content from this Chapter is found in my published work [11].

5.2 Other Ph.D.-related Publications

In addition to the work found in this dissertation, during my Doctor of Philosophy degree, exploration with coherent memory hierarchies on FPGAs was conducted. The content of this work was published [12] [13] [14]. This content is not present in this dissertation as the exploration was not conducted with the aid of the UT-OCL framework. Nonetheless, the contributions of these works could be explored within the UT-OCL framework to provide insight on coherent memory hierarchies within the context of an embedded system using OpenCL.

During the exploration of Shared Virtual Memory, a security flaw in the Xilinx design flow was discovered. This security flaw was exploited to modify the Memory Management Unit (MMU) of the MicroBlaze microprocessor [17], a secure IP from the Xilinx IP library, to create four of the six different approaches presented in Chapter 3. Details of the methodology for extracting the source code of a secure IP was published in [18].

5.3 Future Work

This dissertation presented an open-source OpenCL framework for embedded systems using Xilinx FPGAs. The research potential stemming from this framework can spread many research fields and topics. For example, the implementation of a cache-coherency mechanism between the host and the device subsystems as well as atomic operations can be explored in the system architecture.

Moreover, additional infrastructure can be built to embellish the usability and increase the functionality of the framework. As mentioned in Section 2.3.2, the majority of the infrastructure that enables partial reconfiguration on the FPGA is currently in place. For example, enabling the presence of devices at runtime, as opposed to statically (i.e. at compile time of the hardware system) as it is done now, would benefit from the properties of partial reconfiguration. As a result, the FPGA area will be utilized more efficiently as it is time-multiplexed by multiple device, and the usability of the framework will increase since new devices can be added to the system while it is running.

Currently, the framework uses GNU Project Debugger(GDB) [31] as the debugging tool and a profiling mechanism compliant with the OpenCL standard. However, the GDB environment is not designed for heterogeneous system and the profiling mechanism does not provide in-depth analysis of the functions executed on the device. Given the heterogeneous environment imposed by the OpenCL standard, the debugging paradigm can be explored to enable a more user-friendly experience and the profiling mechanism can be extended to enable in-depth analysis of the functions executed by the device. For example, CodeXL [75], an open-source debugging and profiling tools with graphical user interface made public by AMD, can be incorporated into the framework for a more user-friendly experience.

The UT-OCL framework uses Xilinx’s Embedded Development Kit (EDK) [76], which is now discontinued. The framework is also dependent on PetaLinux version 2013.10 [26], which, at the time of this dissertation, is eight versions older than the current version (version 2016.2) that contains better support and efficient functionality for the target architecture. In addition, the PetaLinux environment is being replaced by a more user-friendly and powerful tool, SDSoC [77]. Future work consists of updating the framework to use Xilinx’s latest embedded

system’s development tool, Vivado [78], and their more mature SDK for embedded systems, SDSoC [77].

In addition, the hardware design can be ported to newer FPGA platforms with an integrated SoC, such as the Zynq Ultrascale+ MPSoC [52]. This platform incorporates a cache-coherency mechanism between the SoC (host) and FPGA fabric (devices). Therefore, if this SoC platform is used, then the framework can leverage this cache-coherency mechanism and the task of building this mechanism in hardware can be avoided. In the end, the selection of the FPGA platform to port the hardware system is dependent on many factors, for example, the future direction of the framework. If the future direction of the framework is towards production, then a platform with reconfigurable fabric integrated with an SoC would be the ideal as the hardened system components can increase the performance of the system. If the future direction of the framework is towards academic research, then an FPGA with solely reconfigurable fabric would be ideal as the fabric can be used to explore the system architecture.

5.4 Recommendation for the OpenCL Standard

Through the development of the pipe-hw software driver (Section 4.3), the number of work-items in a multi-dimensional work group was needed. The number had to be calculated using other built-in functions. Given that this number could be useful in computing the workload of a work-group, a function similar to *get_global_linear_id* is recommended to be added to the OpenCL specification, where the multi-dimensional space representing the work items is linearised to compute the total number of work items in a work group. To follow the same naming convention of the Work-Item Built-in Functions (Section 6.13.1 of the OpenCL Specification), the function should have the following signature: *get_local_linear_size()*.

Bibliography

- [1] Forum, Message P. MPI: A Message-Passing Interface Standard. Technical report, Knoxville, TN, USA, 1994.
- [2] Khronos OpenCL Working Group. The OpenCL Specification Version: 1.0 Document Revision: 48, 2009.
- [3] Altera Inc. Altera Opens the World of FPGAs to Software Programmers with Broad Availability of SDK and Off-the-Shelf Boards for OpenCL. Press Release, May 2013.
- [4] Xilinx Inc. Xilinx SDAccel: A Unified Development Environment for Tomorrow's Data Center. http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-wp.pdf, 2014.
- [5] Xilinx Inc. Zynq-7000 All Programmable SoC Overview. http://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, 2014.
- [6] Xilinx Inc. Partial Reconfiguration User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/ug702.pdf, 2012.
- [7] Khronos OpenCL Working Group. The OpenCL Specification Version: 2.0 Document Revision: 22, 2014.
- [8] Vincent Mirian and Paul Chow. UT-OCL: an OpenCL framework for embedded systems using xilinx FPGAs. In *International Conference on ReConfigurable Computing and FPGAs, ReConFig 2015, Riviera Maya, Mexico, December 7-9, 2015*, pages 1–6, 2015.
- [9] Vincent Mirian and Paul Chow. Using an OpenCL framework to evaluate interconnect implementations on FPGAs. In *24th International Conference on Field Programmable*

- Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–4, 2014.
- [10] Vincent Mirian and Paul Chow. Evaluating shared virtual memory in an OpenCL framework for embedded systems on FPGAs. In *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015, Riviera Maya, Mexico, December 7-9, 2015*, pages 1–8, 2015.
- [11] Vincent Mirian and Paul Chow. Exploring pipe implementations using an OpenCL framework for FPGAs. In *2015 International Conference on Field Programmable Technology, FPT 2015, Queenstown, New Zealand, December 7-9, 2015*, pages 112–119, 2015.
- [12] Vincent Mirian and Paul Chow. FCache: a system for cache coherent processing on FPGAs. In *Proceedings of the ACM/SIGDA 20th International Symposium on Field Programmable Gate Arrays, FPGA 2012, Monterey, California, USA, February 22-24, 2012*, pages 233–236, 2012.
- [13] Vincent Mirian and Paul Chow. Managing mutex variables in a cache-coherent shared-memory system for FPGAs. In *2012 International Conference on Field-Programmable Technology, FPT 2012, Seoul, Korea (South), December 10-12, 2012*, pages 43–46, 2012.
- [14] Vincent Mirian and Paul Chow. An implementation of a directory protocol for a cache coherent system on FPGAs. In *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012, Cancun, Mexico, December 5-7, 2012*, pages 1–6, 2012.
- [15] Michael Adler, Kermin Fleming, Angshuman Parashar, Michael Pellauer, and Joel S. Emer. Leap scratchpads: automatic memory and cache management for reconfigurable logic. In *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 25–28, 2011.
- [16] Hsin-Jung Yang, Kermin Fleming, Michael Adler, and Joel S. Emer. LEAP Shared Memories: Automating the Construction of FPGA Coherent Memories. In *22nd IEEE Annual*

- International Symposium on Field-Programmable Custom Computing Machines, FCCM 2014, Boston, MA, USA, May 11-13, 2014*, pages 117–124, 2014.
- [17] Xilinx Inc. MicroBlaze Processor Reference Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/mb_ref_guide.pdf, 2012.
- [18] Vincent Mirian and Paul Chow. Extracting Designs of Secure IPs Using FPGA CAD Tools. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI, GLVLSI 2016, Boston, MA, USA, May 18-20, 2016*, pages 293–298, 2016.
- [19] Xilinx Inc. ML605 Hardware User Guide. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf, 2012.
- [20] Xilinx Inc. PetaLinux SDK User Guide: Board Bringup Guide. http://www.xilinx.com/support/documentation/sw_manuals/petalinux2013_10/ug980-petalinux-board-bringup.pdf, 2012.
- [21] Xilinx Inc. LibXil FATFile System (FATFS) (v1.00.a). http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/oslib_rm.pdf, 2012.
- [22] Xilinx Inc. Local Memory Bus (LMB) V10 (v2.00.b). http://www.xilinx.com/support/documentation/ip_documentation/lmb_v10/v2_00_b/lmb_v10.pdf, 2011.
- [23] Xilinx Inc. LogiCORE IP Mutex (v1.00a). http://www.xilinx.com/support/documentation/ip_documentation/mutex.pdf, 2010.
- [24] Xilinx Inc. AXI-to-AXI Connector (v1.00.a). http://www.xilinx.com/support/documentation/ip_documentation/ds803_axi2axi_connector.pdf, 2010.
- [25] Khronos OpenCL Working Group. OpenCL 2.0 Reference Card. Reference Card provided by Khronos Group, 2014.
- [26] Xilinx Inc. PetaLinux SDK User Guide: Installation Guide. http://www.xilinx.com/support/documentation/sw_manuals/petalinux2013_10/ug976-petalinux-installation.pdf, 2012.
- [27] CLang: A C Language Family Frontend for LLVM. <http://clang.llvm.org/>.

- [28] The LLVM Compiler Infrastructure. <http://llvm.org/>.
- [29] QEMU: Open Source Processor Emulator. <http://www.qemu.org/>.
- [30] Xilinx Inc. PetaLinux SDK User Guide: QEMU System Simulation Guide. http://www.xilinx.com/support/documentation/sw_manuals/petalinux2013_10/ug982-petalinux-system-simulation.pdf, 2012.
- [31] GDB: The GNU Project Debugger. <https://www.gnu.org/software/gdb/>.
- [32] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A performance-portable opencl implementation. *International Journal of Parallel Programming*, 43(5):752–785, 2015.
- [33] Hiroyuki Tomiyama, Takuji Hieda, Naoki Nishiyama, Noriko Etani, and Ittetsu Taniguchi. SMYLE OpenCL: A Programming Framework for Embedded Many-Core SoCs. In *ASP-DAC*, pages 565–567, 2013.
- [34] Sen Ma, Miaoqing Huang, and David L. Andrews. Developing Application-Specific Multiprocessor Platforms on FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1–6, 2012.
- [35] Eugene Cartwright, Sen Ma, David L. Andrews, and Miaoqing Huang. Creating HW/SW Co-Designed MPSoPC’s from High Level Programming Models. In *HPCS*, pages 554–560, 2011.
- [36] Wesley Peck, Erik K. Anderson, Jason Agron, Jim Stevens, Fabrice Baijot, and David L. Andrews. Hthreads: A Computational Model for Reconfigurable Devices. In *FPL*, pages 1–4, 2006.
- [37] Enno Lübbers and Marco Platzner. ReconOS: Multithreaded Programming for Reconfigurable Computers. *ACM Trans. Embedded Comput. Syst.*, 9(1), 2009.
- [38] David L. Andrews and Douglas Niehaus and Razali Jidin and Michael Finley and Wesley Peck and Michael Frisbie and Jorge L. Ortiz and Ed Komp and Peter J. Ashenden. Pro-

- gramming Models for Hybrid FPGA-CPU Computational Components: A Missing Link. *IEEE Micro*, 24(4):42–53, 2004.
- [39] Aws Ismail and Lesley Shannon. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *FCCM*, pages 170–177, 2011.
- [40] Taneem Ahmed. OpenCL Framework for A CPU, GPU and FPGA Platform. Master’s thesis, University of Toronto, Toronto, Canada, September 2011.
- [41] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From Opencl to High-Performance Hardware on FPGAS. In *FPL*, pages 531–534, 2012.
- [42] Muhsen Owaida, Nikolaos Bellas, Konstantis Daloukas, and Christos D. Antonopoulos. Synthesis of Platform Architectures from OpenCL Programs. In *FCCM*, pages 186–193, 2011.
- [43] Xilinx Inc. LogiCORE IP AXI DataMover v4.02.a. http://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v4_02_a/pg022_axi_datamover.pdf, 2012.
- [44] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, 2001.
- [45] Xilinx Inc. Vivado Design Suite User Guide High-Level Synthesis. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf, 2014.
- [46] Xilinx Inc. AXI Interconnect (v1.06.a). http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v1_06_a/ds768_axi_interconnect.pdf, 2012.
- [47] ARM Inc. AMBA AXI and ACE Protocol Specification, 2013.

- [48] Michael Papamichael and James C. Hoe. CONNECT: re-examining conventional wisdom for designing NoCs in the context of FPGAs. In *FPGA*, pages 37–46, 2012.
- [49] William Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.
- [50] Altera Inc. Arria 10 Device Overview. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/arria-10/a10_overview.pdf, 2016.
- [51] Muli Ben-yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. The price of safety: Evaluating IOMMU performance. In *In Proceedings of the Linux Symposium*, 2007.
- [52] Sagheer Ahmad, Vamsi Boppana, Ilya Ganusov, Vinod Kathail, Vidya Rajagopalan, and Ralph Wittig. A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform. *IEEE micro*, March/April:48–62, 2016.
- [53] Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Specification (Revision 2.00), 2011.
- [54] Intel. Virtualization Technology for Directed I/O: Architecture Specification, 2014.
- [55] Muli Ben-yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, and Elsie Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *In Proceedings of the Linux Symposium*, 2006.
- [56] Ho-Cheung Ng, Yuk-Ming Choi, and H.K.-H. So. Direct virtual memory access from FPGA for high-productivity heterogeneous computing. In *2013 International Conference on Field-Programmable Technology*, pages 458–461, Dec 2013.
- [57] Klaus Danne. Memory Management to Support Multitasking on FPGA Based Systems. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*, page 21, 2004.

- [58] G. Kornaros, K. Harteros, I. Christoforakis, and M. Astrinaki. I/O virtualization utilizing an efficient hardware system-level Memory Management Unit. In *2014 International Symposium on System-on-Chip (SoC)*, pages 1–4, Oct 2014.
- [59] Siddharth Choudhuri and Tony Givargis. Software Virtual Memory Management for MMU-less Embedded Systems. Technical report, 2005.
- [60] Lange, H. and Koch, A. An Execution Model for Hardware/Software Compilation and its System-Level Realization. In *International Conference on Field Programmable Logic and Applications, 2007.*, pages 285–292, Aug 2007.
- [61] Lange, H. and Koch, A. Low-Latency High-Bandwidth HW/SW Communication in a Virtual Memory Environment. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 281–286, Sept 2008.
- [62] C. Meenderinck, A. Molnos, and K. Goossens. Composable Virtual Memory for an Embedded SoC. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 766–773, Sept 2012.
- [63] Altera Inc. Altera SDK for OpenCL. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf, 2016.
- [64] K. Jasrotia and Jianwen Zhu. Hardware Implementation Of A Memory Allocator. In *Digital System Design, 2002. Proceedings. Euromicro Symposium on*, 2002.
- [65] Kenneth C. Knowlton. A Fast Storage Allocator. *Communication ACM*, 8(10), October 1965.
- [66] H. Cam, M. Abd-El-Barr, and S.M. Sait. A High-Performance Hardware-Efficient Memory Allocation Technique And Design. In *Computer Design, 1999. (ICCD '99) International Conference on*, pages 274–276, 1999.
- [67] Xilinx Inc. SDAccel Development Environment User Guide: Features and Development Flows. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug1023-sdaccel-user-guide.pdf, 2016.

- [68] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java, 2nd Edition*. Wiley, 2000.
- [69] Xilinx Inc. LogiCORE IP Mailbox (v1.01b). http://www.xilinx.com/support/documentation/ip_documentation/mailbox/v1_01_b/pg088-mailbox.pdf, 2012.
- [70] Xiaowen Chen, Shuming Chen, Zhonghai Lu, A. Jantsch, Bangjian Xu, and Heng Luo. Multi-FPGA Implementation Of A Network-on-Chip Based Many-core Architecture With Fast Barrier Synchronization Mechanism. In *NORCHIP, 2010*, pages 1–4, Nov 2010.
- [71] M. Saldana and P. Chow. TMD-MPI: An MPI Implementation For Multiple Processors Across Multiple FPGAs. In *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug 2006.
- [72] Antonino Tumeo, Christian Pilato, Gianluca Palermo, Fabrizio Ferrandi, and Donatella Sciuto. HW/SW Methodologies for Synchronization in FPGA Multiprocessors. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '09*, 2009.
- [73] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1), February 1991.
- [74] Herman J. Blinchikoff and Anatol I. Zverev. *Filtering in the time and frequency domains*. 1976.
- [75] AMD Developer Tools Team. CodeXL Quick Start Guide: Version 2.0 Revision 1. https://github.com/GPUOpen-Tools/CodeXL/releases/download/v2.0/CodeXL_Quick_Start_Guide.pdf, 2016.
- [76] Xilinx Inc. Embedded System Tools Reference Manual (EDK). http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/est_rm.pdf, 2012.
- [77] Xilinx Inc. SDSoc Environment: User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug1028-intro-to-sdsoc.pdf, 2016.

- [78] Xilinx Inc. Vivado Design Suite User Guide: Embedded Processor Hardware Design. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_1/ug898-vivado-embedded-design.pdf, 2016.