Abstracting OpenCL for Multi-Application Workloads on CPU-FPGA Clusters

by

Graham Mark Holland

B.A.Sc. (Hons.), Simon Fraser University, 2014

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Applied Science

in the
School of Engineering Science
Faculty of Applied Sciences

Approval

Name:	**Graham Mark Holland**
Degree:	**Master of Applied Science (Engineering)**
Title:	**Abstracting OpenCL for Multi-Application Workloads on CPU-FPGA Clusters**
Examining Committee:	**Chair:** Anita Tino Lecturer
	Lesley Shannon Senior Supervisor Professor
	Zhenman Fang Supervisor Assistant Professor
	Arrvindh Shriraman Internal Examiner Associate Professor School of Computing Science
Date Defended:	**December 3, 2019**

Abstract

Field-programmable gate arrays (FPGAs) continue to see integration in data centres, where customized hardware accelerators provide improved performance for cloud workloads. However, existing programming models for such environments typically require a manual assignment of application tasks between CPUs and FPGA-based accelerators. Furthermore, coordinating the execution of tasks from multiple applications necessitates the use of a higher-level cluster management system. In this thesis, we present an abstraction model named CFUSE (Cluster Front-end USEr framework), which abstracts the execution target within a heterogeneous cluster. CFUSE allows tasks from multiple applications from unknown workloads to be mapped dynamically to the available CPU and FPGA resources and allows accelerator sharing among applications. We demonstrate CFUSE with an OpenCL-based prototype implementation for a small cluster of Xilinx FPGA development boards. Using this cluster, we execute a variety of multi-application workloads to evaluate three scheduling policies and to determine the relevant scheduling factors for the system.

Keywords: Field-programmable gate arrays; runtime task scheduling; OpenCL; hardware acceleration; FPGA clusters

Acknowledgements

Completing this thesis would not have been possible without the help and support of many people. Firstly, I would like to thank my graduate supervisor, Dr. Lesley Shannon, for the wealth of advice and guidance she has given me and for always having faith in my abilities. Thanks to my family and friends for their encouragement and to the members of the Reconfigurable Computing Lab, thanks for the thoughtful discussions and good company.

To my wife Jennifer, thank you for always believing in me. You have kept me focused on my goals and have been the person I could turn to whenever I needed help. I would not have been able to finish this without you.

Table of Contents

List of Tables

List of Figures

List of Listings

List of Acronyms

ACP	Accelerator Coherency Port
API	application programming interface
APU	Application Processor Unit
AWS	Amazon Web Services
AXI	Advanced eXtensible Interface
BRAM	block RAM
CAD	computer-aided design
CFUSE	Cluster Front-end USEr framework
CMA	Contiguous Memory Allocator
CPU	central processing unit
DAG	directed acyclic graph
DMA	direct memory access
DSP	digital signal processing
EC2	Elastic Compute Cloud
FF	flip-flop
FPGA	field-programmable gate array
FPU	floating-point unit
FUSE	Front-end USEr framework
GP	General Purpose

GPU	graphics processing unit
HDL	hardware description language
HLS	high-level synthesis
HP	High Performance
IOMMU	I/O memory management unit
IP	Intellectual Property
JIT	just-in-time
JNI	Java Native Interface
JVM	Java Virtual Machine
LKM	loadable kernel module
LUT	lookup table
MPI	Message Passing Interface
MPSoC	multiprocessor system-on-chip
OS	operating system
PaaS	Platform as a Service
PCIe	PCI Express
PL	Programmable Logic
POCL	Portable Computing Language
POSIX	Portable Operating System Interface
PS	Processing System
RTL	register-transfer level
SCU	Snoop Control Unit
SIMD	single instruction multiple data
SoC	system-on-chip

Glossary

behavioural synthesis The process within the FPGA CAD tool flow in which a circuit description at the register-transfer level is transformed into a netlist of gates.

bitstream Configuration data that is loaded onto an FPGA allowing the FPGA to implement custom digital logic.

compute unit OpenCL *devices* are logically divided into *compute units*, which contain *processing elements* and local memory.

data-parallel programming model An OpenCL programming model involving *kernel* instructions that operate on a memory object for points in an *NDRange* index space.

device A compute resource capable of executing a *kernel* function using an appropriate binary. Examples include CPU cores, GPUs and FPGA fabric.

high-level synthesis The process in which a circuit description at the register-transfer level is generated from an algorithm description in a high-level programming language such as C, C++ or OpenCL C.

kernel A software-defined function marked for acceleration on an OpenCL *device*. Unrelated to an *operating system kernel*.

NDRange A one-, two- or three-dimensional index space that defines a grouping of *work-items* and that must be specified when a OpenCL *kernel* is submitted for execution.

platform An OpenCL *platform* consists of a *host processor* and one or more *devices*.

processing element A *compute unit* within an OpenCL *device* is logically divided into *processing elements*, which execute *work-items*.

task-parallel programming model An OpenCL programming model involving single *work-item kernels* executing concurrently without the need for an *NDRange* index space.

work-group A collection of related *work-items*, executed by a *compute unit*. *Work-items* within a group share local memory.

work-item One of a group of parallel *kernel* execution instances.

Chapter 1

Introduction

For many years, data centres have consisted of racks of networked server systems featuring general-purpose processors. However, the end of Dennard scaling in the previous decade has meant that power consumption is now the limiting factor for single-threaded central processing unit (CPU) performance improvements. This limitation is particularly problematic for the data centre, where client applications demand continuous, efficient scaling, and where data centre operators seek to curb growing power costs.

As a result, data centre server systems now feature customized hardware, such as graphics processing units (GPUs) and field-programmable gate arrays (FPGAs), to offload some computation from the CPU. These devices are capable of achieving improved performance over CPUs for certain workload types due to their massively parallel architectures. However, high-end GPUs typically have high power requirements, which can be problematic for deployment in a power-constrained data centre [1].

To address the power concerns with GPUs, cloud providers have begun to deploy reconfigurable hardware, such as Amazon making FPGAs available on their Amazon Web Services (AWS) Elastic Compute Cloud (EC2) F1 instances. [2]. These deployments have been driven by research showing that reconfigurable hardware has the potential to significantly improve the throughput of cloud applications [3]. However, the integration of such reconfigurable hardware into cloud environments brings with it a large and diverse set of challenges. Among these are issues related to the following topics:

programmability Addressing the lack of portability of FPGA configurations (bitstreams) between different devices, and between different partial reconfiguration regions within a device.

synthesis Providing abstractions to allow software-based definitions of hardware accelerators and associated hardware system infrastructure. Addressing the long compile times for FPGA platforms.

hardware-software integration Managing hardware accelerator use from software applications.

resource management Managing the availability of all execution resources (e.g. CPUs, hardware accelerators) and managing device reconfiguration to enable different sets of hardware accelerators.

scheduling Enabling efficient use of the compute resources available in the cluster by choosing a sequence for task execution via the use of specific scheduling algorithms.

security Ensuring that client applications are sufficiently isolated from one another when they share resources during execution.

reliability Ensuring robustness in the presence of errors, including hardware failures on compute nodes as well as network errors.

In this thesis, we address a subset of the hardware-software integration, resource management and scheduling challenges. We develop an abstraction model for execution targets within a network cluster. Our abstraction allows tasks from multiple applications to be dynamically scheduled for execution on either a CPU and FPGA accelerator, without requiring an explicit mapping from the programmer. We provide a demonstration of our abstraction by leveraging an existing high-level synthesis (HLS) tool to generate FPGA hardware accelerators from software descriptions. These accelerators are integrated into a cluster with compute nodes featuring both CPUs and FPGA hardware.

1.1 Motivation

From a programming perspective, FPGAs have historically been utilized by describing computations at the register-transfer level (RTL) with hardware description languages (HDLs). Subsequently, this has generally limited their use to hardware engineers with a background in digital systems design. Advances in high-level synthesis (HLS) tools have helped to raise the level of FPGA design entry toward more traditional software programming languages including C and C++. Researchers and industry continue to work to close the performance gap between HLS-generated designs and hand-optimized RTL. As a result, the hardware-specific knowledge required to create performant systems with reconfigurable hardware is decreasing, making FPGA-based systems more accessible to non-experts.

At the same time, OpenCL has emerged as a popular and effective framework for unifying the programming of heterogeneous systems, including those featuring FPGAs. Using the functionality provided by the OpenCL host application programming interface (API), programmers schedule software-defined kernel functions for accelerated execution onto the devices available in a given platform. In the case of OpenCL implementations targeting FPGAs, HLS tools are used to generate custom hardware accelerators from OpenCL kernel source code.

A large amount of prior research work has sought to overcome the challenge of effectively using FPGA accelerators from within application programs. Among these is the Front-end

USEr framework (FUSE) [4]. FUSE presents a Platform as a Service (PaaS) model for hardware accelerators, by providing a programming interface analogous to software threads. Using FUSE, a developer models application compute tasks as threads, without specifying whether these tasks will be executed by a CPU core or an FPGA-based hardware accelerator. Instead, the tasks are allocated to an appropriate target by the FUSE runtime, which tracks hardware accelerator availability and uses execution via software as a fallback mechanism. In our work, we seek to adapt this central concept of FUSE for use within a cluster.

1.2 Objective

The primary objective of this work is to create a framework allowing for the execution of kernel functions from multiple applications across CPUs and FPGA-based hardware accelerators within a networked cluster. We wish to provide an abstraction of the available devices in the cluster, by automating the selection of an appropriate execution target for each application kernel. To meet this requirement, we will base the work on the underlying concept of the FUSE project.

Beyond extending the concept of FUSE to a CPU-FPGA cluster, we will also address some of the limitations of the original FUSE implementation. These include supporting multi-core processor platforms, adding direct memory access (DMA) capability for accelerators, and automating accelerator generation via HLS tools.

While the original FUSE implementation used Portable Operating System Interface (POSIX) threads, our framework demonstration will be based on OpenCL in order to leverage existing HLS capability. This also allows our framework to maintain a level of compatibility with the familiar OpenCL execution model, in which kernels are submitted to device command queues for execution. Furthermore, this allows existing OpenCL applications to be modified to use the new framework with less developer effort.

While OpenCL implementations targeting FPGAs, such as Xilinx SDAccel [5] and Intel FPGA SDK for OpenCL [6], have enabled a more accessible, software-driven approach for defining hardware accelerators and scheduling accelerator tasks, there are limitations inherent to the OpenCL models. In the course of implementing our framework, we seek to address a subset of the following limitations:

- Typical OpenCL implementations for FPGA platforms do not use the system processor for executing kernels. Instead, all kernels are executed by FPGA hardware accelerators, while the system processor executes the OpenCL runtime library.
- While OpenCL defines a portable kernel programming language, kernel performance between different device targets can vary substantially. To achieve sufficient performance, kernel code must often exploit device-specific features.

- The runtime library implementation of the OpenCL host API limits the view of system resources, making simultaneous use of devices by multiple applications difficult.
- The OpenCL platform model is restricted to a single network node. There is no method available to utilize devices on remote nodes.
- OpenCL is a low-level framework. The application programmer is responsible for managing device work queues and moving kernel data between the host processor and devices.

1.3 Contributions

The contributions of this thesis can be summarized as follows:

- The Cluster Front-end USEr framework (CFUSE): an abstraction model allowing for the dynamic mapping of tasks from multiple applications to CPU and FPGA hardware accelerators within a network cluster of heterogeneous nodes.
- A demonstration of CFUSE using an OpenCL-based framework prototype.
- An initial analysis of the scheduling performance within the framework using sets of benchmark application workloads.

The CFUSE framework presented in this thesis, raises the abstraction level over the OpenCL execution model by applying the FUSE concept to a cluster of CPU-FPGA nodes. It leverages the Xilinx Vivado HLS tool [7] to create hardware accelerators from kernel source code and provides a simplified mechanism for constructing heterogeneous CPU-FPGA systems with these accelerators. CFUSE utilizes the Portable Computing Language (POCL) [8] OpenCL implementation to enable application kernels to execute on CPU targets in addition to FPGA accelerators.

We present a prototype implementation of CFUSE on a small network cluster of Xilinx Zynq FPGA development boards and perform a series of characterization experiments to investigate the framework overhead. Finally, we perform an analysis of the scheduling capability of the framework to uncover the factors relevant to determining an execution schedule for kernels from a mix of benchmark applications.

1.4 Thesis Organization

This thesis is organized in the following manner. Chapter 2 presents relevant background information on FPGAs, the OpenCL framework and its models, and presents a summary of related research including the prior work of FUSE. Chapter 3 provides a high-level overview of the CFUSE framework, while the implementation details of the prototype are covered in Chapter 4. Chapter 5 describes measurement techniques and presents a characterization of the prototype CFUSE implementation. We present the results of our investigations on scheduler policies for benchmark application workloads in Chapter 6. Finally, Chapter 7 summarizes the work, outlines areas for future research effort, and lists potential system improvements to CFUSE.

Chapter 2

Background

This chapter provides relevant background information about the technologies, hardware devices, software tools and frameworks used in this thesis. We begin with a discussion of field-programmable gate array (FPGA) architecture, synthesis and programming and then move on to discuss the OpenCL framework and OpenCL platforms supporting FPGAs. We provide an overview of two previous works that we make use of in this thesis: the Portable Computing Language (POCL) OpenCL implementation and the Front-end USEr framework (FUSE). Finally, we present an overview of related research.

2.1 Field-Programmable Gate Arrays

FPGAs are hardware reconfigurable integrated circuits, capable of implementing custom digital logic designs. In the following sections, we discuss their internal architecture and the computer-aided design (CAD) tool flow used to create a custom configuration from a circuit design specification. After this, we describe the device configuration process.

2.1.1 Architecture

Figure 2.1 shows a simplified model of a modern FPGA architecture. As shown, FPGAs are made up of a programmable fabric, consisting of an array of *logic blocks* that implement circuit functions, connected by a *programmable interconnect. I/O blocks* are used to interface with the physical device pins and are capable of implementing various digital I/O standards and voltages.

Logic blocks differ between FPGA vendor and device family, but at a minimum contain a lookup table (LUT), flip-flop (FF) and multiplexer. LUTs may be programmed with arbitrary Boolean functions to implement combinational circuits. With the addition of flip-flops to the logic block, sequential circuits such as registers and distributed memories can also be implemented. Modern FPGA architectures feature significantly more complicated logic blocks than what is shown here. They may include structures allowing for the implementation of shift registers and distributed RAM, as well as a carry chain used to efficiently

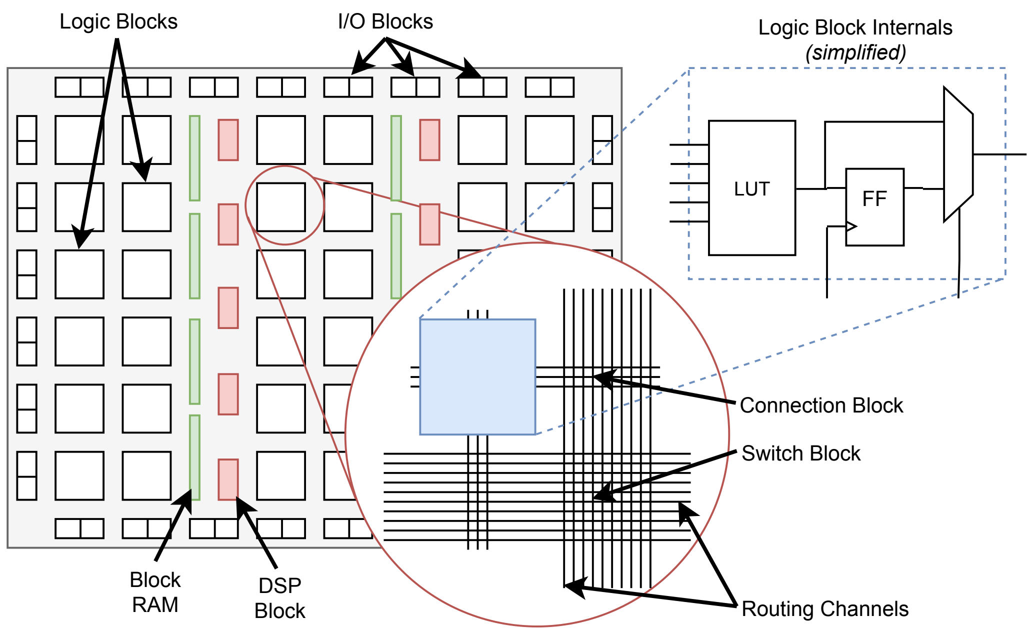


Figure 2.1: Simplified FPGA architecture

implement arithmetic functions. Furthermore, modern architectures use fracturable LUTs that can either implement a single k-input LUT or two $(k-1)$-input LUTs with shared inputs.

The interconnect consists of routing tracks organized into horizontal and vertical *channels*, used for routing signals between logic blocks. Logic blocks connect to this interconnect at a *connection block* and connections are made between intersecting channels at *switch blocks*. Connection and switch blocks contain programmable switches, allowing for arbitrary connections to be made by loading an FPGA configuration, known as a bitstream.

In addition to these configurable blocks, the FPGA fabric also includes various fixed logic blocks, used to implement functionality common to many classes of digital circuits. These include embedded memories known as block RAMs (BRAMs), digital signal processing (DSP) blocks for performing multiply-accumulate operations, analog-to-digital converters, high speed I/O blocks, such as PCI Express (PCIe) and DDR memory interfaces, and dedicated clock circuitry. Some FPGAs also contain embedded central processing unit (CPU) cores with a set of interfaces enabling communication to the remainder of the programmable fabric.

2.1.2 Synthesis, Programming and Reconfiguration

Due to the size and complexity of the circuits that may be implemented on a modern FPGA, designers typically use hardware description languages (HDLs), such as VHDL and Verilog, to specify their designs. These languages describe circuits at the register-transfer level (RTL). Given a set of HDL design files, a CAD tool is used to synthesize a bitstream. This

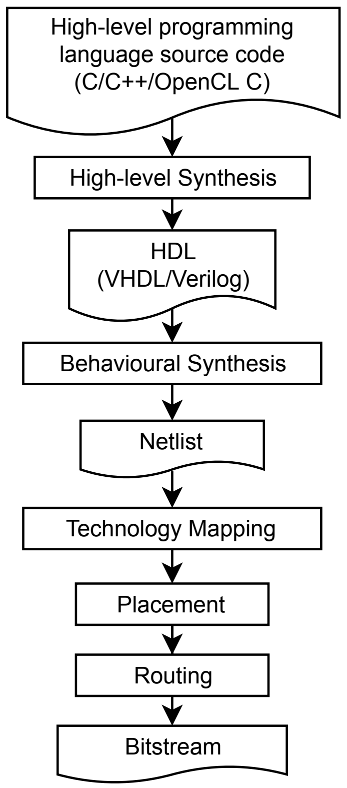


Figure 2.2: FPGA CAD tool flow

process involves the following steps, outlined in Figure 2.2: *behavioural synthesis*, *technology mapping*, *placement* and *routing*. In behavioural synthesis, RTL sources are transformed into a *netlist*, a generic circuit representation consisting of logic gates and connections. Device independent circuit optimizations may then be performed using this representation. These gates are then packed into the primitives (LUTs, flip-flops, BRAMs, DSP blocks etc.) available on the target FPGA device during technology mapping. During placement, specific locations are chosen for each element and these elements are wired together during the routing stage.

High-level synthesis (HLS) is the process of generating RTL code for hardware synthesis from a higher level programming language such as C, C++ or OpenCL C. Increasingly, FPGA-based design is performed using HLS tools. Such tools are useful to those without extensive knowledge of digital circuit design and RTL-based design entry. Advances in HLS optimization have brought the resulting circuit performance closer to what would be generated from hand-optimized RTL.

Once a bitstream is generated using HLS or from RTL sources, it must be programmed onto the FPGA for the device to implement the circuits in the design. This is typically done when the device is initially powered on, but may also be performed during runtime. Some FPGAs support dynamic partial reconfiguration, whereby a subset of the FPGA fabric may be reconfigured, while the circuits in the remaining regions of the fabric continue to operate. To achieve this, the designer must use floorplanning CAD software to divide the device into partially reconfigurable and static regions. Then, multiple partial bitstreams are generated for the partially reconfigurable region as well as a bitstream for the static part of the design. These partial bitstreams are generally not portable between different partial reconfiguration regions.

2.2 OpenCL

OpenCL (Open Computing Language) [9] has begun to see widespread use for developing programs targeting heterogeneous systems, including those featuring graphics processing units (GPUs) and FPGAs. The OpenCL standard defines an application programming interface (API), implemented via a runtime library, and a portable, parallel programming language called OpenCL C. An OpenCL *platform* consists of a *host processor* that executes the runtime library and application code, and one or more computation *devices*. Devices execute specially qualified functions called *kernels* that are written in the OpenCL C language. Examples of such devices include multi-core CPUs, discrete GPUs, FPGAs and DSP chips.

2.2.1 OpenCL Models

The OpenCL specification divides the OpenCL architecture into a hierarchy of four models:

- *Platform model* – Divides the system into a host processor and devices.
- *Execution model* – Defines the use of device command queues to control kernel execution.
- *Memory model* – Defines memory regions and memory access rules for host applications and kernels.
- *Programming model* – Defines the data-parallel and task-parallel models.

The platform model divides a system into a *host processor* and one or more compute *devices*. Devices are further divided into *compute units* that are themselves made up of *processing elements*. It is important to note that these divisions are logical abstractions and how a given device is subdivided into these components is defined by the specific implementation of OpenCL on a system. It is valid for a device to consist of a single compute unit with a single processing element.

The execution model logically divides application program execution. Kernels execute on devices, while the OpenCL API implementation and remaining application code execute on the host processor. All device interactions, including memory transfers between the

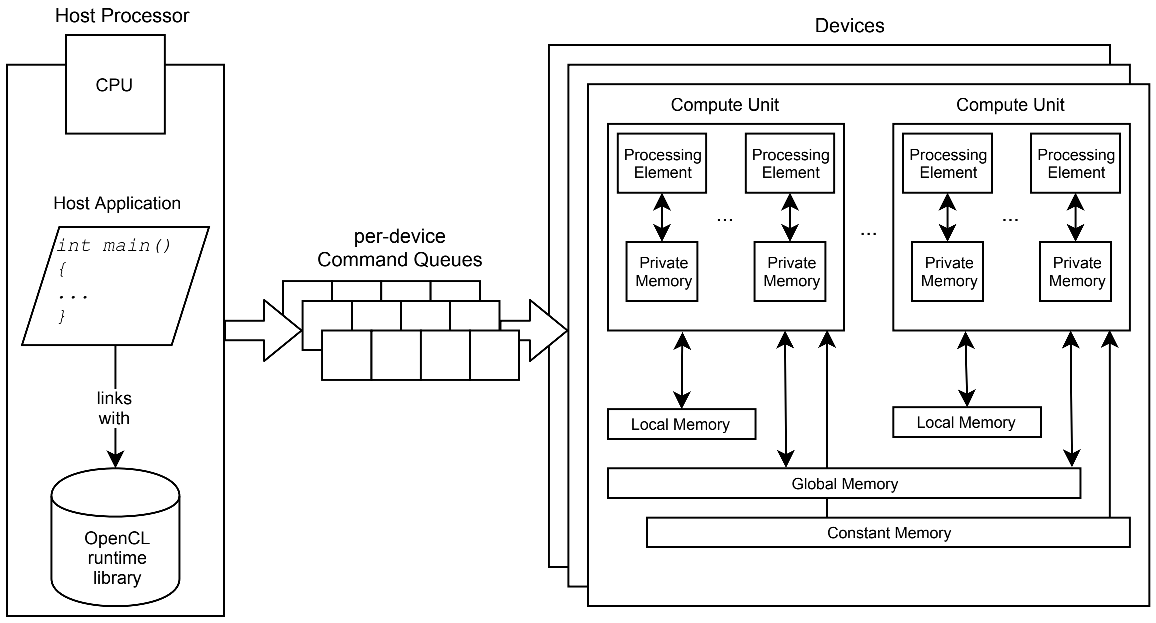


Figure 2.3: OpenCL platform and memory models

host processor and device as well as kernel executions, occur by submitting commands to a *command queue* data structure. A command queue is mapped to a specific device. Kernel invocations are defined over an index space called an *NDRange*, with a single kernel invocation being executed for each point in this index space. These kernel instances are called *work-items*, which are organized into *work-groups*. Each processing element executes a work-item, while each compute unit executes a work-group.

The memory model divides system memory into regions. *Global* and *constant* memories are accessible to both the host and device, while *local* and *private* memories are accessible only to work-groups and work-items respectively.

Finally, OpenCL distinguishes between two possible programming models: data-parallel and task-parallel. The data-parallel programming model involves kernel instructions operating concurrently on the elements of a memory object for points in an index space, with one point for each work-item. This model represents a single instruction multiple data (SIMD) approach to parallelism. Alternately, the task-parallel programming model uses single work-item kernels that execute without the need for an index space. In the task-parallel case, parallelism can be achieved by using vector data types or by enqueuing multiple kernels for concurrent execution. If supported by the specific OpenCL implementation, OpenCL extensions may be used via pragmas in kernel source code for device-specific optimizations such as loop unrolling and pipelining.

An OpenCL implementation is provided by a vendor and consists of both an OpenCL runtime library and a kernel compiler. The runtime library is an implementation of each function from the OpenCL C host API. This is provided as a shared object library. The kernel compiler is used to generate device-specific kernel binaries from OpenCL C source code.

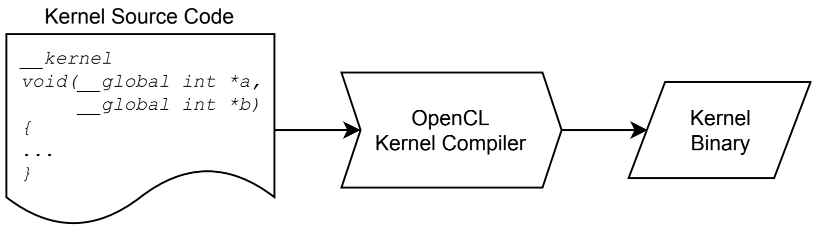


Figure 2.4: OpenCL kernel compilation. Kernel source code is compiled to a device-specific binary format either offline or during application runtime via the OpenCL host API.

2.2.2 Typical OpenCL Host Application Flow

A typical OpenCL application will perform a number of steps to define kernel tasks, setup memory buffers, perform data transfers between the host processor and a device and finally launch kernels to be executed on a device. Host applications primarily interact with OpenCL devices via command queue objects. The application submits commands to device-specific queues to manage data within host and device memory and to launch kernels. The act of submitting a command generates an event object, which may be used to synchronize subsequent commands via an event wait list. Using events in this manner allows the programmer to specify dependencies between commands. Figure 2.5 outlines the primary actions taken by an OpenCL host application during its execution. These steps are as follows:

1. *Query the system for OpenCL platforms.* The OpenCL standard allows multiple platforms (OpenCL vendor implementations) to co-exist on a system. The host API provides functions to query the system for the available platforms and discover their capabilities.

2. *Query the selected platform for available devices.* Once a platform has been selected, the application will query the platform for a list of available compute devices. The application may also check device properties.

3. *Setup a context for the selected device(s).* An OpenCL context includes one or more devices and encapsulates other OpenCL objects: command queues, memory buffers, programs and kernels. These objects cannot be shared between contexts.

4. *Create a command queue.* In order to launch kernels on a device, a command queue must be created. Command queue objects are specific to a single device and context.

5. *Setup memory objects.* OpenCL buffers are specified as one-dimensional collections of elements and are used to specify input and output arguments to kernels. Buffer objects are specific to a context.

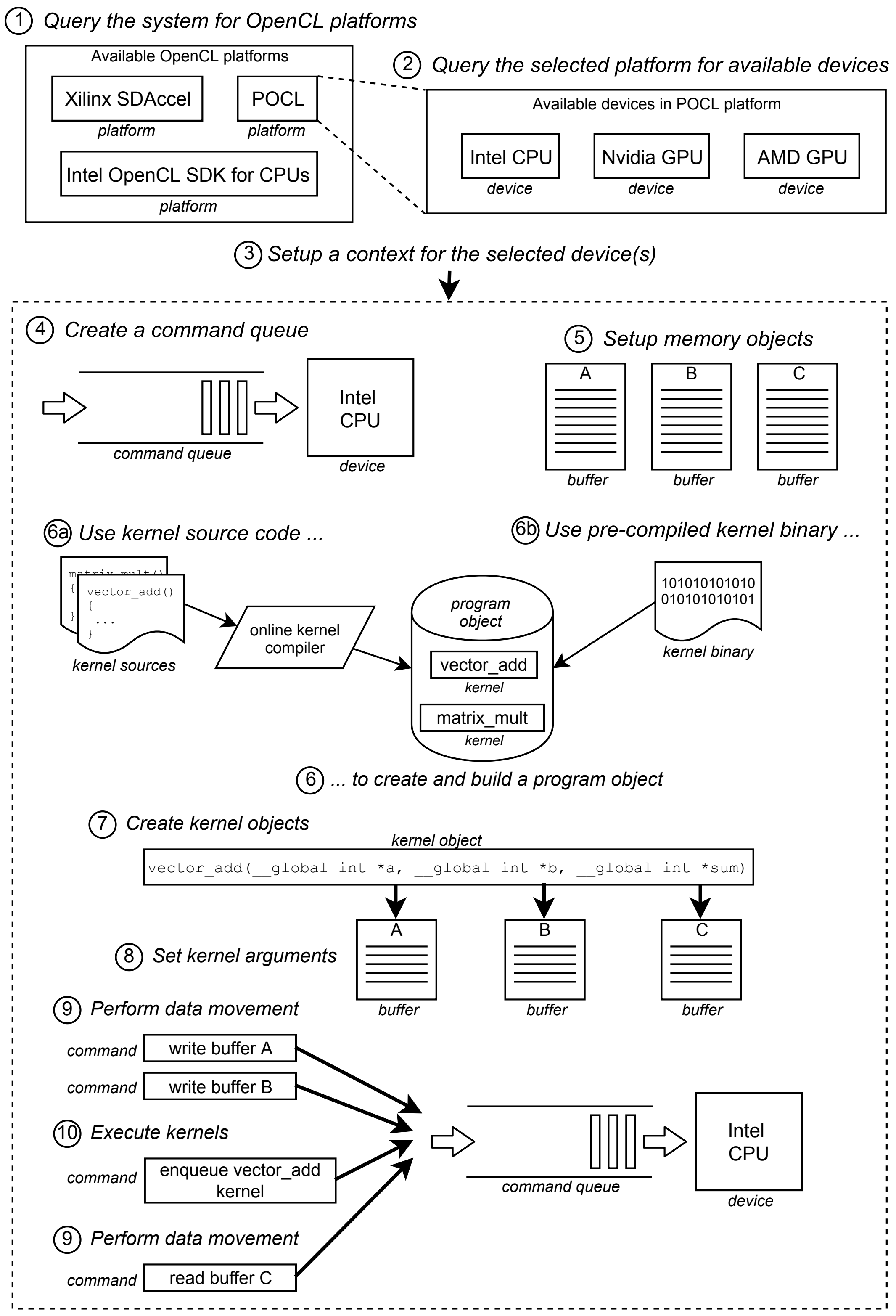


Figure 2.5: Typical OpenCL host application program flow

6. *Create and build program objects.* An OpenCL program object consists of a set of kernels, which are functions written in OpenCL C source code and identified with the `__kernel` attribute. Program objects may be created from either (a) OpenCL C source code by using the vendor-supplied kernel compiler at application runtime or (b) from a prebuilt binary image generated prior to execution.

7. *Create kernel objects.* A kernel object encapsulates a specific `__kernel` function that exists within a program object along with the argument values to be used during execution.

8. *Set kernel arguments.* Memory objects and plain data types are passed as arguments to a kernel function using the `clSetKernelArgument` function.

9. *Perform data movement.* Operations on memory buffers to move data between host and device memory are specified by submitting commands to the command queue.

   The OpenCL host API defines a number of command-enqueuing functions for performing different memory operations. As with all command-enqueuing functions, the application can use event objects to specify dependencies between commands.

10. *Execute kernels.* Kernels are executed by calling the `clEnqueueNDRangeKernel` function and specifying a device command queue and a kernel object. This function also requires the specification of an N-dimensional index space (NDRange) on which to operate.

    An NDRange is used to divide a kernel invocation into instances called work-items that are organized into work-groups. The OpenCL execution model views a device as being made up of one or more compute units, each containing one or more processing elements. Each compute unit of a device executes a work-group and each processing element within the compute unit executes a single work-item.

The source code for an example host application and corresponding kernel function is provided in Appendix A for reference.

2.2.3 OpenCL FPGA Implementations

FPGAs are fundamentally different from the more typical CPU and GPU OpenCL device types since their underlying architecture allows for the realization of custom hardware to implement kernel functions. Since kernel compilation for FPGAs involves the synthesis of kernel cores and bitstream generation, compilation times are orders of magnitude longer than for a CPU or GPU. Therefore, offline compilation must be used to create kernel binaries rather than performing compilation during application runtime. Furthermore, the OpenCL data-parallel programming model, while well suited for massively parallel devices such as GPUs, does not necessarily map well to FPGA hardware [10]. Instead, performance gains on FPGA hardware are typically achieved through HLS-specific optimization techniques such as array partitioning, loop unrolling and pipelining [11] [12]. For this reason, kernels are usually written to use only a single work-item, therefore following the task-parallel programming model [13].

2.3 Amdahl's Law

Amdahl's Law [14] is commonly used to estimate the potential speedup for parallel programs. However, it can also be used to determine the theoretical speedup for serial programs, where some portion of the program can be optimized or accelerated. With this formulation, the equation for program speedup is given in Equation 2.1.

$$Speedup = \frac{1}{(1-p)+\frac{p}{s}} \tag{2.1}$$

The theoretical speedup of the entire program is a function of p, the proportion of the original execution time that benefits from acceleration and s, the speedup of the accelerated portion.

2.4 Related Work

In this section, we discuss background and related work from academia, industry and the open source software community.

2.4.1 POCL

Portable Computing Language (POCL) [8] is an open source implementation of the OpenCL standard, providing a portable OpenCL runtime library implementation and a kernel compiler. The POCL kernel compiler utilizes Clang [15] as a front end for OpenCL C code and LLVM [16] for optimization and code generation to binary formats for different device targets. One of the goals of the POCL project is to make it easy to add support for new devices. This is made possible by having their implementation of the OpenCL runtime library make calls to an intermediary device support API. POCL provides an implementation of this device API for a generic pthread CPU target that maps kernel work-items to Portable Operating System Interface (POSIX) threads. Furthermore, the LLVM-based compiler back end for the pthread device supports both online and offline use, as POCL defines a custom binary format for CPU kernels containing object code and kernel metadata. This allows POCL to be used on embedded platforms, which may not be able to host the complete kernel compiler.

2.4.2 FUSE

This work is based on the Front-end USEr framework (FUSE) architecture [4]. FUSE is a framework for abstracting the use of FPGA hardware accelerators. The concept of the FUSE architecture is to allow computation tasks to be executed either as software on a CPU or using an FPGA hardware accelerator, without an explicit choice being specified in the application source code. The decision on where to execute a computation task is made during application runtime by the FUSE scheduling policy and carried out by the mechanisms of the FUSE library. This architecture abstracts the use of hardware accelerators from the application programmer.

The implementation of the FUSE system uses the POSIX thread (pthread) library as the underlying model for describing computations. The programmer defines pthread functions to be used for CPU execution. Hardware accelerators providing the same functionality as their software counterparts are also required, along with a driver implemented as a Linux kernel module. A kernel table that maps thread functions to equivalent hardware accelerators is used by the FUSE scheduler when deciding where to execute a computation task.

A task is scheduled for execution by calling the FUSE library `thread_create` function. An overview of the implementation of the `thread_create` function is shown in Figure 2.6. The implementation is as follows. If a matching hardware accelerator is found and is not in use, FUSE loads the Linux kernel module driver for that accelerator (if necessary) and launches the task on the accelerator. Otherwise, the task is scheduled for execution on the CPU by spawning the matching software thread with a call to `pthread_create`.

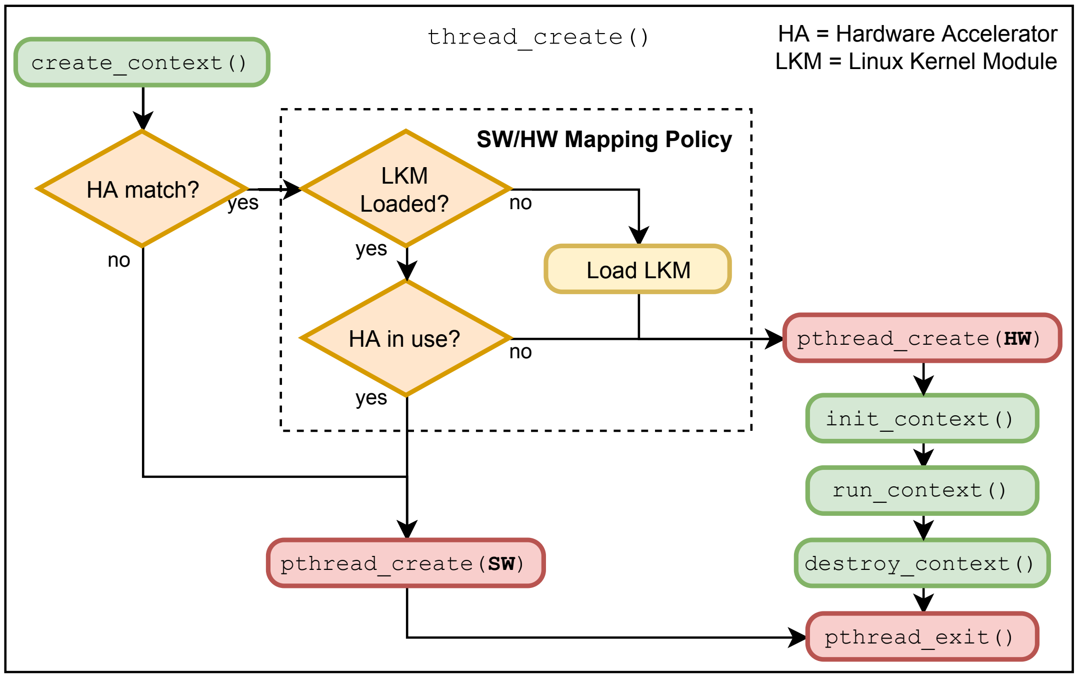


Figure 2.6: FUSE thread creation and scheduling. Adapted from [4].

Some limitations of the FUSE system implementation include the following:

- The target platform is an FPGA development board with a single-core soft processor and accelerators implemented in the FPGA fabric.
- There is no integration with HLS tools. Therefore, there is no means to generate hardware accelerators from software functions.
- Each hardware accelerator requires the development of a custom driver, implemented as a Linux kernel module.
- There is no support for accelerators to use direct memory access (DMA) operations. Data is transferred to accelerators using register accesses via the driver `ioctl` or `mmap` functions.

2.4.3 Task Scheduling in Heterogeneous Systems

Scheduling computation tasks onto the resources available within a heterogeneous system is a widely studied research area. Here we focus our discussion on prior work that utilizes OpenCL and OpenCL-like models.

Taylor et al. [17] develop a method for mapping kernels onto the ARM big.LITTLE mobile platform, a heterogeneous multicore architecture that combines small power efficient cores with larger high performance cores. They use an offline model, trained on OpenCL kernel features, that selects which processor and clock frequency to use for each kernel invocation. Their model performs a trade-off between kernel runtime and power consumption. The approach relies on a just-in-time (JIT) compilation of unseen kernels as they are submitted for execution.

Perina and Bonato [18] extract features from OpenCL code and use a machine learning framework to statically decide between GPU and FPGA accelerators based on estimated energy consumption. They use generic OpenCL kernel benchmarks without FPGA specific optimizations.

MultiCL [19] augments the OpenCL context and command queue objects with scheduling properties. These are used as hints to a runtime system that manages a pool of all device command queues. Their scheduler utilizes device and kernel profiling techniques that are possible only on devices that support online kernel compilation such as GPUs. Further, they restrict the scope of scheduling to kernels within a single application.

A number of related works extend the scope of kernel scheduling to multiple OpenCL applications. Wen et al. [20] use potential speedup as a heuristic for scheduling kernels from multiple applications onto a CPU or GPU. They use machine learning to train a predictor of performance speedup using features extracted from a static kernel code analysis. Their goal is to optimize throughput and application turnaround time. Unlike some other related work, they do not split up kernel tasks, which they define as kernel executions plus associated data transfer.

Conversely, FluidiCL [21] splits kernel executions at the work-group level and enables cooperative kernel execution by CPU and GPU. It manages the extra data transfers to devices automatically and merges the partial results computed by each device. Unlike the work of Wen et al. it does not require an offline training step.

HeteroPDP [22] predicts performance degradation due to memory contention of co-located OpenCL and system applications. The model is based on regressions of performance characterization data and is capable of selecting between CPU and GPU execution within a single platform.

2.4.4 Resource Managers for Clusters and Distributed Systems

A number of previous research projects have looked at enabling OpenCL for use in distributed systems, but only for clusters featuring GPUs. SnuCL [23] makes OpenCL devices on remote nodes available to the local OpenCL context and automatically manages data movement between nodes. However, the programmer must still manually determine the target for each kernel. Similarly, Hybrid OpenCL [24], dOpenCL [25] and Distributed OpenCL [26] all share in common the need to use separate buffers for each device and manually map kernels to devices.

Conversely, DistCL [27] abstracts the locality of the remote GPUs as well, treating all the cluster GPUs as a single device. The DistCL runtime requires that the programmer write kernel meta-functions that specify the memory access patterns of each kernel. These meta-functions are used to determine how kernel executions should be split and to assign work-items to devices while minimizing remote data transfers.

CloudCL [28] utilizes dOpenCL, and therefore also requires manual splitting of kernels, but adds a two-tier scheduler to maintain fairness among multiple executing kernels. The first tier schedules kernels without knowledge of the cluster or the kernel splitting and the second schedules to specific GPU devices using a policy based on recorded kernel performance history.

These distributed OpenCL works (SnuCL, Hybrid OpenCL, dOpenCL, Distributed OpenCL, DistCL and CloudCL) solve a related but distinctly different problem to ours. All of these works make remote devices available to a single OpenCL host application and split kernel tasks among the available devices. We do not split kernel tasks among devices, as we schedule kernels from multiple applications that execute concurrently. Furthermore, we provide an abstraction of the execution target type, by automatically selecting between a CPU or FPGA hardware accelerator at runtime.

The libwater [29] project also aims to ease development for heterogeneous clusters featuring GPUs. It provides a simplified library wrapper over the OpenCL host API and implements a device query language for selecting targets among the many potentially available within a cluster. Notably, the libwater runtime builds a directed acyclic graph (DAG) of commands as they are submitted, based on the dependencies between commands that are specified using event objects. In their initial work, the DAG is used to recognize individual point-to-point communications between nodes, so that they may be replaced with more efficient collective operations. Similarly to our work, kernels are not split over multiple devices. However, our framework abstracts execution targets for kernels as the runtime provides automatic device selection. With libwater, selecting an execution target for each kernel must be performed manually using the provided device query language.

In addition to the distributed OpenCL works, are a variety of commercial and open source cluster resource managers including TORQUE [30] and SLURM [31]. These resource managers are designed to enable batch processing of applications on large clusters. SLURM allocates cluster resources to user applications, provides a means for starting and monitoring work on these compute resources and also manages contention for shared resources using a work queue. The fundamental resource that can be allocated by SLURM is a compute node. Our work takes a more fine-grained approach to application task scheduling. Application kernels are individually scheduled as they are submitted for execution, instead of allocating cluster resources to an application for fixed time periods.

2.4.5 FPGAs in the Data Centre

As mentioned in the introduction, the integration of FPGA hardware into data centre environments is an area of active research by both academia and industry. Here we outline a subset of previous work addressing the challenges of using FPGAs at data centre scale.

One of the first major deployments of reconfigurable hardware into a data centre was by Microsoft with the Catapult project [1]. The architecture consists of custom FPGA daughtercards attached to server systems with PCIe. Notably, these FPGA cards are directly connected to each other via a secondary 10 Gbit network, configured in a two-dimensional torus. The authors describe a thread-safe low latency communication interface between the CPU and local FPGA. Their method avoids system call overhead for data transfers by mapping user pages and supports concurrent buffer accesses by dividing buffers into slots and statically assigning slots to threads. We employ a similar memory mapping scheme in our DMA buffer implementation. Furthermore, we note the use of a Gigabit Ethernet network in our work is a limitation we seek to improve in future work.

The follow-up work to Catapult, the Configurable Cloud Architecture [3], removes the need for a secondary network by placing FPGA fabric between network switches and servers. This tightly couples accelerator hardware to the data centre network allowing for the acceleration of network functions such as encryption. The Configurable Cloud uses a global resource manager that tracks FPGA resources which is similar to our approach. However, we also track CPU availability, as our framework schedules kernel tasks to both CPU and FPGA devices.

Several academic works have investigated methods for integrating FPGAs with big data frameworks including Apache Spark. Since Apache Spark is hosted on the Java Virtual Machine (JVM), access to FPGA accelerators must occur over the Java Native Interface (JNI), which can involve significant overhead.

Chen et al. [32] explore the use of fine-grained accelerators within Apache Spark. These are accelerators where the execution time is short, but where the application requires many iterations. They use a DNA sequencing accelerator that exhibits this fine-grained behaviour and present a method for batching groups of tasks when offloading to the FPGA to minimize the JNI-FPGA data transfer overhead.

Ghasemi and Chow [33] also investigate integrating FPGA accelerators with Apache Spark. Notably, their integration method provides direct access to shared memory to limit the number of JNI-FPGA transfers. This enables the efficient transfer of large buffers, therefore limiting data transfer overheads.

Blaze [34] exemplifies a more general approach for integrating with big data frameworks by exposing FPGA accelerators as a service (FaaS). It provides APIs allowing big data processing applications to leverage FPGA accelerators and is demonstrated with Apache Spark. Blaze also extends the Hadoop YARN job scheduler to use accelerator-centric scheduling, allowing for multiple applications to share the use of accelerators. In the initial proof of concept, kernel tasks are only scheduled to FPGA accelerators, though the authors mention CPU and FPGA co-working as an improvement for future work.

Tarafdar et al. [35] present a cluster provisioning tool that provides client users a simple means to map streaming kernel functions to hardware accelerators spread across multiple FPGAs. Users of the framework describe a logical configuration and the communication between kernels in an application. Using this information, the framework allocates FPGA resources from within the cluster.

Eskandari et al. [36] introduce Galapagos, a modular hardware deployment stack for deploying FPGA accelerators in a heterogeneous data centre. This stack consists of communication, network/middleware, hypervisor and physical hardware layers. They provide a communication layer implementation called HUMboldt that utilizes HLS, allowing for applications that link against it to be portable between CPUs and FPGAs. We view the work of HUMboldt as complementary to ours, in that it seeks to provide means for enabling portability of applications between CPUs and FPGAs.

2.4.6 OpenCL for FPGAs

Both the major FPGA vendors, Xilinx and Intel, provide OpenCL platform support for their devices via dedicated software tools: Xilinx SDAccel [5] and Intel OpenCL SDK for FPGAs [6], respectively. SDAccel targets PCIe attached FPGA cards on x86 platforms and uses an HLS back-end to generate accelerator cores to implement kernel functions using FPGA resources. A unique feature of SDAccel is that it allows kernel sources to be written in C, C++ and OpenCL C or allows accelerators to be described directly in RTL, bypassing the HLS step [37]. Both FPGA vendor OpenCL implementations add some non-standard extensions, enabled via source code pragmas and new OpenCL host API functions. These enable further optimizations for the generated hardware but limit the portability of kernel sources between vendors.

In addition to the FPGA vendor OpenCL implementations, are several academic works exploring how best to support OpenCL on FPGAs. Both OpenRCL [38] and the work of Ma et al. [39] execute OpenCL kernel functions using soft processor cores on FPGAs, rather than using custom accelerators as in our approach. While the works of both Shagrithaya [40] and Hosseinabady [41] map kernel functions to accelerators generated by HLS, they require an initial source-to-source translation on the OpenCL C code. We avoid the need for source-to-source translations by using a more recent version of Xilinx Vivado HLS that natively supports OpenCL C as a source language. We use a similar integration of the hardware accelerators with the host CPU in our prototype platform to that of Hosseinabady, as they also target a Zynq device. UT-OCL [42] implements OpenCL for an embedded FPGA platform using a MicroBlaze soft processor as the host CPU with hardware accelerators also implemented in FPGA fabric. Communication between the host and the device subsystem is via a streaming interconnect.

Chapter 3

The CFUSE Framework

The main contribution of this thesis is the introduction of the Cluster Front-end USEr framework (CFUSE). CFUSE is a framework that allows kernels from multiple applications to be executed on the hardware resources available within a network cluster. The scheduling of kernels is handled dynamically as each kernel is submitted for execution by individual client applications. In this chapter, we provide a high-level design overview of the CFUSE framework. A more in-depth discussion of the prototype implementation is left for Chapter 4.

3.1 Background

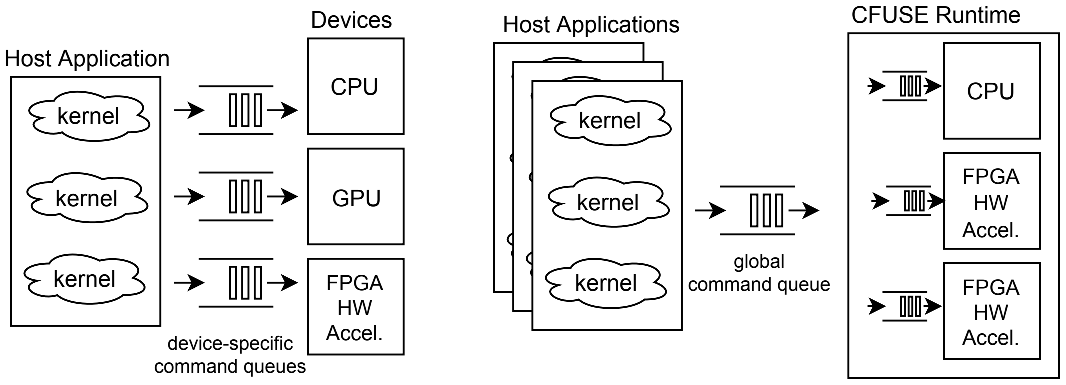
The objective of our work is to develop an abstraction over the possible execution targets within a network cluster. As multiple applications queue work for execution, our abstraction must choose an execution target from the set of possible devices within the cluster. For the purposes of this work, we have chosen to use a cluster with heterogeneous nodes containing a central processing unit (CPU) and field-programmable gate array (FPGA) fabric, used to host custom hardware accelerator cores. While we could also investigate the use of graphics processing units (GPUs) in the cluster, we have restricted our scope to CPU and FPGA devices only.

To realize our abstraction model, which we name CFUSE, we seek to utilize a lower-level programming framework with the ability to specify application tasks and submit them for execution. For this purpose, we have chosen to create CFUSE using the OpenCL framework [9]. OpenCL *kernels* are application functions that are marked for acceleration and may be written in OpenCL C, a portable parallel programming language. OpenCL also defines a runtime application programming interface (API) used to submit kernels for execution on devices. However, there are several OpenCL limitations that we must overcome to make it suitable for use with our abstraction.

The OpenCL specification defines a platform model consisting of a host processor attached to one or more devices. The host processor runs the application code and manages the launching of kernels that are executed exclusively on devices. For a typical FPGA

OpenCL implementation, the host processor is the system CPU, while a device is a region of the FPGA fabric. Modern cluster system CPUs are generally multi-core processors, but since the system processor acts as the OpenCL host, these cores are not available to run application kernels. Furthermore, OpenCL restricts its platform model to a single network node and there is no method available for distributing kernel executions onto other nodes within a cluster. Finally, OpenCL is a low-level framework. The application programmer is responsible for managing device work queues and moving program data between the host processor and devices.

3.2 Execution Model



(a) OpenCL execution model (b) CFUSE execution model

Figure 3.1: OpenCL and CFUSE execution models. OpenCL requires a command queue per device. CFUSE abstracts the use of a specific device using a global command queue.

CFUSE is designed to follow the OpenCL execution model, whereby applications specify kernel functions for accelerated execution on capable hardware devices. As with OpenCL, in the CFUSE framework, kernel functions are executed by submitting commands to a command queue. However, instead of selecting the target device explicitly by submitting to a device-specific queue, the CFUSE model uses a shared queue for all devices. In this way, CFUSE adapts the OpenCL execution model to the concept of the Front-end USEr framework (FUSE) [4], where the runtime system is responsible for choosing a suitable target device. The mapping of kernels onto specific devices is abstracted from the application developer.

CFUSE can be viewed as an extension of the original FUSE work for cluster systems, which has re-targeted the underlying execution model from pthreads to OpenCL. While the initial FUSE implementation managed CPU and FPGA-based hardware accelerators within a single network host for a single application, CFUSE provides multiple applications a means to execute kernels on these device types, both on the local host and remote nodes within a cluster.

3.3 Overview

The runtime of the CFUSE framework consists of the following components:

- proxy
- kernel scheduler
- resource manager
- request handler
- runtime library

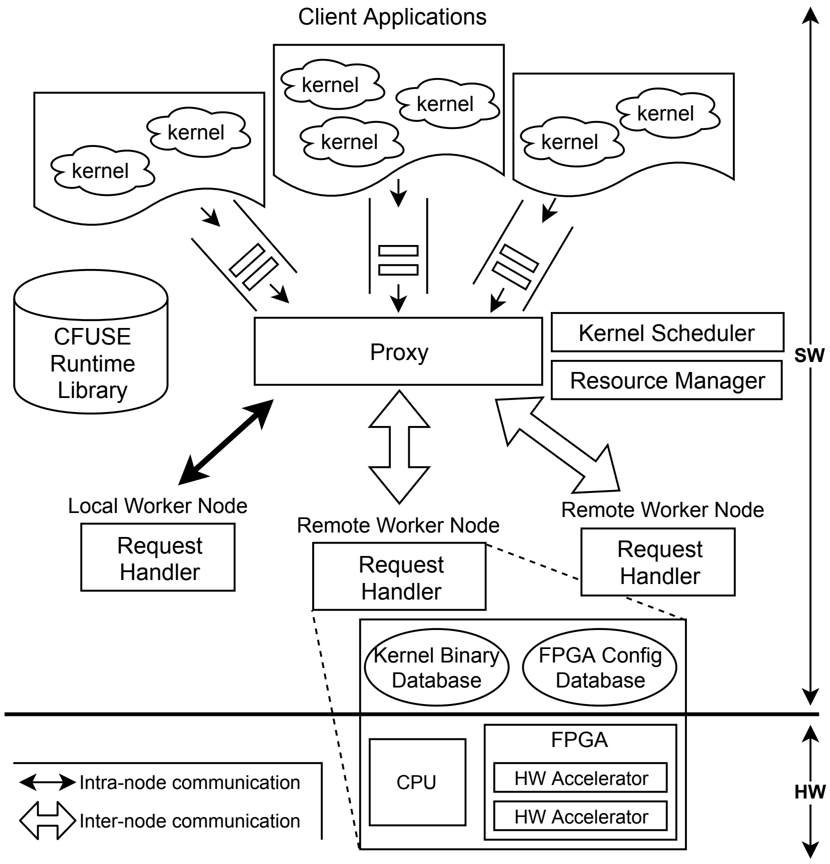


Figure 3.2: CFUSE runtime system overview

Figure 3.2 shows the interaction between these framework components and the basic process by which kernels are mapped to the compute resources. To summarize, a *runtime library* provides client applications with an interface to the facilities of the CFUSE runtime by exporting an object-oriented API. The main object provided by this library is a command queue, which client applications use to submit kernel requests for execution by the runtime. The *proxy* collects requests sent to these queues and forwards them to an appropriate *worker node*, where they are serviced by a *request handler*. The *kernel scheduler* is responsible for selecting the most suitable target device for each request. To make this decision, the scheduler consults the current status of the possible target devices in the cluster. This status information is maintained by the *resource manager*.

A request handler running on a worker node is responsible for managing the devices on that node. The handler executes kernel requests as they are received and sends a response containing kernel output data back to the proxy. This response may be used to update the resource manager. Finally, the proxy forwards this response back to the client application which initiated the request.

3.4 Component Details

In the following subsections, we describe the function of each of the components of the CFUSE framework runtime in more detail.

3.4.1 Proxy

The primary role of the proxy is to enable communication between client applications and the request handlers operating on each worker node. At system startup, the proxy establishes connections to each request handler and creates an endpoint that client applications use to establish connections to the proxy. Client applications submit kernel requests, which are sent to the proxy via these connections. The proxy then forwards these requests to an appropriate worker, as determined by the scheduler. The proxy is also responsible for forwarding kernel responses, which consist of kernel output argument data, back to the correct client application.

3.4.2 Resource Manager

The resource manager is responsible for tracking the availability of all the possible execution devices in the cluster. For each network node, this includes the CPU device and any hardware accelerators that are currently configured in the FPGA fabric of that node. This device status information is maintained globally within the resource manager and made available to the kernel scheduler. Device status is initialized at system startup and then updated both when a kernel request is scheduled or a response is received.

A primary design choice for the resource manager is between a centralized or distributed design. For our implementation, we have elected to use a centralized resource manager hosted on a single network node. Therefore, the resource manager has a global view of the device status for each node in the cluster. However, maintaining status information with this approach is more expensive. Status updates take longer to complete since they must propagate through the network.

3.4.3 Kernel Scheduler

The role of the kernel scheduler is to select a target device, either CPU or accelerator, for each kernel request received by the proxy. To do so, the scheduler uses kernel metadata stored within the request. This includes information such as the kernel function name used to find matching hardware accelerators, and the size of the kernel argument data that must be transferred. The scheduler uses this kernel metadata combined with the current device status information from the resource manager to make a scheduling decision based on a specific scheduling policy. The scheduler then adds the selected target device to the header of the request and informs the proxy of which node the request should be forwarded to for handling.

3.4.4 Request Handler

A request handler running on a worker node manages access to the devices on that node. The request handler receives requests from the proxy, reads the execution target from the request header and signals the target device to execute the kernel. Once a kernel request is completed by the handler, a response is sent back to the proxy to be forwarded to the user application that originally sent the request. The request handler is also responsible for managing a database of compiled kernel binaries for the CPU targets, and a database of FPGA configurations containing accelerators. These databases are specific to each worker node.

3.4.5 Runtime Library

The runtime library provides an object-oriented interface to the CFUSE framework for individual client applications. The main purpose of the library is to hide implementation details related to how the remaining components of the CFUSE framework interact to execute kernels on behalf of an application.

The API provided is designed as a light abstraction over the OpenCL execution model. User applications link against the runtime library which provides objects allowing for the specification of kernels, memory buffers to be passed as kernel arguments, and a command queue to which kernel requests are submitted for execution. This command queue is not associated with a specific device, as is the case with the OpenCL host API command queue. As previously mentioned, this allows the CFUSE runtime to select the most appropriate target device for kernel requests as they are received.

Chapter 4

CFUSE Prototype Implementation

This chapter discusses the prototype implementation of the CFUSE framework including the platform hardware and the software we developed. We present the implementation details beginning with a discussion of our cluster hardware and the accelerators used to implement kernel functions. We then describe the design of the various software components that make up the CFUSE framework, including the device drivers, system daemons and the runtime library for client applications. Finally, we provide an overview of the automation tool flow used to both simplify the creation of field-programmable gate array (FPGA) configurations, and to build the embedded Linux operating system (OS) that runs on each board.

4.1 Cluster Hardware

In order to have full control over the hardware interfaces and software stack used in the prototype, we opt not to use a public cloud such as a set of Amazon Web Services (AWS) Elastic Compute Cloud (EC2) F1 instances [2]. Instead, we use a custom-built cluster. Our target system for the prototype is a cluster consisting of four FPGA development boards that each contain a multi-core CPU and reconfigurable FPGA fabric. These boards are connected over a Gigabit Ethernet network formed by directly connecting each board to an Ethernet switch. Figure 4.1 shows a photograph of the prototype cluster system.

We have selected the Xilinx ZC706 Evaluation Kit [43] for the FPGA development boards in the cluster. The ZC706 board features a Xilinx Zynq-7000 XC7045 All Programmable system-on-chip (SoC) [44], which integrates a dual-core ARM processor system with FPGA fabric. These boards also feature a Gigabit Ethernet network interface, which is used to connect each board to a 8-port switch, forming the cluster network. We use a TP-Link TL-SG108 [45] for the network switch, which is an off-the-shelf unmanaged Gigabit switch. This switch offers a switching capacity of 16 Gbps with theoretically equal throughput for each port.

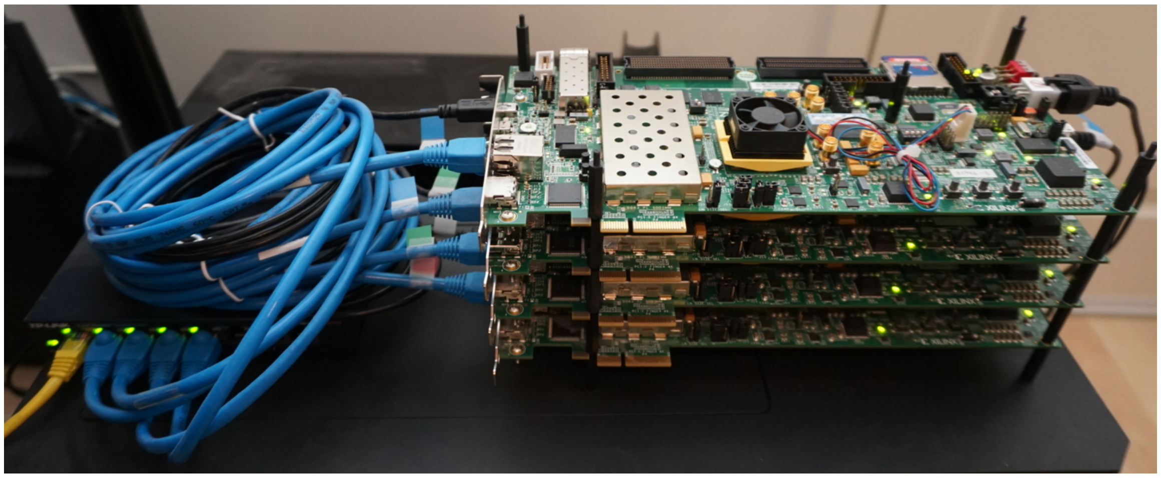


Figure 4.1: The target platform cluster for the CFUSE prototype implementation. Four Xilinx ZC706 FPGA development boards are connected to a TP-Link TL-SG108 8-port Gigabit Ethernet switch.

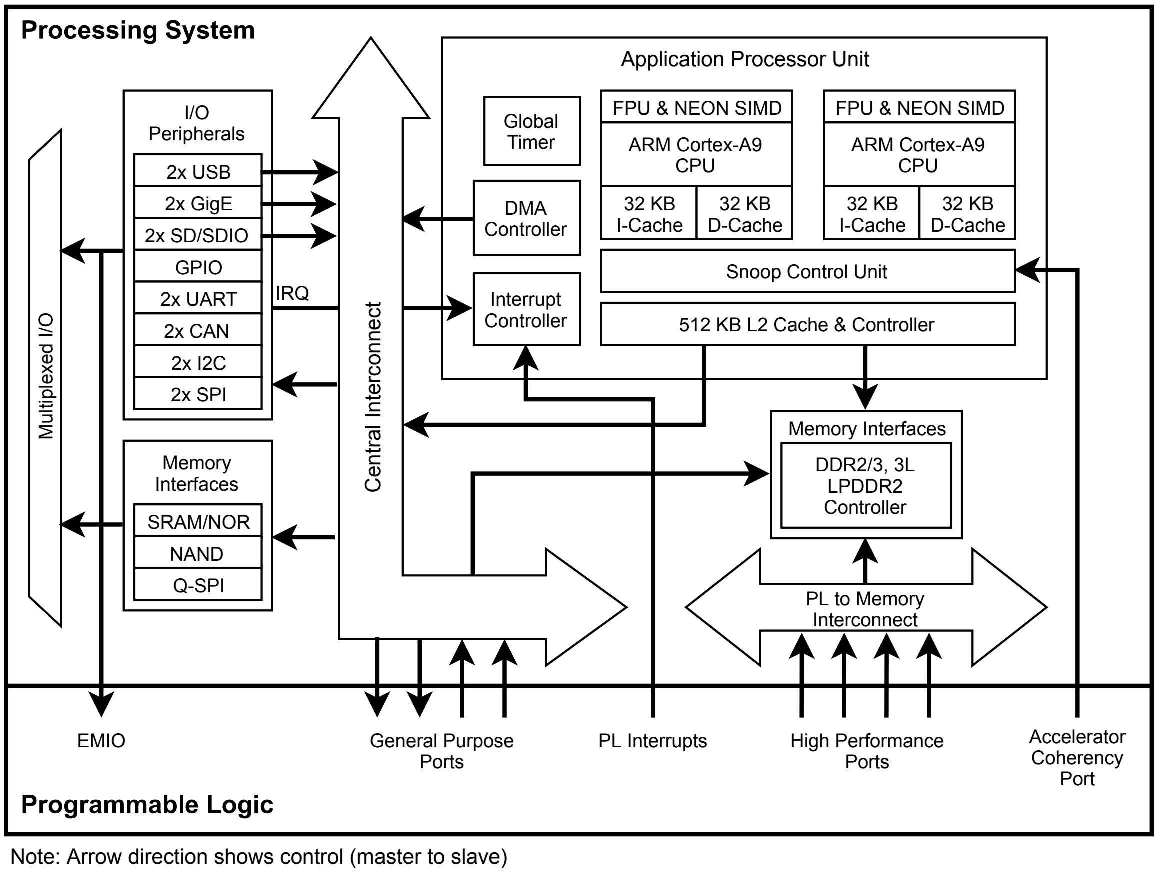


Figure 4.2: Zynq-7000 device block diagram. Adapted from [44].

4.1.1 Zynq-7000 System-on-Chip

A simplified block diagram of the Xilinx Zynq-7000 family of devices is shown in Figure 4.2. The Zynq device is divided into the *Processing System (PS)* and *Programmable Logic (PL)* regions. The Processing System contains the ARM multiprocessor system-on-chip (MP-SoC), which includes the Application Processor Unit (APU), memory interfaces to off-chip memory, and I/O peripheral cores. The Programmable Logic contains the FPGA fabric and reconfiguration hardware. For the prototype, we configure the processors to use a clock speed of 667 MHz and the logic in the PL uses a common 100 MHz clock. We selected this fabric clock speed as it is a reasonable frequency target for the accelerator cores of our benchmark applications.

The Application Processor Unit includes two ARM Cortex-A9 processors which feature 32 KB L1 instruction and data caches, a floating-point unit (FPU) and a NEON single instruction multiple data (SIMD) media engine. The APU also contains a unified 512 KB L2 cache and Snoop Control Unit (SCU), which manages processor cache coherency and enables coherent access to the L2 cache from the PL via the Accelerator Coherency Port (ACP).

The Zynq device uses hardware interfaces following the Advanced eXtensible Interface (AXI) protocol throughout. The AXI protocol is a bus architecture that defines three related interfaces [46]. Memory-mapped AXI is a high performance interface supporting variable sized burst transfers and interface data widths ranging from 8 to 1024 bits. AXI Lite is a simplified version of the full AXI protocol without burst transfer support and is intended for low throughput applications. AXI Stream is a point to point streaming protocol without address channels.

Three sets of AXI interfaces enable communication across the PS-PL boundary: four *High Performance (HP) ports*, four *General Purpose (GP) ports* and one *Accelerator Coherency Port (ACP)*. Each of these interfaces is intended for a particular use case. The High Performance ports provide high bandwidth access from the Programmable Logic to external DDR memory. The General Purpose ports are meant for register style accesses between the Processing System and Programmable Logic. Finally, the Accelerator Coherency Port provides low latency, coherent access to the APU L2 cache from the Programmable Logic. A set of PS-PL interrupts are also available, allowing for custom hardware in the Programmable Logic to use interrupt-driven I/O. In the following section, we describe how we use these PS-PL interfaces to integrate custom hardware accelerators.

4.2 Hardware Accelerators

In the CFUSE framework, custom hardware accelerators are used to implement kernel functions in hardware. For the prototype, these are instantiated in the Programmable Logic of the Zynq device and connected to the Processing System, as shown in Figure 4.3. CFUSE utilizes high-level synthesis (HLS) to produce synthesizeable HDL (hardware description language) code from kernel function source code written in either C++ or OpenCL C. We use the Xilinx Vivado HLS 2016.4 tool to perform this conversion [7]. Vivado HLS is capable of generating hardware Intellectual Property (IP) cores with a variety of different hardware interfaces both for controlling the IP core and transferring data to and from the core. While the design of each individual interface is fixed and specified by Vivado HLS, the specific interfaces to use for a given accelerator may be customized.

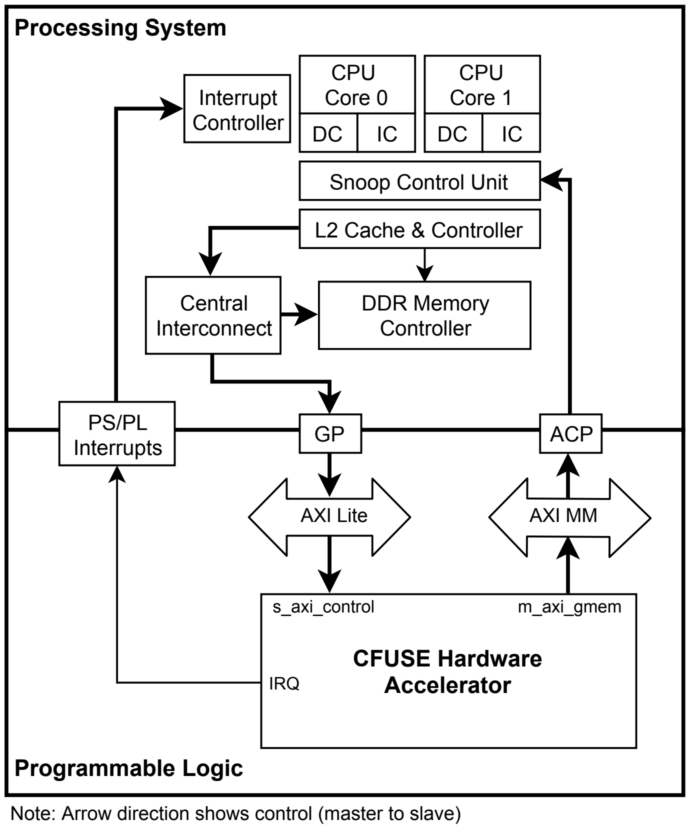


Figure 4.3: CFUSE hardware accelerator architecture

We choose to use a common interface for all the hardware accelerators we generate for the prototype. A common interface allows a single OS device driver to control each accelerator instance from software. This accelerator interface consists of the following:

- An AXI Lite slave for register based control (`s_axi_control`) that connects to a General Purpose port.
- A memory-mapped AXI master interface (`m_axi_gmem`) that connects to the Accelerator Coherency Port.
- An interrupt request output (`IRQ`).

The memory-mapped AXI master interface carries kernel argument data for kernel function parameters that are of pass-by-reference (pointer and array) type. The AXI Lite slave is used to access accelerator registers. These registers are used to control the accelerator operation, to transfer pass-by-value kernel arguments, and to allow software to specify direct memory access (DMA) buffer addresses for each pass-by-reference kernel argument. The accelerator control interface contains the following registers:

- Control Register
- Status Register
- Interrupt Enable Register
- Interrupt Status Register
- an Address Register for each pass-by-reference kernel function argument
- a Value Register for each pass-by-value kernel function argument

The primary design choice for integrating our hardware accelerators within the Zynq device is deciding which PS-PL interface the memory-mapped AXI interface should be connected to. For the prototype implementation, we chose to use the ACP to provide accelerators coherent access to the CPU L2 cache. This choice was motivated by the benchmark applications we used, which generally utilized smaller data sets. We note that depending on the memory access patterns of specific kernels, connection to one of the High Performance ports may offer better performance.

4.2.1 Accelerator Operation

The basic process for using a hardware accelerator to execute a kernel function is as follows:

1. Allocate DMA buffers for pass-by-reference arguments. For input arguments, initialize the corresponding DMA buffer(s) with argument data.

2. Write kernel arguments.

   (a) For pass-by-reference arguments, write the physical address of the DMA buffer containing argument data to the Address Register for that argument.

   (b) For pass-by-value arguments, write the argument value directly to the Value Register for that argument.

3. Enable interrupts for the accelerator by writing to the Interrupt Enable Register.

4. Start the accelerator by writing to the Control Register. During operation the accelerator will perform DMA transfers to and from the buffers specified, interleaved with computation.

5. Wait for an interrupt from the accelerator. Once the accelerator has completed its operation, it raises an interrupt.

6. Clear the interrupt by writing to the Interrupt Status Register.

7. Output argument data is now available in the corresponding DMA buffer(s).

4.3 Software Runtime Components

Figure 4.4 provides a hierarchical view of the software components that make up the CFUSE runtime prototype implementation. The runtime consists of two Linux loadable kernel modules (LKMs) that implement the accelerator device driver and DMA support respectively, as well as two system daemons and a shared object library. In the following subsections, we describe our design and implementation for each of these software components and their role within the CFUSE framework.

4.3.1 Hardware Accelerator Device Driver

The Linux device driver for CFUSE hardware accelerators is used to control accelerator hardware from software and exposes accelerators as Linux character devices. Each accelerator instance has an associated filesystem node, which application software uses as a handle to the accelerator. The device driver implements two system calls that are exported via the filesystem node: `mmap` and a custom `ioctl` operation. The driver `mmap` implementation memory maps the accelerator control register interface. This allows higher-level software to set accelerator arguments by writing to the accelerator Address Registers and Value Registers accordingly.

This memory-mapped register interface cannot be used to directly access the accelerator Control Register and start the accelerator. As mentioned in Section 4.2.1, accelerators use interrupts to signal that they have completed their operation. However, interrupt handling from software must occur within OS kernel-space. For this reason, the driver also exports a blocking `ioctl` function, which starts the accelerator and blocks the calling thread until accelerator completion is signalled by an interrupt. An interrupt handler implemented within the driver, acknowledges the interrupt, clears the interrupt bit and unblocks the calling thread.

4.3.2 DMA Buffer Module

As mentioned in Section 4.2, the HLS-generated accelerators use DMA buffers to transfer pass-by-value argument data. The Zynq platform lacks an I/O memory management unit (IOMMU) for translating user-space virtual addresses into physical bus addresses for use by accelerator hardware. Therefore, accelerator DMA buffers must be physically contiguous. On a Linux platform, physically contiguous buffers may only be allocated by the OS. To provide user-space with a means to allocate physically contiguous DMA buffers via the OS, we implemented the *DMA Buffer Module* as an LKM. This module creates a single Linux character device and implements the `mmap` system call. The implementation of this `mmap` allocates a contiguous buffer of the specified size and memory maps the buffer into the address space of the calling process.

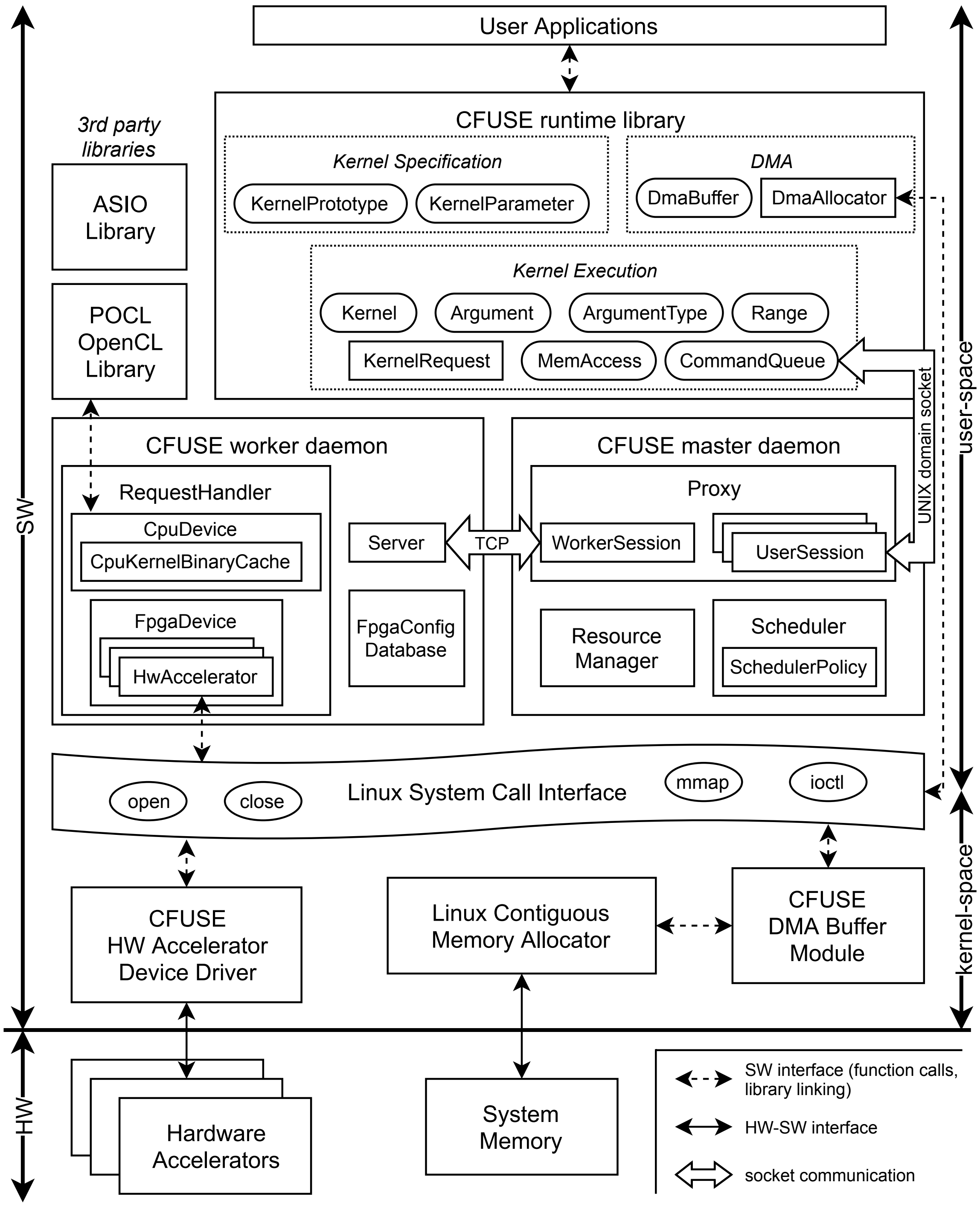


Figure 4.4: Overview of the CFUSE runtime prototype implementation

We make use of the *Linux Contiguous Memory Allocator (CMA)* capability to allocate these buffers. The CMA system works by reserving a fixed-size block of physically contiguous OS memory during system boot-up. The CMA allocator then uses this memory region at runtime to satisfy memory allocation requests from other OS components. In the prototype implementation, we have configured the CMA system to reserve 10 % of the 1 GB system memory, for a total of 102.4 MB.

4.3.3 CFUSE Daemons

Our prototype implementation contains two system daemons (master and worker) that implement the primary runtime components introduced in Chapter 3. The *master daemon* (cfused-master), runs on a single designated master node within the cluster. A *worker daemon* (cfused-worker) is started on each designated worker node. We distinguish between *local* and *remote* workers. The local worker refers to the worker daemon that runs on the same network node as the master daemon, while remote workers run on remote nodes.

cfused-master

The role of the master daemon is to receive kernel execution requests from user applications, choose a target device (either CPU or hardware accelerator) on a specific worker node and forward the request there. Once a kernel request is completed on a worker, the master forwards the response containing kernel output data, back to the user application that originated the request.

To manage forwarding between user applications and workers, the master daemon establishes connections to each worker node when the cluster is brought online. Similarly, to allow connections from user applications, the master sets up a local communication endpoint during initialization. For the prototype system, user applications are started on the same node on which the master daemon runs. This allows the master to use a UNIX domain socket for the communication endpoint to user applications. Kernel requests from multiple applications are received by the master using this socket in a first-come, first-served fashion.

To select a suitable target for a kernel request, the master contains a *scheduler* and a *resource manager*. The resource manager maintains a global view of the status of the CPU and accelerator devices within the cluster. This status information is used by a specific *scheduler policy*. We have made an effort to make it easy to introduce additional scheduling policies within the CFUSE framework. The resource manager updates status information when kernel requests are scheduled and when responses are received.

Different scheduling policies may require access to different types of information from the resource manager. Therefore, in our prototype the scheduling policy and resource manager are tightly coupled, allowing each policy to customize the data structures that store resource status information. This allows for status information to be efficiently retrieved and updated, given the specific requirements for each policy.

cfused-worker

As mentioned, a worker daemon runs on each worker node. In our prototype cluster implementation, we also run an instance of the worker daemon on the same node as the master. This is known as the *local worker*. The primary purpose of the worker is to manage the devices available locally on a node and execute kernels as they are received. On startup, the worker daemon initializes the local FPGA with a bitstream configuration that is statically assigned to each worker as part of the cluster configuration. Although we do not have a scheduler policy that utilizes runtime reconfiguration in the current prototype, we have implemented the required FPGA management infrastructure to support it in the future. The worker daemon then waits for a connection to be established with the master daemon. As kernel requests are received by the worker, they are passed on to the appropriate device as indicated in the header of the request.

Kernel requests bound for the CPU are executed using the Portable Computing Language (POCL) OpenCL implementation [8]. As mentioned in Section 2.4.1, POCL provides a kernel compiler based on Clang and LLVM that produces object code for CPU targets and maps kernel work-items to threads. As our prototype hardware platform is an embedded system, we choose to pre-compile sets of kernels offline rather than use the online compilation capability of the OpenCL host application programming interface (API). We store the sets of kernel program binaries in a database, so that they may be loaded as required. To reduce the latency of kernel executions on the CPU, we also cache kernel objects as they are created from these binaries.

Daemon Communication

Communication between master and worker daemons occurs over the Gigabit Ethernet network and involves the use of the TCP/IP stack on the processor via socket connections. We initially evaluated the use of Message Passing Interface (MPI), (specifically the OpenMPI implementation), for enabling communication between master and worker daemons. However, OpenMPI implements the MPI non-blocking operations using busy waiting. We found that this busy waiting led to high CPU usage within the daemons while they were idle, causing starvation of other system processes. Therefore, we decided to implement communication using TCP sockets via the Asio library [47].

Asio is a C++ library that provides an asynchronous programming framework and abstractions of network and other I/O functionality such as sockets. In Asio, asynchronous operations, such as reads and writes, are started on a socket by specifying a callback function known as a completion handler. Submitting an asynchronous operation starts a non-blocking call on the underlying socket descriptor. Once the operation has completed, Asio schedules and runs the corresponding completion handler function. Completion handlers may in turn start additional asynchronous operations, leading to a chain of asynchronous operations being executed.

4.3.4 Runtime Library

The CFUSE runtime library is implemented as a shared object library that user applications link against to access the functionality of the CFUSE runtime. The library provides an object-oriented C++ API serving as a light abstraction over the objects available in the OpenCL host API. We can generally divide the classes exported by the runtime library into the following categories based on their use: *kernel specification*, *kernel execution* and *DMA*.

Of these objects, the *CommandQueue* provides the primary method with which a client application will interface to the rest of the CFUSE runtime system. When a command queue is created by an application, the runtime library establishes a connection to the previously created endpoint of the master daemon. When kernels are submitted to the command queue within the application, the library generates a kernel request message. A kernel request contains a header with fields describing the kernel function, a range specification and information about each kernel argument. The payload of the kernel request is variable in size, as it contains the data for kernel function arguments. Once created the request is sent to the master daemon for scheduling. On the prototype we launch client applications on the same node as master, allowing us to use a Unix domain socket for the communication endpoint between the master daemon and user applications.

4.4 Hardware and Software Build Automation

The final component of the CFUSE framework implementation is a series of scripts and software build infrastructure that helps automate the process of creating FPGA configurations, generating an embedded Linux image for the cluster boards, and compiling the runtime software components. Our prototype implementation utilizes *FPGA configurations*, which we define as containing an FPGA bitstream, an overlay for the Linux device tree and configuration metadata.

The entry point for creating a configuration is a definition of accelerator metadata, which is a description of the accelerators to be included in the configuration. This metadata specifies kernels by their function declaration, including the function name, parameter list (parameter names and types) and the location of the kernel source code files. We provide a Python script that wraps the use of various computer-aided design (CAD) tools to create an FPGA configuration. The wrapper script performs the following steps:

1. *Create accelerators from kernel source code.* This step involves configuring the Vivado HLS tool for each kernel and then running HLS to generate an accelerator IP core.

2. *Create a Vivado project.* This step involves creating a project for the Xilinx FPGA CAD tool Vivado. A system containing accelerator cores and the required system infrastructure is created automatically.

3. *Generate a bitstream.* This step runs the Vivado tool performing behavioural synthesis, place and route, and bitstream generation.

4. *Create a device tree overlay.* Using the configuration metadata and the Linux device tree compiler, the wrapper script creates a device tree overlay file. This overlay is applied onto the base Linux device tree when an FPGA configuration is loaded. The device tree contains hardware information about the accelerators in a configuration and is read by the accelerator device driver.

The embedded Linux image is generated using Buildroot [48]. Buildroot is a tool used to simplify setting up a cross compilation toolchain, as well as configuring and cross-compiling the Linux kernel, an embedded bootloader and packaging various system software into a root filesystem. For this prototype we use a Linux version 4.6 operating system kernel provided by Xilinx and a cross compiler toolchain from Linaro with hardware floating-point capability based on the GNU Compiler Collection (GCC) version 5.3.

Chapter 5

Experimental Framework Characterization

This chapter outlines the measurement techniques and experiments used to characterize the CFUSE prototype implementation. First, we discuss the need for a more accurate interval timing mechanism on the prototype platform and introduce our cycle counter-based solution. We then outline the instrumentation method used to record execution times of program events within the CFUSE daemons. Finally, we discuss the benchmarking methodology used to determine the characteristics of data transfers as well as the overhead introduced by the physically contiguous direct memory access (DMA) buffers.

5.1 Low Overhead Interval Timer

To compare the runtime performance of benchmark programs, it is important to use a low-overhead, monotonic timer to record time intervals. On Linux and other Portable Operating System Interface (POSIX) platforms, the standard approach is to record start and stop times by reading values from a high-resolution clock source. The POSIX standard specifies the `clock_gettime` function for this purpose, which returns a time value to nanosecond resolution. However, on the ZC706 platform, this function is implemented using a system call and therefore includes the overhead of a context switch into the operating system (OS) kernel mode. Using a simple benchmark program, we determined the overhead of the `clock_gettime` call to be approximately 1 µs.

To more accurately measure small time intervals of executing code sections, we implemented a method to read the cycle count from the Global Timer, a 64-bit incrementing counter that is part of the ARM Cortex-A9 system-on-chip (SoC) [49]. The implementation consists of a Linux kernel module used to map the Global Timer cycle count registers into user space. This allows for direct access to these registers from user programs, eliminating the need for a system call and removing the overhead of the associated context switch.

5.1.1 Timer Read Procedure

While the Global Timer is a 64-bit counter, the ARM architecture is 32-bit. Therefore, the procedure for reading a single cycle count involves reading two 32-bit register values that correspond to the upper and lower 32 bits of the counter. Care must be taken to ensure that the upper 32 bits do not change during the time it takes to read the lower 32-bit value, which necessitates a minimum of three register read operations. Listing 5.1 shows the source code for this read procedure, which includes handling for the case where the upper 32 bits change during the read of the lower 32 bits.

```
uint64_t read_cycle_count()
{
    uint64_t cycle_count;
    uint32_t lower;
    uint32_t upper, old_upper;

    upper = read_register(GT_UPPER);
    do {
        old_upper = upper;
        lower = read_register(GT_LOWER);
        upper = read_register(GT_UPPER);
    } while (upper != old_upper);

    cycle_count = upper;
    cycle_count <<= 32;
    cycle_count |= lower;
    return cycle_count;
}
```

Listing 5.1: Global Timer read procedure

To convert the cycle count value read from the Global Timer to a time value in seconds, it is necessary to multiply by the clock period of the Global Timer clock source. On the prototype platform, the Global Timer is clocked at half the central processing unit (CPU) core clock frequency of 667 MHz, which corresponds to a clock period of 3 ns.

5.1.2 Timer Accuracy Characterization

To determine the overhead of recording a time interval using the Global Timer, we used a benchmark of the form shown in Listing 5.2. The benchmark simply records a start and stop time using consecutive calls to `read_cycle_count` and records the intervals into an array of samples. In doing so, the benchmark simulates the procedure that would be used to measure a time interval in practice. Therefore, because the benchmark records an empty time interval, the time recorded will approximate the timer overhead.

```
uint64_t start, stop;
uint64_t samples[NUM_SAMPLES];
for (i = 0; i < NUM_SAMPLES; i++) {
    start = read_cycle_count();
    stop = read_cycle_count();
    samples[i] = stop - start;
}
```

Listing 5.2: Interval timer benchmarking procedure

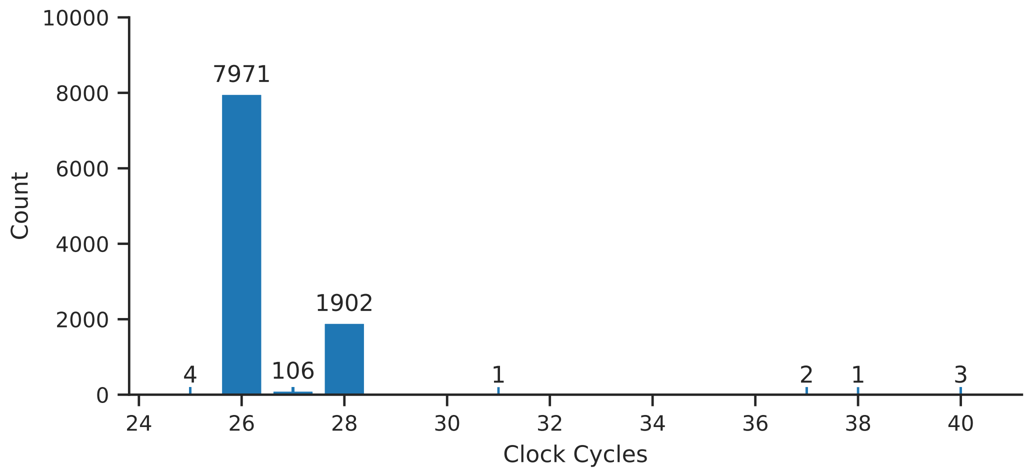


Figure 5.1: Histogram of interval timer overhead. Data points are shown as ticks on the x-axis where the histogram bars would not appear due to the plot scale. Values counts are shown above each histogram bar.

Table 5.1: Interval timer overhead statistics

time unit	mean $\pm\ \sigma$	(min, max)
clock cycles	26.40 ± 0.84	(25, 40)
nanoseconds	79.20 ± 2.53	(75, 120)

Figure 5.1 shows the histogram of the results from the timer overhead benchmark run with 10 000 samples. Table 5.1 shows the summary statistics from these benchmark runs in both clock cycles and scaled by the 3 ns clock period to give time in nanoseconds. Due to the mechanics of the read procedure outlined in Section 5.1.1, the overhead for recording a time interval is variable. The time required to read the 64-bit clock cycle count varies depending on whether the upper 32 bits of the cycle count register changed during either the call to read the start time, the stop time or during both calls. The results in Figure 5.1 show this spread, but with nearly all the samples in the range of 25 to 28 cycles.

With these results, we can determine the minimum time interval that may be measured to a given accuracy. For a conservative estimate of this minimum time interval, we use the maximum overhead value seen of 120 ns. For example, the minimum time interval that may be measured to within 1% accuracy can be calculated as $120\,\text{ns} \times 100 = 12\,\mu\text{s}$.

5.2 CFUSE Daemon Instrumentation

For recording the runtime of specific code paths within the CFUSE daemons, we implemented an instrumentation method based on the logging of time interval values. These time intervals are measured by recording start and stop times using the cycle count based interval timer described in Section 5.1. These intervals are stored in memory during program execution and written to a log file periodically. We instrumented the following program events from the CFUSE daemons:

- `master::read_kernel_request_data` – Read kernel input argument data sent from a client application to master.
- `master::schedule` – Choose an execution target for a given request.
- `worker::read_kernel_request_data` – Read kernel input argument data sent to the worker.
- `worker::execute_kernel` – Execute a kernel on a device.
- `master::read_kernel_response_data` – Read kernel output argument data returned by the worker.
- `master::update_scheduler` – Update any scheduler bookkeeping data upon receiving a response.

The set of program events that may be instrumented is somewhat limited by the use of asynchronous operations within the runtime implementation. As mentioned in Section 4.3.3, we use the Asio library to perform asynchronous socket reads and writes for communication between client applications and the master daemon, and between the master and worker daemons. Due to the use of completion handler callback functions in Asio, it is only possible to determine when an asynchronous operation has completed and not when it started executing. Since measuring the runtime of an event involves recording a cycle count at the start and end of the event, this restricts our instrumentation to events that don't use asynchronous operations at all and events that use chains of asynchronous operations, such as the reading of kernel request and response messages.

We are able to record the time required to read kernel request data since a kernel request message consists of a header and a variable-sized payload representing kernel data. Reading a single request message therefore requires a chain of two asynchronous read operations: one to read the header and one for the kernel data. We can record the start time from within the completion handler for the header read and the stop time during the completion handler for the kernel data read. The procedure is the same for kernel responses, which have a similar header and payload format as kernel requests.

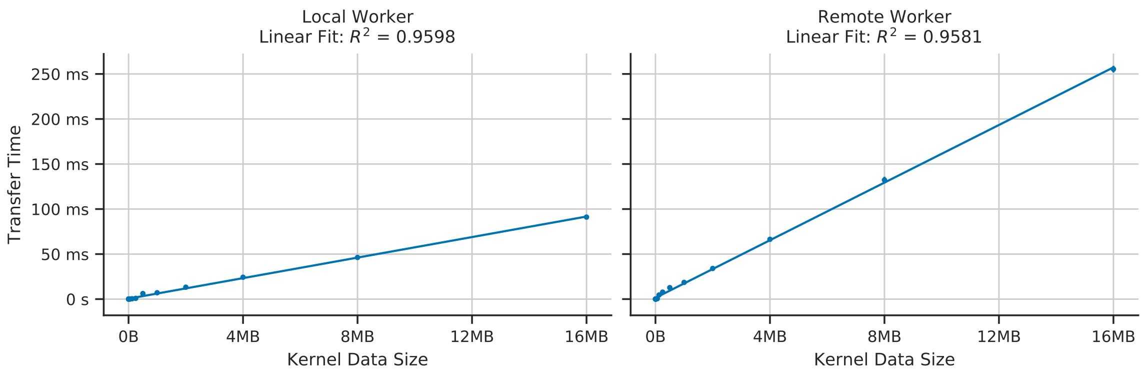
5.3 Data Transfer Characterization

To characterize the data transfer between daemons, we implemented a `memcpy` kernel that copies a specified number of bytes from a source buffer to a destination buffer. We run this kernel using the CFUSE framework with workers hosted on both the local node and remote nodes and vary the data size to determine the latency of both local and remote transfers. Our objective is to fit a model to this latency data, which would allow scheduler policy implementations to estimate data transfer overhead when determining a target worker for a kernel request.

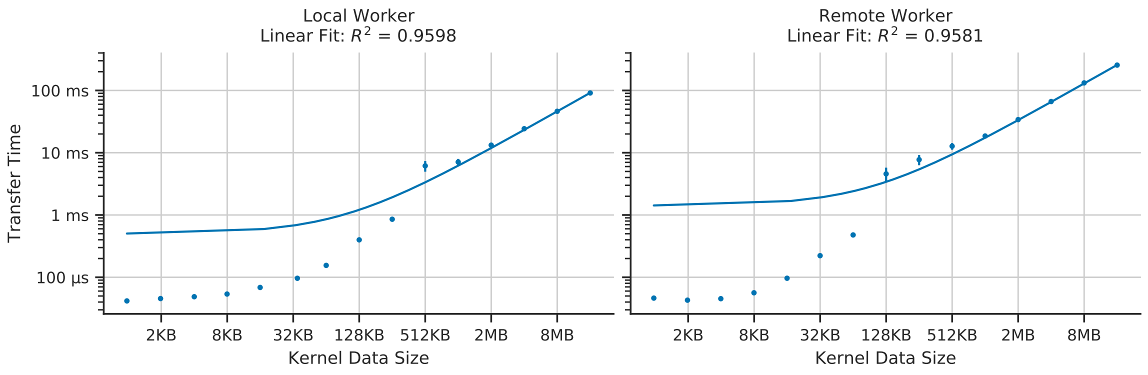
The data transfer time from master to local and remote workers is shown in Figure 5.2 and Figure 5.3. From the linear scale plot of transfer time vs. data size in Figure 5.2a, we find that transfer times to remote workers are longer than to the local worker as expected. This is due to the fact that data transfer to the local worker does not require network communication, using a local TCP socket for inter-process communication. Furthermore, the relationship between transfer time and data size appears linear. We achieve what appears to be a well-fitted linear regression line with R^2 values of 0.9598 and 0.9581 for local and remote workers respectively.

Figure 5.2b shows a $\log_{10}$-$\log_2$ scale plot, of the same latency data and linear regression. Using this scale it is easier to see the data points for small data sizes. It is clear at this scale that the regression line fits poorly for data sizes below approximately 512 KB for the local worker, and 64 KB for the remote workers.

The transfer time to data size relationship is clearly non-linear for smaller data sizes and transitions to a linear relationship for data sizes above a certain threshold. Therefore, using the linear model to estimate transfer times in a scheduling policy, would lead to large inaccuracies for data sizes within this range. From the log scale plots, these inaccuracies



(a) Linear scale



(b) $\log_{10}$-$\log_2$ scale

Figure 5.2: Kernel data transfer time from master to local and remote workers. Solid line shows linear regression.

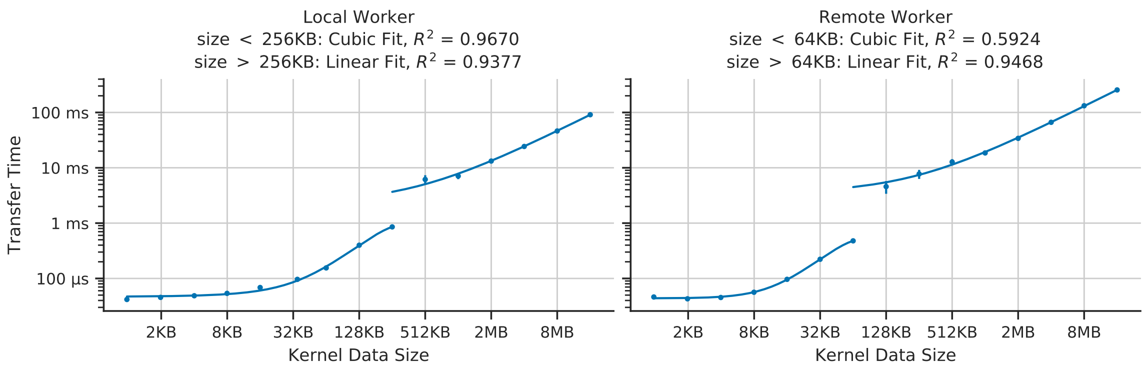


Figure 5.3: Kernel data transfer time from master to local and remote workers. Solid lines show segmented regression with cubic and linear components.

are on the order of 10x. To overcome this problem, we refit the data using a segmented approach. For data points below the threshold data size, we refit using a polynomial regression. For both the local and remote data a cubic polynomial was found to fit the data points reasonably well without overfitting. For the data above the threshold in the linear region, we fit using a linear regression.

Figure 5.3 shows the result of this segmented fitting. As can be seen, the model becomes much more accurate for small data sizes. The segmented model comes at a cost of greater complexity and introduces a discontinuity between the two segments.

5.4 DMA Buffer Overhead

As explained in Section 4.2, the accelerators on the prototype platform require argument data to be located in physically contiguous DMA buffers. To quantify the overhead involved with the use of these buffers, we developed a set of three microbenchmarks. `BM_DmaBuffer_create` performs a single DMA buffer allocation of a specified size followed immediately by a free. `BM_DmaBuffer_write` writes a block of memory to the DMA buffer using `memcpy`, while `BM_DmaBuffer_read` uses `memcpy` to read a block of memory from a DMA buffer. We executed these benchmarks on the ZC706 board and varied the buffer sizes ranging from 1 KB to 16 MB. This range was chosen to cover the input and output data sizes of each of the benchmark kernels for the sample data provided.

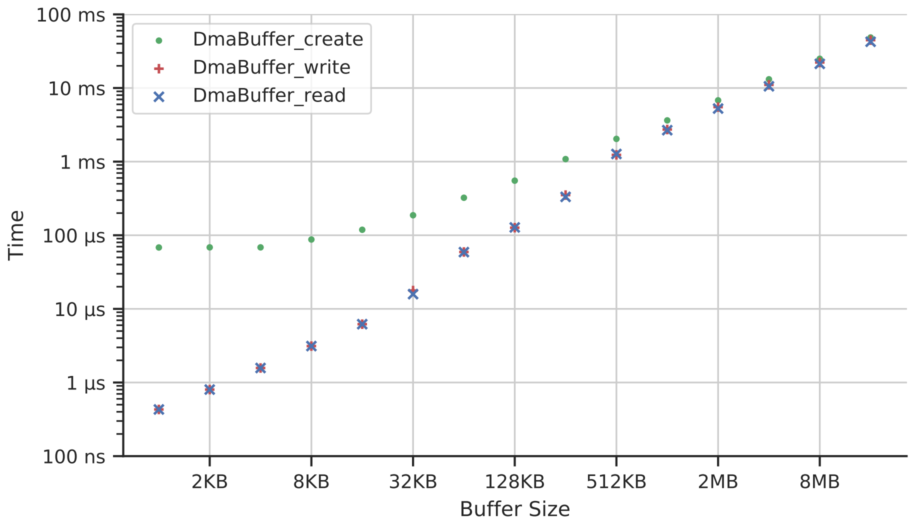


Figure 5.4: DMA buffer overheads

Figure 5.4 shows the results from the microbenchmarking of the DMA buffer operations. From the plot, we see that the time to allocate a DMA buffer is constant for buffer sizes up to 4 KB. This is a direct result of the implementation of the DMA Buffer Module. As mentioned in Section 4.3.2, DMA buffers are made accessible to user-space via a `mmap` call that maps entire memory pages. Therefore, the minimum allocation size is a single page, which is 4 KB on the platform. The plot also shows that the read and write times are essentially identical. This follows from the fact that both operations are implemented using the `memcpy` function.

Chapter 6

Experimental Results and Discussion

In this chapter, we introduce the benchmark programs and methodology we used to create application workloads for evaluating the CFUSE framework prototype. Using single application workloads, we determine the baseline performance of each benchmark accelerator. We then describe the implementation of three scheduling policies with varying capability and sophistication. Finally, we outline the workload metrics of interest and present an evaluation of scheduler performance as it relates to total workload runtime.

6.1 Benchmark Applications

We use a selection of benchmark programs taken from the Rosetta benchmark suite [50] for creating application workloads for our system evaluation. Rosetta is a collection of image processing and machine learning benchmark applications designed for use on field-programmable gate array (FPGA) platforms. Each benchmark application features a single kernel function to be accelerated and the Rosetta suite provides three separate implementations of each kernel: a baseline software version in C++ targeting the central processing unit (CPU), and kernel versions in C++ or OpenCL C designed for use with high-level synthesis (HLS) targeting both the embedded ZC706 board and an Amazon Web Services (AWS) F1 FPGA instance.

We have chosen a subset of four benchmarks of the possible six available in Rosetta. The Rosetta authors have indicated that the Binarized Neural Network benchmark requires non-trivial modifications to run with an OpenCL kernel so we have excluded it. The Optical Flow benchmark uses a kernel with an associated accelerator that is designed to run only once before requiring a system reset. Since we use workloads comprised of multiple benchmark applications, we require that the benchmark accelerators can run multiple times. Therefore we have also excluded the Optical Flow benchmark from the available set.

Table 6.1 summarizes the benchmarks chosen from the Rosetta suite. Each benchmark is classified as either compute-bound or memory-bound. For each benchmark, we also list the operations that make up the majority of the benchmark kernel functionality. The data sizes of the input arguments and output arguments are also provided. We note that the spam-filter benchmark is the only memory-bound benchmark and uses the largest data set size. Furthermore the image processing benchmarks (3d-rendering and face-detection) use smaller data sets compared to the machine learning benchmarks (digit-recognition and spam-filter). In the subsections that follow, we describe each benchmark and its kernel function in more detail.

Table 6.1: Rosetta benchmark application summary

Benchmark	Application Domain	Categorization	Primary Operations	Sample Data Size	
				Input	Output
3d-rendering	Image processing	Compute-bound	Integer arithmetic	28 KB	64 KB
digit-recognition	Machine learning	Compute-bound	Bitwise operations Hamming distance k-NN voting	625 KB	1.95 KB
face-detection	Image processing	Compute-bound	Integer arithmetic Image scaling Cascaded classifiers	75 KB	1.57 KB
spam-filter	Machine learning	Memory-bound	Fixed-point arithmetic Dot product Scalar multiplication Vector addition Sigmoid function	19.54 MB	4 KB

6.1.1 3D Rendering

3D Rendering is an image processing benchmark that renders 2D images given a mesh of 3D triangles as input. The benchmark kernel function implements a 5-stage image processing pipeline with the following stages:

1. *Projection*: Convert a 3D triangle description into 2D
2. *Rasterization*: Search for pixels in a 2D triangle within the bounding box
3. *Z-culling*: Hide or display pixels according to their z-value (depth)
4. *ColoringFB*: Colour the frame buffer according to the z-value buffer

The input data consists of an array of 3D triangle objects represented by 9, 8-bit values corresponding to the coordinates of the triangle vertices in the x, y and z dimensions. The sample data consists of an array of 3192 3D triangle objects, which is rendered to a 256×256 8-bit per pixel grayscale image. This corresponds to a kernel input argument size of 28 KB and an output size of 64 KB. The benchmark kernel is classified as compute-bound, with the main operations consisting of integer arithmetic. The main HLS optimization used in the implementation is dataflow pipelining.

6.1.2 Digit Recognition

The Digit Recognition benchmark is from the machine learning domain and classifies images of hand-written digits using the k-nearest neighbours (k-NN) algorithm. The kernel function consists of two parts: the calculation of Hamming distance and the k-NN calculation, The benchmark kernel is classified compute-bound and the primary operations are bitwise operations. The main optimizations in the HLS kernel implementation are loop unrolling and loop pipelining.

The benchmark sample data consists of images of handwritten digits taken from the MNIST database and are divided into a training set of 18 000 digits and a test set of 2000 digits. The images have been binarized and downsampled to 14×14 pixels so that they may be packed into a 196-bit value with each bit being represented by a single pixel. This allows the sets of images to be efficiently transferred to the accelerator using a customized data path width. The benchmark outputs a 1-byte label for each image that specifies which digit the handwritten image was matched to. This sample data corresponds to kernel input argument data totalling 625 KB and output argument data of 1.95 KB.

6.1.3 Face Detection

The Face Detection application detects human faces from an image using the Viola-Jones algorithm. The benchmark kernel creates an image pyramid from which an integral image is generated. A fixed-sized window from the integral image is passed to a set of cascaded Haar feature classifiers. Integer arithmetic operations make up the majority of the benchmark kernel which is classified as compute-bound. The main HLS optimizations used are memory customization and datatype customization. The sample data provided with the benchmark is a 320×240 8-bit per pixel grayscale image containing a number of human faces. The output produced is a set of bounding box coordinates which define the locations of the detected human faces in the input image. This sample data corresponds to 75 KB of kernel input argument data and 1.57 KB of output data.

6.1.4 Spam Filter

Spam Filter is a machine learning benchmark that uses stochastic gradient descent to train a logistic regression for classifying emails as spam. The benchmark kernel is memory-bound as there is insufficient on-chip memory to store the training and test sets so data must be streamed to the accelerator. The main kernel operations are dot product, scalar multiplication, vector addition, and the calculation of the sigmoid function. The main HLS optimizations used are dataflow pipelining, memory customization and communication customization.

The sample benchmark data consists of 5000 emails, each represented as a 1024-element vector of features. These features are relative word frequencies stored as fixed-point values. From the sample emails, 4500 are used as a training set and 500 are used for the test set. The HLS kernel version uses custom bit-width data types to work with the fixed-point values. This sample data corresponds to 19.54 MB of kernel input argument data and 4 KB of output data.

6.1.5 CPU Kernel Conversion Process

As previously mentioned, Rosetta provides three implementations of each of the benchmark kernel functions, with each version provided for use on a different execution target. The software versions targeting the CPU are written in C++. However, as we described in Section 4.3.3, we use the Portable Computing Language (POCL) OpenCL implementation [8] to execute kernels on CPU devices within the CFUSE framework. The kernel compiler provided by POCL for the CPU target currently only supports kernels written in the OpenCL C language. Therefore, we must convert the Rosetta CPU kernels from C++ to OpenCL C.

While mostly compatible with the C programming language, OpenCL C provides some extensions, including extra built-in functions and vector data types. It also imposes some restrictions on the use of pointers, requires address space qualifiers and limits the use of the `static` storage-class specifier. Table 6.2 summarizes the incompatible language differences we encountered when converting the C++ kernel code to OpenCL C and the steps taken to resolve them.

To verify that these modifications did not significantly alter the runtime performance of the benchmark programs, we ran both the C++ and OpenCL versions of the benchmarks on the ZC706 platform and recorded the runtime of the kernel functions. For the C++ version, the kernel runtime is simply the time to call the kernel function. However, the OpenCL version requires the use of the OpenCL host application programming interface (API) to execute kernels. For this case, we calculate the kernel runtime as the time to set the work-item and work-group size, set kernel arguments and enqueue the kernel for execution. The results of this comparison are shown in Table 6.3.

As shown, there are some significant differences between the kernel runtimes for the C++ and OpenCL C versions. To determine the cause of these discrepancies, we examined the static object code generated for each version and noticed significant differences in the amount of function inlining, the total number of assembly instructions and the number of vector instructions generated by the compiler. These differences can be attributed to the

Table 6.2: C++ and OpenCL C language differences and resolutions

Language Difference	Resolution
No support for C++ reference types.	Replace the use of references with pointers.
Top-level kernel function parameters that are pointers must have an address space qualifier.	By default choose the global address space. When passing such variables as arguments to sub-functions, those functions must include the address space qualifier in their definition also.
All program scope variables must be declared in the constant address space.	Disallow the use of non-constant program scope variables and place constant program scope variables in the constant address space. Alternately convert program scope constants to C preprocessor object-like macros.
OpenCL C versions 1.2 and prior do not support static local variables.	Convert static local variables to program scope variables in the constant address space where possible. Otherwise add these as additional kernel function arguments.
OpenCL C provides additional built-in functions that may conflict with program identifiers.	Rename any identifiers that conflict with OpenCL built-in functions. E.g. `popcount`, `dot`
OpenCL C provides additional built-in math functions.	Prefer the use of OpenCL built-in math functions over math functions from the standard C library. E.g. prefer overloaded `exp(x)` function to `expf(double x)`.

Table 6.3: Rosetta benchmark kernel CPU runtimes

Benchmark	C++ Kernel Runtime	OpenCL Kernel Runtime	Relative Runtime
3d-rendering	32.136 ms	17.223 ms	0.536
digit-recognition	15.823 s	8.432 s	0.533
face-detection	501.429 ms	401.325 ms	0.800
spam-filter	859.398 ms	954.636 ms	1.111

fact that different compilers are used depending on the kernel source language. The C++ software kernels are built using the cross compiler for the platform, Linaro GCC, while the OpenCL kernels are built using the POCL kernel compiler that is based on Clang/LLVM. Furthermore, POCL uses the SLEEF vectorized math library [51] to implement OpenCL C built-in functions. These differences in the static object code are summarized in Table 6.4.

Table 6.4: Comparison of object code for Rosetta benchmark CPU target kernels. Object code statistics are static counts of functions, assembly instructions and vector instructions for the ARM NEON SIMD engine.

Benchmark	Kernel Language	Kernel Runtime	Functions	Total Instructions	Vector Instructions	
					Count	% of Total
3d-rendering	C++	32.136 ms	6	534	0	0.0
	OpenCL	17.223 ms	5	496	6	1.2
digit-recognition	C++	15.823 s	4	1010	0	0.0
	OpenCL	8.432 s	3	241	29	12.0
face-detection	C++	501.429 ms	1	1080	65	6.0
	OpenCL	401.325 ms	3	708	64	9.0
spam-filter	C++	859.398 ms	5	322	196	60.9
	OpenCL	954.636 ms	3	191	73	38.2

The 3d-rendering and digit-recognition benchmarks exhibit the largest relative differences between the C++ and OpenCL C versions. From the object code comparison, we see that the C++ kernels for these benchmarks do not utilize any vector instructions. Furthermore, the only benchmark that executes slower with an OpenCL version is spam-filter. For this benchmark, the OpenCL version has a lower proportion of vector instructions compared to the C++ version. Therefore, we propose that the use of vector instructions largely determines the relative runtime difference between C++ and OpenCL kernel versions for a given benchmark.

6.1.6 Benchmark Hardware Accelerators

For each of the Rosetta benchmarks, we created a system containing a single kernel accelerator instance. Table 6.5 shows the total FPGA area usage for each system. The total resource usage for the systems can be divided into the device area required for the accelerator and the system infrastructure, which includes the interconnect and reset core. Table 6.6 and Table 6.7 list the resource usage of the accelerator and infrastructure components respectively.

As shown, the system infrastructure resource usage is directly related to the data width of the accelerator memory-mapped AXI interface, which is used to transfer kernel argument data. A wider interface generally requires a larger interconnect and therefore more area. The

Accelerator Coherency Port (ACP) is a 64-bit interface. Therefore no data width re-sizing logic is required if the accelerator AXI interface is also 64-bit. This case results in the lowest resource usage for the interconnect.

Table 6.5: Total system resource usage

Accelerator	LUT	(%)	FF	(%)	BRAM	(%)	DSP	(%)
3d-rendering	5 990	(2.7)	6 308	(1.4)	36.5	(6.7)	9	(1.0)
digit-recognition	29 204	(13.4)	13 615	(3.1)	184.5	(33.9)	0	(0.0)
face-detection	47 804	(21.9)	47 616	(10.9)	89.5	(16.4)	79	(8.8)
spam-filter	6 676	(3.1)	9 563	(2.2)	69.0	(12.7)	224	(24.9)

Table 6.6: Accelerator resource usage

Accelerator	LUT	(%)	FF	(%)	BRAM	(%)	DSP	(%)
3d-rendering	5 051	(2.3)	4 975	(1.1)	36.5	(6.7)	9	(1.0)
digit-recognition	27 878	(12.8)	11 955	(2.7)	184.5	(33.9)	0	(0.0)
face-detection	46 865	(21.4)	46 283	(10.6)	89.5	(16.4)	79	(8.8)
spam-filter	6 124	(2.8)	8 771	(2.0)	69.0	(12.7)	224	(24.9)

Table 6.7: System infrastructure (interconnect and reset) resource usage

Accelerator	LUT	(%)	FF	(%)	Accelerator Data Bus Width
3d-rendering	939	(0.4)	1333	(0.3)	32
digit-recognition	1326	(0.6)	1660	(0.4)	256
face-detection	939	(0.4)	1333	(0.3)	32
spam-filter	552	(0.3)	792	(0.2)	64

6.2 Baseline Kernel Performance

The goal of the baseline performance characterization is to establish an initial performance level for each benchmark and to investigate the overhead of executing kernels via the CFUSE runtime. Using the daemon instrumentation outlined previously in Section 5.2, we seek to understand the source of these overheads and how they vary between the different benchmarks. For this evaluation we execute only a single instance of each benchmark application. We repeat the benchmark targeting each of the four possible devices types: the CPU on the local node, an accelerator on the local node, a CPU on a remote node and an accelerator on a remote node.

Figure 6.1 shows a comparison of the total application runtime on each of the target types for each benchmark. Runtime is shown relative to the local CPU runtime. From the

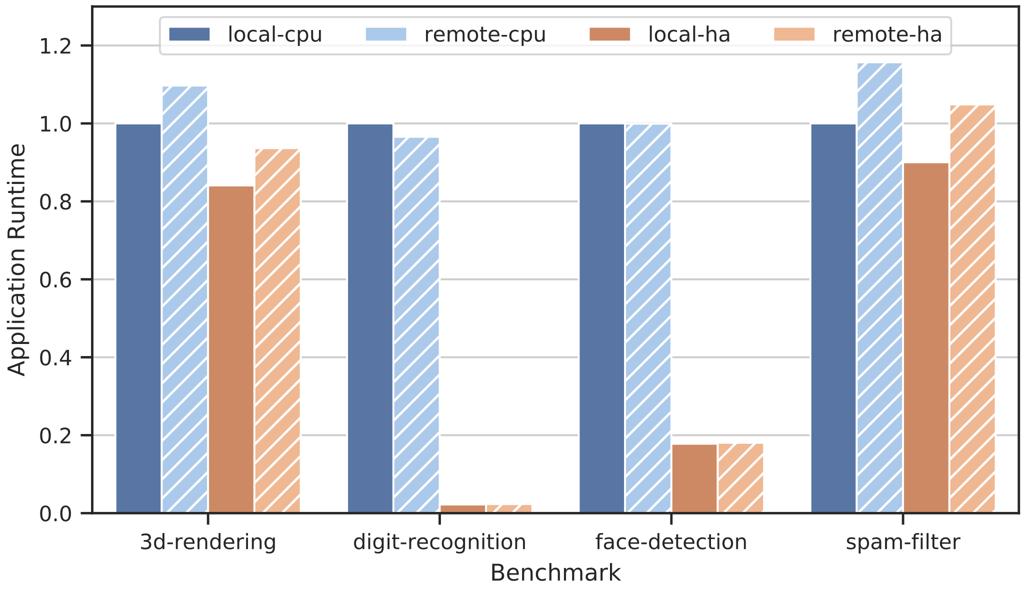


Figure 6.1: Total benchmark application runtime on each target type. Runtime is shown relative to the local CPU runtime for each benchmark.

plot, we see that digit-recognition exhibits the greatest total application speedup, which we attribute to the significant speedup of its accelerator.

In contrast, spam-filter achieves the poorest application speedup when its kernel is executed on a hardware accelerator. This is because spam-filter has a very large input data set that must be copied to a direct memory access (DMA) buffer before the accelerator starts. The overhead of this required DMA transfer significantly impacts the accelerator speedup. Furthermore, we note that the spam-filter benchmark executes faster on the local CPU than on a remote accelerator. This is again a consequence of the large data set that must be transferred over the network to the remote node, before execution may begin.

Tables 6.8 through 6.11 summarize the results of the baseline benchmark characterization across each of the possible kernel execution devices. Each table shows the total benchmark application runtime (`client::program_total`), the time for the kernel execution as measured by the client application (`client::enqueue_kernel`), followed by a breakdown of the kernel execution into each of the events within the master and worker daemons.

Comparing the results shown in these tables, we find that the time to schedule a kernel and update the scheduler data structures is independent of the benchmark, as we would expect. Both the time to schedule and the time to update the scheduler, take on the order of 10 µs.

Furthermore, we notice that both digit-recognition and face-detection achieve slightly better runtime on a remote CPU than on the local CPU. This is because the local CPU is used to run application host code, kernels as well as the master daemon, whereas the remote nodes are only used to run application kernels. Therefore, there is less contention for the

Table 6.8: 3d-rendering runtime breakdown on each target device

	Local Worker Runtime (s)		Remote Worker Runtime (s)	
`Event`	CPU	HW Accel	CPU	HW Accel
`client::program_total`	0.058 538	0.049 216	0.064 222	0.054 823
`client::enqueue_kernel`	0.018 502	0.008 113	0.024 167	0.013 753
`master::read_kernel_request_data`	0.000 089	0.000 090	0.000 090	0.000 090
`master::schedule`	0.000 008	0.000 011	0.000 008	0.000 012
`worker::read_kernel_request_data`	0.000 070	0.000 069	0.000 138	0.000 251
`worker::execute_kernel`	0.016 442	0.006 061	0.016 335	0.005 914
`master::read_kernel_response_data`	0.000 177	0.000 178	0.004 877	0.004 683
`master::update_scheduler`	0.000 016	0.000 027	0.000 015	0.000 026

Table 6.9: digit-recognition runtime breakdown on each target device.

	Local Worker Runtime (s)		Remote Worker Runtime (s)	
`Event`	CPU	HW Accel	CPU	HW Accel
`client::program_total`	8.099 577	0.179 292	7.820 986	0.188 164
`client::enqueue_kernel`	7.948 584	0.028 190	7.669 979	0.037 024
`master::read_kernel_request_data`	0.002 571	0.002 577	0.002 588	0.002 643
`master::schedule`	0.000 010	0.000 014	0.000 010	0.000 014
`worker::read_kernel_request_data`	0.003 318	0.003 480	0.012 235	0.010 911
`worker::execute_kernel`	7.940 850	0.020 213	7.644 529	0.020 051
`master::read_kernel_response_data`	0.000 036	0.000 036	0.000 036	0.000 038
`master::update_scheduler`	0.000 014	0.000 026	0.000 014	0.000 027

Table 6.10: face-detection runtime breakdown on each target device type.

	Local Worker Runtime (s)		Remote Worker Runtime (s)	
`Event`	CPU	HW Accel	CPU	HW Accel
`client::program_total`	0.430 078	0.076 566	0.429 810	0.077 616
`client::enqueue_kernel`	0.401 483	0.048 141	0.401 162	0.049 212
`master::read_kernel_request_data`	0.000 215	0.000 220	0.000 219	0.000 218
`master::schedule`	0.000 008	0.000 012	0.000 009	0.000 012
`worker::read_kernel_request_data`	0.000 225	0.000 231	0.000 879	0.000 603
`worker::execute_kernel`	0.399 521	0.046 155	0.397 704	0.045 914
`master::read_kernel_response_data`	0.000 042	0.000 039	0.000 040	0.000 039
`master::update_scheduler`	0.000 014	0.000 025	0.000 014	0.000 024

Table 6.11: spam-filter runtime breakdown on each target device type.

	Local Worker Runtime (s)		Remote Worker Runtime (s)	
`Event`	CPU	HW Accel	CPU	HW Accel
`client::program_total`	1.302 164	1.172 340	1.506 367	1.365 376
`client::enqueue_kernel`	1.160 857	1.030 938	1.364 707	1.223 124
`master::read_kernel_request_data`	0.086 566	0.087 227	0.087 524	0.087 886
`master::schedule`	0.000 010	0.000 013	0.000 011	0.000 014
`worker::read_kernel_request_data`	0.109 411	0.109 445	0.314 278	0.308 722
`worker::execute_kernel`	0.955 475	0.824 684	0.949 263	0.815 186
`master::read_kernel_response_data`	0.000 038	0.000 038	0.004 179	0.002 116
`master::update_scheduler`	0.000 013	0.000 025	0.000 014	0.000 027

CPU and caches on remote nodes leading potentially better runtime for longer running compute-bound benchmarks.

6.3 Scheduler Policies

For our evaluation, we implemented three different scheduling policies with varying levels of sophistication and awareness of the cluster system. These policies are: *Round-Robin CPU-only*, *Prefer Hardware Accelerator* and *Oracle*. In the following subsections, we describe the implementation and the scheduling objective of each.

6.3.1 Round-Robin CPU-only Policy

The Round-Robin CPU-only (RR-CPU) scheduler always selects CPU targets and uses a round-robin queue in an attempt to distribute requests among the available nodes. It does not distinguish between the local CPU from CPUs on other nodes. The design of RR-CPU is purposefully simplistic to provide a baseline for our later evaluation of more complex scheduling policies.

6.3.2 Prefer Hardware Accelerator Policy

The goal of the Prefer Hardware Accelerator (Prefer-HA) policy is to prioritize scheduling kernels onto hardware accelerators over CPUs. The scheduler maintains a separate queue of idle accelerators for each kernel type and a round-robin queue for all CPUs. Upon receiving a kernel request, the scheduler inspects the accelerator queue corresponding to the requested kernel type. The next accelerator in this queue is selected as the target and removed from the queue. If the accelerator queue is empty, meaning there are matching accelerators that are idle, then the next CPU from the round-robin CPU queue is used as the target. When a kernel response is received, the accelerator that executed the request is pushed onto the appropriate idle queue, as this accelerator is now idle.

As with the RR-CPU scheduler, Prefer-HA does not differentiate the available devices by network locality. This scheduler is the most straightforward re-implementation of the scheduler from the original Front-end USEr framework (FUSE) work for use with a cluster.

6.3.3 Oracle Policy

The Oracle scheduler is designed to balance the following factors when scheduling: network data transfer overhead, hardware accelerator speedup and worker load. To do so, this scheduler classifies all of the targets in the cluster into one of the following types:

- local CPU
- remote CPU
- local accelerator
- remote accelerator

The primary feature of the Oracle scheduler is the use of estimates of total kernel runtime on each of these possible target types. For each kernel request, the Oracle scheduler attempts to determine the target with the lowest estimated runtime. To perform these estimates the Oracle scheduler uses a formula of the form in Equation 6.1.

$$T_{total} = T_{data\_transfer} + load\_factor \times T_{runtime} \tag{6.1}$$

We note that the form of this equation stipulates that the data transfer of kernel arguments is not overlapped with kernel execution. In our prototype implementation, the worker does not begin executing a kernel until the network transfer of kernel argument data from the master has completed. However, the hardware accelerators do overlap DMA buffer accesses with computation during their operation.

As we determined from the characterization in Section 5.3, master to worker data transfer time is variable and largely dependent on the amount of the kernel data to be transferred. We use the results of the data transfer characterization to provide the Oracle scheduler a means with which to estimate data transfer times. The scheduler computes an estimate of transfer time using the segmented regression model we derived.

To estimate $T_{runtime}$, we performed characterization runs of each benchmark kernel on both a CPU device and a hardware accelerator. For our initial implementation, this runtime data was statically added to the Oracle scheduler implementation in the form of a software look-up table. More sophisticated methods for estimating kernel runtime on different target devices can be explored in the future.

The purpose of the $load\_factor$ is to scale the estimated kernel execution time depending on the current CPU load on each worker. Intuitively, a worker that is handling many kernel requests (higher load), will take longer to handle an additional request compared to an idle worker. This is due to anticipated contention for shared resources including caches, the network interface and the system CPU.

6.4 Workload Experiments

For an evaluation of the CFUSE framework, we build workloads consisting of multiple application benchmarks and execute these workloads on the prototype cluster. For these experiments, we set up the prototype cluster with four ZC706 boards. One board is designated the master and is used to host the CFUSE master daemon and the remaining three boards are designated as remote workers. We also run an instance of a worker daemon on the same node as the master, known as the local worker. The workload applications are launched on the master node.

We use three classes of workloads: *compute workloads* consisting of compute-bound benchmarks, *memory workloads* consisting of only memory-bound benchmarks and *mixed workloads* that contain a combination of the two benchmark types. We also vary the number of applications per workload in multiples of 4, ranging from 4 to 20, while the specific applications used within each workload are chosen randomly from the appropriate category. Table 6.12 lists each workload and shows the applications contained in each.

Table 6.12: Summary of benchmark application workloads

		Applications in Workload			
Number of Applications	Workload Category	3d-rendering	digit-recognition	face-detection	spam-filter
4	compute	2	1	1	-
4	memory	-	-	-	4
4	mixed	-	-	2	2
8	compute	3	2	3	-
8	memory	-	-	-	8
8	mixed	2	1	2	3
12	compute	3	4	5	-
12	memory	-	-	-	12
12	mixed	4	4	1	3
16	compute	3	5	8	-
16	memory	-	-	-	16
16	mixed	5	4	4	3
20	compute	7	9	4	-
20	mixed	2	4	8	6

We note that there is no memory workload with 20 applications. This is due to the large data requirements of the spam-filter benchmark. Due to the limited amount of system memory on the embedded boards, we are unable to run the 20 instances of the spam-filter benchmark simultaneously without experiencing memory allocation failures.

For each application, we record the start time (t_{start}) and stop time (t_{stop}), which can be used to determine the total runtime of each application. The total workload execution time ($t_{workload}$), can also be computed from these application times using Equation 6.2.

$$t_{workload} = \max\{t_{stop1}, t_{stop2}, \dots\} - \min\{t_{start1}, t_{start2}, \dots\} \tag{6.2}$$

6.4.1 FPGA Configurations

Part of the experimental setup for the cluster involves choosing a configuration for each FPGA in the cluster, which determines the set of available hardware accelerators. Here we describe our two-step process for selecting these initial configurations for each workload.

To limit the total possible number of configurations, we started with a set of four FPGA configurations each containing different sets of accelerators, but prioritizing the inclusion of accelerator instances from a single benchmark. For example, one of the four configurations was designated to prioritize including 3d-rendering accelerators. We then added accelerators for the other benchmark kernels to the configuration to fill the remaining FPGA area, based on the runtime performance impact of an accelerator on its corresponding benchmark application. The runtime performance impact is estimated using Amdahl's Law. Table 6.13 lists the four configurations and the number of accelerator instances of each type contained in each.

Table 6.13: Hardware accelerator instances within each FPGA configuration

Configuration	Accelerator Instances			
	3d-rendering	digit-recognition	face-detection	spam-filter
R	3	1	2	-
DR	1	2	1	-
FD	1	1	1	-
SF	1	1	1	3

With these configurations created, the next step is to select the set of configurations to use on the cluster board for each of the workloads. The process here is similar to the first step, however, we now factor in the locality of each accelerator.

6.4.2 Workload Results and Discussion

Figure 6.2 shows the total runtime for all workloads using each of the scheduling policies. Figures 6.3, 6.4 and 6.5 show the same data, but with the workload types split over three plots. We find that the memory workloads and to a lesser extent the mixed workloads, do not benefit as much when using a scheduler that utilizes accelerator cores. This is largely due to the poor speedup of the spam-filter accelerator over the CPU implementation, which in turn is a direct result of the large data set size of this benchmark. The spam-filter accelerator suffers from high overhead from the required use of large DMA buffers, which significantly affects the overall speedup.

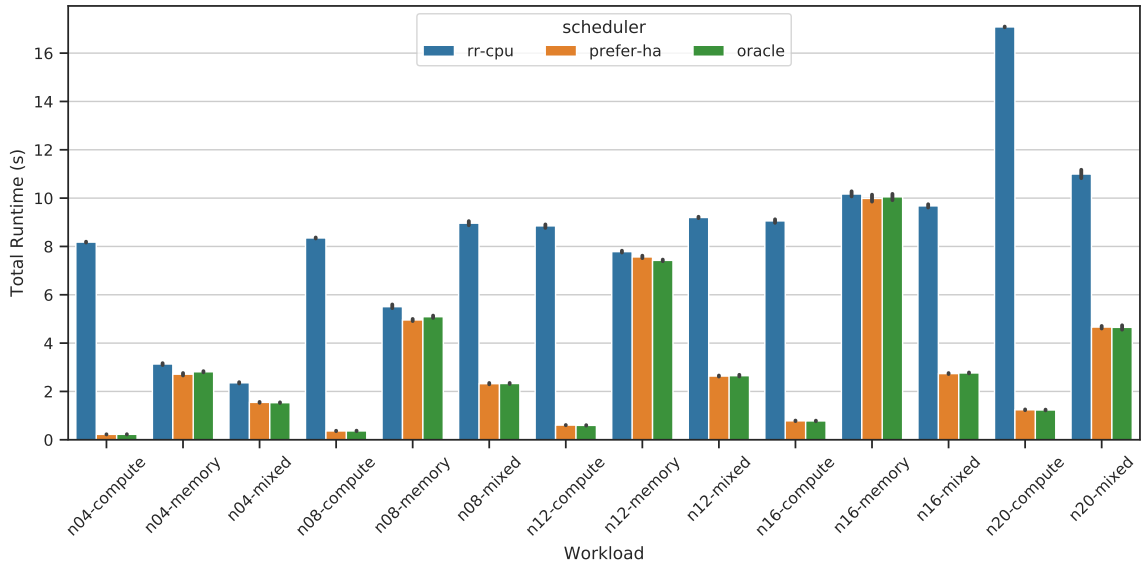


Figure 6.2: Total execution time per workload

The compute workload time for the RR-CPU scheduler is dominated by the number of digit-recognition applications in the workload. This benchmark has a significantly longer running time of approximately 8 seconds on the CPU. There are enough CPU cores on the platform, with two on each node, to allow for the parallel execution of this benchmark for all workloads except for workload n20-compute. This is the only workload that includes more than 8 instances of this benchmark and therefore the only compute workload with significantly longer runtime with the RR-CPU scheduler.

We also notice that memory and mixed workloads exhibit a larger amount of variability in total workload runtime, between individual workload runs. This is likely because these workloads contain the memory-bound spam-filter application. Spam-filter utilizes a large data set which must be transferred between master and worker and it is likely this variability in data transfers that increases the overall workload variability for these workload types.

We see from the plot that the Prefer-HA and Oracle schedulers generally perform similarly, with only slight differences in overall workload runtime between the two. For a clearer comparison, Figure 6.6 shows the same workload runtime data for only the Prefer-HA and Oracle schedulers.

The Prefer-HA scheduler prioritizes accelerator targets above all else and does not factor in the network location of a target device. For our experimental setup, this proves to be largely sufficient for decreasing the total workload execution time. This is a result of the fact that on the prototype cluster the penalty for remote data transfer is relatively small for

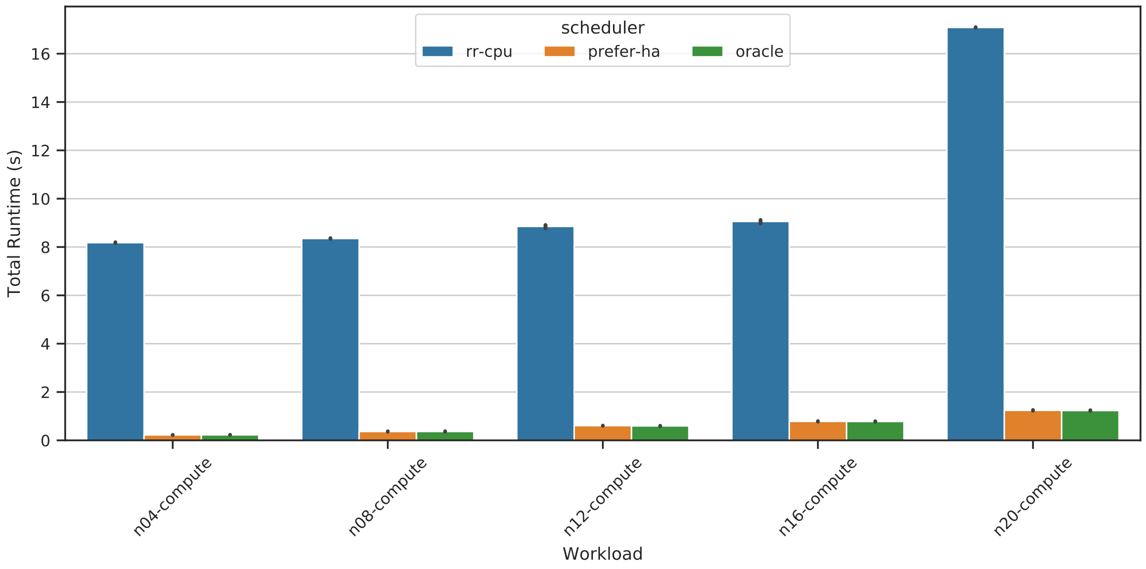


Figure 6.3: Total execution time of compute workloads

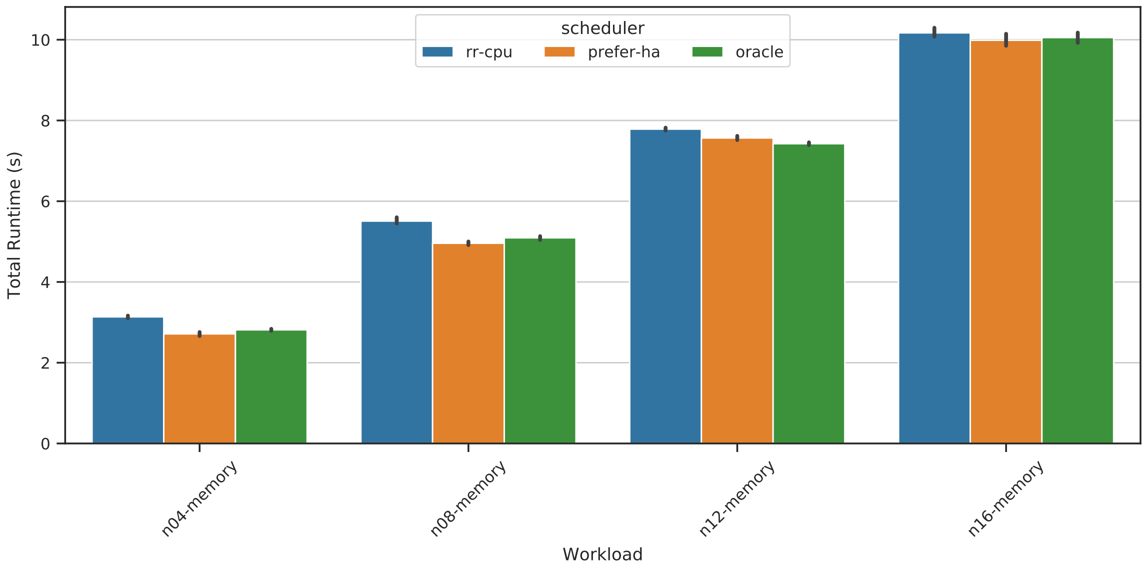


Figure 6.4: Total execution time of memory workloads

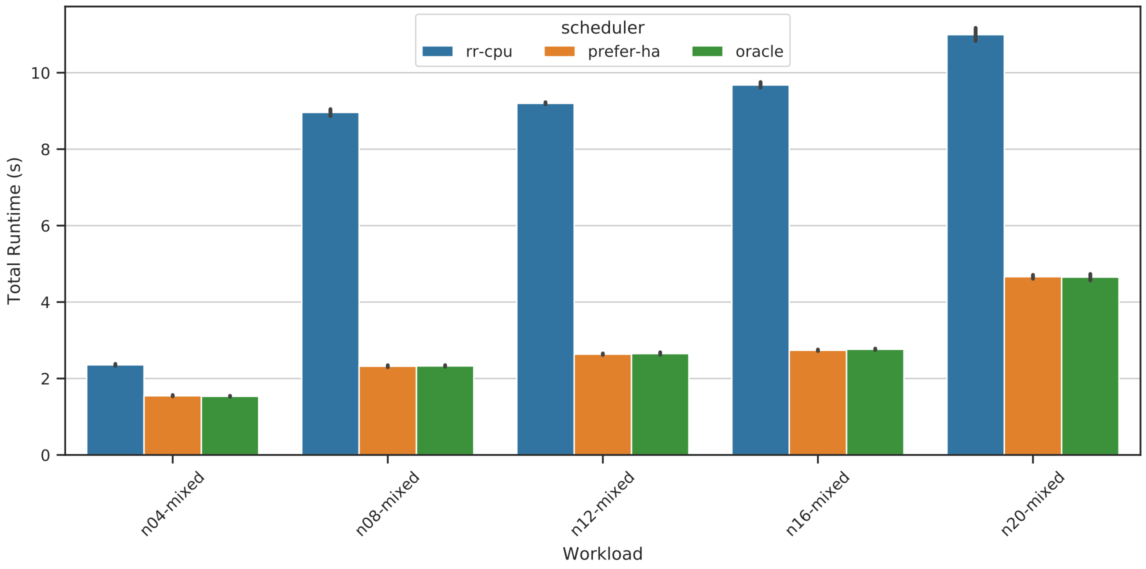


Figure 6.5: Total execution time of mixed workloads

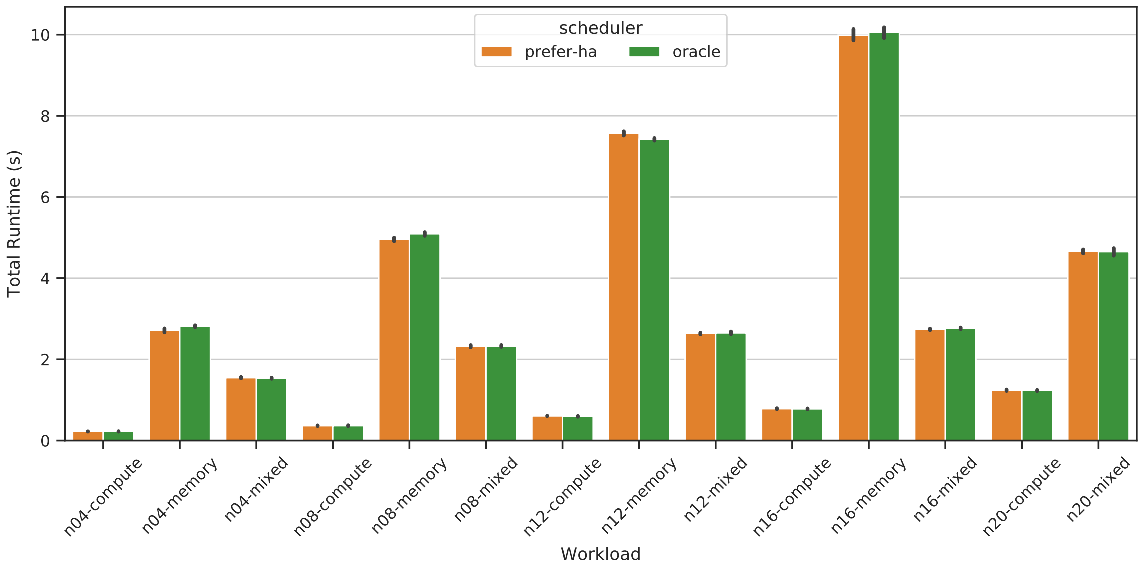


Figure 6.6: Total execution time per workload: Prefer-HA and Oracle schedulers only

smaller data sizes. That is, data transfer time to a remote worker is generally not a significant portion of the overall kernel runtime particularly for small data sizes. The majority of the benchmarks use data set sizes within this smaller range. Furthermore, the cluster generally has enough accelerator resources to run the workload application kernels and so Prefer-HA and Oracle will therefore usually make the same scheduling decision.

The few workloads where the Oracle scheduler outperforms Prefer-HA, all involve the spam-filter benchmark. This benchmark is the only one with a kernel that executes faster on a local CPU than on a remote accelerator, although the difference is small. By using its estimates of total kernel runtime including data transfer, the Oracle scheduler can detect this case and schedule appropriately. We presume that given a larger difference between remote and local transfer time, there would be more benchmarks that also exhibit this behaviour. The benefit of the Oracle scheduler over Prefer-HA would be more pronounced in this case.

Chapter 7

Conclusions and Future Work

OpenCL has helped to raise the level of design entry for field-programmable gate arrays (FPGAs) above the traditional register-transfer level (RTL) approach, and has made access to high-level synthesis (HLS) technology accessible to non-experts. However, limitations in the OpenCL models and the unique reconfigurable architecture of FPGA devices, make the simultaneous use of FPGA and central processing unit (CPU) resources challenging using OpenCL. This is particularly true for clusters of systems that feature general-purpose processors and attached reconfigurable hardware, which are becoming more prevalent in cloud environments.

7.1 Conclusions

In this thesis, we presented the Cluster Front-end USEr framework (CFUSE) that enables scheduling OpenCL kernels from multiple applications across FPGA and CPU devices within a networked cluster. Basing our work on the Front-end USEr framework (FUSE) concept, we were able to remove the burden of device selection from the application developer. We presented a prototype implementation of CFUSE on a cluster of FPGA development boards, with a runtime library to simplify the simultaneous use of multiple OpenCL devices.

We performed a series of experiments to characterize the prototype system, including an evaluation of network data transfer and direct memory access (DMA) buffer overhead. Finally, we presented a comparison of three scheduling policies using workloads comprised of applications from both the image processing and machine learning domains. Our findings indicate that scheduling on our prototype is sensitive to data transfer overheads and hardware accelerator speedup.

7.2 Future Work

One of the primary limitations imposed by the prototype platform is the lack of an I/O memory management unit (IOMMU) that would allow accelerators direct access to user-space memory pages. As such, we required the use of physically contiguous DMA buffers allocated by the operating system. The use of these buffers adds considerable overhead particularly for memory-bound application kernels executed via CFUSE. Therefore, a logical improvement to the system would be to investigate IOMMU usage on the prototype. The simplest design for integrating an IOMMU into the current prototype would likely involve using a software-managed IOMMU per accelerator core. Such a design would allow accelerator cores to directly access user-space virtual memory, avoiding the need for the use of physically contiguous DMA buffers and associated overhead.

Furthermore, we would like to investigate other techniques for inter-process communication between the master daemon and client applications, including moving some functionality of the master daemon into the operating system (OS) kernel. Also related to the area of improving communications performance and lowering associated overhead, would be to employ a dedicated hardware connection between FPGAs on remote hosts.

We would like to continue experimentation with more benchmark applications from a greater variety of application domains and with larger, more varied sample data sets. Using more complex benchmarks that contain multiple interdependent kernels would likely provide useful insight to the scheduling problem and more opportunity for useful optimizations. A potential difficulty with finding suitable benchmarks is that we require implementations of kernel functions both for the CPU and that are suitable for use with HLS tools.

Therefore, a tangential research area worth investigating is the use of a single source code specification for kernel functions. This is particularly important for a framework such as CFUSE, which must manage execution across multiple types of devices. OpenCL does provide a portable kernel language for this purpose. However, to ensure reasonable execution performance, a developer must often customize a kernel to match the features of the specific target device, which can vary significantly between CPUs, graphics processing units (GPUs) and FPGAs. This is particularly true for kernels targeting FPGAs, as most FPGA OpenCL platforms recommend the use of single work-item kernels and platform-specific extensions to enable HLS optimizations.

Finally, there is also area for further research on additional scheduling policies. Of particular interest are classes of schedulers utilizing the runtime reconfiguration capability of the FPGA. This would allow the cluster to adapt the set of available accelerators to specific workload demands.

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Jun. 2014, pp. 13–24.

[2] "Amazon EC2 F1 Instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/

[3] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[4] A. Ismail and L. Shannon, "FUSE: Front-end user framework for O/S abstraction of hardware accelerators," in *Proceedings of the 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2011, pp. 170–177.

[5] Xilinx Inc., *SDAccel Environment User Guide*, May 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1023-sdaccel-user-guide.pdf

[6] Intel Inc., *Intel FPGA SDK for OpenCL Programming Guide*, Sep. 2019. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[7] Xilinx Inc., *Vivado Design Suite User Guide: High-Level Synthesis*, Nov. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug902-vivado-high-level-synthesis.pdf

[8] P. Jääskeläinen, C. S. d. L. Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "Pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, Oct. 2015.

[9] Khronos OpenCL Working Group, *The OpenCL Specification*, Nov. 2012. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf

[10] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proceedings of SC'16: The International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2016, pp. 409–420.

[11] Intel Inc., *Intel High Level Synthesis Compiler: Best Practices Guide*, Sep. 2019. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls-best-practices.pdf

[12] Xilinx Inc., *Vivado HLS Optimization Methodology Guide*, Apr. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf

[13] Intel Inc., *Intel FPGA SDK for OpenCL Pro Edition: Best Practices Guide*, Sep. 2019. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf

[14] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 1967, pp. 483–485.

[15] "Clang C Language Family Frontend for LLVM." [Online]. Available: https://clang.llvm.org/

[16] "The LLVM Compiler Infrastructure Project." [Online]. Available: http://llvm.org/

[17] B. Taylor, V. S. Marco, and Z. Wang, "Adaptive optimization for OpenCL programs on embedded heterogeneous systems," in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Jun. 2017, pp. 11–20.

[18] A. B. Perina and V. Bonato, "Mapping estimator for OpenCL heterogeneous accelerators," in *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, Dec. 2018, pp. 294–297.

[19] A. M. Aji, A. J. Peña, P. Balaji, and W. Feng, "Automatic command queue scheduling for task-parallel workloads in OpenCL," in *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, Sep. 2015, pp. 42–51.

[20] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *Proceedings of the 2014 21st International Conference on High Performance Computing (HiPC)*, Dec. 2014, pp. 1–10.

[21] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices," in *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2014, pp. 273–283.

[22] S. Lee and C. Wu, "Performance characterization, prediction, and optimization for heterogeneous systems with multi-level memory interference," in *Proceedings of the 2017 IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2017, pp. 43–53.

[23] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, Jun. 2012, pp. 341–352.

[24] R. Aoki, S. Oikawa, T. Nakamura, and S. Miki, "Hybrid OpenCL: Enhancing OpenCL for distributed processing," in *Proceedings of the 2011 IEEE 9th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, May 2011, pp. 149–154.

[25] P. Kegel, M. Steuwer, and S. Gorlatch, "dOpenCL: Towards a uniform programming approach for distributed heterogeneous multi-/many-core systems," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, May 2012, pp. 174–186.

[26] B. Eskikaya and D. T. Altılar, "Distributed OpenCL distributing OpenCL platform on network scale," *IJCA Special Issue on Advanced Computing and Communication Technologies for HPC Applications*, vol. ACCTHPCA, no. 2, pp. 25–30, Jun. 2012.

[27] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger, "DistCL: A framework for the distributed execution of OpenCL kernels," in *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Aug. 2013, pp. 556–566.

[28] M. Plauth, F. Rösler, and A. Polze, "CloudCL: Distributed heterogeneous computing on cloud scale," in *Proceedings of the 2017 5th International Symposium on Computing and Networking (CANDAR)*, Nov. 2017, pp. 344–350.

[29] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer, "LibWater: Heterogeneous distributed computing made easy," in *Proceedings of the 27th International ACM Conference on Supercomputing (ICS)*, Jun. 2013, pp. 161–172.

[30] G. Staples, "TORQUE resource manager," in *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*, 2006.

[31] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer, 2003, pp. 44–60.

[32] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei, "When Apache Spark meets FPGAs: A case study for next-generation DNA sequencing acceleration," in *Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, Jun. 2016.

[33] E. Ghasemi and P. Chow, "Accelerating Apache Spark big data analysis with FPGAs," in *Proceedings of the 2016 International IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, Jul. 2016, pp. 737–744.

[34] M. Huang, D. Wu, C. H. Yu, Z. Fang, M. Interlandi, T. Condie, and J. Cong, "Programming and runtime support to blaze FPGA accelerator deployment at datacenter

scale," in *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, Oct. 2016, pp. 456–469.

[35] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, and P. Chow, "Enabling flexible network FPGA clusters in a heterogeneous cloud data center," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2017, pp. 237–246.

[36] N. Eskandari, N. Tarafdar, D. Ly-Ma, and P. Chow, "A modular heterogeneous stack for deploying FPGAs and CPUs in the data center," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2019, pp. 262–271.

[37] Xilinx Inc., *SDAccel Programmers Guide*, May 2019. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1277-sdaccel-programmers-guide.pdf

[38] M. Lin, I. Lebedev, and J. Wawrzynek, "OpenRCL: Low-power high-performance computing with reconfigurable devices," in *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2010, pp. 458–463.

[39] S. Ma, M. Huang, and D. Andrews, "Developing application-specific multiprocessor platforms on FPGAs," in *Proceedings of the 2012 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec. 2012, pp. 1–6.

[40] K. Shagrithaya, K. Kepa, and P. Athanas, "Enabling development of OpenCL applications on FPGA platforms," in *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, Jun. 2013, pp. 26–30.

[41] M. Hosseinabady and J. L. Nunez-Yanez, "Optimised OpenCL workgroup synthesis for hybrid ARM-FPGA devices," in *Proceedings of the 2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2015, pp. 1–6.

[42] V. Mirian and P. Chow, "UT-OCL: An OpenCL framework for embedded systems using Xilinx FPGAs," in *Proceedings of the 2015 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec. 2015, pp. 1–6.

[43] Xilinx Inc., *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC User Guide*, Aug. 2019. [Online]. Available: https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf

[44] ——, *Zynq-7000 SoC Technical Reference Manual*, Jul. 2018. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf

[45] TP-Link, *5/8-Port 10/100/1000Mbps Desktop Switch: TL-SG105/TL-SG108*, 2015. [Online]. Available: https://static.tp-link.com/res/down/doc/TL-SG105-108.pdf

[46] ARM, *AMBA AXI and ACE Protocol Specification: AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite*, 2013. [Online]. Available: https://static.docs.arm.com/ihi0022/e/IHI0022E_amba_axi_and_ace_protocol_spec.pdf

[47] "Asio C++ Library." [Online]. Available: https://think-async.com/Asio/

[48] "Buildroot - Making Embedded Linux Easy." [Online]. Available: https://buildroot.org/

[49] ARM, *Cortex-A9 MPCore Technical Reference Manual*, 2011. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0407g/index.html

[50] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, W. Wang, and Z. Zhang, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, Feb. 2018, pp. 269–278.

[51] "SLEEF Vectorized Math Library." [Online]. Available: https://sleef.org/

Appendix A

OpenCL Application Example

This appendix contains the source code for an example OpenCL host application that performs vector addition. The source code consists of two files: `opencl_example.c`, which contains the host code and `vector_add.cl`, which contains the OpenCL C kernel code for a vector addition kernel.

The host code illustrates the use of the OpenCL application programming interface (API) for setting up a single device to run the vector addition kernel. This kernel computes the sum of two integer arrays. Furthermore, the example code shows how program objects can be created from either a kernel source file or from a pre-compiled binary file. Although not strictly necessary in this simple example, OpenCL event objects are used to synchronize the commands submitted to the device queue to illustrate the technique. Note that handling of potential errors from the OpenCL functions is omitted for brevity.

A.1 opencl_example.c

```c
#include <stdlib.h>
#include <CL/cl.h>
#include "read_file.h" /* defines read_file() helper function */

int main(void)
{
  /* Initialize host memory buffers: will be used as kernel arguments */
  const size_t array_size = 8;
  int A[] = {0, 1, 2, 3, 4, 5, 6, 7};
  int B[] = {0, 2, 4, 6, 8, 10, 12, 14};
  int sum[] = {0, 0, 0, 0, 0, 0, 0, 0};

  /* Get number of available OpenCL platforms */
  cl_uint num_platforms = 0;
  clGetPlatformIDs(0, NULL, &num_platforms);

```

```
    /* Get list of platform IDs */
    cl_platform_id *platform_ids;
    platform_ids = calloc(sizeof(cl_platform_id), num_platforms);
    clGetPlatformIDs(num_platforms, platform_ids, NULL);

    /* Use first available platform */
    cl_platform_id platform = platform_ids[0];

    /* Query for number of devices in platform */
    cl_uint num_devices = 0;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);

    /* Get list of device IDs */
    cl_device_id *device_ids = calloc(sizeof(cl_device_id), num_devices);
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, device_ids, NULL);

    /* Use first available device */
    cl_device_id device = device_ids[0];

    /* Create context */
    cl_context_properties properties[] = {CL_CONTEXT_PLATFORM,
                                          (cl_context_properties)platform, 0};
    cl_context context = clCreateContext(properties, num_devices, device_ids,
                                         NULL, NULL, NULL);

    /* Create command queue (use default properties) */
    cl_command_queue queue = clCreateCommandQueueWithProperties(context, device,
                                                                NULL, NULL);

    /* Setup memory objects */
    size_t buffer_size = sizeof(int) * array_size;
    cl_mem A_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, buffer_size,
                                     NULL, NULL);
    cl_mem B_buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, buffer_size,
                                     NULL, NULL);
    cl_mem sum_buffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, buffer_size,
                                       NULL, NULL);

    /* Create program object ... */
    char *kernel_file;
    size_t file_size;
    cl_program program;
  #if defined(USE_KERNEL_SOURCE)
    /* ... using kernel source code */
    kernel_file = read_file("vector_add.cl", &file_size);
    program = clCreateProgramWithSource(context, 1, (const char **)&kernel_file,
                                        &file_size, NULL);
```

```
#elif defined(USE_KERNEL_BINARY)
  /* ... using precompiled kernel binary */
  kernel_file = read_file("vector_add.bin", &file_size);
  program = clCreateProgramWithBinary(context, 1, &device, &file_size,
                                      (const unsigned char **)&kernel_file,
                                      NULL, NULL);
#endif

  /* Build program */
  clBuildProgram(program, 1, &device, NULL, NULL, NULL);

  /* Create kernel object */
  cl_kernel kernel = clCreateKernel(program, "vector_add", NULL);

  /* Set kernel arguments by index */
  clSetKernelArg(kernel, 0, sizeof(cl_mem), &A_buffer);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &B_buffer);
  clSetKernelArg(kernel, 2, sizeof(cl_mem), &sum_buffer);

  /* Write kernel input argments (A and B buffers) to device memory */
  cl_event A_event, B_event;
  cl_bool blocking = CL_FALSE; /* use non-blocking commands */
  clEnqueueWriteBuffer(queue, A_buffer, blocking, 0, buffer_size, A,
                       0, NULL, &A_event);
  clEnqueueWriteBuffer(queue, B_buffer, blocking, 0, buffer_size, B,
                       0, NULL, &B_event);
  cl_event write_events[] = {A_event, B_event};

  /* Execute kernel */
  size_t global_work_size[1] = {array_size};
  size_t local_work_size[1] = {1};
  cl_event kernel_event;
  clEnqueueNDRangeKernel(queue, kernel, 1,
                         NULL, global_work_size, local_work_size,
                         /* wait on input buffers to be written */
                         2, write_events, &kernel_event);

  /* Read kernel output arguments (sum buffer) to host memory */
  cl_event read_event;
  clEnqueueReadBuffer(queue, sum_buffer, blocking, 0, buffer_size, sum,
                      1, &kernel_event, /* wait on kernel execution command */
                      &read_event);
  clWaitForEvents(1, &read_event);

  /* Release OpenCL resources */
  clReleaseMemObject(A_buffer);
  clReleaseMemObject(B_buffer);
  clReleaseMemObject(sum_buffer);
  clReleaseKernel(kernel);
  clReleaseProgram(program);
  clReleaseCommandQueue(queue);
  clReleaseContext(context);
}
```

A.2 vector_add.cl

```
__kernel void vector_add(__global int *a, __global int *b, __global int *sum)
{
  size_t i = get_group_id(0);
  sum[i] = a[i] + b[i];
}
```