Thesis Title: Integrated framework for heterogeneous embedded platforms using OpenCL

Author: Kulin Seth

Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree:

Thesis Advisor: Prof. David Kaeli

Thesis Committee: Prof. Gunar Schirner

Thesis Committee: Dr. Rafael Ubal

Head of Department:

Graduate School Notified of Acceptance:

Associate Dean of Engineering: Yaman Yener
INTEGRATED FRAMEWORK FOR HETEROGENEOUS EMBEDDED PLATFORMS USING OPENCL

A Thesis Presented
by
Kulin Seth
to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements for the degree of
Master of Science in Electrical and Computer Engineering
Northeastern University
Boston, Massachusetts
March 2011
Abstract

The technology community is rapidly moving away from the age of computers and laptops, and is entering the emerging era of hand-held devices. With the rapid development of smart phones, tablets, and pads, there has been widespread adoption of Graphic Processing Units (GPUs) in the embedded space. The hand-held market is now seeing an ever-increasing rate of development of computationally intensive applications, which require significant amounts of processing resources. To meet this challenge, GPUs can be used for general-purpose processing. We are moving towards a future where devices will be more connected and integrated. This will allow applications to run on handheld devices, while offloading computationally intensive tasks to other compute units available. There is a growing need for a general programming framework which can utilize heterogeneous processing units such as GPUs and DSPs on embedded platforms. OpenCL, a widely used programming framework has been a step towards integrating these different processing units on desktop platforms. Extending the use of OpenCL to the embedded space can potentially lead to the development of a new class of applications in the embedded domain. This thesis describes our efforts made in this direction. The main idea behind this thesis is to utilize OpenCL to benefit embedded applications as run on GPUs. This work provides an integrated toolchain, with a full-system simulation environment to support a platform
with an ARM device and embedded GPU on it. The use of an integrated framework provides visibility and extensibility to perform end-to-end study of these platforms. This thesis pursues different levels of optimizations that can be carried out, namely source, compiler and micro-architectural level optimizations. Case studies presented consider the interaction between these levels and provide guidelines as a result of this student. The final goal of the thesis is to study performance improvements by running computationally intensive tasks on embedded GPU. Over 20 benchmarks, taken from OpenCL SDKs, are studied in this work. They are broadly categorized as signal processing kernels, image processing applications and general computational task. The simulated results were compared between different configurations of the GPU with CPU as the reference and an average speedup of around 348 times was seen for kernel execution time. This thesis work can be used as a research tool to study GPGPU computing on embedded devices.
Acknowledgements

I would like to acknowledge Prof. David Kaeli for being patient with me and supporting me to finish this project. His guidance and enthusiasm for the work kept going thorough the dark phases of the project.

I would like to acknowledge the open-source tools OVPsim, GPGPU-Sim without which this project wouldn’t have been possible.

Lastly and most importantly I would like to thank my wife Preethi for being supportive and understanding for long nights in the lab and providing comments and valuable feedback on the thesis.
Contents

Abstract iv

Acknowledgements vi

List of Figures x

List of Tables xii

1 Introduction 1

1.1 Motivation 3

1.2 Contributions 5

1.3 Related Work 6

1.4 Outline of the Thesis 10

2 OpenCL Programming Framework 11

2.1 What is OpenCL? 12

2.1.1 Platform Model 12

2.1.2 Execution Model 13

2.1.3 Memory Model 14

2.1.4 Programming Model 15
3 Simulation Architecture

3.1 Overview

3.2 ARM Model

3.2.1 Simulation Details

3.3 GPU Model

3.3.1 GPGPUsim

3.3.2 PTX Parser

3.4 OVP Simulation Environment

4 Methodology and Optimizations

4.1 Implementation Overview

4.1.1 Installation & Initial Testing

4.1.2 Reading Code

4.1.3 Restructured Simulator

4.1.4 Separating ARM Model and GPU Model

4.1.5 OVPsim Environment

4.2 Optimization Framework

5 Discussion

5.1 Benchmarks Overview
List of Figures

1.1 Embedded GPU hardware battling out in the handheld space 4
1.2 Latency (µs): kernel execution time of ARM vs GPU (baseline) 5.3 6

2.1 OpenCL Platform model: Host with Compute devices, Compute Units and Processing elements 13
2.2 OpenCL Memory model: shows different memory spaces 14
2.3 OpenCL framework divided into blocks of OpenCL application, OpenCL runtime with compiler support and the target platform 18
2.4 OpenCL runtime and API calls 19
2.5 OpenCL compilation flow with following parts: (A) The compilation flow for the host device, (B) compilation flow for GPU which is offline till PTX generation and online compiling PTX to GPU binary, (C) subset of part B, PTX statistics for the scheduler, (D) is the platform with CPU and GPU 20

3.1 Hardware vendors for embedded GPUs: (A) ARM with Mali GPUs, (B) Imagination Technologies, Power VR SGX, (C) ZiiLabs, (D) NVIDIA with Tegra GPUs 24
3.2 Overall block diagram of the simulator: (i) ARM 9 ISS model from OVPsim, (ii) GPU model extended from GPGPUsim (iii) OVPsim simulation environment. ................................................. 25
3.3 The overall block diagram of ARM system used. ..................... 27
3.4 The overall block diagram of the GPU model. ......................... 28
3.5 Internals of the shader block. ........................................ 30
4.1 Implementation overview: Mapping of the ARM+GPU logical model using TCP/IP to the embedded simulation environment. ................. 35
5.1 The latency measurements for benchmarks for different no. of shaders. 48
5.2 The latency measurements for benchmarks for different no. of regs/shader core. .......................................................... 49
5.3 The latency measurements for baseline configuration vs perfect mem. 50
5.4 Instruction classification for PTX instructions. ....................... 51
5.5 Classification for mem instructions. .................................. 52
List of Tables

5.1 Kernel benchmarks and their characteristics . . . . . . . . . . . . . . 41
5.2 Benchmarks characteristics . . . . . . . . . . . . . . . . . . . . . . . . 42
5.3 Shows the latency in ms for ARM vs GPU (baseline). . . . . . . . . 44
5.4 Statistics generated for the baseline configuration . . . . . . . . . . 46
5.5 Different configurations for embedded GPU . . . . . . . . . . . . . 47
5.6 MatrixMultiplication case study: Different levels of optimizations . . 53
1

Introduction

In the recent years embedded space has seen a tremendous increase in processing power and graphics processing capabilities\[51\]. With the increasing growth of smart phones\[41\], tablets and other hand held devices, there is a widespread use of GPUs in embedded market. In the desktop market, GPUs became popular when the Unified programmable Shader models were introduced and GPUs have already proved to be quite useful in the desktop space. Until sometime back, the graphics units in embedded platform were dedicated hardware units. But now, the GPUs employed on embedded platforms have programmable shader units and hardware which has also evolved to support general purpose computations like floating point computations etc. Various hardware vendors currently support this programmability in GPUs.

The introduction of Unified Shader models in the embedded domain is paving the way for GPGPU computing in embedded domain. There is a lot of interest in embedded space to exploit the capabilities of the GPU for compute operations such as advanced imaging, augmented reality, graphics, web based content and GPS based navigation. Tasks like video acceleration, image processing etc. require extra
hardware muscle. The applications like augmented reality and GPS based navigation which are already introduced also require a lot of processing power. We are moving towards a future where devices will be more connected and integrated. The use of GPU for general purpose processing can allow applications to run on handheld devices while offloading computationally intensive tasks to other processing units which will be available on chip, off chip or distributed. While these embedded platforms with their current hardware capabilities can be utilized to perform GPGPU tasks, the reality is that the usage of GPU in embedded platform is still predominantly limited to User Interface framework and graphics.

We are now at a point where the increase in hardware capabilities in embedded space has vastly outpaced the software framework available to exploit those resources. The introduction of GPU to embedded space provides an existing infrastructure which can be leveraged for other compute intensive tasks. This thesis talks about “heterogeneous computing” in embedded platform. The term is broadly used to refer to computation on processors/cores that have attributes different from one another. But here the focus is exclusively on the system with with CPUs and GPUs. Carrying out general purpose computations on GPUs in embedded space requires a disciplined, unified programming standard. A comprehensive programming framework having a language with a rich feature set to support the heterogeneity in the embedded space, a compiler to compile the language and libraries to load the binaries is needed to enable us to tap into the available computational resources. As todays computer systems often include highly parallel CPUs, GPUs and other types of processors, it is important to enable software developers to take full advantage of these heterogeneous processing platforms.
1.1 Motivation

OpenCL, an open-source heterogeneous programming framework is a good fit to the problem being faced. OpenCL is a widely used programming framework in desktop platforms and is capable of integrating different processing units in addition to providing compiler and library support. Extending the use of OpenCL to embedded space can provide the required framework for the development of computationally advanced applications in the embedded domain. This would require a hardware structure capable of supporting OpenCL along with provision of drivers.

While, companies like ARM, Qualcomm, and Imagination Technologies have already announced support of OpenCL for GPGPU related tasks in the embedded domain, there is still a shortage of applications utilizing OpenCL based computations in embedded domain. Google has recently introduced a new feature in their Honeycomb release to provide GPGPU related functionality in Android. It uses Renderscript, a new API targeted at high-performance rendering and compute operations. It is based on C99 with some extensions very similar to OpenCL. The goal of using Renderscript is to provide lower-level, high performance API to the Android developers to maximize the overall performance of the application. With the growing number of android developers, there will soon be a large rise in the use of OpenCL based applications in the embedded domain. From all of this, it is clear that the industry is considering frameworks similar to OpenCL as a platform for utilizing GPU for compute tasks very seriously.

This project evaluates OpenCL as the programming framework for the embedded platforms with graphics processing capabilities. It is an open standard developed by Khronos group which also maintains OpenGL ES. This is the defacto API standard used by embedded GPUs. Further, OpenCL is an open-source standard and hence
can be used by independent researcher.

This thesis is broadly divided into two parts, the first part involves the development of OpenCL tools and the second part involves running the application on a suitable simulation platform. Being able to see the benefits of using a heterogeneous programming framework requires a micro-architectural platform which can be used to do design space exploration. The simulation environment with detailed hardware blocks and be a very useful tool for research. It can provide useful statistics which can feed into architectural enhancements more tailored for OpenCL applications. The primary motivation for this work includes being able to study the application performance improvement that can be brought about by micro architectural enhancements.

Figure below shows some of the embedded GPU hardware vendors in the arena.

Figure 1.1: Embedded GPU hardware battling out in the handheld space[51].
1.2 Contributions

Implementing OpenCL on embedded platforms requires the development of OpenCL libraries, drivers for the simulation platform and the simulation platform itself. The major contributions of this thesis are:

1. Provide integrated tools and simulation platform for OpenCL programming framework in embedded domain. This allows end to end study of compute kernels on embedded platform.

2. Develop simulation platform at micro-architectural detail. This provides the capability to perform design space exploration and enables us to use different hardware blocks and study their effect on performance. Detailed statistics are collected at the end of the simulation runs to understand which tools work best for micro-analysis. Having a simulation platform further provides transparency to see the effects of optimizations at developer level, compiler level, runtime level and architectural level.

3. OpenCL libraries and simulation platform driver model.

The thesis provides complete visibility into the OpenCL application flow enabling optimization at any level. The thesis also provides end-to-end simulation of application benchmarks. All the components of the application can be modified to study the corresponding effect on performance. This provides flexibility to the developer to experiment with different architectures and evaluate the suitability of OpenCL as a programming model. The following graph shows speedup (Y-axis) in the execution time of the benchmarks shown on the X-axis by running them on the GPU (baseline configuration) as compared to the execution time on ARM.
1.3 Related Work

The OpenCL programming framework is widely used in the desktop world. In the embedded space due to limited bus bandwidth, power constraints and memory availability, it has not been taken seriously. There are already announcements from many hardware vendors to release support of GPGPU framework in handheld devices. Many of them will be using OpenCL as their framework. Leskela et al. developed OpenCL embedded profile prototype for a mobile device. They note the importance of the GPU usage for new use-cases. The paper discusses the performance and energy consumption of an image processing workload for a CPU+GPU configuration. Their findings were promising and they suggest room for optimization. The study undertaken used GPU back-end compiled into a CPU program containing the OpenGL ES 2.0 shader as a string, which was used in run-time. The paper suggests
some possible scheduling improvements but these were limited by the access to the driver. Some of the recommendations were to run GPU thread at higher priority, asynchronously overlap data transfers with GPU execution and load balancing. The main finding in this performance measurement is that GPU was clearly faster than CPU and operations on floating point numbers, especially vectors, is much faster on GPU than on the performance-compromised CPU. NEON should perform well with floating point vectors but requires more pipelined vector computations, otherwise the performance boost is easily lost on NEON register conversions. This thesis because of its end-to-end visibility of the OpenCL framework can incorporate the suggestions and run them on various workloads to see the performance gains. This work provides the tools and the simulation platform to study the framework more holistically.

On the embedded GPU architectural simulation research front, Del Barrio et al. [13] present a cycle-level execution-driven simulator for GPU architectures. OpenGL framework is used to feed the simulator with traces from applications. The presented ATTILA simulator supports the simulation of a whole range of GPU configurations and architectures. This work is different as the simulator uses graphics pipeline and OpenGL to drive the simulator, in contrast this thesis, which focuses on using GPU for compute tasks and OpenCL as the framework. Lee et al. talk about using OpenCL framework for heterogeneous accelerator multicore architectures with local memory in their paper. The paper discusses a generic scenario of general-purpose processor core and multiple accelerator cores that has small local memory. OpenCL runtime does source-source transformations to boost performance and suggests new techniques like work-item coalescing, web-based variable expansion and preload-postload buffering. Work-item coalescing is a procedure to serialize the concurrent executions of the work-items in a single work-group in the presence of barriers, and to sequentially execute
them on a single accelerator core. Preload-poststore buffering is a buffering technique that eliminates software cache access overhead and individual DMA operation overhead by overlapping multiple DMA transfers together. The techniques suggested are orthogonal to the approach taken in this thesis and can be incorporated in the current framework suggested. The thesis proposes an approach of source-source transformation at the compiler level. It suggests optimizing by PTX-PTX passes, which is at a lower level than what is discussed in the paper.

Bakhoda et al.\cite{4} have developed a detailed micro-architectural simulator that runs NVIDIA’s Parallel Thread eXexution (PTX) virtual instruction set. The paper characterizes the non-graphics workloads written in NVIDIA’s CUDA programming model, demonstrating varying levels of performance improvement on GPU hardware versus a CPU-only sequential version of the application. They further do the design space exploration of interconnect topology, use caches, design memory controller, parallel workload distribution mechanisms and memory request coalescing hardware. The work resulted in the “GPGPUsim” simulator which was later extended to support the running of OpenCL applications. The simulator is open-source and was used in the thesis to model the GPU device in the embedded platform. GPGPUsim is completely configurable right from shader cores, threads per core to L1 cache size, shared mem etc. in a “.config” file which was modified for the embedded GPU device. The simulator was extended in two ways, the simulator side and the tools. Firstly, a CPU model in the form of an ARM device along with a bus model was introduced. The full-system simulation capability was lacking in the GPGPUsim, but was added in this thesis. This allows us to measure bus traffic, analytically determine power impact which is important in embedded space. Secondly, the OpenCL runtime libraries were extended in functionality and simple driver interface between CPU and GPU was
developed.

Diamos et al. [15, 14] developed a dynamic compilation framework designed to map the explicitly data parallel execution model used by NVIDIA CUDA applications onto diverse multithreaded platforms. The main idea behind this work is a dynamic binary translator from PTX to many-core processors that leverages the LLVM code generator to target x86 and other ISAs. The dynamic compiler is able to execute existing CUDA binaries without recompilation from source and supports switching between execution on an NVIDIA GPU and a many-core CPU at runtime. The paper uses the approach of converting the PTX code to an intermediate representation called PTX IR which is later converted to the LLVM IR. They leverage the existing LLVM infrastructure to execute the LLVM IR on the multi-core CPU machines. The PTX IR supports PTX-PTX transformations which was integrated in the existing toolchain to perform optimizations on static level. Zhang et al.[54] proposed a technique of eliminating thread divergence elimination. The technique is applied at runtime and uses thread-data remapping to tackle the problem. The paper discusses interesting ideas about the reference redirection and data layout transformation for remapping thread and data. They study the major challenges affecting the remapping techniques like the conflict between runtime overhead and need for remapping on the fly. They propose techniques to overcome this, like doing CPU-GPU pipelining. One more paper by Gummaraju et al. [21] tackles the optimizations at the runtime level. They propose a software platform for heterogeneous computing with CPU+GPU architectures that executes code originally targeted for GPUs, efficiently on CPUs as well. This supports load-balancing between the CPU and GPU, and enables portability of code between these architectures and CPU-only environments. These papers make specific contributions on the runtime techniques which can be well suited to the
model proposed in the thesis.

There are publicly available OpenCL implementations released from some industry-leading companies: Apple, AMD, IBM and NVIDIA [24, 23, 22, 11].

1.4 Outline of the Thesis

The thesis is organized in the following way: Chapter 2 introduces the relevance of OpenCL programming framework to this work. This chapter goes into details of the implementation of the OpenCL framework, more specifically the runtime libraries and the compilation flow. In Chapter 3, a detailed discussion of the simulator is provided. This chapter is divided into subsections which discuss the various components and tools used in the development and integration of the simulation platform. It comprises of the CPU (ARM) model, GPU model and the simulation environment to embed these models. In Chapter 4, the previous two chapters are tied together by changing the parameters in the simulator or/and compilation flow to optimize the performance. This chapter looks into the details of the optimizations and their effects on performance. The optimizations are classified at different levels depending where they were carried out and the resulting effects are studied. Chapter 5 discusses the results of various benchmarks run on the integrated framework. Some metrics and guidelines were developed which can be helpful to future developers. Chapter 6 concludes the work and provides pointers towards future directions.
This chapter discusses the design and implementation of Open Compute Language (OpenCL) framework that targets heterogeneous accelerator embedded architectures. The underlying architecture generally comprises of CPU core (usually ARM) with programmable GPUs. Parallelism has been adopted as a significant route to achieve increased performance in modern processor architectures. Central Processing Units (CPUs) faced challenges with higher clock speeds in a fixed power envelope and now, performance improvement is carried out by adding multiple cores. Following a similar path, Graphics Processing Units (GPUs) have also now evolved from fixed function rendering devices into programmable parallel processors. As embedded systems now often include CPUs, GPUs, DSPs and other types of processors, there is a growing need to enable software developers to take complete advantage of these heterogeneous processing platforms. Creating applications for heterogeneous parallel processing platforms is a challenge as conventional approaches for programming multicore CPUs and GPUs are very different. On embedded space the GPUs are still
graphics oriented. OpenCL provides general purpose GPU programming model provides way to address complex memory hierarchies and vector operations. It abstracts the underlying heterogeneous hardware and presents the developer with a generic programming model.

2.1 What is OpenCL?

OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms. OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction[27]. OpenCL is particularly suited to provide interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines. This is specifically important in the embedded space as there is a burst of game playing applications of tablets and smartphones. OpenCL consists of an API for coordinating parallel computation across heterogeneous processors; and a cross-platform programming language with a well-specified computation environment.

2.1.1 Platform Model

The platform model for OpenCL consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more Compute Units (CUs) which are further divided into one or more Processing Elements (PEs). Computations on a device occur within the processing elements. An OpenCL application runs on a host
Figure 2.1: OpenCL Platform model: Host with Compute devices, Compute Units and Processing elements[27].

according to the models native to the host platform. The OpenCL application submits commands from the host to execute computations on the processing elements within a device. The processing elements within a compute unit execute a single stream of instructions as SIMD units (execute in lockstep with a single stream of instructions) or as SPMD units (each PE maintains its own program counter).

2.1.2 Execution Model

OpenCL program executes on GPU in the form of kernel(s) that execute on one or more OpenCL devices and a host program that executes on the CPU. The host program defines the context for the kernels and manages their execution. Kernel defines an index space depending on the application. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, by a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item by querying its ID.

Work-items are organized into work-groups. Work-groups are assigned a unique
work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit. The index space is called an NDRange which is an N-dimensional index space, where N is one, two or three.

\subsection{Memory Model}

Work-item(s) executing a kernel have access to four distinct memory regions. This is distinctly shown in Figure 2.2.

- **Global Memory**: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending
on the capabilities of the device.

- **Constant Memory**: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

- **Local Memory**: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

- **Private Memory**: A region of memory private to a work-item. Variables defined in one work-items private memory are not visible to another work-item.

### 2.1.4 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. This thesis focuses on the data parallel.

#### Data Parallel

A data parallel programming model is a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items. In a strictly data parallel model, there is a one-to-one mapping between the work-item and the element in a memory object over which a kernel can be executed in parallel. OpenCL implements a relaxed version of the data parallel programming model where
a strict one-to-one mapping is not a requirement. OpenCL provides a hierarchical data parallel programming model. There are two ways to specify the hierarchical subdivision. In the explicit model a programmer defines the total number of work-items to execute in parallel and also how the work-items are divided among work-groups. In the implicit model, a programmer specifies only the total number of work-items to execute in parallel, and the division into work-groups is managed by the OpenCL implementation.

Task Parallel

Task parallel programming model defines a single instance of a kernel executed independent of any index space. Under this model, vector data types implemented by the device, enqueuing multiple tasks and/or enqueuing native kernels developed using a programming model orthogonal to OpenCL.

2.2 OpenCL Framework

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The generic framework contains the following components:

- **OpenCL Platform Layer** The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts. The application consists of a host program and kernels. The host program executes on the host processor and submits commands to perform computations on the PEs within a compute device or to manipulate memory objects.
• **OpenCL Runtime** The runtime allows the host program to manipulate contexts once they have been created. The API calls manage OpenCL objects such as command-queues, memory objects, program objects, kernel objects in a program. It allows to enqueue commands to a command-queue such as executing a kernel, reading, or writing a memory object. This sequence of runtime calls is shown in Figure 2.4.

• **Compiler** The OpenCL compiler creates program executables that contain OpenCL kernels. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism.

The implementation of the OpenCL framework on the embedded platform requires runtime libraries compiler support for the OpenCL C frontend and compiling it to GPU binary with various optimizations, and OpenCL driver for low-level talking to the hardware or simulator. Figure 2.3 summarizes the components discussed above.

### 2.2.1 OpenCL C Language

The OpenCL C programming language is based on C99 language specification [16] with specific extensions and restrictions OpenCL kernels written in OpenCL C are represented as strings defined with a C-based The main additions to C include:

1. Vector data types and rich set of data-type conversion functions
2. Image types and image read/write functions using sampler objects
3. Qualifiers for kernels and address spaces
4. Functions to query work-item identifiers
Figure 2.3: OpenCL framework divided into blocks of OpenCL application, OpenCL runtime with compiler support and the target platform [42].

5. Synchronization primitives

There are also extensions for double precision floating-point types and functions, atomic functions, 3D Image writes, byte addressable stores and built-in functions to support half types. Main restrictions are recursion, function pointers and C99 standard headers.

2.2.2 Embedded Profile

The OpenCL 1.0 Embedded Profile is defined to be a subset of OpenCL which be identified via a macro from the kernel language and through parameters of platform and device info in the OpenCL API. Its requirements are more relaxed than the full profile:

1. Online compiler is optional for embedded devices
2. 3D image support is optional and requirements of 2D images are relaxed to the level of OpenGL ES 2.0 textures

3. The minimum memory/object size requirements as well as requirements on floating point types are scaled down

### 2.3 Compilation Toolchain

This section covers all the tools developed for the OpenCL implementation on the embedded platform. The major parts are OpenCL runtime libraries, compiler support and GPU driver. The OpenCL compilation flow is shown in Figure 2.5

The Figure 2.5 has three major parts which are depicted as: A, B, and D.
Figure 2.5: OpenCL compilation flow with following parts: (A) The compilation flow for the host device, (B) compilation flow for GPU which is offline till PTX generation and online compiling PTX to GPU binary, (C) subset of part B, PTX statistics for the scheduler, (D) is the platform with CPU and GPU.

2.3.1 Host Compilation Flow

The part A is the compilation flow which takes place for the host (CPU) which in the current case is ARM device. The OpenCL application written in C/C++ reads the OpenCL kernel in the form of a string. The application is compiled and linked with the OpenCL runtime libraries. Since, in this work ARM was the host, application was cross-compiled for ARM as the target.

OpenCL specification provides two options of online and offline compilation models. Generally when the OpenCL kernel is launched it undergoes online compilation to the GPU hardware specific binary. In the offline compilation model, kernel code is compiled offline. It either uses a pre-built binary or the cached copy of the previous run of the kernel. In this thesis, since the target is embedded platform, a mid ground between offline and online compilation route is taken, to avoid the compilation...
overhead. In this process, the OpenCL C kernel code is separately compiled using the OpenCL kernel driver to intermediate format which is stored in the ARM binary. This step happens offline hence avoids the overhead of the CPU. During the execution of the kernel, this intermediate code is parsed and compiled for target GPU hardware by the backend compiler. This is loaded by the driver to the GPU and executed on it along with the data and other initialization routines.

2.3.2 GPU Compilation Flow

The part B depicts the GPU compilation flow which typically happens online, but in this thesis part of it is offline and the other part is online. As explained above the OpenCL driver used in this case is the NVIDIA’s OpenCL kernel driver which converts the .cl code to an intermediate format called PTX. It is a low-level parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device[39]. More details about PTX can be found in Appendix A. The kernel code is stored in the form of PTX within ARM in a fat binary format. The binary is loaded and executed on ARM which, when launches the kernel does the second part of the compilation happening at runtime. The kernel is launched and the online execution of the PTX starts. The PTX parser and PTX info parser libraries are loaded on ARM and are executed on the ARM. The parser outputs the execution details of the kernel in the form of data structures. More details are provided in the next chapter.

There are two main advantages of using this split online/offline compilation approach:

1. Static level optimizations: the offline compilation option allows one to apply more comprehensive and heavy optimizations which would have been limited if
were applied online. In this model having visibility at the PTX level exposes some of the underlying architectural details at the same time it allows developer to apply static level optimizations. There are tools which can be used to PTX-PTX level transformations, one of which is used in this thesis[14].

2. Less CPU overhead: It also reduces the compilation overhead on the host processor.

The part C shown in Figure[2.5] is a subset of the part B. This process takes place along with the offline compilation model. It deals with gathering PTX statistics which are used by the hardware scheduler to execute the kernel on the GPU. The part more specifically find out the shared memory usage and per-thread register usage of the kernel to be executed on the GPU. This information is used to schedule the threads as viewed by the developer in OpenCL programming model to the hardware thread as the GPU cores see it. Part D in the figure denoted the underlying platform with CPU and GPU on it. This will covered in more detail in the next chapter.

OpenCL execution can happen in following three ways:

1. Only on CPU can do native execution of the kernel, in this case the kernel is compiled to run on CPU. This can be useful to achieve load-balancing when in some case the GPU is running heavily and CPU can take some tasks.

2. Run it on the GPU by compiling it to GPU binary.

3. Can run it on emulator to check the functional correctness of the code. Functionally emulate the code.
Simulation Architecture

The simulation framework is modelled to test the results and performance of the application. The goal was to have the simulator for the GPU with focus on compute side of the GPU. The simulator, with micro-architectural details, will increase the accuracy of the simulation, give flexibility with design space exploration and visibility to the underlying hardware architecture. There is limited information available from the hardware vendors in embedded space regarding the GPU architecture. The hardware configuration and micro-architectural details for the underlying GPU model was inferred from datasheets, websites and online forums of the embedded GPU vendors. Figure 3.1 shows some of the information about the GPU on the block level by the hardware vendors [46, 25, 10].

The OpenCL benchmarks required simulating the results on reference platform. This chapter describes various details of the simulator and its components. Simulation platform comprises of three major components:

1. ARM model

2. GPU model
Figure 3.1: Hardware vendors for embedded GPUs: (A) ARM with Mali GPUs, (B) Imagination Technologies, Power VR SGX, (C) ZiiLabs, (D) NVIDIA with Tegra GPUs.

3. Simulation environment

Each model is discussed in detail in the following sections. Simulation platform was developed by the integration of ARM model and GPU model and embedding it in the simulation platform. The two models are connected by a simple bus interface.

3.1 Overview

The simulation framework was designed with certain goals in mind. One of the important goals was to have an extensible platform so that more components can be added later. This will enable future research in various directions such as having shared memory on this platform and its impact of boosting performance. The other
component which can be added to the platform is a DSP. This allows certain computational tasks to be offloaded to the DSP and the load balancing which can be carried out on the system can be studied. This will enable us to study traffic on the bus when the various resources are available on the platform and are being used on the OpenCL platform. OpenCL provides a unified way to program these heterogeneous elements on the simulation platform, so extensibility will be helpful in achieving that.

The other important factor for the simulation platform is the availability of the compilation tools with debugging capability. This becomes more important when the simulation is of the heterogeneous platforms. Here it will be important to know the call stack, and have the ability to set the breakpoints to see what values are being passed before the control is transferred to GPU model where debugging is done on a log file basis. It would also be helpful to know the pointer values etc. The simulator requirements were set so that we have a more detailed simulation of the GPU on the

Figure 3.2: Overall block diagram of the simulator: (i) ARM 9 ISS model from OVP-sim, (ii) GPU model extended from GPGPUsim (iii) OVPsim simulation environment.
cycle-level and simulation of the CPU was at the instruction accurate level with much higher MIPS. This was because, the research was more focused towards the GPU and CPU was merely used as driver for the GPU. This can be very well, extended to have more detailed simulation of the CPU and do studies on load-balancing between CPU and GPU[21].

3.2 ARM Model

ARM device was used as the CPU model for the embedded platform developed. This was because of the wide usage of the ARM devices on the embedded platform with GPUs. The ARM platform was taken from the library of available models, platform and peripherals in OVP simulator environment. Details about which will be discussed in the next section. The ARM platform referred to as “ARM IntegratorCP” is configured for ARM926EJ-S and Linux2.6. ARM processor model includes ARMv4, ARMv5, and ARMv6 instructions.

3.2.1 Simulation Details

There is a RAM disk with all the benchmarks which will run on the ARM model. There are simple script files to unpack the RAM disk image file to the linux file system so that disk image can be easily modified by adding or deleting files. Then another script file can pack it back into the format which can be used by the ARM platform.

Listing 3.1: Set of commands to use ARM platform

```
unpackImage.sh fs.img fs
packImage.sh fs fs.img
<platform_executable> <linux_image> <ram_disk>
```
The ARM platform model is written in a C source code file. This makes API calls into the OVP simulation interface to build the platform. This gets compiled and linked into an ARM platform executable. This executes on top of the OVP simulator which does instruction accurate modelling of the ARM code using binary translation. The other parameters passed to the platform executable are the Linux image (zImage format) and the RAM disk image which has all the benchmarks loaded on it.

### 3.3 GPU Model

The GPU model is based on the GPGPU sim simulation model developed by Bakhoda et al.\[3\]. In the programming model GPU is treated as a co-processor onto which an application running on a CPU can launch a massively parallel compute kernel. The
kernel is comprised of a N-dimensional grid of scalar threads. Each thread is given an unique identifier which can be used to help divide up work among the threads. Within a grid, threads are grouped into thread blocks. Within a single thread block, threads have access to a common fast memory called the local memory (also referred to as shared memory) and can perform barrier synchronizations.

### 3.3.1 GPGPUsim

The GPU consists of a collection of small data-parallel compute cores, labeled shader cores in Figure 3.4 connected by an interconnection network to memory modules and the external memory. Each shader core is a unit similar in scope to a core computational unit on a GPU. Threads are distributed to shader cores at the granularity of thread blocks, The thread block resources, registers, shared memory space are not freed until execution is completed. Multiple CTAs can be assigned to a single shader core, thus sharing a common pipeline for their execution. The GPU model omits graphics specific hardware as the model is used for GPGPU related tasks. The shader
core has a SIMD width of 8. It uses an 6 stage, in-order pipeline with a super pipelining degree of 4 without forwarding. The stages in the pipeline are motivated by the number of threads that are needed to avoid stalling for true data dependencies between consecutive instructions from a single thread. It has six logical pipeline stages namely fetch, decode, execute, memory1, memory2 and writeback. Threads are scheduled to the SIMD pipeline in a fixed group of 32 threads called a warp/wavefront. All 32 threads in a given warp execute the same instruction with different data values over four consecutive clock cycles in all pipelines which means that the SIMD cores are effectively 8-wide. Threads running on the GPU have access to several memory regions global, private, constant, texture, and shared. The simulator models accesses to each of these memory spaces. In particular, each shader core has access to highly-banked per-core shared memory, texture memory and global constant memory with a per-core constant cache. Multiple memory accesses from threads within a single warp to a localized region are coalesced into fewer wide memory accesses to improve DRAM efficiency.

Thread scheduling inside a shader core is performed with zero overhead on a fine-grained basis. Every 4 cycles, warps ready for execution are selected by the warp scheduler and issued to the SIMD pipelines in a loose round robin fashion that skips non-ready warps, such as those waiting on global memory accesses. In other words, whenever any thread inside a warp faces a long latency operation, all the threads in the warp are taken out of the scheduling pool until the long latency operation is over. Meanwhile, other warps that are not waiting are sent to the pipeline for execution in a round robin order. The many threads running on each shader core thus allow a shader core to tolerate long latency operations without reducing throughput. In order to access global memory, memory requests must be sent via an interconnection.
network to the memory controllers. Each on-chip memory controller then interfaces to off-chip GDDR3 DRAM chips. When memory accesses within a warp cannot be coalesced into a single memory access, the memory stage will stall until all memory accesses are issued from the shader core.

The micro-architecture details of the GPU model are completely configurable. It uses a .config file as an input to set up the parameters to run the simulation.

Looking at the simulator from the software organization perspective. The gpu-sim.cc file contains the code for the gpu-init, run-gpu-sim loop and the gpu-sim loop. They are the main loops which run, over all the threads running on the shader. In the one pass it goes over all the shaders and runs the warps of thread which are scheduled to run. The instructions are stored in the instructions.cc file which functionally executes the GPU instructions. They are stored in the form of function calls which are made from the device. Then there are files which deal with the
interconnect topology used in the model and option parsing interface used to read
the configuration file passed to the simulator.

3.3.2 PTX Parser

PTX parser is written using the PTX ISA documentation provided by NVIDIA and
parsing tools like flex and bison. The parser provides the details of the PTX and how
it maps to the code in the simulator. The parser generates the C source files which
traps into callback functions to populate the data structures. The data structures are
used to drive the GPU and are later sent to the GPU device.

The other aspect of the parser is the PTX info parser. This provides the informa-
tion regarding the per-thread register usage and shared memory usage. The infor-
mation is used by the scheduler to group threads into thread blocks, assign threads
and thread blocks to different shaders. This information is stored in the file which
is generated at the stage of offline compilation. This was discussed in the previous
chapter, during the compilation flow. The file is parsed along with the PTX parsing
stage and generates data structures which are used by the scheduler.

3.4 OVP Simulation Environment

This section discusses the OVP simulation environment which provides the interface
to embed the above discussed models. OVP provides processor models that are
instruction accurate. The instruction accurate models execute each instruction using
the correct number of clock cycles and perform their I/O operations close to the right
place within the instruction. The OVP models are functional models. They progress
one instruction at a time. When an OVP platform is put together, each processor will
advance one instruction at a time, as will any other processors in the platform. They may operate at different instruction frequencies, but the exact timing relationship between them has no conceptual meaning. When they attempt to use a shared resource, such as a memory, the model will arbitrate the access to the model and ensure that each transaction is conducted in a safe and repeatable manner. However, the order in which those access happen cannot be guaranteed.

OVP is made up of four C interfaces: ICM for creating platforms, VMI for processors, BHM for behavioral blocks and PPM for peripherals. ICM is the interface that ties together the basic blocks of the system, such as processors, memory sub-systems, peripherals and other hardware blocks. The ICM interface is a C interface that when compiled and linked with those of each of the models and some object files produces an executable model. Given that it is based on standard C code, any C compiler can be used to create the model. OVP uses a code morphing approach which is coupled with a just-in-time (JIT) compiler to map the processor instructions into those provided by the host machine. In between are a set of optimized opcodes into which the processor operations are mapped, and OVPsim provides fast and efficient interpretation or compilation into the native machine capabilities. This is very different from the traditional ISS approach which interprets every instruction. Another capability of VMI is to allow a form of virtualization for capabilities such as file I/O. This allows direct execution on the host using the standards libraries provided.

PPM, the peripheral modeling interface, is very similar to the fourth interface BHM, which is intended for more generalized behaviors. These models run in a second portion of the simulator that is called a Peripheral Simulation Engine (PSE). This is a protected runtime environment with a separate address space for each model and restricts communications to the mechanism provided by the API. The principal
difference between the two interfaces is that the PPM interface understands about busses and networks.

OVP models the bus as a pure one-to-one communications channels with their memories and peripherals. It helps to identify the general traffic levels on each bus, and provides the ability to identify the peaks.
4

Methodology and Optimizations

This chapter discusses the overall methodology applied on the project.

4.1 Implementation Overview

Figure 4.1 gives a block diagram of the implementation overview of the simulator developed for the thesis. The development of the simulator was done over following stages.

4.1.1 Installation & Initial Testing

The first step involved installation and initial testing of GPGPU-Sim [4] the GPU model for the simulator. Similar effort was taken for the OVP simulator and testing the ARM model. The OVP platform is well documented with active support forum which helped and expedited the process. The GPGPU sim model was experimented with various configurations for available benchmarks both on OpenCL and CUDA. This was required to understand the interface of the simulator. The results generated
were in a certain format and had to be familiarized.

4.1.2 Reading Code

Next step, was understanding structure of the code for the GPU model. This was important as the code had to be modified to add the CPU model to the existing simulator, which it currently doesn’t support. The entry points for the code was required which was helped by the presentation by the same group which developed the simulator[4]. In the presentation they cover in detail the GPGPUsim internals like micro-architectural details modelled, software organization and the modifications which can be done to the model. The GPGPUsim has two modes of simulation, functional and cycle-level. The various options can be configured using a “gpgpusim.config” file in the simulator. The code in GPGPUsim simulator is logically

Figure 4.1: Implementation overview: Mapping of the ARM+GPU logical model using TCP/IP to the embedded simulation environment.
structured into two blocks. The runtime compilation block for the PTX into the data structures that drive the GPU model and the various hardware blocks of the GPU model itself. The compilation model, takes place on a CPU, which in this case is an ARM device. This required cross-compilation of the code to target ARM device. In this thesis, cross-compiler used was provided by the Code Sourcery tools[8]. The compilation tools provided by Code Sourcery are well maintained with tools support. The correctness was tested by functional mode simulation of the PTX code on ARM.

4.1.3 Restructured Simulator

This step required a block model organization at the complete system level of the simulator. The ARM model runtime compilation model was divided into:

1. PTX parser generated by flex[40] and bison[18] tools.

2. Cross compilation of the OpenCL runtime libraries which were later statically linked to the applicaiton.

3. A simple driver interface which will interact with the GPU model to load the data structures generated by the PTX parser on to the GPU and. This requires a server and client relationship between the ARM model and the GPU model.

The driver interface developed required populating the data structures and transferring them over the bus to drive the GPU model. The main issue, with this was that GPGPUIm generates data structures in the form of C++ objects and uses them to load the GPU model. Since there is no concept of bus and separate CPU and GPU model in the GPGPUIm, this worked fine. But, to accomplish the goal of full-system simulation a more flexible and extensible platform was required. To solve this problem, BOOST serialization libraries were used[33].
The Boost “serialization” libraries do a reversible deconstruction of an arbitrary set of C++ data structures to a sequence of bytes. Such a system can be used to reconstitute an equivalent structure in another program context, which in present scenario is in the GPU. This can be used implement object persistence, remote parameter passing or other facility. In this system, the term “archive” is used to refer to a stream of bytes which are passed from the sender to the receiver. So, the boost libraries convert the C++ objects into a stream of bytes which can then be sent over a bus interface and reconstructed on the GPU to be used to drive the model. This enables, the developer to have a simple bus interface and study the general traffic pattern between CPU and GPU at a more coarser granularity.

4.1.4 Separating ARM Model and GPU Model

Separating of the execution space of the ARM model with the GPU model is critical in realistically modelling the CPU and GPU. This was needed to make the communication between ARM and GPU model more explicit. This thesis, models the CPU+GPU platform with both the devices having their separate address spaces. This was achieved using a TCP/IP client-server model to test the GPU and ARM model.

4.1.5 OVPsim Environment

The final step was integration of the ARM model with the GPU model. These models were embedded in a OVPsim simulator which has a Peripheral Simulation Engine(PSE). The GPU model was modelled as a peripheral model for the ARM system. A peripheral model is compiled into an ELF format executable for the PSE
processor architecture. It is dynamically loaded by OVPSim simulator tools. Peripheral model also needs an XML file to be created to describe the interface to the PSE executable. The PSE processor runs in its own private (simulated) memory space, which is isolated from the host environment. Interception of functions defined in the peripheral model allows the use of features of the host system in the implementation of the behavior of a peripheral. This feature was used to model the GPU model. The ARM platform had the GPU as a peripheral. When simulating this system, the GPU model in the form of a shared object (.so) was loaded into the PSE execution space. The final integration of the GPU on to the ARM system was done by memory mapping the registers of the GPU peripheral to the bus. ARM then programs these registers with callback functions which were intercepted by the shared libraries, as discussed earlier. This model can be extended to provide DMA support for transferring GPU configuration structures and data to and from the GPU. This infrastructure can be used to study CPU/GPU load balancing also.

4.2 Optimization Framework

The main idea behind the proposed optimization framework, is to clearly define the levels at which the optimizations can be applied for the GPGPU tasks on the GPU. The next step is to make a set of optimizations which can be applied at each level and to choose the optimizations such that they work coherently. In this thesis, the OpenCL tools are integrated with the underlying simulator platform with visibility. This provides us the flexibility of having control over the compilation flow and provides opportunity to apply the optimizations and see that the change is propagated down.

This work suggests four levels:
• Developer level: It suggests the optimizations which can be applied at the source level by the developer. For ex. this can be to optimize the application to use shared memory efficiently looking at the application requirements and the underlying hardware, or it can be using the parallelism in the code.

• Compiler level: This refers to the static level optimizations which can be applied to the code by the compiler. There are whole bag of optimizations which can be applied like loop-unrolling, dead code elimination, if-conversion etc. This can be used in this work at the PTX level. The OpenCL driver emits the PTX code from the OpenCL C code. There are tools which can be used to do PTX-PTX transformations. The tool used in this work is Ocelot\textsuperscript{[14]}, which is a dynamic compilation framework for the PTX.

• Runtime level: These set of optimizations are applied at runtime when the kernel is launched for execution on the GPU. There are set of techniques like thread-data remapping, dynamic warp formation etc. which can be applied at this level.

• Architectural level: This refers to the set of enhancements which can be done at the ISA level and/or the micro-architectural level. It can be related to the shared memory banks, number of shader cores, registers per core, L1 cache, L2 cache etc. This thesis has provision to apply these enhancements and study them with the benchmarks.

This optimization framework can be used as guidelines for the developer. A case study using static level optimizations on the benchmark is presented in the next section. The performance improvements were noted using baseline GPU.
Discussion

5.1 Benchmarks Overview

Benchmarks were selected from the NVIDIA and AMD OpenCL SDKs. The kernel benchmarks represent a diverse application base such as signal, image processing, computational and control etc. The benchmarks require data-parallel architectures for performance.

The benchmarks and their characteristics is shown in the following Table 5.1

5.1.1 Workloads

This section discusses the benchmarks covered in this thesis more in detail. The section covers the datasets used in the benchmark applications, size of the data and its format. It also looks at the:

1. The number of threads created, number of workgroups

2. How many times the kernel is called
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESEncryptDecrypt</td>
<td>Encryption on an image, computational</td>
</tr>
<tr>
<td>BitonicSort</td>
<td>Control intensive</td>
</tr>
<tr>
<td>DCT</td>
<td>Signal processing</td>
</tr>
<tr>
<td>DwtHaarID</td>
<td>Signal processing</td>
</tr>
<tr>
<td>FastWalshlTransform</td>
<td>Signal Processing</td>
</tr>
<tr>
<td>FFT</td>
<td>Signal Processing</td>
</tr>
<tr>
<td>FloydWarshall</td>
<td>Graph based algorithm</td>
</tr>
<tr>
<td>MatrixMultiplication</td>
<td>Computational</td>
</tr>
<tr>
<td>MatrixTranspose</td>
<td>Computational</td>
</tr>
<tr>
<td>PrefixSum</td>
<td>Tree-based algorithm</td>
</tr>
<tr>
<td>Reduction</td>
<td>Computational, control</td>
</tr>
<tr>
<td>VectorAdd</td>
<td>Computational</td>
</tr>
<tr>
<td>DotProduct</td>
<td>Computational</td>
</tr>
<tr>
<td>URNG</td>
<td>Gaussian, Computationally intensive</td>
</tr>
</tbody>
</table>

Table 5.1: Kernel benchmarks and their characteristics

3. Details of the kernel

There are characteristics of the benchmarks captured in the table 5.2. It clearly depicts the programmer’s view of OpenCL characteristics in the benchmark. There are details like grid dimensions, number of threads and the work-group size which suggests the index space in which the application is working. The scheduler maps this index space on to the available hardware details. Then, there are memory oriented details and characteristics like shared-memory usage, constant memory etc. which reflects on the kernel’s memory usage patterns. This information is instrumental in understanding of the performance results of these benchmarks with these varying characteristics. The memory accesses are a large part of the bottlenecks where performance takes the hit. So it is important to understand the memory access patterns and layout of the data to provide maximum benefit. Techniques like memory coalescing, avoiding shared memory and cache bank conflicts etc. can be useful to help boost the performance. The GPU model in the simulator does have these features
modelled and are measured in the results to study how application is behaving with these characteristics.

Below is a description of the algorithm and the OpenCL implementation of the benchmarks used in this thesis. This helps throw more light at the programmer's view of the application.

**AESEncryptDecrypt**

This AES encryption and decryption algorithm uses Galois multiplication algorithm to implement the kernel. The application makes tiling of 1X4 tiles and does local optimizations. globalThreads are (Image-width)/4 and height. The encryption key is stored in the local memory to avoid fetching it for encryption algorithm. keysize equal to 128 bits. Workgroups created are 512*128. Local values Sbox is for encryption and RBox is for decryption. In the kernel code there is a loop for rounds. There is a global fetch and 2 barrier operations. This implementation is in 2 Dimensions for the application.
**PrefixSum**

This implementation is in 1 Dimension. There is globalThreads=length/2 and localThreads=length/2 It uses the shared memory to cache the computational window. This is used to make the Prefix Tree. First it builds the sum in place up the tree. It uses barrier at the beginning of the loop to sync all the threads in the loop and then does in place calculation. Here the shared memory is used heavily and can have bank conflicts. This is what was seen in the simulation results.

**BitonicSort**

It works by creating bitonic sub-sequences in the original array, starting with sequences of size 4 and continuously merging the subsequences to generate bigger bitonic subsequences. Finally, when the size of this subsequence is the size of the array, it means the entire array is a bitonic sequence. This algorithm works only for arrays with length of a power of 2. For an array of length 2N, the sorting is done in N stages. The first stage has one pass; the second has two passes; the third stage has three passes, and so on. Every pass does length/2 comparisons; that is, 2(N-1). Let us call this value numComparisons. The kernel compares once and writes out two values. So, for every pass there are numComparisons invocations of the kernel; that is, numComparisons is the number of threads to be invoked per pass.

**DCT**

Discrete Cosine Transform (DCT) is a common transform for compressions of 1D and 2D signals such as audio, images and video. This kernel implements a two-dimensional variation of the DCT that operates on blocks of size 8x8. This variation, is used often in image and video compression. The input matrix is partitioned into blocks of size
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ARM(ms)</th>
<th>baseline(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESEncryptDecrypt</td>
<td>238.76</td>
<td>1.096255</td>
</tr>
<tr>
<td>BitonicSort</td>
<td>9.753</td>
<td>0.092345</td>
</tr>
<tr>
<td>DCT</td>
<td>65.334</td>
<td>0.025148</td>
</tr>
<tr>
<td>DwtHaar1D</td>
<td>16</td>
<td>0.004754</td>
</tr>
<tr>
<td>FastWalsh Transform</td>
<td>38</td>
<td>0.012246</td>
</tr>
<tr>
<td>FFT</td>
<td>22</td>
<td>0.014123</td>
</tr>
<tr>
<td>FloydWarshall</td>
<td>128.74</td>
<td>0.322511</td>
</tr>
<tr>
<td>MatrixMultiplication</td>
<td>169.383</td>
<td>0.091803</td>
</tr>
<tr>
<td>oclDotProduct</td>
<td>13.833</td>
<td>0.13744</td>
</tr>
<tr>
<td>oclVectorAdd</td>
<td>30.064</td>
<td>0.059649</td>
</tr>
<tr>
<td>PrefixSum</td>
<td>4.666</td>
<td>0.131842</td>
</tr>
<tr>
<td>Reduction</td>
<td>0.073</td>
<td>0.005665</td>
</tr>
<tr>
<td>transpose</td>
<td>120.667</td>
<td>0.155775</td>
</tr>
</tbody>
</table>

Table 5.3: Shows the latency in ms for ARM vs GPU (baseline).

8x8 and each block is performed the DCT.

**DwtHaar1D**

Basic one-dimensional Haar Wavelet transform, is a one-dimensional image with a resolution of four pixels, having values 9, 7, 3, 5. Image in the Haar basis by computing a wavelet transform. The wavelet transform (also called the wavelet decomposition) of the original four-pixel image to be the single coefficient representing the overall average of the original image, followed by the detail coefficients in order of increasing resolution.

**FastWalshTransform**

Fast walsh transform is the matrix product of a square set of data and a matrix of basis vectors consisting of Walsh functions. The kernel performs a butterfly operation and it runs for half the total number of input elements in the array. In each pass of the kernel two corresponding elements are found using the butterfly operation on an array of numbers and their sum and difference is stored in the same locations as the numbers.
**FFT**

A fast Fourier transform (FFT) is an efficient algorithm to compute the Discrete Fourier transform (DFT) and its inverse. This computes an in-place complex-to-complex FFT. \( x \) and \( y \) are the real and imaginary arrays of power of 2 points. \( \text{Dir} = 1 \) gives forward transform while \( \text{Dir} = -1 \) gives reverse transform.

**FloydWarshall**

The Floyd-Warshall algorithm computes the shortest path between each pair of nodes in a graph. It is a dynamic programming approach that iteratively refines the adjacency matrix of the graph until each entry in the matrix reflects the shortest path between the corresponding nodes. This implementation is a randomly generated adjacency matrix.

**Reduction**

Reduction consists of dividing the input array into blocks, calculating the sum of blocks and then add them up. Each block invocation of the kernel reduces the input array block to a single value; it then writes this value to output and reduces the block sums to a final result, which is sent to the host. Each work-item loads its data from an input array to a shared memory of block. The reduction of each block is done in multiple passes. In first pass, half work-items are active, and they update their values in shared memory by adding other half values in shared memory. In subsequent passes, the number of active threads are reduced to half, and they keep updating their value with other half values of shared memory.
Table 5.4: Statistics generated for the baseline configuration

MatrixMultiplication

There are two implementations of the matrix multiplication algorithm. One uses the shared memory and the other implements without it. In the implementation without shared memory, each thread reads four floats from matrix A, then the corresponding four float4s from matrix B. Each thread then calculates the partial matrix multiplication and updates the partial sum. Thus, each thread computes four floats. The number of global threads = width/4, height/4. In the second implementation shared local memory a tile of matrix A is loaded into shared memory, which supports broadcasting of values. A work-group loads a tile of matrix A into shared memory. The outer loop runs for the number of work-groups that can fit in matrix A’s width. The inner loop consists of loading four float4 values from matrix A’s tile, and the corresponding values from matrix B’s global memory, to compute a partial sum. Each work-item computes four float4s values, and writes to matrix C’s global memory. The number of global work-items = width/4, height/4.
<table>
<thead>
<tr>
<th>Configuration parameters</th>
<th>Baseline Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptx instruction classification</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>sim mode</td>
<td>0</td>
<td>0=cycle_level, 1=functional</td>
</tr>
<tr>
<td>n shader</td>
<td>8</td>
<td>Number of shader cores</td>
</tr>
<tr>
<td>spread blocks</td>
<td>1</td>
<td>Thread blocks across cores</td>
</tr>
<tr>
<td>n mem</td>
<td>4</td>
<td>No. of memory channels</td>
</tr>
<tr>
<td>clock domains</td>
<td>325:650:800.0</td>
<td>Core:Interconnect:DRAM</td>
</tr>
<tr>
<td>shader regs</td>
<td>4096</td>
<td>Regs/shader core</td>
</tr>
<tr>
<td>pipeline</td>
<td>256:32:32</td>
<td>Registers:warp size:pipeline simd width</td>
</tr>
<tr>
<td>shader cta</td>
<td>4</td>
<td>No. of thread blocks per shader core</td>
</tr>
<tr>
<td>pdom sched type</td>
<td>8</td>
<td>Branch divergence algo</td>
</tr>
<tr>
<td>shmem bkconflict</td>
<td>1</td>
<td>Shared memory bank conflicts</td>
</tr>
<tr>
<td>cache bkconflict</td>
<td>1</td>
<td>Cache bank conflicts</td>
</tr>
<tr>
<td>n cache bank</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>shmem pipe speedup</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>shmem port per bank</td>
<td>2</td>
<td>Ports/shared memory</td>
</tr>
<tr>
<td>cache port per bank</td>
<td>2</td>
<td>Ports/cache</td>
</tr>
<tr>
<td>const port per bank</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>interwarp mshr merge</td>
<td>6</td>
<td>MSHRs used</td>
</tr>
<tr>
<td>const cache:dl1</td>
<td>64:64:2:L</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5: Different configurations for embedded GPU

**Uniform Random Noise Generator**

This kernel generates noise in an image by using a linear congruential generator which generates a uniform deviation in the range (0, 1) and which is multiplied by a noise factor to produce the final noise. This application has each thread having its own private values and the local memory is used as indexed arrays to use global memory instead of registers. Therefore this also has shared memory conflicts in the banks.

The configuration details of the baseline configuration is covered in the table 5.5. It shows all the different configurations possible for the GPU.

There were other benchmarks like VectorAdd, DotProduct and MatrixTranspose also studied, but they had standard implementations.
5.2 Results & Discussion

This section is broadly divided into two sections, first is where the results presented are a result of design space exploration of the GPU. Secondly there is a case study of the benchmark with some optimizations done at the PTX level.

5.2.1 Design Space Exploration

There were different variations of the baseline configuration done and tested against it. The Figure 5.1 shows the simulation results of the number of cycles it took to run the benchmarks for varying number of shader cores.

Similarly in Figure 5.2, the number of registers per shader core were varied, with taking baseline configuration. So, the number of shader cores are fixed but the number of registers on each shader core is different. This affects the scheduling of the threads.
on the shader cores. It is because the scheduler looks at the number of threads, per-thread register usage and the number of registers/shader core to do the mapping. If the register count goes down it affects the number of threads that can be scheduled and also the number of work groups which can be scheduled on the shader core. This can affect the performance as, if the number of thread groups on the shader core go down, the scheduler on chip of the shader will not have enough thread groups to schedule when there is stall due to long latency operations or global memory accesses.

In Figure 5.3 presents the results for the comparison of latency numbers for the baseline configuration as compared to the case with perfect memory configuration. In the perfect memory configuration the memory access latency is not measured, which gives the shader core a flat memory structure. The results provide an interesting perspective to look at the kernels, because the bigger difference between the latency
The latency measurements for baseline configuration vs perfect mem. shows that the kernel is memory bound. The performance can be improved for these kernels by changing their memory access patterns. This analysis can be further benefitted by having the memory instruction classification.

The analysis presented above correlates with the finding in the Figure 5.4 which is the graph of different instruction classification on the benchmarks. In the benchmarks like DotProduct and MatrixTranspose which have lot of memory operations, result in a bigger latency difference between baseline configuration and the perfect memory case. So, we can conclude that the performance of the benchmark is bottlenecked because of the memory operations. To, further validate the findings, looking at the OpenCL C kernel source code for DotProduct benchmark reveals that all the memory operations are global accesses and that is causing the bottleneck. The other Figure 5.5 regarding the memory classification can help to further categorize the memory operations to the shared memory, constant memory accesses, global accesses,
load instructions, store etc.

This way the simulator can give insight into the application, which can help tune the application to perform better depending on the underlying structure.

5.2.2 Case Study: Optimizations

The study was done on the MatrixMultiplication benchmark kernel. The optimizations were applied on all the four levels mentioned before in the Optimizations framework section.

The table 5.6 shows two separate kernel variants of the MatrixMultiplication kernel where one uses shared memory with one buffer and the other with two shared memory buffers. The static level transformations are reverse-if-conversion, structural-transform, remove-barriers etc. These were provided by the Ocelot[14] compilation
framework which has PTX-PTX transformations. There were some experiments at the runtime level with dynamic warp formation of the threads [4] but that didn’t have a positive effect on the performance. The results were tested on the baseline configuration of the simulator. The static level optimizations didn’t give much performance benefits, but are instrumental in arriving at a different structure of the code. This can be useful in different scenarios. The overall improvement of 2% was observed.
<table>
<thead>
<tr>
<th>MatrixMultiplication</th>
<th>kernel configuration</th>
<th>Developer 1</th>
<th>baseline</th>
<th>Developer 2</th>
<th>baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_cycle</td>
<td>28954</td>
<td>28951</td>
<td>28950</td>
<td>28922</td>
<td>28917</td>
</tr>
<tr>
<td>sim_load</td>
<td>142864</td>
<td>141800</td>
<td>141800</td>
<td>141568</td>
<td>141568</td>
</tr>
<tr>
<td>sim_loadmem</td>
<td>142864</td>
<td>141800</td>
<td>141800</td>
<td>141568</td>
<td>141568</td>
</tr>
<tr>
<td>shared</td>
<td>43194</td>
<td>43385</td>
<td>43570</td>
<td>43397</td>
<td>43572</td>
</tr>
<tr>
<td>shared buffers</td>
<td>43194</td>
<td>43385</td>
<td>43570</td>
<td>43397</td>
<td>43572</td>
</tr>
<tr>
<td>completel2com</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>total_load</td>
<td>28954</td>
<td>28951</td>
<td>28950</td>
<td>28922</td>
<td>28917</td>
</tr>
<tr>
<td>total_loadmem</td>
<td>142864</td>
<td>141800</td>
<td>141800</td>
<td>141568</td>
<td>141568</td>
</tr>
<tr>
<td>total</td>
<td>43194</td>
<td>43385</td>
<td>43570</td>
<td>43397</td>
<td>43572</td>
</tr>
</tbody>
</table>

Table 5.6: MatrixMultiplication case study: Different levels of optimizations
Conclusions and Future Directions

This thesis is an attempt to study the OpenCL programming framework for the heterogeneous embedded platforms. Specifically, the goal is to use the GPUs widely available on the embedded market for compute purposes. There are already published which suggest that embedded GPUs can be used for compute intensive tasks like signal processing, video processing, medical monitoring etc. The thesis takes a step forward and integrates the OpenCL tools with simulation framework. This provides a platform to do design space exploration, tools development and performance improvement guidelines. This research shows clear performance gains for benchmark kernels studied when run on GPUs as compared to the ARM implementation. The results showed an average speedup of 385 times across benchmarks. The thesis also provides complete end to end visibility of the compilation flow, simulation platform and runtime model. This gives flexibility to study optimizations across different levels. One such case study on the benchmark kernel was performed. This added further performance boost of 2%. This integrated framework can be used as a tool to develop performance metrics and guidelines for the developers. The developer can
tune their applications to get maximum benefit from number of available and future architecture.

6.1 Work Ahead

This work lays down the foundation for using embedded GPUs in compute domain. The work can be taken forward in many directions.

1. A power analysis tool can be integrated with the current model to give approximate power consumption numbers for using GPUs

2. It can be used to study the load balancing on CPU+GPU environment

3. Sophisticated bus interface can be developed to monitor the traffic because in embedded space bus traffic plays an important role in making architectural decisions

4. Tools infrastructure can be improved to add functionality and precision, like OpenCL runtime libraries, driver model etc.

5. Extend the current model to support OpenGL, this can help study the optimizations which can be done when both OpenCL and OpenGL commands are enqueued

6. Here the kernels were taken from OpenCL SDKs which are tuned to perform well on GPUs, as compared to when run on CPUs, so comprehensive benchmarks can be used to see the performance gains in real-life scenarios
Appendix

7.1 Appendix A: PTX

PTX defines a virtual machine and ISA for general purpose parallel thread execution. PTX programs are translated at install time to the target hardware instruction set. The PTX-to-GPU translator and driver enable NVIDIA GPUs to be used as programmable parallel computers.

PTX provides a stable programming model and instruction set for general purpose parallel programming. It is designed to be efficient on NVIDIA GPUs supporting the computation features defined by the NVIDIA specific architecture. High level language compilers for languages such as CUDA and C/C++ generate PTX instructions, which are optimized for and translated to native target-architecture instructions.

The major objective for PTX is to provide a stable ISA that spans multiple GPU generations. It is capable of achieving performance in compiled applications comparable to native GPU performance. It provides a scalable programming model that
spans GPU sizes from a single unit to many parallel units. It provides a machine-independent ISA for C/C++ and other compilers to target. It acts as a code distribution ISA for application and middleware developers and a common source-level ISA for optimizing code generators and translators, which map PTX to specific target machines. It also facilitates hand-coding of libraries, performance kernels, and architecture tests.
Bibliography


[18] GNU. Bison - gnu parser generator, March 2011. 36


[22] IBM. Opencl development kit for linux on power, March 2011. 10

[23] AMD Inc. Opencl zone: Amd developer central, March 2011. 10


61


[41] Email Marketing reports. Smartphone statistics and market share, March 2011.


Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such.

The thesis work was conducted from 04/2010 to 04/2011 under the supervision of Prof. David Kaeli at Northeastern University.