ADDIS ABABA UNIVERSITY
ADDIS ABABA INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Acceleration and Energy Reduction of Object Detection on Mobile Graphics Processing Unit

---

By:
Fitsum Assamnew Andargie

Supervisor:
Prof. Jonathan ROSE
University of Toronto

Co-supervisor:
Dr-Ing. Dereje Hailemariam
Addis Ababa University

A dissertation submitted in partial fulfilment of the requirements for the degree of **Doctor of Philosophy**

in

Computer Engineering

June 2019

Addis Ababa University
Addis Ababa Institute of Technology
School of Electrical and Computer Engineering

Doctor of Philosophy

Acceleration and Energy Reduction of Object Detection on Mobile Graphics Processing Unit

by Fitsum Assamnew Andargie

APPROVAL BY BOARD OF EXAMINERS

Dr. Yalemzewd Nagash _____
Dean, SECE, AAiT Signature

Prof. Jonathan ROSE _____
Supervisor Signature

Dr-Ing. Dereje Hailemariam _____
Co-supervisor Signature

Prof. Jason Anderson _____
External Examiner Signature

Dr. Mesfin Kifle _____
Internal Examiner Signature

Declaration of Authorship

I, Fitsum Assamnew Andargie, declare that this dissertation titled, "Acceleration and Energy Reduction of Object Detection on Mobile Graphics Processing Unit" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this dissertation has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this dissertation is entirely my own work.
- I have acknowledged all main sources of help.
- Where the dissertation is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

ቀስ በቀስ እንቁላል በእግሩ ይሄዳል!
የኢትዮጵያውያን አባባል።

In time, an egg will walk on its feet.
Ethiopian Proverb.

Addis Ababa University
Addis Ababa Institute of Technology
School of Electrical and Computer Engineering

Doctor of Philosophy

Acceleration and Energy Reduction of Object Detection on Mobile Graphics Processing Unit

by Fitsum Assamnew Andargie

Abstract

The evolution of high performance computing in today's smartphones is enabling their use in compute-intensive applications. As the compute requirement increases, the energy required to do the computation cannot increase in proportion because the cost of providing that energy available and cooling would become prohibitive. An alternative, potentially power-reducing approach is to use graphics processing units or special accelerator cores. Today's smartphones are equipped with system-on-chip (SoC) devices that house many cores such as graphics processing units, digital signal processors, and special multimedia encoder/decoder hardware along side multi-core central processing units. Their inclusion enables applications that require greater computational power such as real-time computer vision. In this work, we study the capability of the recently introduced general-purpose graphics processing unit (GPU) in a smartphone SoC to enable energy-efficient object detection. This will include understanding the architecture of the recent GPUs that will be used (the Adreno 320 and Adreno 420 from Qualcomm), the implementation and optimization of the object detection algorithm used in the Open Computer Vision library (OpenCV) using these GPUs and measuring the energy consumption of this implementation. We implemented the Viola-Jones based object detection on the GPU in an Android tablet. The implementation is 35% faster on average than the same algorithm running on the CPU on the same device. The implementation also reduces the average energy consumption by 68% compared to the CPU on the same device. An application that utilized the object detector on the mobile GPU to detect Ringworm skin disease was developed. A classifier was trained for this application and it has an accuracy of 75%.

Keywords: *Smartphone, GPU, Object Detection, Acceleration, Ringworm*

Acknowledgements

I had the chance of being in three countries while working on this dissertation, namely; Ethiopia where I am originally from, Canada where I started working on my PhD, the US where I had a six month fellowship. I was fortunate enough to meet amazing people who are kind and accepting. It makes great sense to organize my acknowledgements by country.

I would like to start by expressing my heartfelt gratitude to Professor Jonathan Rose of the University of Toronto who is my thesis supervisor for his unwavering support, guidance, and patience. He has been a continued source of wisdom in the methods of doing quality research, in good teaching skill and striving to learn. I would like also take this chance to thank Professor Rose's family for hosting me whenever I was in Toronto. Next, I would like to thank Braiden Brousseau and Dr. Jason Luu for making me feel at home and teaching me how to survive the winter when I first moved to Toronto and for their technical support with my studies. I would also like to thank Alex Radionov, Daniel Di Mateo and Dr. Henry Wong for their insights which were very helpful in advancing my work. I would like to thank Sergi Lopez-Torres and Brandon Janke who were my room mates for teaching me about the Spanish and Canadian culture and their friendship. I would like to thank my friends Micaela Cristiano, Beatrice Stefanescu, Ross Mujica, Cata Morales, Michael Crimi, Matthew Patience, Olivia Marasco, Erin Hamlyn, Eranga De Zoysa and Aaron Kligerman. They have made my time in Toronto exciting and unforgettable. I would like to thank Tseganesh Temesgen and her family for treating me to Ethiopian culture while in Toronto so I didn't feel homesick.

I am forever grateful for Professor Valeria Bertacco and Professor Todd Austin for their mentorship and willingness to host me as University of Michigan African Presidential Scholar at their Lab. I would like to also thank their students Dr. Biruk Wondimagegn Mamo, Dr. Salessawi Ferede, Zelalem Awoke, Misiker Tadesse, Abraham Lamesgin and Doowon Lee. The discussions I had with them led me to some of my breakthroughs. I would like to specifically thank Doowon Lee for lending me some of the hardware that I experimented on.

I am very proud to be part of the School of Electrical and Computer Engineering community and forever thankful for their support to accomplish my goals. I would like to thank my co-supervisor Dr-Ing. Dereje Hailemariam for his continued support. My colleagues Getachew Teshome, Leul Beyene, Kibrework Alemayehu, Azeb Mekuria for making my time at SECE enjoyable. Last but not least, I would like to thank my parents Ato Assamnew Andargie and W/ro Haimanot Tassew for making me understand the value of education at an early age and for believing in me and my siblings that we can achieve anything we aspire to. Also, sincere gratitude is due my siblings Bekalu Assamnew, Selam Assamnew, Samra Assamnew and Yodit

Abate along with their spouses for their continuous support and unfaltering love. Although not listed here I am grateful to everyone I cross path with everyday for the support and kindness you have shown me.

Thank you!
Fitsum Assamnew Andargie

Contents

List of Figures

List of Tables

List of Abbreviations

1-D	1-Dimension
2-D	2-Dimension
3-D	3-Dimension
AMD	Advanced Micro Devices
AOT	Ahead Of Time
API	Application Programming Interface
ART	Android Run Time
CPU	Central Procesing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GFLOP	Giga Floating Operations Per Second
GHz	Giga Hertz
GP GPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
GTP	Growth and Transformation Plan
HD	High Definition
HDR	High Dynamic Range
HOG	Histogram of Oriented Gradients
JIT	Just In Time
KB	Kilo Byte
LPDDR3	Low Power Double Data Rate version 3
MB	Mega Byte
MHz	Mega Hertz
MLP	Multi Layer Perceptron
NDRange	N-Dimensional Range
NPR	Non-Photorealistic Rendering
OpenCL	Open Computing Language
OpenCV	Open Computer Vision
OpenGL ES	Open Graphics Library for Embedded Systems
PE	Processing Element
RAM	Random Access Memory
SIFT	Scale Invariant Feature Transform
SIMD	Single Instruction Multiple Data
SoC	System On Chip
STDCL	STandard Compute Layer
SURF	Speeded UP Robust Feature
SVM	Support Vector Machines

Dedicated to my parents Assamnew Andargie and Haimanot Tassew.

Chapter 1

Introduction

The progress made by the electronics industry fuelled by Moore's law (the exponential increase over the past 50 years in the number of transistors that is possible to fabricate in a semiconductor chip[1]) has enabled many modern miracles. This includes low-cost and high-performance computers as well as a revolution in handheld smart devices such as smartphones. The increase in the number of transistors in a chip enabled the design of multi-core central processing units (CPU) and many core graphics processing units (GPUs) that are capable of parallel processing. Originally, GPUs had many parallel but fixed-function graphics pipelines which later on were made more flexible so as to do other computations in addition to graphics processing. The capabilty to do more general computations other than graphics tasks in parallel has seen the wide adoption of GPUs for scientific, image processing and many kinds of data parallel applications [2, 3]. As a result, these GPUs were refered to as general purpose GPUs (GP GPUs) (however, in this work we will refer to GP GPU as GPU because virtually all GPUs these days are capable of general purpose computation and there is no need for this distinction). Indeed, many of the top supercomputers, as listed in top500.org [4], are made of hybrids of GPUs and CPUs, leveraging the enhanced computational power available from GPUs. Moreover, since the modern imperative is to achieve higher computational performance per unit of energy required, it is clear that GPUs have contributed to improvements on that front as well [5].

The progress in computing capacity seen in server/desktop processors has also occurred in smartphone and tablet processors. This progress enabled the smartphone to become a more general-purpose computational device rather than mainly being a communications device. A key issue in the evolution of smart phones is the energy storage capacity of their batteries. Battery capacity has not followed the same rate of progress in the electronics industry as dictated by Moore's law. As a result new features that consume too much energy cannot be added to smartphones due to the impact on how long a single charge of the battery lasts. Manufacturers have tried to alleviate this problem in many ways, from integrating special co-processors such as media encoders/decoders and regular GPUs for graphics processing to compute

cores that would only be turned on whenever they are needed. These special hardware cores are designed to be energy efficient. In addition, the GPUs in smartphones have become programmable for general-purpose computation in recent years [6, 7]. In this way they followed in the footsteps of general purpose computers [8].

A second method for reducing energy consumption in smartphones is to optimize the software written in the applications. A first-order way to reduce energy is to find ways to finish a computation more quickly, allowing a processor to go into an energy-saving mode sooner. Such an approach, however, requires understanding of the architecture of the smartphone processor and the tools provided for development. In the very popular Android platform [9], most applications are developed using the Java programming language that runs on top of a Java Virtual Machine called Dalvik [10]. For most applications that are not compute-intensive, developing in Java is sufficient. On top of that, the older Dalvik runtime was equipped with a just in time compiler (JIT) to increase the speed of code sections that are compute-intensive at run time. The more recent Android runtime (ART) introduced Ahead of Time (AOT) [11] compiling of application code to the native machine code. ART later on included the JIT for futher optimization of applications at runtime. Developers can also optimize parts of applications that are compute-intensive using low level programming langugages like C or C++ and make use of capabilities such as the vector processing capability available in many CPU cores. (For example ARM's NEON vector processor[12]).

A third option for reducing energy consumption of an application is to offload some of the computation to the GPU while utilizing the CPU for some other computation. A different approach for the application is to let the processor sleep (saving energy) until the GPU finishes the computation offloaded to it. In addition, there are many ways of optimizing applications that are inherently data-parallel to run in an energy-efficient way on a GPU. For example, as there are different types and sizes of memory in the GPU, how these are accessed and utilized in an algorithm greatly affects performance and energy consumption. Also, the way an algorithm is organized to run on the GPU affects performance.

As stated earlier, data parallel applications have the potential to significantly benefit from the parallel computational capability of a GPU. One class of applications that is highly data parallel are those related to computer vision. Most algorithms in computer vision apply a series of instructions on all pixels or a set of pixels in an image or a video frame at a time. Given the data size in an image or video frame, processing many pixels in a sequential manner places a significant workload on a serial processor. Serial processors by nature lack parallelism, and spend significant amount of time and energy for each scalar operation that they execute. While this is improved with the introduction of vector-based instruction sets, their performance and energy consumption per operation are significantly worse than GPUs

[13]. As such, it makes sense to consider implementing computer vision applications on GPUs, and this has been widely done in the desktop/server space [14, 15, 16, 17, 18]. One can apply the same operation on multiple pixels or set of pixels at the same time on the GPU, which means there is a chance the application can finish much earlier than the previous case and consume less energy than the CPU.

In the early days of general purpose GPU computing, graphics libraries were used to implement general purpose applications for the GPU. This had a significant effect on software engineering costs as general problems had to be transformed into a graphics problem. This issue was first solved by the introduction of the compute unified device architecture (CUDA) [19] and associated programming tools. Using CUDA, programmers no longer need to cast their problems as graphics problems. Non-graphics problems were directly and comparatively easily developed for the GPU. However, CUDA was proprietary and only worked with GPUs, specifically with NVIDIA GPUs. And as discussed earlier, compute systems are becoming more heterogeneous by containing different kinds of processors and co-processors. To enable efficient software development for this heterogeneous compute environment, a number of tools and standards such as the Open Compute Language (OpenCL) [20] have been created. In this work, a smartphone-based GPU is used to accelerate and reduce energy consumption of a computer vision application compared to the same application running on the smartphone CPU. OpenCL is used as the programming tool for the GPU.

1.1 Objectives

The goal of this thesis is to explore if a computer vision application can be accelerated using the GPUs found in smartphone processors while saving energy. We will focus on one of the computer vision algorithms found in the Open Computer Vision (OpenCV) [18] library: in particular the Viola-Jones [21] object detection algorithm. One of the reasons this algorithm is selected is that the Viola-Jones algorithm is popularly used for detecting objects in particular faces in images or videos [22, 23, 24, 25, 26, 27]. The other reason is that this algorithm has parts that are compute intensive and can be parallelized. Going forward the research will focus on understanding the architecture of the mobile GPU and exploring how it can best be utilized in accelerating the Viola-Jones object detection algorithm.

The specific objectives of this work include:

1. The study and characterization of the architecture of the mobile GPU to understand the underlying architecture and use it to optimize the object detection

algorithm.

2. Implementing the object detection algorithm with different optimization techniques.
3. Measuring the runtime performance and energy consumption of each version of the algorithms implemented.
4. Comparing the results with benchmark (OpenCV) implementation of the Viola-Jones algorithm [21] on the mobile CPU to gauge the improvement achieved.
5. Applying the results to implement a medical image classification application on a smartphone - in particular to the detection of the Ringworm skin disease .

1.2 Methodology

In this section the process followed to achieve the objectives of this research is discussed. The tasks involved are performance characterization of mobile GPUs, implementation and evaluation of the object detection algorithm and development of a medical image classification application.

1.2.1 Performance Characterization of the Architecture

The purpose of this first step is to study and measure the parameters and the performance of components of mobile GPUs. The measurements will help in deciding which kind of optimization techniques to use. This is necessary, as the vendor of the GPU we use, Qualcomm Inc., reveals almost nothing about its GPU architecture that is helpful to global optimization of programs. For example, the OpenCL framework lists distinct memory systems, which are referred to as the Host side memory and the Device side memory. The Device side memory is further separated into units with differing locality, size and speed. Measuring the performance of these different kinds of memory helps in determining which memory type to use frequently and which ones to limit the use of. Other measurements to consider are the latencies of common arithmetic operations, the number of work items per work group and the like. The measurements conducted are listed below:

1. The data transfer rate between the Host CPU and the GPU.
2. The throughput and latency of the different types of memory in the GPU.
3. The structure of the memory caching system – how many levels, and size of levels.
4. The aggregate arithmetic computation performance of the GPUs.

1.2.2 Implementation of the Viola-Jones Object Detection Algorithm

In this second step, the Viola-Jones object detection algorithm [21] will be implemented on the mobile GPU using OpenCL. In chapter 2, we will describe the core Viola-Jones object detection algorithm [21] which includes something called 'integral image', 'square integral image', along with the core cascade detection algorithm. Our implementation will employ the following steps:

1. Implement the integral and square integral image computation.
2. Optimize the integral and square integral image computation.
3. Implement the Cascade detection algorithm.
4. Optimize the Cascade detection implementation.

Afterwards the performance of our implementation of the Viola-Jones object detection algorithm [21] on the mobile GPU will be compared with the benchmark implementation. The benchmark implementation that will be used is the multi-threaded Viola-Jones object detection algorithm [21] that resides in the OpenCV library running on the CPU of the same mobile device. The process includes:

1. Measurement of run time for each implementation.
2. Measurement of the energy consumed for each implementation.
3. Comparing of run time and energy consumption with the benchmark implementation.

1.2.3 Application Development

In the last step of our work, we develop an application that utilizes the outcomes of our research. We have selected a medical image classification application in particular the automatic detection of ring-worm skin disease from images. The process of developing this application includes

1. Preparing positive and negative datasets for training.
2. Training of many ring-worm classifiers to choose from.
3. Evaluation of the trained classifiers to choose the best one.

Section 1.3 explains the motivation for this exercise.

1.3 Relevance and Application to Ethiopia

The growth and transformation plan (GTP) of the Ethiopian government stresses the importance of science and technology in the economic growth plans of the country. The health, education, agriculture, industrial and infrastructure sectors are focused on development. Research in these areas is ongoing in different institutions of the country. This research is one of such efforts, and the benefit of is two-fold: first, it helps with the development of educated manpower with proper training in doing research and expertise in mobile applications. Secondly, the outcome of the research can be applied in many of the sectors mentioned above.

In agriculture and health, for example, the government has trained millions of extension workers to help with the development effort. Also, the government has expanded and is developing the coverage of mobile communications infrastructure to cover most of the country. Therefore, in a scenario where an [1]extension worker is equipped with a mobile phone or tablet with applications such as the one developed in this work and given the complete lack or intermittent behaviour of electricity access in their work area, the challenge becomes how long a single charge of the battery used in the smartphone or tablet lasts. The result of this work could be applied to improve the runtime performance and energy efficiency of the applications used in these devices.

1.4 Contributions

In this work, three contributions are made, namely; the performance characterization of mobile GPU [28], the implementation and optimization of the Viola-Jones [21] object detector on a mobile GPU [29] and the training of a cascade classifier for the detection of the ringworm skin disease from an image of a person's skin.

- In the first contribution, measurements were done to understand the mobile GPU's architecture and its related performance. Understanding the architecture helps guide our work and the work of others to enhance performance of applications on the mobile GPU. We show that the CPU and GPU in smartphones are connected to a single main memory hence data can be directly shared by programs running on both systems unlike their desktop counterparts. This presents an opportunity for an improved application performance

[1]Extension worker: is an employee of the government who has a basic training in the area of health and agriculture. The goal of extension worker is to provide basic training to the masses with regards to health and agriculture practices.

where explicit data copying is avoided as we have used this in our implementation. However, the access patterns to this memory used on the GPU can significantly affect performance. In order to overcome these effects, memory access should be coalesced as our measurements show. Additional architectural measurements include a quantification of the amount of parallelism available in the specific GPU, which is hidden by the vendor. Knowing this available parallelism helps with planning the arrangement of processing threads for a particular algorithm. This is important because some device manufacturers do not specify how many parallel processing elements are in their device and how threads are scheduled on the processing elements.

- The second contribution in this work is the implementation and optimization of the Viola-Jones [21] object detection on a specific mobile GPU. Our version of the object detector ported to the mobile GPU was measured to be 35% faster in runtime on average and more than two fold faster in runtime for the best case on the largest and hardest Full HD images as compared to an optimized parallel implementation on the CPU of the same mobile device. The energy consumption measurement for the ported object detector on the mobile GPU shows a best case of 84% reduction for Full-HD images. The object detector on the GPU was slower than the CPU-based object detector for images that are smaller than the Full-HD images. However, it had an improved energy consumption where the minimum energy consumption reduction measured was $\sim 35\%$. This indicates that porting data parallel applications to run on the mobile GPU makes them energy efficient even if the applications do not gain in runtime improvement.

- The third contribution is using our object detector for the ringworm skin disease detection on the mobile GPU. We chose the ringworm skin disease as it has a large prevalence in sub-saharan Africa [30]. Also, it easy for a person to take a picture of skin using a mobile phone and the image fed to object detector. However, the object detector needs a classifier to be used in the detection process. Hence, a classifier for the ringworm skin disease detection was trained from a set of positive and negative samples. This trained classifier with the available dataset has an accuracy of 75%. In order to improve the accuracy of the classifier a higher number of positive and negative samples are needed during training.

1.5 Thesis Organization

The remainder of the dissertation is organized in the following way: Chapter Two will give background on general-purpose GPU computation using OpenCL, describe the Viola-Jones [21] object detection algorithm and give some related work

in the use of the mobile GPU to accelerate various applications. In Chapter Three, the methodology used, the experiments done and results collected for the performance characterization of the mobile GPU architecture will be detailed. Chapter Four will describe the implementation of the object detection algorithm on the mobile GPU and the different modifications done to it to improve runtime performance and energy consumption. Then, Chapter Five will give a description of the process used to develop a medical image classification application, namely: ringworm skin disease detection on the mobile GPU. Lastly, Chapter Six will conclude on the work and give future research directions.

Chapter 2

Background

The first section of this chapter will give a brief overview of the Open Computing Language (OpenCL) which is used for general purpose computation on highly-parallel systems. The next section will describe a widely-used method for object detection: the Viola-Jones object detection algorithm [21]. The last section will summarise work related to the goal of this project: making use of a mobile GPU for general purpose computing.

2.1 General Purpose Programming on GPUs

Ever since graphics processing units evolved from the graphics display processing units and became relatively programmable, researchers have used them for general purpose computation. Early on, the general purpose algorithms needed to be represented in terms of graphics API primitives because the first standards and languages developed for these GPUs were targeted to representing graphical objects, such as triangles, polygons and textures. Owens et.al. [2], describe many efforts that have been made to develop high-level programming languages for GPUs. All such efforts strive to abstract the underlying hardware architecture so that it can be programmed easily. The earlier languages still needed the programmer to understand how to map the computation into graphics primitives such as vertex and texture shading. The authors note that the streaming programming model is the dominant model employed.

The first widely accepted high level programming language for general GPU computing was developed by NVIDIA. This C-based programing language was called CUDA [19]. It exposes much of the architecture of NVIDIA's GPUs to the programmer. There was no need to represent algorithms as graphics objects. Programmers can directly allocate buffers on the GPU and utilize them as they see fit. In addition, threads could be organized in a way that best suits the application. However, CUDA was proprietary and only worked on NVIDIA GPUs.

Another high level programing language that was initiated by Apple Inc. was the open compute language (OpenCL). OpenCL was Apple's way of standardizing the way general purpose computing is done on GPUs and/or any other accelerator co-processor. Many vendors have joined the consortium started by Apple and the Khronos group was given the responsibly to maintain and evolve the OpenCL standard [20].

2.2 Open Computing Language (OpenCL)

The open computing language standard was developed to work on any kind of compute device. These compute devices can be CPUs, GPUs, Digital Signal Processors (DSPs) , Field Programmable Gate Arrays (FPGAs) and so on. The main purpose of OpenCL is to express a standard compute model and develop definition of APIs for the standard thereby allowing one to express parallelism that might be successfully deployed accross a range of architectures. Every manufacturer/vendor of a compute device is responsible to develop an OpenCL library/driver to implement the APIs and kernel compilers for their device. OpenCL gives the programmer specific ways to express parallelism present in the code. The same code can be run on different devices with no or minimal alterations. However, the performance of the code on different architectures is not guaranteed to be the same [31]. In cases where performance degrades, optimizations need to be done by understanding the underlying architecture.

2.2.1 The OpenCL Architecture

The OpenCL Architecture defines a hierarchy of models [32]. These are platform model, memory model, execution model and programming model. The first two models define an abstraction of a hardware architecture that is easy to understand. These two models clearly describe how the available computational devices and their associated memory are organized. The execution model describes how computation can be divided and scheduled between the available computational devices. The programming model defines how parallelism can be achieved; data parallel or task parallel.

The Platform Model

The platform model pre-supposes the existence of a ***host*** computer, which is typically a general purpose CPU, and several other compute devices that act as accelerators. A compute device is further subdivided into compute units (CUs). Each

compute unit usually has one or more processing elements (PEs). The processing elements in each compute unit execute a single stream of instructions as Single-Instruction Multiple-Data (SIMD) units [33]. The compute devices in the platform model can be from different vendors. The combination of one or more compute devices and associated drivers from a particular vendor are referred to as a platform in the platform model. There can be one or more platforms in a system. Figure 2.1 shows the platform model.

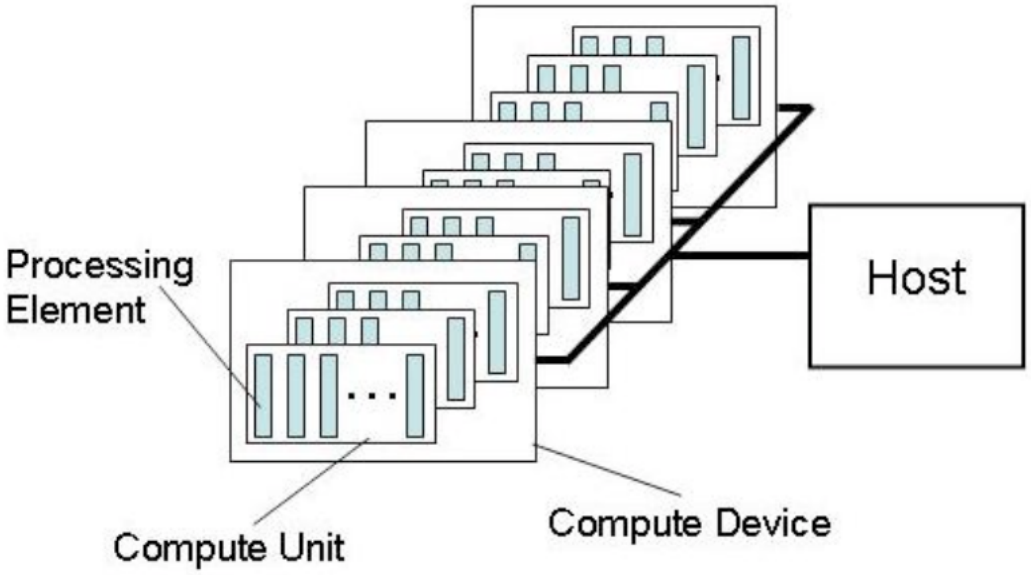


FIGURE 2.1: OpenCL Platform Model (Reproduced from[32])

The host is responsible for identifying the OpenCL platform available in a system. As many platforms can exist in a system, the host code needs to choose/handle which platform to use for the execution of the OpenCL code. The OpenCL platform usually provides a compiler that converts the OpenCL code to run on the OpenCL device. This compiler can be an offline compiler, where the OpenCL code needs to be compiled before runtime, or an online compiler that compiles OpenCL code at runtime. OpenCL defines two platform profiles: the embedded profile and the full profile. The embedded profile has restricted functionalities and is not expected to have an online compiler whereas the full profile is expected to do so. As these compilers are device and/or vendor specific, device vendors are at their discretion to include online compilers for the embedded profile while they are required to provide the online compiler for a full profile device.

The host is also responsible for creating and associating a context with devices in the platform. A context keeps track of the memory objects and kernels associated with a particular application. Defining a context is necessary as there may be other applications running on the platform at the same time. On the other hand, if the application wants to partition parts of the work it can create multiple contexts. This

way it can ensure the consistency of data used in different parts of the code.

OpenCL Memory Model

The OpenCL memory model describes how memory is organized on an OpenCL platform, and is illustrated in Figure 2.2. There is host side memory, which is usually the main system memory, and device side memory in a typical OpenCL platform. The memory on the OpenCL compute device is organized hierarchically as global memory, constant memory, local memory and private memory. The large global memory and the constant memory are accessible by all compute units as well as each processing element in the compute unit. This makes the global memory visible to all threads (threads are referred as workitems in OpenCL nomenclature) running in an OpenCL based application. The global memory is significantly slower than the smaller constant, local and private memories which means kernels must organize global memory access to hide its access latency. The local memory is very small in size as compared to the global memory and is shared by the processing elements in a compute unit. However, the local memory is usually an order of magnitude faster than the global memory. The private memory is also a shared resource on a compute unit that can be allocated to each work-item/thread residing in a processing element. A variable allocated on the private memory by a work-item/thread is not accessible by any other work-item/thread. The global, constant, local and private memories are programmable and the programmer manages them explicitly. However, there is often a separate hardware-controlled cache for the global and constant memory on the OpenCL device.

In the case where many OpenCL devices exist in a system or there is a hardware support for sharing a part of the host memory, a shared virtual memory address space is used. The shared virtual memory address space allows for direct data sharing between the devices and/or with the host with out issuing explicit copy commands on the host side. Consistency of the shared data is assured at synchronization points and atomic operations if supported. The synchronization can happen both from the host and device side.

When defining data structures in OpenCL programs, address space qualifiers need to be used to specify on which part of the memory hierarchy an object is to be allocated. These qualifiers are **__global**, **__constant**, **__local** and **__private**. The address space qualifiers can be used without the double dash prefixes as **global**, **constant**, **local** and **private** (Listing 2.2 and Listing 2.3 show both usages). If a variable in a kernel is defined without an address space qualifier, it will be allocated on the default (generic) address space which is the private memory. The **const** qualifier can also be included with the **__global** qualifier to specify the object is read only. All kernel and device function arguments are defined on the private memory address

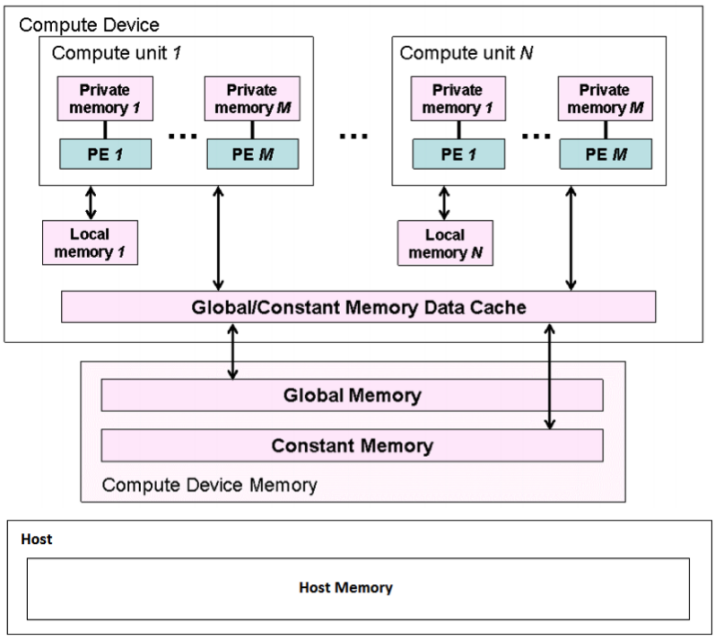


FIGURE 2.2: OpenCL Memory Model (Reproduced from[32])

space but in case of pointers they can point to the other address spaces. Listing 2.1 gives a vector addition example that shows how these address qualifiers are used in a kernel (kernel: OpenCL functions that run on the OpenCL device).

LISTING 2.1: Vector addition kernel

```
1 __kernel  void vector_add(__global const float* A,
2 __global const float* B,__global float* C,int dataSize)
3 {       //tid and dataSize are allocated on the private memory
4         int tid=get_global_id(0);
5         if(tid<dataSize)
6         C[tid]=A[tid]+B[tid];
7 }
```

OpenCL Execution Model

In the OpenCL execution model there is code that runs on one or more OpenCL devices and code that runs on the host. The host code manages the code that runs on the OpenCL device and the context it runs on. When a kernel is submitted for execution, an index space is created on the context. An instance of the kernel runs for each item in the index space. This kernel instance running on the device is called a work-item. Each work-item resides on a processing element when it is executing. Work-items are grouped in to a work-group that are scheduled to run on a single compute unit. Each work-item is assigned global-id that uniquely identifies it in the index space. Similarly, each work–group is assigned a unique index. In addition,

each work-item in a work-group is given a unique local-id that identifies it in the work-group.

Threads/work-items in an OpenCL program can be arranged in an N-Dimensional Range (NDRange), where N may be 1, 2 or 3. A range is defined as index space from which reference id can be assigned to a thread/work-item. The index space can be 1-Dimensional, 2-Dimensional or 3-Dimensional and its bounds vary from device to device. The arrangement of threads/work-items in this way can help with managing algorithms in a more natural way. Consider the following 2-Dimensional index space in Figure 2.3. Such arrangement is ideal for matrix manipulation, 2-D image processing and the like. Work-items are usually grouped together into a work-group. The purpose of this grouping is to schedule a group of work-items to execute in a given compute unit. Another benefit of grouping work-items, be it explicit or defined by the OpenCL implementation, is in synchronizing the various kids of parallelism available in OpenCL. However, synchronization is limited to work-items in a work-group. OpenCL does not provide built in global work-item synchronization.

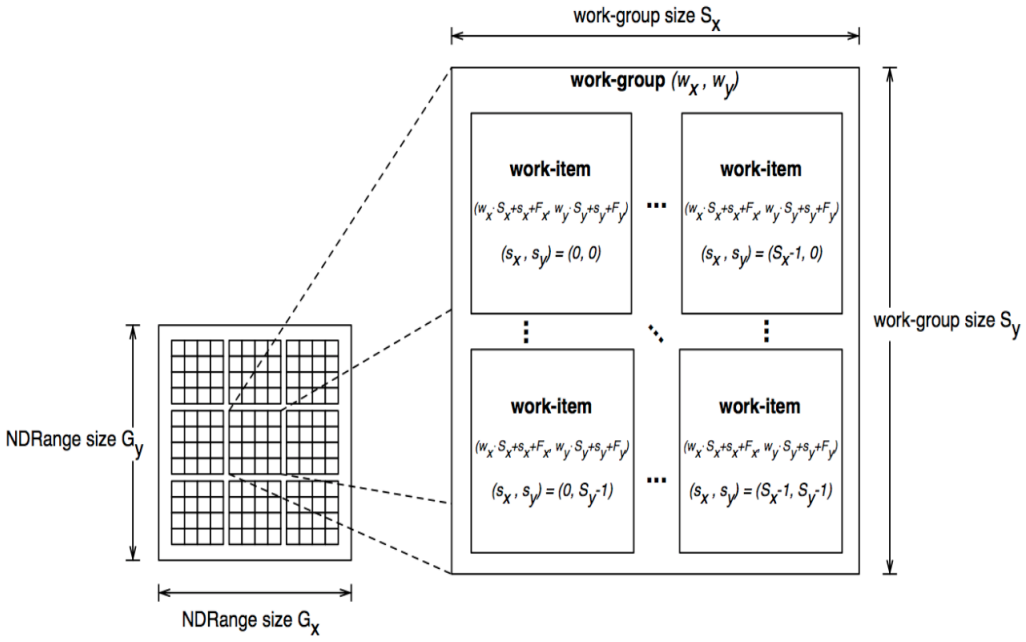


FIGURE 2.3: An example of two dimensional work-item index space.(Reproduced from[32])

For the vector addition example given in Listing 2.1, the ideal set up of the work-items will be a single dimensional range (1D-Range) because vectors are linear data structures. We can instantiate as many work-items as the number of vector elements until the allowed maximum work-item size for the OpenCL device is reached. (One can query the device to know what this maximum number is.) In the case where the number of vector elements is greater than the max allowed work-item size, each work item can be made to work on more than one vector element. The next step is to define how these work-items are grouped. The OpenCL runtime is able to determine

the best configuration or grouping can be specified by the programmer. However, finding a good grouping may not be straight forward. It is advised to experiment with different work-group sizes and use what works well.

We use matrix multiplication as another example to illustrate how work-items can be organized. Here, setting up the work-items as a 2-dimensional grid (2D-Range) is ideal because of the 2-dimensional nature of matrices. The single-threaded matrix multiplication C-code given in Listing 2.2 can be converted as the OpenCL Kernel shown in Listing 2.3. It can be seen that the outer two loops of the C-code have been converted into an OpenCL kernel. Each work-item now will have two global_ids that correspond to indices from both dimensions of the matrix. The length of each dimension for our particular case is set to $(RowA, ColB)$. This way, it is ensured that there will be enough work-items to do the job. For example, if matrix **A** and **B** have dimensions of $(204, 576)$ and $(576, 367)$ respectively, then ideally we will require $204\ x\ 367\ =\ 74,868$ work items to do the job. This particular number of work-item dimension size would work if the work-group size is set to $(1, 1)$ which is very inefficient on most OpenCL enabled systems. Depending on the system, a more reasonable work-group size should be chosen, although this may lead to the readjustment of the global work-item sizes. For instance, if one chooses an $(8, 8)$ work-group size, the global work-item size for the previous example needs to be adjusted to the nearest multiple of 8 as $208\ x\ 368\ =\ 76,544$. However, as more work-items are instantiated than actually needed, one has to insert checkpoints in the kernel code as done on $line\ 9$ of Listing 2.3 to avoid index out of bound (buffer overflow) errors.

LISTING 2.2: Matrix multiplication in C

```
1  void C_MatMul(float* A,float* B, float* C,
2  int RowA,int ColRowAB,int ColB )
3  {
4    float sum=0.0;
5    for(int i=0;i<RowA;i++)
6          for(int j=0;j<ColRowAB;j++)
7          {
8            sum=0.0;
9            for(int k=0;k<ColB;k++)
10                 sum+=A[i*ColRowAB+k]*B[k*ColRowAB+j]
11           C[i*ColRowAB+j]=sum;
12         }
13 }
```

LISTING 2.3: Matrix multiplication in OpenCL

```
1  __kernel void OpenCL_MatMul(global float* A,global float* B,
2  global float* C,int RowA,int ColRowAB,int ColB)
3  {
4
5     int i = get_global_id(0);    //work-item id in dimension 1
```

```
6       int j = get_global_id(1);   //work-item id in dimension 2
7       float sum = 0.0;
8
9       if ((i < RowA) && (j < ColRowAB)) {
10          sum = 0.0;
11          for(int k = 0; k < ColB; k++)
12          {
13              sum += A[i*ColRowAB + k]*B[k*ColRowAB + j] ;
14          }
15          C[i*ColRowAB + j] = sum;
16      }
17  }
```

An OpenCL application has host side code and device side code (kernel code) as mentioned earlier. In order to successfully run the application and get results, the following procedures need to be done on the host side code

1. Initialize OpenCL Device
    - Get available platforms
    - Get available devices in the discovered platforms
    - Create a context/contexts on the selected platform/platforms
    - Create command queues and associate with a context/contexts
2. Build Program
    - Read kernel source/binary from a file/files
    - Create a program/programs with the source/binary
    - Build the program/programs
    - Check if the program/programs compiled successfully
3. Create Buffers
    - Create buffers on the selected context with appropriate access behaviour and sizes
    - If explicit copying of data is needed, enqueue buffer write operation
4. Set arguments and Enqueue kernels
    - Set arguments to the kernels
    - Set the sizes of global work-items and work-groups
    - Enqueue the kernel
5. Get back results
    - Enqueue Read buffer operation to read data back from the GPU if explicit copying is required.

The OpenCL APIs used in the host code usually give feedback on the successful completion of the task at hand. Therefore, it is advised that the host side code should always check the success of any OpenCL API call. The error code returned from these API calls can be interpreted to help with identifying problems/bugs in the OpenCL application. Appendix A gives the complete Matrix Multiplication application source code with commentaries.

2.3 Computer Vision

Computer vision algorithms allow computers to understand their environment from visual inputs. The visual inputs can come from stored images, videos, or a camera. The purpose of computer vision is to interpret this visual input for some purpose, such as object detection and modelling of an environment from a stored video or camera feed. The main barrier to the advancement of computer vision is its requirement for large amounts of computational power. This computational requirement arises from the immense data generated by visual inputs and the inherent difficulty of the vision task itself. There is an ongoing effort to improve existing algorithms to perform better as well as create new algorithms that are better and can utilize alternate computing facilities available in a system. This is most relevant and important when we seek to achieve real-time computer vision, which has hard performance constraints. Many of the algorithms developed (prior to the recent advances in Deep Neural Nets in this area) are incorporated in a library called the Open Computer Vision (OpenCV) library [18].

The Open Computer Vision collaboration is engaged in continuous development of a library that can be used free of charge for many kinds of computer vision applications. This collaboration collects many of the widely accepted computer vision algorithms and packages them in a library that is well tested and ready to use. This OpenCV library incorporates more than 2500 computer vision algorithms grouped in modules such as image processing, video analysis, machine learning, object detection and so on. These algorithms are available for different processor architectures and different programming languages. The OpenCV library supports the acceleration of the computer vision algorithms on GPUs using CUDA and OpenCL (but not yet on mobile GP GPUs as of OpenCV version 3.1). It also supports the Windows, Linux, Mac OS, iOS, and Android operating systems.

The object detection module in OpenCV is used for detecting objects in images and videos. There are several different object detection algorithms, including the Support Vector Machine (SVM) approach [34], Histogram of Oriented Gradients (HOG) [35], and Cascade Classifier [21]. Since the Cascade Classifier based object detector is the focus of this work, we will describe it in more detail next.

2.4 Cascade Classifier based Object Detector

The cascade classifier based object detector was first introduced by Viola and Jones [21]. Their object detector uses a set of basic binary classifiers arranged in stages with each stage consisting of a set of features used to evaluate some feature of the object. These basic classifiers are called *weak classifiers* as they are expected to be correct slightly more than 50% of the time. However, the combination of weak classifiers in a cascade of stages gives rise to a *strong classifier*. The classifier stages can consist of a few weak classifiers in the early stages and upto hundreds of weak classifiers at later stages.This strong classifier is called a cascade classifier as it is composed of cascade of classifier stages.

The cascade classifier is produced by training it with positive and negative samples, where positive samples are images of the object of interest and negative samples are background images that do not include the object. The training process uses the AdaBoost algorithm [36] to learn the weak classifiers that give rise to the strong cascade classifier. A fixed size of image of the object is used during training which will later be used in the detection process.

In the process of detection, a sliding window approach is used. A window in this context means a region of an image where the presence of the object of interest is being checked and is equal to the size of the object at training time. In the detection process, the first stage of the cascade classifier is applied to a window. The next stage classifier will only continue on the same window when the first stage of the cascade classifier passes. If the first stage fails, there is no need to continue with the subsequent classifier stages, as the object is deemed not found in this window. The object is deemed to be found in this window if all the stages pass. Figure 2.4 illustrates this process. Once the decision is made on the particular window, the same procedure is repeated on the subsequent, adjacent windows until the whole image has been explored.

There are two reasons the object may not be detected in the image/window. One reason is that the object is indeed missing or is too small to be detected. The other is in case the scale of the image is wrong that the object does not fit into the window under test. In order to address this later issue, the Viola-Jones [21] method scales down the image by a certain constant factor. The scaling is needed as the basic classifiers are trained on a fixed window size. Once the image is scaled down, the detection procedure is applied at the new scale. The recursive scaling down process terminates at the point that the entire image fits into a single window.

The features used in the Viola-Jones [21] approach are called Haar-like features that compute local oriented intensity difference using rectangular blocks. The rectangular blocks used in their work are upright rectangles where the rectangles lie

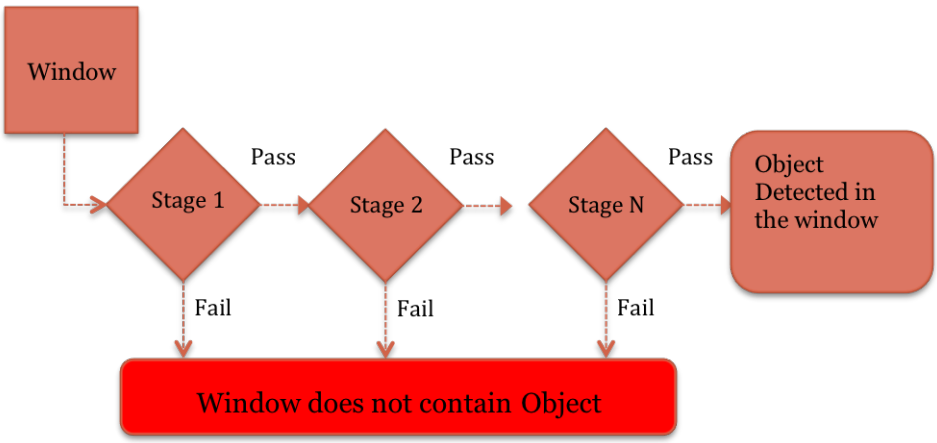


FIGURE 2.4: The Cascade Classifier Process

horizontally or vertically never at an angle. The work by Leinhart and Maydt [37] introduced rotated Haar-like features that improved the quality of the object detector. The inclusion of the rotated feature rectangles lowered the false alarm rate. Both works use a cascade of boosted classifiers in the object detection process. In this work however we will only be focusing on the work by Viola and Jones [21] and the terms "object detector" or "cascade classifier" will mean the Viola-Jones object detector and will be used interchangeably.

The key concepts in the Viola-Jones [21] object detector discussed earlier will be detailed in the following sections beginning with the key mechanism used in the classifiers, the Haar-like features. The structure of the classifier cascade will then be discussed. The process of the detection procedure will be discussed followed by the optimizations used in the detection procedure.

2.4.1 Haar-like Features and Classifier Matching

The Haar-like features are rectangles that represent some feature of an object. Figure 2.5 shows some Haar-like features used in the object detection algorithm. A feature used in a classifier is specified by its shape, its position within a region of interest and the scale. The coloring of the rectangles shows the contribution of the colored region when taking the sum of underlying pixel values of the input image. Black pixels make positive contribution while white pixels make negative contribution. Listing 2.4 shows the representation of Haar-like features in a program.

Listing 2.4 shows a similar representation of Haar-like features as found in OpenCV in **C** programming language. As one can see, the **HAAR_feature** type definition has a boolean data to indicate the presence and absence of tilted features. However, it is to be remembered that in this work only upright rectangle features are

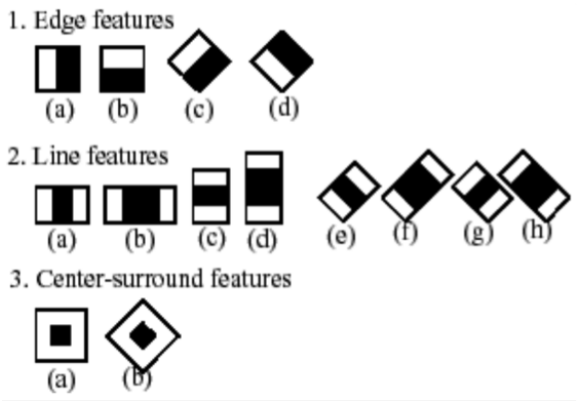


FIGURE 2.5: Haar-like features (Reproduced from[17])

used. The way a feature is represented is by a set of rectangles with associated weights as depicted in Listing 2.4 by the **rects** object. These associated weights have a positive or negative values depending on their color as shown in Figure 2.5. **MAX_HAAR_FEATURE_SIZE** is a macro definition that allows the user to specify the maximum number of rectangles per feature as the number of rectangles per feature can vary depending on the object that is being sought after.

LISTING 2.4: Represenation of the HAAR-like features in a program

```
typedef struct
{
  bool tilted; //tilted features present??
  struct
  {
    Rect f;              //feature rectangle
    float weight;        //feature rectangle weight
  }rects[MAX_HAAR_FEATURE_SIZE]; //rectangles per feature
} HAAR_feature;
```

Cascade Classifier

The building blocks of the cascade classifier are a set of weak classifiers. These weak classifiers are composed of one or more Haar-like features. The Haar-like features in these weak classifiers are evaluated to test for the existence of a certain feature of the object. The term weak classifier is given because the level of certainty expected from these classifiers is just over 50%. If these weak classifiers are grouped together, they tend to push the detector to the right direction in detecting the object. In other words, if the collective certainty is below a threshold, the object is definitely not present in the current location of the image, meaning further scrutiny of the location is not necessary. Conversely if the threshold is surpassed, the object probably is found in the current location and more testing is needed. Hence, when taken together a group

of weak classifiers is called a strong classifier. The strong classifier is sufficient to detect if the object exists in an image. Figure 2.6 shows the structure of a stage of a strong classifier.

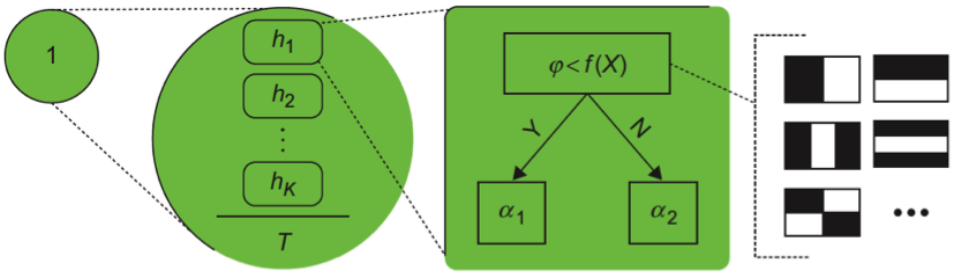


FIGURE 2.6: One stage of the cascade classifier in OpenCV (Reproduced from [17])

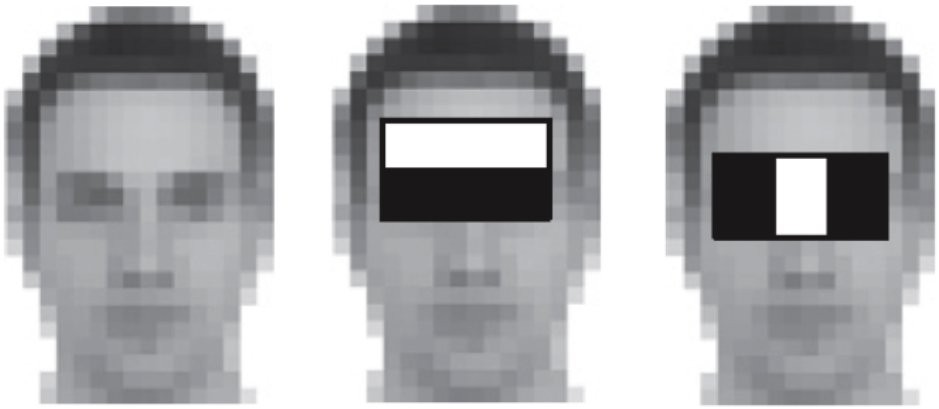
As one can see from Figure 2.6, each weak classifier $h_i(X)$ is evaluated on the region of interest. These evaluations result in either constant α_1 or constant α_2 value for each weak classifier. α_1 is a value less than the threshold for the specific weak classifier while α_2 is above the threshold. However, these individual values do not determine the the existence of the object in the particular window by themselves unless the stage is composed of only one weak classifier. Instead, the results of each weak classifier in a stage are summed up and compared with a threshold for the stage. If the sum is greater than the stage threshold, then the object is likely to be found in the window requiring further scrutiny with the subsequent stages. In the case the sum is below the threshold, the object is deemed not found in the current window and no further processing of the window is required. The computation process is shown in equation 2.1.

$$H(X) = \begin{cases} 1, & \sum_{i=1}^{K} h_i(X) \geq T \\ 0, & \text{otherwise} \end{cases} \qquad h_i(X) = \begin{cases} \alpha_1, & f_i(X) > \varphi_i \\ \alpha_2, & \text{otherwise} \end{cases} \tag{2.1}$$

where X is the region of interest(window) defined by MxN pixels.

Figure 2.7 shows an example of feature evaluation in the detection process discussed above. The object being detected in this example is face of a person as shown in 2.7a. The first feature overlayed on the face in Figure 2.7b checks for the forehead because usually the forehead is brighter than the area around the eyes in comparison. Similarly the next feature as in Figure 2.7c evaluates for the eyes with the assumption that the nose bridge is brighter than the eyes. This way all the features determined during the training process will be applied on the window of interest through the weak classifiers discussed above.

The way the cascade classifier is represented in a **C** program is given Listing 2.5.



(a) (b) (c)

FIGURE 2.7: Haar-like Feature Evaluation Example (Reproduced from [17])

The cascade classifier used in this work is based on the stump weak classifier. A stump weak classifier has a two level binary tree of features. This means after a feature in the root node is evaluated, a choice is made to evaluate the left child feature if the feature evaluated is greater than weak classifier threshold or right child feature is selected otherwise. Hence, the **HAAR_Treenode** object in Listing 2.5 represents a node of the binary tree described above. The **HAAR_Weak_Classifier** object contains a count of nodes in the decision tree, a pointer to root node of the tree and a pointer to the alpha values to be selected. The next object is the **HAAR_Stage_Classifier** that is used to represent one stage of the cascade classifier. This object contains the count of weak classifiers in the stage, the stage threshold and a pointer to the list of weak classifiers in the stage. The last object in the representation of the cascade classifier is the **HAAR_Classifier_Cascade**. It stores the count of stages in the classifier cascade and keeps a pointer to the list of classifier stages among other values.

LISTING 2.5: Represenation of the Cascade Classifier in a program

```
typedef struct
{
  short left;   //points to left node
  short right;  //points to right node
  short featureIndex; //points the feature to evaluate
  double threshold;      //weak classifier threshold
} HAAR_TreeNode;

typedef struct
{
  short count; // number of nodes in the decision tree
  HAAR_treeNode *treeNode;      //pointer to the tree structure
  /* pointer to alpha values chosen based on the evaluated
     threshold */
  double *alpha;
```

```
16 } HAAR_Weak_Classifier;
17
18 typedef  struct
19 {
20   int count;  // number of stages in the battery
21   double threshold; // stage threshold
22   // pointer to array of weak classifiers in a stage
23   HAAR_Weak_Classifier* classifier;
24 } HAAR_Stage_Classifier;
25
26 typedef struct HAAR_Classifier_Cascade
27 {
28   int count; // number of stages
29   /*original object size the classifier is trained for*/
30   Size orig_window_size;
31   Size real_window_size; // current object size
32   /* pointer to the array of stages */
33   HAAR_Stage_classifier* stage_classifier;
34 } HAAR_Classifier_Cascade;
```

Integral Image

The features are evaluated by subtracting the weighted sum of pixel values under the black rectangular regions of the input image from the ones under the white rectangular regions. There can be hundreds of weak classifiers in a particular cascade classifier each of which may have one or more features. Also, the classifier is applied to windows of the image at different scales which results in a large set of windows to be tested. If the sum of pixels is computed every time a feature is evaluated, the computational need would be so large that the use of this object detector would be ineffective. One innovative approach Viola-Jones [21] introduced was an intermediate image representation called the ***integral image***. In order to reduce the computational complexity, the sum of pixel values preceding and including the pixel value at every point in the entire image was computed and stored in an integral image (equation 2.2). The integral image is also computed for all the scales of the original image. This way, we can avoid the redundant computations and computing sum of pixels in a rectangular region of the image can be done in constant time ($O(1)$).

$$IntImg(x, y) = \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} image(i, j) \tag{2.2}$$

where $image$ is the original image and $IntImage$ is the resulting integral image.

Figure 2.8 demonstrates the efficiency of the integral image representation. For example, if one wants to compute the sum of the pixels in the rectangular region D. In order to achieve this, one only needs to consider points 1, 2, 3 and 4 of the integral

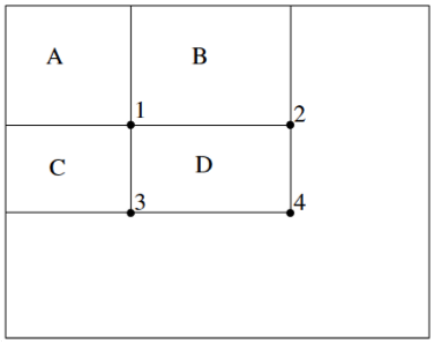


FIGURE 2.8: Integral Image Representation Example (Reproduced from[17])

image. The value stored at point 1 is sum of the pixel values in region A, at 2 we have $A + B$, at 3 we have $A + C$ and at 4 the value is $A + B + C + D$. Therefore, the sum of pixel values in region D can easily be found by four array references as shown in equation 2.3.

$$IntImg(4) + IntImg(1) - IntImg(2) - IntImg(3) = sum(D) \tag{2.3}$$

The square integral image, which is similar to the integral image, is the sum of the squares of each pixel before and including the point as shown in equation 2.4. It is used when computing the variance to normalize the image.

$$sqIntImg(x, y) = \sum_{i=0}^{x-1} \sum_{j=0}^{y-1} image(i, j)^2 \tag{2.4}$$

where $sqIntImage$ is the resulting square integral image.

Normalization of Lighting Conditions

The sample images used during the training and the detection process are usually taken at different lighting condition. This would affect the quality of the classifier trained as the lighting condition can introduce a bias. In order to minimize the effect of lighting conditions the classifiers are trained on a variance normalized image. Also, the image to be used for detection needs to be variance normalized as well. The variance of a region of an image can be easily computed from the integral image and square integral images as shown in equation 2.5. To be variance normalized the pixels of the region need to be divided by the standard deviation, which is the square root of the variance.

$$Var(image) = \frac{1}{MN} \sum_{i=0}^{M} \sum_{j=0}^{N} image(i, j)^2 - (\frac{1}{MN} \sum_{i=0}^{M} \sum_{j=0}^{N} image(i, j))^2 \tag{2.5}$$

The Viola-Jones Object Detection Algorithm

The complete algorithm for the Haar-like based object detection as listed in [17] is reproduced in Algorithm 1. In this algorithm, the integral image and square integral image are computed first for all scales of the image. This means the target image needs to be resized into all the scales. The nearest neighbor and the bilinear interpolation image resizing techniques can be used for this purpose.

Algorithm 1 Cascade Classifier-based Object Detector

```
1: Build integral image I(F) and square integral image SI(F)
2: Set curScale = 1.0
3: for all scales do
4:     curScale* = S
5:     for curRegion in all regions X on the current scale do
6:         for H_i(x) in all cascade stages do
7:             StageSum = 0
8:             for h_j(x) in all weak classifiers of H_i(x) do
9:                 StageSum+ = calculate weak classifier h_j(x) using I(F) and SI(F)
10:                if StageSum < StageThreshold then
11:                    Mark region as non-object and proceed to next region
12:        Mark region as object
13: Partition and filter regions marked as objects
```

2.5 Previous Related Work

Several researchers have previously tried to utilize the low power mobile GPU to improve the power consumption and performance efficiency of mobile applications while performing general purpose computing on it. In this section we present a survey of prior work on the acceleration of general purpose algorithms on a mobile GPU. We have identified that many earlier works used the open graphics library for embedded systems (OpenGL ES) for representing their general problem as a graphics problem on the mobile GPU while recent research uses OpenCL for direct implementation of algorithms on the GPU. The rest of this section is organized according to the programming framework the research used on the mobile GPU.

OpenGL ES for General Purpose Computing

A survey done by Pulli et al. [38] identified programming frameworks such as OpenGL ES and OpenCL to be used for implementing applications on the mobile

GPU. They suggested that mobile GPUs can be used to accelerate augmented reality applications including object detection and tracking, and scene modelling. They also explored the computational photography applications high dynamic range (HDR) imaging and panorama capture.

The work by Cheng et. al [6] mapped the face recognition problem to a graphics-rendering paradigm. The face detection part of this work was done using an Android API. For the recognition part, they implemented the Gabor wavelet using a Fast Fourier Transform. They used OpenGL ES on an NVidia Tegra SoC. The face recognition took about 8.5 seconds on the CPU while taking only 4.6 seconds on the GPU and consuming $16.3J$ of energy in contrast to the CPU only implementation's $29.8J$. This shows an almost $2x$ speed up was gained using the GPU while also lowering the energy consumption by 45.3%.

In the work done by Rister et al. [39], the Scale-Invariant Feature Transform (SIFT) detector was implemented on a mobile GPU. SIFT is an image descriptor for image-based matching and recognition developed by David Lowe [40]. In their approach, data was partitioned between the CPU and GPU in order to maximize efficiency. The parts of the SIFT detector that can be quickly computed in the CPU were scheduled to run on the CPU. The ones that take longer and can be computed in parallel (the algorithms that are data parallel) were scheduled to run on the GPU. In this way, they were able to minimize data movement between main memory and GPU memory, which is a slow process. Another innovation of this work is the use of data compression by pixel reordering. Since a SIFT detector works on gray scale images, 4 gray scale pixels can be copied into one texture pixel which expects red, green, blue and alpha values. This effectively reduced the data transfer requirement by up to four times. They report a significant energy consumption reduction (87%) as compared to CPU only implementation when using the CPU+GPU in combination.

Lee et. al [41]] used a mobile GPU for an augmented reality application where they applied computer-vision techniques for tagging spaces for augmentation. Their work involved the learning of a patch of space to be augmented and then detecting and tracking the tagged space. In addition, they used the phones sensor's for pose estimation. They conclude that the GPUs in the phones enable a near real time Anywhere Augmentation.

Ensor and Hall [42] implemented the Canny edge detection on mobile GPUs using OpenGL ES 2.0. The implementation moved the entire pipeline in the Canny edge detector to the GPU. The Gaussian blurring, the gradient vector computation, the non-maximum suppression, double threshold and their own tweaks to the Canny algorithm are all done on the GPU. They report significant frame rate improvement with the GPU implementation for some of the mobile devices that were tested with $640x480$ resolution.

Hofmann et. al [43] implemented an upright speeded up robust features (SURF) descriptor on a mobile GPU called uSURF-ES. They used OpenGL ES 2.0 and C++ for programming their application. Their implementation on the GPU was compared against the upright SURF in OpenCV, which is not multi-threaded. The comparison was made on different mobile devices and tablets, and speed-ups ranging from $2x$ up to $14x$ were reported.

Singhal, Park and Cho [44] used OpenGL ES 2.0 to implement an image processing tool kit. The image processing tool kit includes algorithms such as Gaussian smoothing, edge detection, color conversion, and bilateral filtering. Optimization techniques such as floating point control, loop unrolling, branching control, sharing computation load between the vertex and fragment shaders and texture compressing were considered when the image processing toolkit was implemented. Cartoon style non-photorealistic rendering (NPR), speeded up robust features (SURF) and stereo matching applications were implemented on a system equipped with an ARM Cortex A8 at 1GHz and a PowerVR SGX 540 GPU at 200MHz to measure the GPUs performance using their tool kit. The authors reported speed up of $5x$, $1.7x$ and $5x \sim 7x$ for the Cartoon style NPR, SURF and stereo matching, respectively, when comparing the GPU implementation vs CPU. However, the authors have not reported the energy consumption measurements.

OpenCL for general purpose computing

The SIFT detector was also implemented in the work by Wang et al. [45] using C++ and OpenCL. In this work, the optimizations mentioned in the work by Rister et al. were used with the addition of a fast Gaussian blur pyramid generation. With these optimizations, they were able to get about 8.5 frames per second for key point detections and 19 frames per second for descriptor generations. A performance speed up of about $1.69x$ for key point detections as compared to an optimized C++ reference implementation was achieved. In addition, energy consumption was reduced by 41%.

The work by Wang et al. [46] implemented object removal from images using an exemplar-based in-painting algorithm on a mobile GPU. Object removal deals with the removal of an object deemed unimportant or for some particular reason from images or videos. In this work, the OpenCL capable GPU from Qualcomm's Snapdragon SoC was used. The object removal algorithm was implemented as a heterogeneous application using OpenCL after profiling revealed the bottleneck part of the algorithm. Further optimizations of their implementation used processing of data in vector form and using data sharing by copying to local memory of the GPU. With other optimizations, the heterogeneous implementation reduced the runtime

required to about 2 seconds on the GPU. On the other hand, the OpenCL implementation on the CPU performed very low at about 393.8 seconds. The work does not give the speed up gained by their implementation.

Jones et al. [47] used mobile GPUs and OpenCL for acceleration of embodied robot simulation. They chose the Stage robot simulator's ray tracing algorithm to be accelerated using OpenCL as it was found to be the most time consuming. They report that 82% runtime performance increase and around 30% drop in energy usage for one of their experiment setups. They also speculate that more performance and energy saving can be achieved with rigorous OpenCL code optimization as the goal of the current implementation was porting the ray tracing algorithm with minimal coding effort.

Ross et al. [48] ported an OpenCL Benchmark developed for the desktop to run on a mobile GPU on an Android system. The benchmark implemented is made of N-Body simulation kernel with auto tuning capability. They report the difficulty of writing an OpenCL program for the mobile GPU because the Android development tools did not directly support it at the time. They also point out that OpenCL is optimized for portability of code not programmability hence they used a high level API called STandarD Compute Layer (STDCL) which makes programming in OpenCL for high performance computing easier. They reported the best N-body simulation on the Adreno 320 mobile GPU achieved 14.7 GFLOPs. They compared this result with a similar kernel running on an ARM Cortex A9, dual Intel Xeon X5650 CPU, and an AMD Radeon HD 6970 GPU. These three processors had 1.09 GFLOPs, 89.8 GFLOPs and 1362 GFLOPs runtime performance respectively. The authors conclude that the Adreno GPU has the potential to close the performance gap with the Intel Xeon CPU in future versions. The authors also suggest that the mobile GPU has a superior energy consumption performance over the Intel Xeon processor even though they have not provided any measured values.

In the earlier days of mobile GPU acceleration the OpenGL ES language was used to utilize the capabilities of the mobile GPUs. This was a difficult task, as it required reinterpreting problems as graphics problems. Also, the proper use of the GPU's vertex and fragment shaders must be understood. The introduction of unified shader GPU architectures and the support of OpenCL on these mobile GPUs recently has made developing software for such systems relatively easier. Although the use of OpenCL is much better than using OpenGL ES for general purpose applications, it is still difficult to develop with OpenCL for the mobile GPU as the development environment is not yet matured as discussed earlier [48].

Computer Vision problems seem to be the main focus of the literature reviewed

with regards to utilizing the general-purpose nature of the mobile GPUs. The reason behind this trend is that computer vision problems are computationally intensive data parallel applications that can benefit from any optimizations available in a system. Applications from simple primitives of image processing to real-time augmented reality were observed as a choice to be implemented on the mobile GPU. All literature reviewed reported a positive runtime performance gain. However, only some of the researchers measured the energy consumption for their application on the mobile GPU. The ones that measured the energy consumption observed that their application on the mobile GPU consumed less energy than the application running on the mobile CPU. However, the resolution of the images used in the experimentation is much smaller than the full high definition inputs (images, videos and cameras) that current mobile systems are expected to handle.

2.6 Summary

In the first section of this chapter, the open compute language (OpenCL) was described. The models present in this language were briefly explained. The process of developing an OpenCL programm was given. In the second section, the Viola-Jones [21] object detector algorithm was described. It was described that this object detector uses a cascade classifier. The components of this cascade classifier and the principle operation on an input image was explained. Also, the integral image used as an optimization technique for the object detector was detailed. In the third section, we presented research that used the mobile GPU for general purpose computation. It was observed that two types of programming languages were used where one is relatively easier to use. The first one was OpenGL ES which required reinterpretation of general problems as graphics problems. The second one was OpenCL that allows direct implementation of problems on the mobile GPU. It was also shown that all the reviewed researches reported speed ups using the mobile GPU.

Chapter 3

Performance Characterization of Mobile GPU

In this chapter, we present measurement that characterizes the nature of two modern Mobile GPU architectures; the purpose was to understand the capabilities of the underlying hardware so that applications developed for it can be well optimized. We describe experiments that were run to determine the throughput of data transfer between host and OpenCL device (the architectre of which is illustrated in Figure 2.2). Further experiments were also done to measure the global and local memory access latencies of the OpenCL device. In addition, the latencies of basic arithmetic operations were measured. The detailed experiments, results and their description are given below.

3.1 Experimental Setup

The devices used to run the experiments were the Qualcomm Snapdragon S4 Pro (APQ8064) and Qualcomm Snapdragon 805 mobile development tablets . The former tablet has the Qualcomm APQ8064 Snapdragon S4 pro applications processor that has 4 Krait 200 processors running up to 1.5 GHz, 2GB of LPDDR2 memory at 533MHz and an Adreno 320 GPU running at 325 MHz. The latter is equipped with 4 Krait 450 processors running up to 2.5 GHz, 3GB of LPDDR3 memory at 800MHz and Andreno 420 GPU running at 600MHz. Figure 3.1 shows the architecture of a dual core Qualcomm Snapdragon S4 pro applications processor as a reference. The Krait CPUs are used as the hosts and the Adreno GPUs are the OpenCL devices.

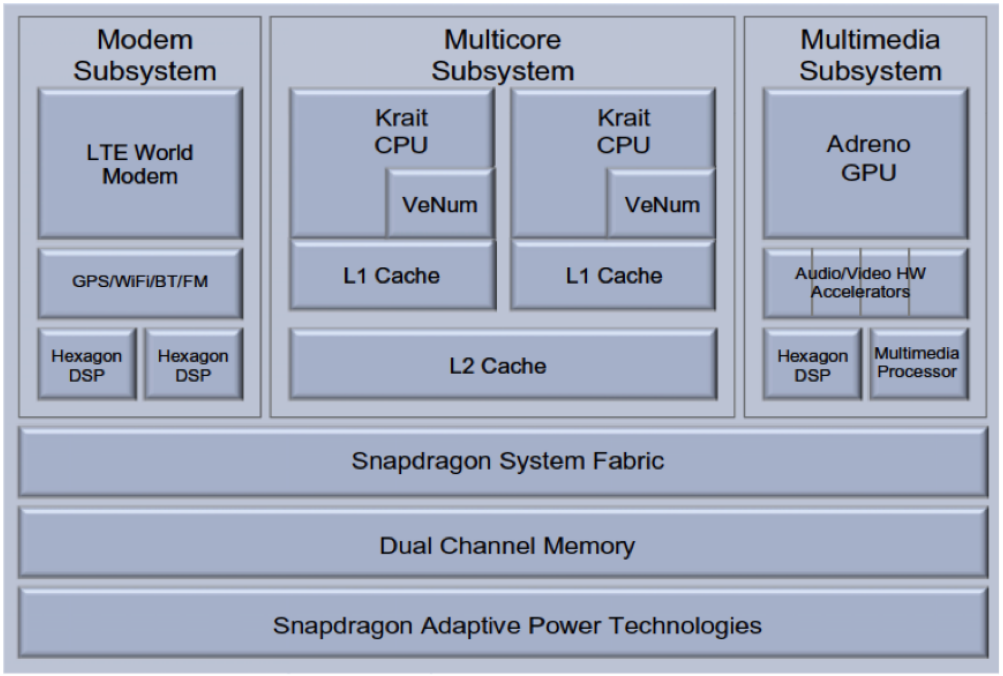


FIGURE 3.1: Architecture of dual core Qualcomm Snapdragon S4 Pro (MSM8960) [23]

3.2 Data Transfer Throughput Measurement

One of the key determinants of performance for GPU systems is the rate at which the host can send and receive information to/from the GPU. Historically, the desktop/server GPUs had physically separate memory, which was connected to the physically separate chips that contained the CPU (typically referred to as the host) and the GPU. In that context data must be copied to the GPU's memory from the CPU's main memory and returned to the CPU's main memory after computations are complete on the GPU. These data transfers often take up a significant fraction of the computation time. In this section we measure the data copying throughput between the CPU and GPU memories.

The OpenCL Software stack provides APIs that explicitly transfer data to the OpenCL device memory from the host memory and back. We used these OpenCL APIs to measure the *cpu-to-gpu* and *gpu-to-cpu* transfer rates. A data size of 256 MB was used in the experiments. Table 3.1 gives the result of the experiments conducted on the two tablets with the two generations of Qualcomm SoCs. The results in Table 3.1 show that the explicit copy actions between the host CPU and the GPU are very slow. Hence, the performance of applications that contain frequent data transfers between the host and the OpenCL device would be significantly affected.

However, in the mobile context, with a single SoC connected to an off chip memory as illustrated in Figure 3.2, there is no separate CPU or GPU memory as both

TABLE 3.1: Measurement of Transfer Rates

APQ8064 (MBytes/s)(Adreno 320/Krait 200)		APQ8084 (MBytes/s)(Adreno 420/Krait 450)	
cpu-to-gpu	gpu-to-cpu	cpu-to-gpu	gpu-to-cpu
1522	264	97	532

the GPU and the host CPU use the same main memory. With recent versions of OpenCL, the *cpu-to-gpu* copy and *gpu-to-cpu* copy can be avoided on such systems where the memory is shared. The OpenCL API ***clCreateBuffer*** can be used with the **CL_MEM_ALLOC_HOST_PTR** flag to utilize this feature. On the SoCs tested, this feature does indeed work, and makes the copy process redundant, making this part of the mobile GPU computation very efficient.

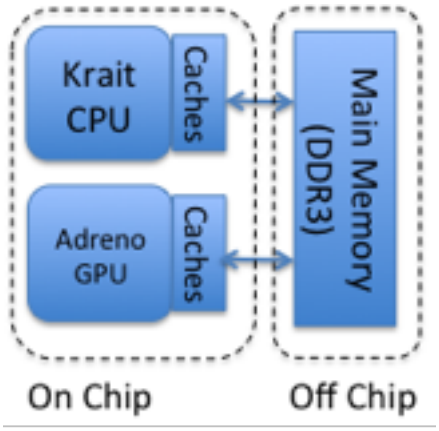


FIGURE 3.2: Abstracted Architecture of Qualcomm Snapdragon SoC

3.3 Memory Throughput and Latency

3.3.1 Global Memory Throughput Measurement

In chapter 2, it was discussed that the global memory of a GPU is usually slower in access speed compared to the constant, local and private memories. However, there is a clever memory architecture feature that allows an improved access speed. The global memory is usually aligned in blocks of ***n*** bytes (where ***n*** can be 32, 64, and 128) that can be accessed by a single memory request. An example of such alignment is shown in Figure 3.3 with 64 and 128 byte aligned memory blocks. In order to hide the slowness of the global memory access, a single memory transaction can be issued for a group of work-items accessing memory with in an aligned segment. This way the number of memory transactions will be reduced and the work-items can begin processing as soon as the data they were waiting on arrives at the same time. However, depending on the pattern memory is accessed in algorithms that run on the GPU, there can be a benefit from this feature or not.

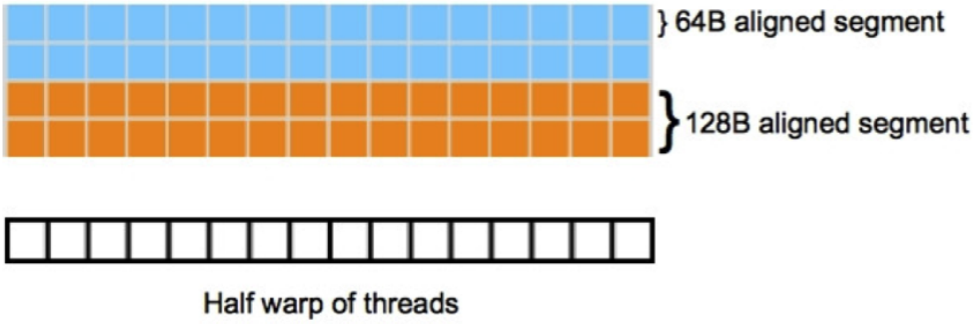


FIGURE 3.3: Linear Memory Segments and half of work-items per wavefront (warp in CUDA) [49]

A simple but efficient memory access pattern is when all (but does not have to be) the work-items in a work-group access an aligned memory segment or a set of sequential aligned memory segments. This pattern of access is called coalesced memory access. In coalesced memory access, work-items can access corresponding memory locations to their arrangement or permuted locations within the aligned memory segment. Hence, the memory transaction issued in GPUs for each aligned segment will be only one. The red enclosing box in Figure 3.4 shows the only memory transaction issued for the work-items requesting data from the aligned segment. A simple OpenCL kernel that illustrates coalesced memory access is given in Listing 3.1. This kernel does an element wise copy of vector **A** on to vector **B** which means all the work-items will be accessing data in aligned memory segments.

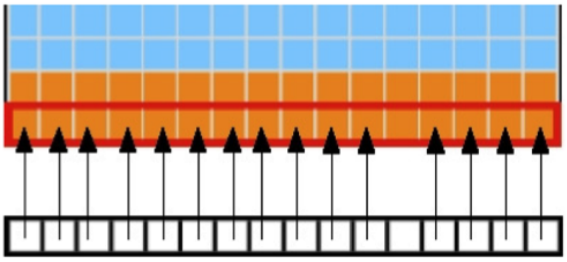


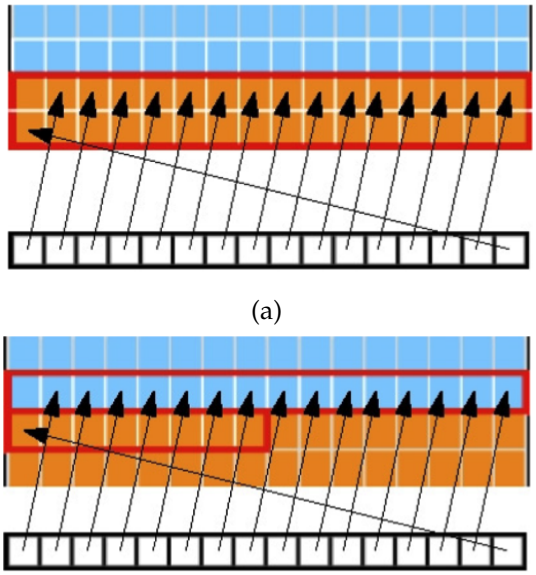
FIGURE 3.4: Coalesced Access: All work items except one access memory. One 64-byte segment is read from memory. [49]

LISTING 3.1: Kernel code for aligned memory access

```
1 __kernel  void coalescedMemoryAccess(__global
2 float* A,__global float* B,int dataSize)
3 {
4         int tid=get_global_id(0);
5         B[tid]=A[tid];
6 }
```

In the case where memory access is shifted by a constant distance relative to index of the work-items, many scenarios can arise. The shift can incur a misalignment

within a 128 byte segment or overflow a 64 byte segment as shown in Figure 3.5a and Figure 3.5b. The former case will still issue a single memory transaction but a 128-byte segment will be read while in the later case two transactions will be made; one 64-byte segment read and one 32-byte segment read. A kernel code to test the effect of this shifted memory access is given in Listing 3.2. In the case of the coalesced memory access, an element of vector **A** located at work-item index tid is copied to the same index location but in vector **B**. In the shifted access case, an element of vector **A** located at $tid + shiftdistance$, where $shiftdistance$ stays constant for a kernel execution, is copied to an element location of tid in vector **B**. The value of $shiftdistance$ can be varied to measure its effect on memory access throughput during different runs of the kernel. However, the effect of shifts on the memory access throughput can vary from device to device. This is because the device architects may chose different sets of compromises in the device design.



(a)

(b)

FIGURE 3.5: Misaligned Sequential Memory Access [49]

LISTING 3.2: Kernel code for misaligned sequential memory access

```
__kernel  void uncoalescedMemoryAccessShift(__global
float* A,__global float* B,int dataSize,int shift)
{
        int tid=get_global_id(0);
        B[tid]=A[tid+shift];
}
```

Another kind of misaligned memory access is memory access with strides. In this kind of memory access, each work-item accesses a memory location, which is a constant number of memory locations away. Listing 3.3 shows memory access with strides. If the stride length is small enough to make the entire memory access lie within 128-byte segment, only one memory transaction is made. Usually, the strides

are long enough to necessitate multiple memory transactions. Hence, such memory access method should be used sparingly. In cases where it must be used, other optimization techniques such as caching on the local memory should be used to minimize its effect.

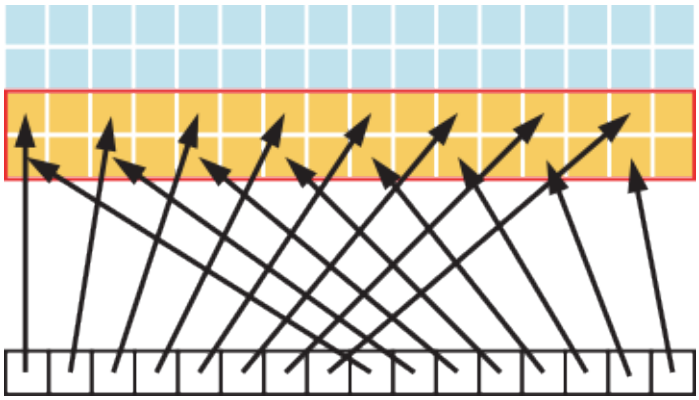


FIGURE 3.6: Strided Memory Access [50]

LISTING 3.3: Kernel code for strided memory access

```
1 __kernel  void uncoalescedMemoryAccessStride(__global
2 float* A,__global float* B,int dataSize,int stride)
3 {
4         int tid=get_global_id(0);
5         B[tid]=A[tid*stride];
6 }
```

Given the above memory access methods, the global memory of the mobile GPU is tested for memory copy throughput, where elements of a buffer are copied from one location of memory to an other. To test the copy throughput coalesced memory access, memory access with a constant shift, and memory access with a constant stride were conducted. Also each test was done for varying numbers of work-items per work-group. The reason this secondary test was employed is to see if the number work-items in a work-group affected memory access throughput.

Coalesced Memory Access

In this experiment, a single precision floating point array **A** resident in the global memory was copied to another array **B** with same size and data type using the kernel shown in Listing 3.1. The kernel was run with a total number 67108864 work-items that is equal to the size of each array. The memory allocated in the global memory then becomes $67108864\ x\ 4\ bytes\ =\ 256MB$ per array. Each work-item makes a single element copy between **A** and **B** corresponding to its index in the index space, where a work-item is identified by a unique integer id. Also, as mentioned above,

the memory copy operation is done with varying number of work-items per work-group between 1 and a *Maximum* value, which is inherent to the device. In our case this value is 128 for the Adreno 320 GPU and 512 for the Adreno 420 GPU.

The memory copy throughput is measured using the time it takes to copy all the elements of the array **A** to array **B**. The kernel in Listing 3.1, which is used to measure the throughput, does one memory read and one memory write operation. Hence, the copy throughput is computed as follows

$$CopyThroughput = \frac{1Read * ArraySize + 1Write * ArraySize}{Total\ Time\ Taken\ to\ Copy\ A\ to\ B} * sizeof(dataType) \tag{3.1}$$

where $ArraySize$ is the number of elements in array **A** and $dataType$ is the data type of array **A**.

However, both the read and write operations are of the same dataType, which means equation 3.1 reduces to

$$CopyThroughput = \frac{2 * ArraySize * sizeof(dataType)}{Total\ Time\ Taken\ to\ Copy\ A\ to\ B} \tag{3.2}$$

The results of this experiment are shown in Figure 3.7. The $x - axis$ represents the number of work-items per work-group in logarithmic scale while the $y - axis$ gives the measured data transfer throughput in Mega bytes per second (MB/s). The blue line shows the copy throughput trend for Adreno 320 GPU while the red line gives the copy throughput trend for the Adreno 420 GPU for the coalesced memory access. The immediate observation from Figure 3.7 is that the vertical gap between the blue and red lines which is caused the memory used with the two GPUs being from different generations with different speeds. Another observation is the copy throughput increased as the size of work-group increased from one to the maximum value. This phenomenon can be explained by more coalescing of memory access happening as a result of more and more work-items accessing contiguous memory locations from within a work-group. More coalescing means fewer and fewer memory transactions are done resulting in higher and higher data copy throughput.

However, the increasing copy throughput trend may not continue indefinitely with increasing number of work-items per work-group in general as the effect of various overheads starts to appear. The decrease in copy throughput observed for a size of 512 work-items per work-group for the Adreno 420 GPU from the maximum throughput at 256 work-items per work-group for the same GPU indicates the effect of overheads. The sheer amount of workload created (memory requests) by the number of work-items and the work-item scheduling used by the GPU can be attributed as the cause for this decrease. In other words, higher number of work-items per work-group create thread/work-item scheduling overhead and saturation

of the memory pipeline by the increased number of memory transaction requests issued even if they are coalesced requests.

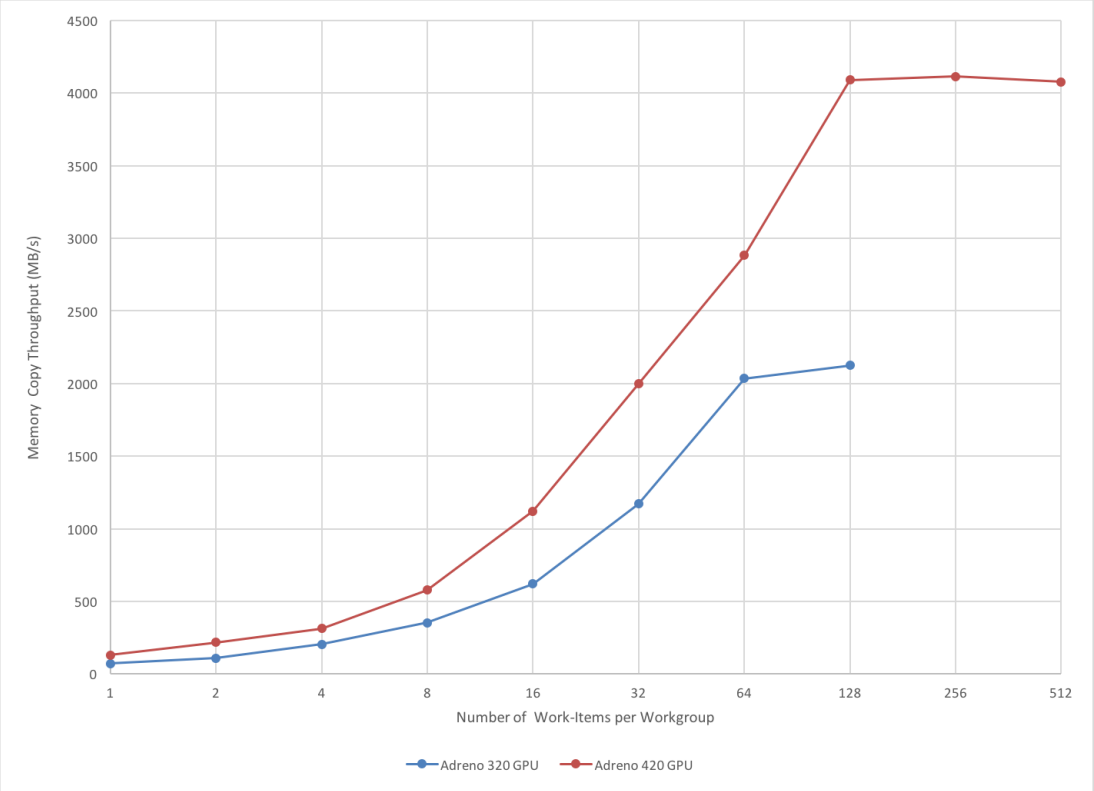


FIGURE 3.7: Coalesced Memory Access

Global Memory Access with Shifts

As shown in Listing 3.2, memory access with shifts does reads and writes memory a constant shift distance away from the current work-item id. Such reads/writes may result in surpassing the boundary of a memory block a work-group can access resulting in the issuance of multiple memory transactions. Experiments were done on the Adreno 320 and Adreno 420 GPUs to test the effects of shifted memory access. The size and type of array used in this experiment was same as the one used for the *coalesced memory access* experiment. Shift distance sizes ranging from 0 to 32 with increments of 1 were used. The kernel execution time was measured for each shift distance value while keeping work-group size fixed. Then the same procedure was repeated for increasing work-group sizes. The results of the experiment are depicted in Figure 3.8.

The $x - axis$ in Figure 3.8 represents the number of shift distances used in the experiments in linearly increasing order while the $y - axis$ shows the throughput achieved in Mega bytes per second (MB/s). The different color lines represent the

results obtained for the varying sizes of the work-groups used on the two GPUs used for experiment as indicated in the legends of Figure 3.8 . It is easily observed from Figure 3.8 that the memory access throughput is at its highest when the work-group size is the largest for both GPUs. This leads us to believe the GPUs are optimized for memory access at work-group sizes of 128 for the Adreno 320 and 512 for Adreno 420. Also, the distance between the bumps in graphs lead us to believe that in the Adreno 320 the memory alignment is 32 bytes while in the Adreno 420 it is 64 bytes.

Another, observation is that the throughput is not significantly affected by most shift values except ones that break the memory block boundary. The results fo the two GPUs show characteristically different graphs. On the Adreno 320 GPU with work-group size of 128 work-items, the bumps in the graph have a positive trend indicating that most of the shifts have an effect in reducing memory access throughput. However, these positive bumps did not start with shift values of 0 indicating that buffers are not allocated in memory at the beginning of an aligned memory segment. This also seems to be the case for the Adreno 420 GPU,where for shift value of 0 the memory access throughput is the lowest for this GPU as well. Apart from this common behaviour, the two GPUs have opposite responses for the shifted access. For the Adreno 320 GPU, the shifts negatively affected throughput keeping it lower until a shift when memory is aligned again. In the case of the Adreno 420, the throughput is not affected for most of the shifts except at intervals of 16 shift distances apart where it gets lower. This shows modern GPU architectures are improving to reduce the effect memory access patterns.

Global Memory Access with Strides

The global memory access with strides experiment is done using Listing 3.3. In this experiment, a memory location that is found a constant multiple away from the current work-item id is accessed. As the size of the constant multiple increases, the gap between two accessed memory locations widens significantly which quickly breaches the aligned memory block boundaries. As a result, more and more memory transactions will be issued. Similar array sizes and measurement techniques were used as in the previous two experiments. But this time, the global work item size was reduced to 4194304 using equation 3.3 where max stride size used in the experiments was 16. The reason for this reduction of work size is to make memory access be within the allocated array boundaries and also keep a similar work size for the different stride sizes.

The number of strides was varied in the experiment from 2 to 16 linearly. There are two reasons for the strides starting from 2. If the strides start from 0, it will mean all the work-items will be accessing the first element of the array used in the experiments. If the strides started from 1, the resulting memory access pattern will

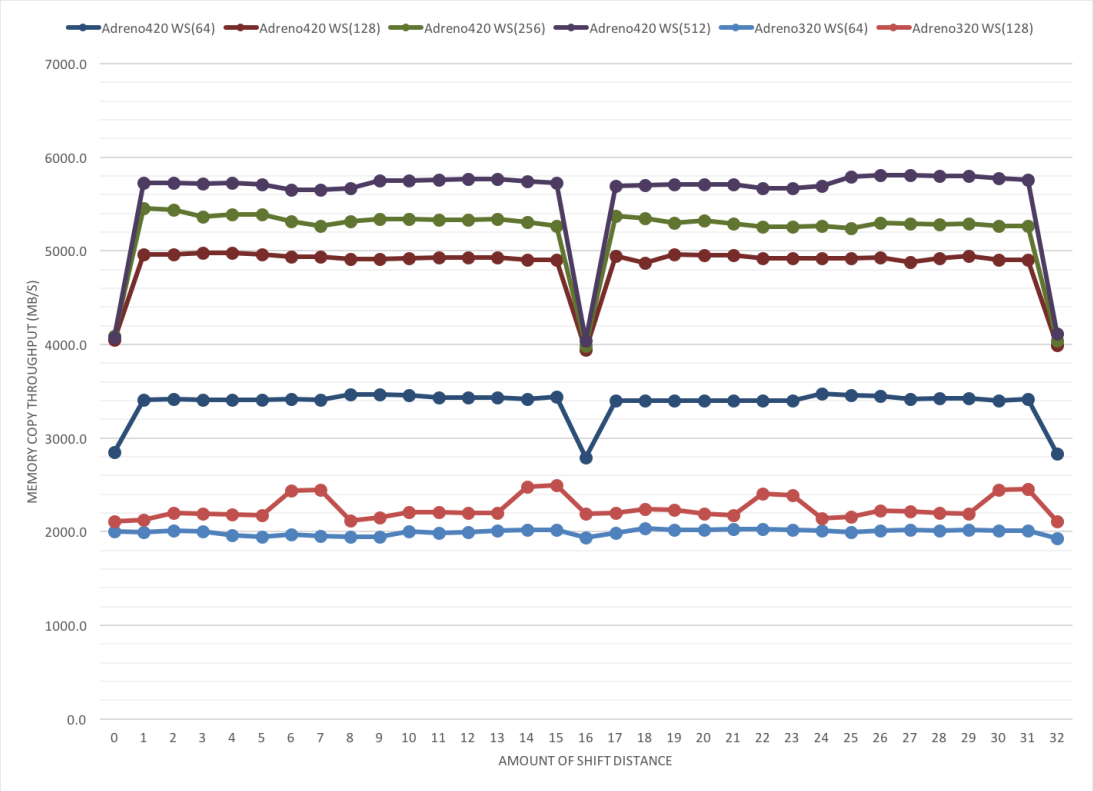


FIGURE 3.8: Global Memory Access with shifts

be coalesced which has been studied and reported earlier. The experiment was also repeated for two different work-group sizes for each GPU. These work-group sizes were chosen based on the previous experiment where we picked the best performing work-group size for both GPUs.

$$GlobalWorkItemSize = \frac{ArraySize}{Max\ Stride\ Size} \tag{3.3}$$

The $x - axis$ in Figure 3.9 represents the value of strides used in the experiments while the $y - axis$ gives memory access throughput in Mega bytes per second. The bold blue and red lines show results for the Adreno 420 GPU while the greyed out blue and red lines show results for the Adreno 320 GPU. The figure shows that the memory access throughput decreased significantly as the number of strides increased for both GPUs. These results indicate that memory accessed with large strides result in many memory transactions hence the observed lower throughput. Therefore, this memory access pattern should be avoided. In case it could not be avoided, moving the data to the local memory and making the memory access with strides in the local memory can improve performance. One can also deduce from the results shown in Figure 3.9 that a random memory access pattern will perform worse in GPUs and should be avoided or mitigation techniques suggested earlier be used in such situation.

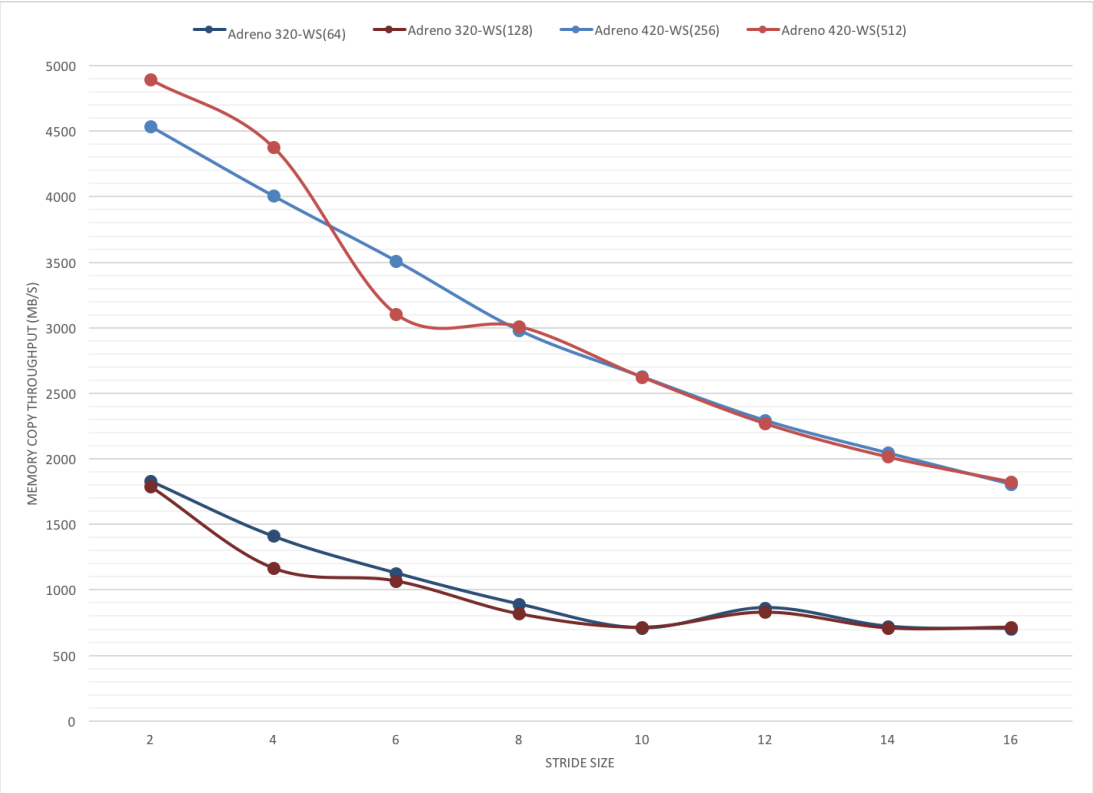


FIGURE 3.9: Global Memory Access with strides

3.3.2 Global Memory Read Latency

In the earlier section we saw, how the pattern of accessing the global memory affected throughput of memory access. In this section, we measured the global memory read latency to identify how long it takes to read an integer from memory; this helps to reveal optimizations in the memory system of the GPU, giving insights into its caching structures. The basic structure of the measurement is to read all elements of an array, and to measure the run time used to make the accesses. The array size is varied from small (likely to fit all inside a cache) to much larger than all the caches. The locality of the accesses is destroyed by making essentially random consecutive accesses, guaranteeing cache misses once the array size is large enough.

The code illustrated in Listing 3.4 is used for this task. It is a version of pointer chasing construct usually used in such measurements [51]. A random sequence of indices have been stored in the array (array A) used in the experiment beforehand to make ensure cache miss occur. The construct $repeat128(str)$ in the listing is a macro defined to repeat a given string in the brackets. It is used for unrolling the loop to lower the effect of the loop time on timing measurement. Also, to prevent the compiler from optimizing away the kernel, a dummy result had to be returned from the kernel as shown on $line\ 9$ of Listing 3.4. To also remove the effect of latency hiding by coalescing only a single work-item is used in this experiment.

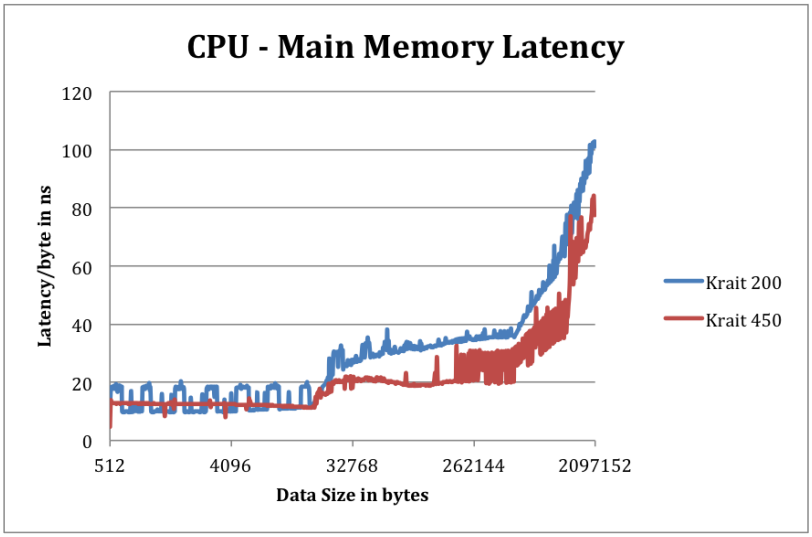
LISTING 3.4: Kernel code for memory read latency measurement

```
1  __kernel void MeasureMemoryLatency
2  (__global unsigned int *A, int dataSize,int iterations)
3  {
4          unsigned int j=0;
5          for(int i=0;i<iterations;i++)
6          {
7                  repeat128(j=A[j];);
8          }
9          A[dataSize-1]=j;
10 }
```

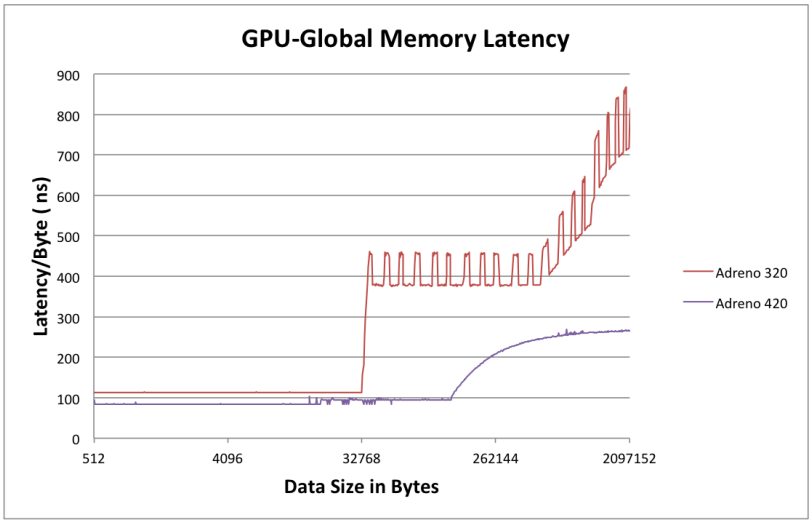
The measurement was done for the CPUs (the Krait 200 and Krait 450) as well as the GPUs (Adreno 320 and Adreno 420) for comparison. Figure 3.10a shows the CPU results and Figure 3.10b shows the GPU latency. In both figures, the $x - axis$ gives data size used in bytes using a logarithmic scale while the $y - axis$ represents the memory read latency measured in nano seconds (ns). In Figure 3.10a the blue line give results for the Krait 200 CPU while the red line depicts the trend of latency measured for the Krait 450 processor. This sub-figure suggests that the Krait 200 and the Krait 450 processors have two levels of cache, and the sizes did not change across the generations: they both have 16 KByte level one (L1) Cache and 512 KBytes of level two (L2) cache. This can be seen by the jumps in latency measurement at these values on the $x - axis$ of Figures 3.10a.

The red line in Figure 3.10b depicts the memory read latency for the Adreno 320 GPU while the violet line gives the results for the Adreno 420 GPU. From this figure, one can see the GPUs also appear to have two levels of cache; the Adreno 320's has an L1 cache of 32Kbytes and a 512Kbyte L2 cache. We suspect that L2 cache of the Adreno 320 GPU is shared with the Krait 200 CPU, as they are the same size and are quite large. The Adreno 420's L1 cache is smaller, at 16Kbytes, but it contains its own L2 Cache at 128 Kbytes, separate from the Krait 450 CPU's L2 Cache.

The CPUs and the GPUs all share the same global memory, as illustrated in Figure 3.1, but the latency to global memory is not the same. For example, the Adreno 320 global memory latency, shown at the right side of Figure 3.10b is approximately $859ns$, whereas the CPU global memory latency is about $100ns$. In the case of the Adreno 420, the GPU global memory access latency has improved to be about three times slower than the corresponding CPU Memory compared to the Adreno 320s eight times slowness against its CPU memory access latency. Although, there is improvement over generations of applications processors, these results point to the need to use the *global memory* sparingly. However, it needs to be remembered that these experiments were done using only one processing element and this larger latency observed for the GPUs can be hidden using coalescing when many work-items are used.



(a)



(b)

FIGURE 3.10: (a)Krait CPUs Main Memory and (b) Adreno GPUs Global Memory Read Latency

3.3.3 Local Memory Latency Measurement

In this section, we present two experiments done to measure the capabilities of the local memory of the GPU. The first experiment was done to determine the latency of the local memory. The second experiment was done to see the effect of memory bank conflicts in the local memory.

Local Memory Latency

The read latency of the local memory was measured by accessing it with strides. The strides were varied from 0 to 255. This was done to spread the reads across the allocated local memory. Using strides in this experiment does not have the behaviour described in section 3.3.1 because only one processing element is used. The indices generated as a result of stride computation were first copied to a buffer in the global memory as the local memory is not directly accessible from the host side. These indices stored in the global memory buffer are then copied to the local memory at the runtime of the kernel. Then, data is read from the local memory in the pointer chasing fashion as shown in listing 3.5. In order to remove the effect of copying from the global memory to the local memory and have a correct runtime measurement, the experiment was run first to measure the average run time of the global-to-local memory copy part of the kernel. Then this average time was subtracted from the overall average kernel run time. For comparison, the latency of global memory was also measured. The device used for this experiment was the Adreno 420 GPU in the Snapdragon 805 SoC.

LISTING 3.5: Kernel code for local memory read latency measurement

```
1  __kernel void dTestLocalMemoryLantency(__global int* gdata,
2  __local int* ldata,int dlength, int iterations)
3  {
4      __local int j;
5      for(int i=0;i<dlength;i++)
6          ldata[i]=gdata[i];
7      j=0;
8      for(int i=0;i<iterations;i++)
9      {
10         repeat256(j=ldata[j];);
11     }
12 }
```

Figure 3.11 shows the results of the experiment conducted. The strides used in the experiment are given in linear scale in the $x-axis$ and the corresponding latency in nano seconds (ns) is given in the $y-axis$. The red line shows the latency values measured for the global memory while the local memory latency is the blue line. The immediate observation from the figure is the gap between the global memory latency and the local memory latency. On average the Adreno 420's global memory latency is $446ns$ while the local memory has a latency of about $12.45ns$. This means the local memory is 37 times faster than the global memory for this GPU. Although, this speed difference is lower than the 100 times difference usually attributed to such memories, it still is very significant and algorithms can benefit from using the local memory. Another observation is that there is no significant change in latency between accessing the same location of local memory as in the case of $stride = 0$ and

different locations with other stride values. However, a slight increase can be noticed for the global memory as the strides increased.

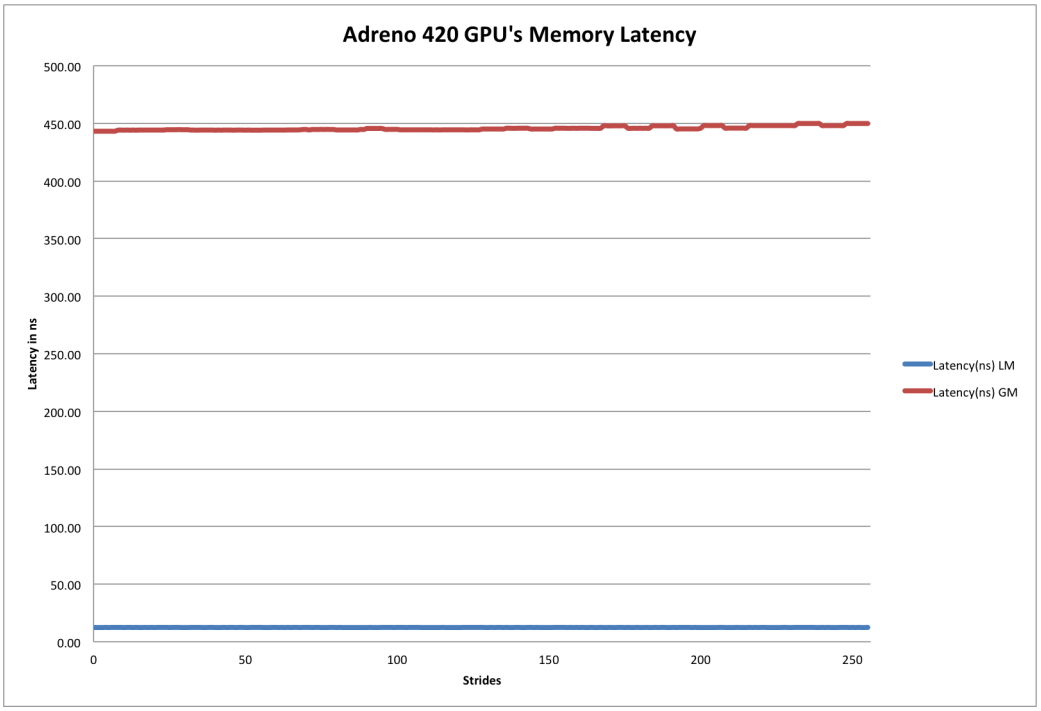


FIGURE 3.11: Read latency for the global and local memories of the Adreno 420 GPU

Local Memory Bank Conflict

The local memory in the GPU is organized in banks to allow for simultaneous read-/write access of data stored in different banks. However, if two work-items try to access data in the same local memory bank, the access will be serialized. The serialization will have impact on the performance of the algorithm running on the GPU. Hence, understanding the behaviour of the local memory banks is essential for optimizing algorithms to avoid bank conflicts. In this second experiment, two work-items are used where the first one accesses data at a base of an array on the local memory while the other is moved with linear offsets to identify conflict points where the offsets are varied from 0 to 256. The conflict points can be inferred from the latencies measured at each offset. The experiment was done on the Adreno 420 GPU, an AMD Radeon HD 6770M from a laptop and a desktop NVIDIA GTX 480 GPU. The laptop and desktop GPUs were included for comparison.

The results of the experiment are given in Figure 3.12. The $x - axis$ gives the linear offsets and the $y - axis$ gives latency in nano seconds (ns). The blue line represents the measured latencies for the different offsets on the Adreno 420 GPU. The red line provides latency measurements for the AMD Radeon HD 6770M GPU

while the green line give measured latencies for the NVIDIA GTX 480. It is expected to see a doubling of access latencies whenever there is a conflict. As one can see from Figure 3.12,the older NVIDIA GTX 480 GPU shows bank conflicts at every 32 offsets which is inferred from the doubling of latencies at these points. This behaviour is clearly described in the documentation [19]. Whereas, the more modern Adreno 420 and AMD Radeon HD 6770M don't show any bank conflict. As a result, one can assume that there is a clever hardware trick in these GPUs that can predict potential conflicts and take appropriate measure to avoid them.

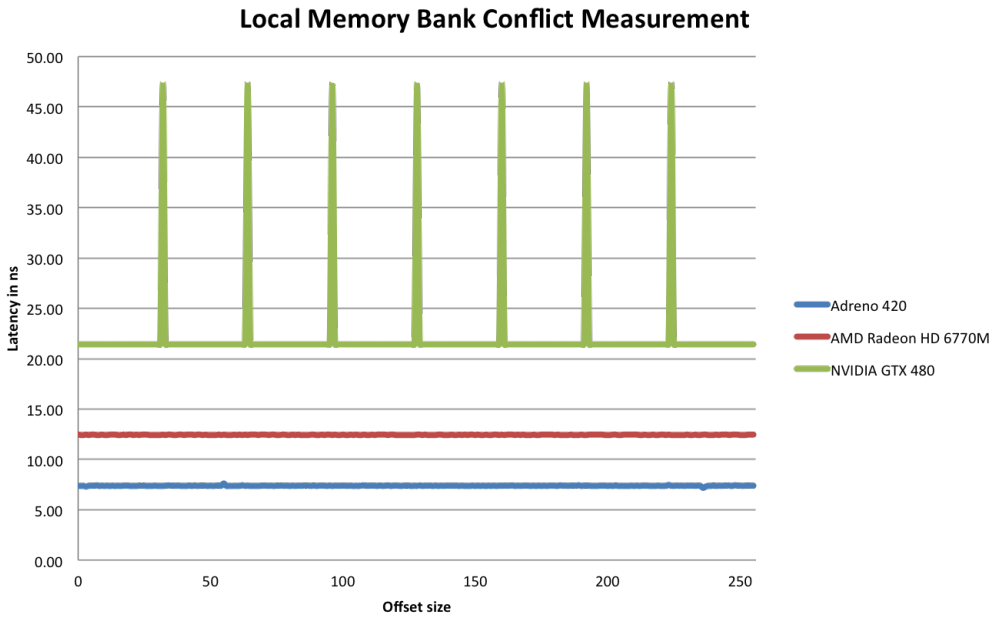


FIGURE 3.12: Local Memory Bank Conflict Measurement

3.4 Arithmetic Operations Latency Test

In this experiment, the latency of the basic arithmetic operations $(+, -, *, /)$ is measured for integer and floating-point data types. The measurement is done following the algorithm shown in Listing 3.6. Two variables, a and b, are instantiated on the private memory (registers) of the GPU to reduce the memory access latency. The operation is done on a in a cumulative way so that the instructions run are processor bound. That is, since a and b are created to be on registers, it is assumed a is moved to the accumulator once and the value of b is operated on a repeatedly as shown. Also to avoid all clever optimizations of the compiler, the kernels are compiled with all compiler optimizations disabled. One other thing to note is that the kernels are run on a single processing element on the GPU.

A similar code was run for the CPU (Krait 450 and Kriat 200) as well for comparison. The code for the CPU was compiled with the default GCC optimizations.

LISTING 3.6: Kernel code for arithmetic latency measurement

```
__kernel void d_measureArithmetic[DataType][Operator]
(int iterations,__global float *result)
{
        DataType a=Const1;
        DataType b=Const2;

        for(int i=0;i<iterations;i++)
        {
                repeat128(a=a[Operator]b;);
        }
        result[0]=a;
}
```

This measurement was run on both systems. Table 3.2 and Table 3.3 show the time it takes to complete a single operation for integer and floating-point data types in nano seconds. It can be seen that the newer GPU i.e Adreno 420 has superior integer and floating-point performance when compared to the Adreno 320 GPU. Interestingly the GPU's floating-point performance and integer arithmetic performance are very similar. For both integer and floating-point, addition and subtraction operations were the fastest followed by multiplication. The division operation consistently took a significantly longer time in all cases except on the Krait 200 Integer arithmetic. Repeated Integer divisions quickly reach a zero values and all subsequent divisions will have a zero nominator. This leads one to believe that there exists a clever hardware optimization that removes the further division operations in the case of the Krait 200 CPU.

TABLE 3.2: Results of Integer Arithmetic Operations latency

Integer	**Adreno 320(ns)**	**Krait 200(ns)**	**Adreno 420(ns)**	**Krait 450(ns)**
Addition	142	29	86	8
Subtraction	143	28	86	8
Multiplication	180	21	94	8
Division	737	12	315	13

TABLE 3.3: Results of Floating Point Arithmetic Operations latency

Float	**Adreno 320(ns)**	**Krait 200(ns)**	**Adreno 420(ns)**	**Krait 450(ns)**
Addition	142	29	86	14
Subtraction	142	28	86	14
Multiplication	143	21	94	15
Division	398	70	176	23

3.5 Measuring Parallelism

In this experiment, the amount of parallelism that exists in GPUs was measured by measuring the latency of arithmetic operations with respect to number of threads/work-items used. A fixed amount of work was given to a work-item and the number work-items was gradually increased. Although, the total work-size increases each work-item will have the same fixed amount of work. This fixed amount of work in our case was a set of simple arithmetic operations performed on integer data in the private memory of the GPU. There is no interdependence between each work-item and the effect of using the global memory is not present. Therefore, we expect to see no change in the latency of the execution of the instructions in Listing 3.7 until we have the work-group size greater than or equal to the number of processing elements in the system. The Qualcomm Adreno 420, the ARM Mali T628 and the Imagination Tech's PowerVR G6400 mobile GPUs as well as the laptop AMD Radeon HD 6770M and the desktop NVIDIA GTX 480 GPUs were used for the experiment. The default OpenCL compiler optimization is enabled in this experiment unlike the previous arithmetic latency measurement where it was disabled. As usual the average of the latencies measured were taken after multiple runs. In addition cache warming runs were made before measurement was taken.

LISTING 3.7: Kernel code for measuring degree of parallelism

```
1  __kernel void dTestWorkLoadParallelism(__global int* gdata,
2   int iterations,int offset)
3  {
4      int a=offset;
5      int b=const1;  //const1 is replaced by a number
6      int x=const2;  //const2 is replaced by a number
7      int j=get_global_id(0);
8      for(int i=0;i<iterations;i++)
9      {
10                 //some dummy arithmetic operation
11         repeat128(x+=a;a+=b;b+=x;);
12     }
13     gdata[j]=x; //return dummy data
14 }
```

As stated earlier, the latency should be the same for a number of work-items less than the number of actual processing elements in the GPU. For example, if the number of processing elements in a GPU is 64 and all work items have similar work load, one should not see a difference in latency between 10 work-items or 58 work-items running on the GPU. However, there should be a difference when the number of work-items is greater than 64 because there are not enough processing elements to handle computation in parallel. The extra work-items will need to be scheduled to run only after resources are available again. The results in Figure 3.13 show clearly the amount of parallelism present in each tested GPU allowing us to determine the

number of processing elements for each. The $x-axis$ in Figure 3.13 gives the number of work-items in linear scale while the $y-axis$ gives the arithmetic load latency in nano seconds. The blue, red, green, violet and cyan colored lines give the measured arithmetic workload latency for the Adreno 420, AMD Radeaon HD 6770M, Nvidia GTX 480, ARM MALI T628 and the PowerVR G6400 GPUs respectively.

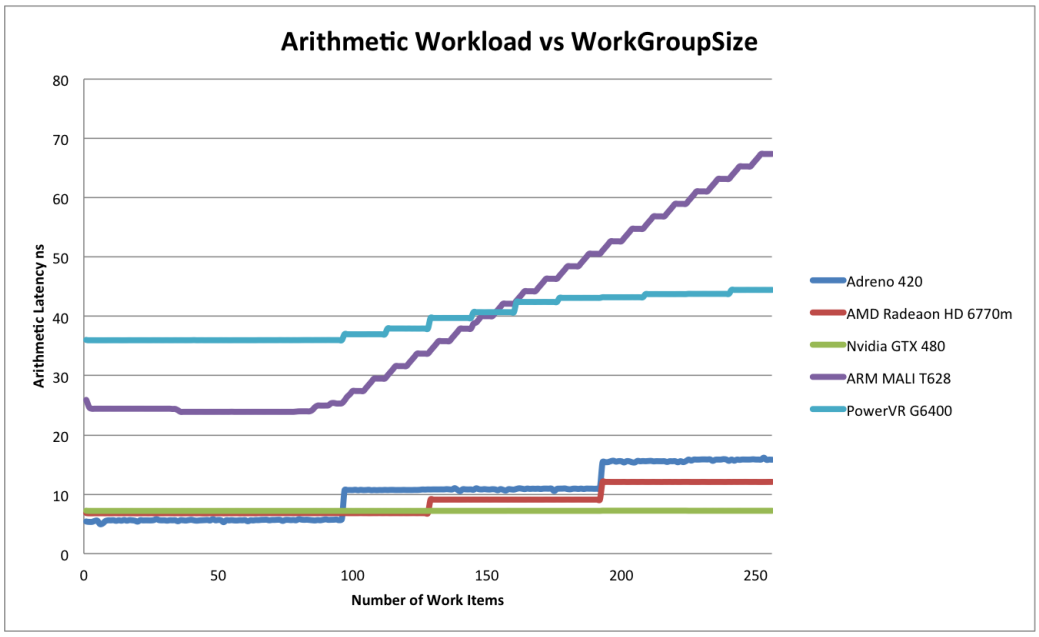


FIGURE 3.13: Arithmetic workload vs work-group-size

The Adreno 420 GPU and the PowerVR G6400 both boast about 96 processing elements while the Mali T628 GPU starts to saturate earlier than its nominal number of 100 processing elements. The latency for the Radeon HD 6770M GPU stayed low until 128 work-items and the GTX 480 had the same latency for all work-item sizes in this test which was limited at 256 work-items. The reason for this behaviour of the GTX 480 is that it has many more processing elements than tested. All except the GTX 480 GPU show a stepping increase in latency after the first set of measured latencies although these steps were different for each GPU. The measured latency for the Adreno 420 GPU comparatively stayed the same until the number of work-items reached 96 and showed a doubling of the latency for the next set of 96 work-items and a tripling for the next set of work-items until the end of the test. The results for the ARM Mali T628 show irregularity for the first 100 work-items.The latency at the beginning lowers from its starting value and lowers again after many work-items then keeps this value until it reached around the 100 work-item mark. After the 100 work-items, the measured latency for the ARM Mali T628 have ramps and landings at regular intervals that lead to quickly rising latency. In the case of the powerVR G6400 GPU, the latency for the first 100 work-items was relatively constant. However, for next set of work-items the latency showed variable length steps. The latency increased at shorter intervals at first and starts to widen at higher

number of work-items. The AMD Radeon 6770M GPU showed the same latency for the first 128 work-items and after that the latency increased at steps of 64 work-items. However, the latency did not double or triple but increased at regular values.

Figure 3.13 also shows that although the mobile GPUs tested were relatively the same generation, the Adreno 420 had the lowest latency. In fact, the Adreno 420 had the lowest latency for the first 128 work-items compared to all GPUs tested. The Mali T628 showed a lower latency than the PowerVR G6400 GPU for the first 154 work-items but after that number of work-items the PowerVR G6400 had a more stable lower increase in latency unlike the Mali T628. One can infer from this behaviour that different computer architects choose different ways of scheduling work-items and sharing resources.

3.5.1 Summary

In this chapter the measurements that were done to understand the mobile GPU's architectural capabilities were discussed. Data transfer throughput between host side and device side were measured giving us the insight to avoid such explicit movement of data on mobile systems. Then we saw how different patterns of accessing data from the global memory of the GPU affects data transfer throughput and what this means for computation. Following this, we determined the sizes of different levels of caching available for both the CPU and GPU in the systems tested. The latency of local memory was measured and compared with the latency of the global memory and this latency measurement technique was also used to study the behaviour of local memory bank conflicts. The latency of arithmetic operations was also tested to get an insight on how GPU processing element bound computations behave. Finally, the arithmetic latency measurement technique was adapted to measure the available parallelism in the GPUs tested. The insights gained from these experiments will be used in the next section for optimizing the object detection algorithm.

Chapter 4

Object Detection on a Mobile GPU

The second objective of this work is to implement the Viola-Jones object detection algorithm [21], described in detail in Chapter 2, on a mobile GPU and to measure the resulting performance both in run time and energy efficiency. In this chapter, the implementation of the components of Algorithm 1 in Section 2.4 on a GPU are described. First, the implementation of the *integral* and *square integral* image computation will be described. The naive implementations of Algorithm 1 on both the CPU and GPU will then follow. After that the different approaches used to improve the performance of the algorithm on the GPU will be described.

The mobile development platform used in this work was the Qualcomm Snapdragon 805 [52] developer tablet. It has four Krait 450 processors that run at frequencies of up to 2.5 GHz, with 3GB of LPDDR3 memory running at 800MHz. It also contains the Adreno 420 GPU running at 600MHz. This GPU has four compute units where each contain 32KB of local memory. It is important to note that both the processor and the GPU share the same RAM which means that it is possible to avoid memory copies as a buffer allocated in memory can be accessed from the CPU and GPU. This is significant as those copies are often bottlenecks [53, 54].

The parallel and serial implementations presented in this chapter are based on the same algorithm as the OpenCV 3.1 version [18] and we will directly compare the quality of results and runtime with the OpenCV 3.1 code. Note that the OpenCV version 3.1 also has an implementation in OpenCL (which is a key goal of this research; we warn the reader that the two acronyms in the preceeding sentence - OpenCV and OpenCL refer to quite different things - OpenCV is the open-source computer vision library, and OpenCL is the open computing language used for accelerating computation) that was indicated to be compatible with Android. However, we were unable to make the OpenCL version of the object detection algorithm from OpenCV work on the mobile device used. That code did work on the desktop environment when the same version of OpenCV was compiled for the desktop environment with the same build options as in the case of the Android version. A closer look at the

OpenCV 3.1 version source code revealed that the OpenCL version of the Viola-Jones based object detection algorithm was limited to only run on Advanced Micro Devices (AMD) and Intel systems. We compiled the OpenCV 3.1 version after removing this limitation and tested it with the same sample images as for the CPU only version. The results obtained showed a great erroneous deviation from the CPU only version suggesting that the OpenCL object detection code is not designed to function correctly on systems other than the ones mentioned earlier.

4.1 Integral Image Computation

In this section we will describe the integral image computation (described in Section 2.4) that is required before the search for objects commences. The integral image is computed for different scales of the input image. The first subsection describes the way the resizing of the input image is done for different scales. It is followed by the description of the implementation of the integral image computation using the prefix sum algorithm.

4.1.1 Image Resizing

Before the integral image computation commences the original image needs to be resized to the appropriate scales i.e. the original image is successively shrunk to create a scaled image pyramid. The scales used for resizing the original image are computed based on the window size a cascade classifier is trained on and the original image size. The bilinear image scaling method shown in Listing 4.1 is used as this method is also used in OpenCV. The bilinear algorithm uses the linear interpolation of the neighbouring pixels to compute the value of the new pixel in the new resized image. This algorithm was straightforward to implement on the GPU. Our GPU implementation is based on the Java implementation of the bilinear resizing algorithm given in [55].

The inputs to the kernel in Listing 4.1 are a gray scale image, a *scaleDataGPU* object and other necessary information. The *scaleDataGPU* object contains the precomputed sizes of the new images at each scale, pointers to the location of memory where the new resized images are going to be stored and other information needed for later computations. The kernel is organized to be run with $[8\ x\ H]$ number of work-items, where H is the image height of the largest scale. We have also selected to organize work-items in an $[8\ x\ 8\ =\ 64]$ work-group following our finding that upto 96 work-items can virtually execute together on a compute unit in chapter 3. Each row of work-items is responsible to iterate over a corresponding row of the

image at the new scale computing new pixel values. These newly computed pixel values are then stored in their corresponding location in memory.

LISTING 4.1: GPU implementation of the Bilinear image scaling

```
__kernel void resizeBilinearGrayGPU(__global uchar* pixels,
int w,int h,__global uchar* sums,
__global ScaledDataGPU* s,int numberOfScales)
{
 int ygid=get_global_id(1);
 int xlid=get_local_id(0);
 __global int* temp;
 int w2,h2;     //dimensions of target image
 for(int scales=0; scales < numberOfScales; scales++)
 {
   w2=s[scales].sizeOfImage.s0; //get target width
   h2=s[scales].sizeOfImage.s1; //get target height
   temp = (__global int*)(sums + s[scales].bufferOffset
           * sizeof(int) );
   int A, B, C, D, x, y, index, gray ;

   float x_ratio = ((float)(w))/w2; // x - scaling ratio
   float y_ratio = ((float)(h))/h2; // y - scaling ratio
   float x_diff, y_diff ;

   int offset=ygid*w2; //used for easy memory reference
   int iterations=0;
   for ( int j=0; j < w2 && ygid < h2; j += HALF_TILE_SIZE )
   {
     int xIndex=mad24(iterations,HALF_TILE_SIZE,xlid);
     x = (int)(x_ratio * xIndex) ;
     y = (int)(y_ratio * ygid) ;
     x_diff = (x_ratio * xIndex) - x ;
     y_diff = (y_ratio * ygid) - y ;
     index = y*(w)+x ;

     // range is 0 to 255 thus bitwise AND with 0xff
     A = pixels[index] & 0xff ;
     B = pixels[index+1] & 0xff ;
     C = pixels[index+w] & 0xff ;
     D = pixels[index+w+1] & 0xff ;

     // Y = A(1-w)(1-h) + B(w)(1-h) + C(h)(1-w) + Dwh
     gray = A*(1-x_diff)*(1-y_diff) +  B*(x_diff)*(1-y_diff)
            + C*(y_diff)*(1-x_diff)   +  D*(x_diff*y_diff);
     temp[offset+xIndex] = (int)(gray) ;
     iterations++;
   }
  }
}
```

4.1.2 Integral Image

Once the image is resized, the next step is to compute the integral images. The integral image is computed using the prefix sum also known scan operation. The serial computation of the prefix sum is straight forward as shown in Figure 4.1. Given an input vector, the algorithm first sets the first element of the output vector to 0. Then starting from $i = 0$ iterates through each element of the output vector doing $output[i + 1] = output[i] + input[i]$ until $i + 1 < input\_vector\_length$. At the end of this operation each element of the output vector will store the sum of elements preceding and including itself. However, as one can see from Figure 4.1 the last element of the input vector is not used. This kind of prefix sum is called an exclusive prefix sum. In order to have in inclusive sum, the length of the output vector has to be increased by one and the iteration should continue until $i + 1 < (input\_vector\_length + 1)$.

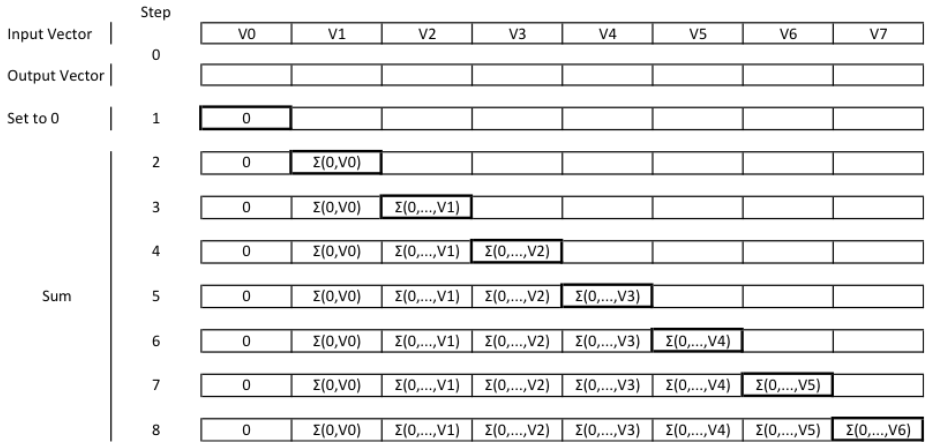


FIGURE 4.1: Sequential computing of an exclusive prefix sum (scan)

There are different ways of computing the prefix sum as shown in [56] but in this work the parallel prefix sum computation method developed by Blelloch in [57] is used. This prefix computation method uses an upward reduction operation on a summation tree and a downward summation pass. Figure 4.2 shows the process of computation of the prefix sum on a SIMD parallel hardware such as GPUs. Given an input vector as shown in *step* 0 of Figure 4.2, pairs of elements of the input vector are summed in parallel and the resulting sums are stored in the same vector avoiding the need for additional memory to store output results. In the second step, pairs of the previously computed sums will be summed and stored in parallel. This computation is continued step by step in a similar manner until finally the last element of the vector carries the sum of all elements in the vector.

Once the upward reduction operation is done, the last element of the vector will be replaced by a 0 indicating the beginning of the downward swap and sum operation. The first of these operations is shown in the fifth step of the example shown in

Figure 4.2. Here a copy of the fourth element of the vector is stored in a temporary memory location. Then the last element in the vector is copied to the fourth location. This is followed by the addition of the value stored in the temporary location to the last element. In the next step, the process described above is applied to levels of the summation tree generated during the upward reduction process. After all the levels in the tree are processed, each element of the resulting vector will have the sum of all preceding elements including itself.

The operations discussed above are designed to be executed on SIMD machines. It can be seen in Figure 4.2 that to compute the prefix sum for a vector size of n, only $n/2$ or $n/2 + 1$ processing elements/threads are required. However, all the processing elements are not used all the time as some of them will be idle for parts of the prefix sum algorithm. Although, these processing elements/threads are not used all the time during the execution of the prefix sum, on SIMD machines, such as a GPU, this algorithm is efficient as there are only O(n) additions done.

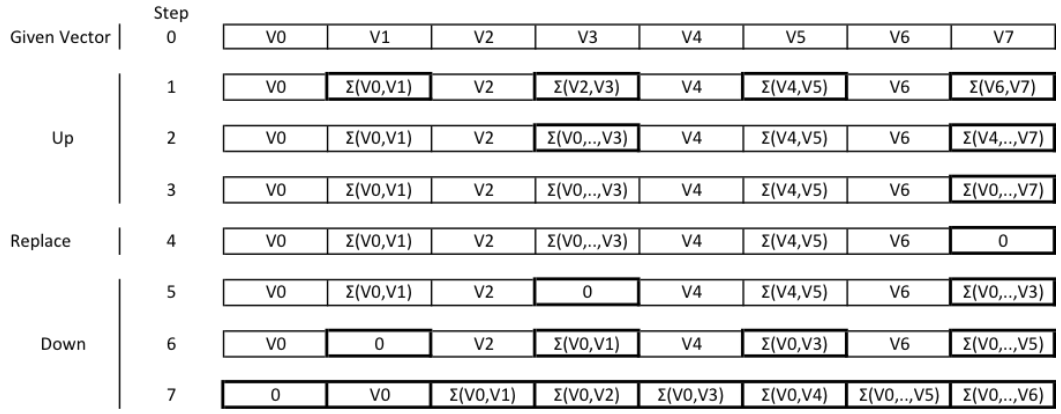


FIGURE 4.2: Process of computing an exclusive prefix sum (scan) in parallel [57, 56]

The code snippet in Listing 4.2, which is a version of the CUDA implementation presented in [56] into OpenCL, shows the implementation of the prefix sum computation discussed earlier on a GPU which is a SIMD device. This snippet does prefix sum computation for a single block of a row in the input gray image. Given X amount of work-items per work-group in one dimension in the GPU, this approach has the capacity to compute the prefix sum for up to $2X$ number of elements in a vector. However, this approach is to be utilized for the computation of the integral image. The integral image is composed of a number of rows that can have more than $2X$ number of elements.This necessitates the improvement of the prefix sum algorithm to handle multiple rows and large number of elements per row.

LISTING 4.2: Code snippet for prefix sum computation on a GPU

```
1 /*copy input data to local memory (temp)- for coalescing*/
2 temp[2*lid]=input[2*lid];       //lid - local work item id
3 temp[2*lid+1]=input[2*lid+1];
```

```
4  /*Build the reduction tree*/
5  for(int d=blockSize>>1;d>0;d>>=1) //blockSize=2*workGroupSize
6  {
7          /*make sure all copies are done*/
8          barrier(CLK_LOCAL_MEM_FENCE);
9          if(lid<d)
10         {
11                 int ai=offset*(2*lid+1)-1;
12                 int bi=offset*(2*lid+2)-1;
13                 temp[bi]+=temp[ai];
14         }
15         offset*=2; //offset is initialized to 1
16 }
17 /*set the last element to zero*/
18 if(lid==0)
19         temp[blockSize-1]=0;
20 /*sum up the tree*/
21 for(int d=1;d<blockSize;d*=2)
22 {
23         offset>>=1;
24         /*make sure all updates are done*/
25         barrier(CLK_LOCAL_MEM_FENCE);
26         if(lid<d)
27         {
28                 int ai=offset*(2*lid+1)-1;
29                 int bi=offset*(2*lid+2)-1;
30                 int tmp=temp[ai];
31                 temp[ai]=temp[bi];
32                 temp[bi]+=tmp;
33         }
34 }
35 /*write back results to global memory*/
36 output[2*lid]=temp[2*lid];
37 output[2*lid+1]=temp[2*lid+1];
```

The prefix sum given in Listing 4.2 can be modified in many ways to compute the sum for the entire image. We will describe two options here. Both approaches use blocks of work-items for computing the prefix sum. The first approach uses half the width ($W/2$) by the height (H) number of work-items for a given input image. These work items are grouped into work-groups of size X by X that are responsible for computation of the prefix sum for a particular block of the image. However, as one can see in Listing 4.2 the last element in the computed sum for each row in a block is set to zero. This is acceptable if the width of the input image is less than or equal to $2X$. Otherwise, the resulting output is not a prefix sum for the entire image but for blocks of it. To correct this problem, the last elements of the computed sums in each block have to be collected in temporary storage located in the global memory. Each block will have a resulting column of intermediate sums. In the next step, these intermediate sums have to be aggregated sequentially ($blkSum[i, j+1]$ +=

$blkSum[i, j]$) which then are added to each row element in the corresponding block in a second pass and so requires a second kernel.

The idea behind the second approach is that instead of instantiating as many as $(W/2)$ work-items per row and have a second aggregation pass to compute the prefix sum for the entire row, instead fix the number of work-items (X) and have them loop over the row. This way, the need for a set of temporary storage per row is replaced by only one temporary storage located in the private memory. In addition, the second aggregation pass is removed as the carried over sums is applied in the next loop iteration. In order to compute the prefix sum for the entire image using this method, X by H number of work-items will be needed.

The computation of the integral image on the GPU in this way requires the computation of the prefix sum on the rows of the image first and then followed by the columns. However, computing the prefix sum for the columns will be very inefficient, as data locality will be affected because the pattern of memory access when operating on the the columns is strided. We know from Chapter 3 that global memory access with strides is very inefficient and is to be avoided. Transposing the resulting image after the computation of the row prefix sum solves the problem of data locality, at some cost of extra data movement effort. Applying the row prefix sum computation on this transposed image will result in an integral image for the given input gray image. Another transpose may or may not be required depending on how subsequent computations are configured to use the resulting integral image. In our case a second transpose was required.

The way the prefix sum kernels and the transpose kernels are organized can affect performance. One can have a row prefix sum kernel followed by a transpose kernel and then a column prefix sum followed by another transpose kernel. This will be a total of four kernel calls which is inefficient because of the overhead associated with kernel calls. In this work, the row prefix sum and the first transpose as well as the column prefix sum and the second transpose were combined into a single kernel each to reduce the cost of kernel call overhead. This means we will have only two kernel calls instead of four. In addition, the computation of the integral and square integral images are combined in the same kernels as the computations are similar, further reducing the number of kernels needed.

The integral image must be computed for all the scales of the original image as required by the object detection algorithm. As mentioned in the previous section, the input gray image was resized and stored in memory for all the scales. Therefore, the prefix sum computation kernels developed were modified to compute the sum for all the scales. This approach reduced the number of kernel calls that would have been needed for each scale in the previous method. This way the the number of kernel calls to compute the integral image are reduced only to two. In all the computation

discussed above data was only accessed on the GPU side.

4.2 Searching for Objects

The next step in the Viola-Jones [21] object detection algorithm is the search for objects in a given image. As already discussed, the input gray image has been resized for all the scales and the integral image computed for each scale. In the process, the integral image for all scales is computed and stored in memory. Doing so reduces the need to allocate memory, resize the image, and compute the integral image on the fly for each scale. However, this has a toll on memory requirement as a large amount of memory has to be reserved. In recent smart phones, this will not be a problem as the trend in the industry is to include sufficient sizes of memory in the device.

In this section we describe the different approaches used to implement the Viola-Jones[21] object detection algorithm on the mobile GPU. First, the naive implementation of the object detection, as described in Algorithm 1 of Section 2.4, on the GPU is discussed. Next, the different techniques applied on the naive implementation for performance improvement are detailed.

4.2.1 The Naive Object Detection on the GPU

The naive implementation of the object detection Algorithm 1 described in Section 2.4 is given in Listing 4.3. In this implementation, it is assumed that each candidate window from the input image will be tested for the existence of the object using a separate work-item/thread. Most of the buffers used for storing the necessary input data for the algorithm are stored in the global memory.

As described in Listing 4.3, given the top left corner of the a window to be tested, the variance norm factor is computed first (in lines 5-11). The variance norm factor is used to reduce the effect of lighting condition in the input image. Next, the outer for loop in Listing 4.3 (in lines 17-69) applies the cascade classifier on the window stage by stage. The inner for loop (in lines 23-66) evaluates the features present in each weak-classifier per stage. It is known that each feature consists of two or three feature rectangles that requires the computation of weighted sum of the underlying pixels. This computation is unrolled instead of using a loop in lines 36-56.

In this particular implementation, the number of rectangles per feature are either two or three and are evaluated accordingly. Once all the weak classifiers present in a stage are evaluated, the computed result is compared against the stage threshold. If the computed result is greater than the stage threshold, the window needs further

testing and the next stage needs to be applied. If, otherwise, the computed result is less than the threshold, the object is deemed to not be present in the current window and further processing is not needed. After the outer loop finishes processing, the iterator is checked to see if it is equal to the number of stages.If this is so, the object is deemed to be found in the current window and it will be marked as containing the object of interest.

LISTING 4.3: Naive Object Detection Kernel

```
/* (xglid,yglid) top-left corner of current window */
 x1=xglid+windowSize.x-1;
 y1=yglid+windowSize.y-1;
/* computation of the lighting normalization */
 mean=GET_SUM2(xglid,yglid,x1,y1,width,sum);
 variance_norm_factor=GET_SQSUM(xglid,yglid,x1,y1,width,sqSum)
                       * window_area - (float)(mean*mean);
 if(variance_norm_factor>0.0f)
     variance_norm_factor=rsqrt(variance_norm_factor);
 else
   variance_norm_factor=1.0f;
/* apply the cascade classifier on the current window */
treeStart=0;
alphaStart=0;
treeIndex=0;
featureIndex=0;
for(i=0;i<numberOfstages;i++)
{
  stageSum=0;
  j=haarStages[i].countWPtr.s0;
  end=haarStages[i].countWPtr.s1+haarStages[i].countWPtr.s0;

  for(;j<end;j++)
  {
      k=0;
      res=0;
      treeStart=weakClassifiers[j].countTPtr.s0;
      alphaStart=weakClassifiers[j].alphaStart;
      /* evaluate the feature */
      do
      {
         treeIndex=treeStart+k;
         featureIndex= (int)treeNodes[treeIndex].featureIndex;
         HAAR_Feature_GPU feature=features[featureIndex];
         /* Evaluate first feaure */
         x0=(feature.f[0].s0)+xglid;
         y0=(feature.f[0].s1)+yglid;
         x1=(feature.f[0].s2)+x0;
         y1=(feature.f[0].s3)+y0;
         res=GET_SUM2(x0,y0,x1,y1,width,sum)
             * feature.weights.s0;
         /* Evaluate second feaure */
         x0=(feature.f[1].s0)+xglid;
```

```
44          y0=(feature.f[1].s1)+yglid;
45          x1=(feature.f[1].s2)+x0;
46          y1=(feature.f[1].s3)+y0;
47          res+=GET_SUM2(x0,y0,x1,y1,width,sum)
48                   * feature.weights.s1;
49          /* Evaluate third feaure if it exists */
50          res+=select(0.0f,GET_SUM2(
51               feature.f[2].s0+xglid,
52               feature.f[2].s1+yglid,
53               feature.f[2].s0+xglid+feature.f[2].s2,
54               feature.f[2].s1+yglid+feature.f[2].s3,
55               width,sum)* feature.weights.s2,
56               (feature.weights.s2!=0));
57         /* apply the variance norm factor */
58         res*=variance_norm_factor;
59        /* choose an alpha value */
60         k=(res<treeNodes[treeIndex].threshold)?
61                    (treeNodes[treeIndex].lr.s0):
62                    (treeNodes[treeIndex].lr.s1);
63
64            }while(k>0);
65       stageSum+=alphas[-k+alphaStart];
66   }
67    /* Object not detected */
68    if(stageSum<haarStages[i].threshold) break;
69 }
70 /* Object detected: add to list of detections */
71  if(i==numberOfstages)
72  {
73      int nObjects=atomic_inc(numberOfFoundObjects);
74      if(nObjects<MAX_OBJECTS){
75          foundObject[nObjects].s0=xglid*scaleFactor;
76          foundObject[nObjects].s1=yglid*scaleFactor;
77          foundObject[nObjects].s2=windowSize.x*scaleFactor;
78          foundObject[nObjects].s3=windowSize.y*scaleFactor;
79      }
80  }
```

There can be as many as $[W\ x\ H]$, where $W = (imgWidth - window.width)$ and $H = (imgHeight - window.height)$, candidate windows when searching the first scale of input image. Subsequent scales will have progressively fewer candidate windows. $imgWidth$ and $imgHeight$ are the sizes of the first scaled image while $window.width$ and $window.height$ are the sizes of the window the cascade classifier is trained for. Each candidate window will be tested for the existence of the object of interest by one instance of the kernel (a work-item) that executes the code snippet in Listing 4.3. Hence, in order to run the object detection kernel in this implementation, a global work-item size of $[W\ x\ H]$ is needed. These work-items can be grouped in many ways but in this work they are grouped into a work-group size of $[8x8]$ work-items using the lessons learned in Chapter 3. Although, subsequent scales do not

require as many as $[W\ x\ H]$ work-items as used for the first scale, these same work-items will be reused for searching the object for the subsequent scales. This approach of reusing work-items is used in all of the implementations on the GPU in this work.

The naive implementation was tested on nine images that were selected based on their diversity in resolution and the number of objects they contain. Table 4.1 presents results on the speed and accuracy of several implementations of the Viola-Jones [21] object detection algorithm that we compare. The first column of Table 4.1 gives the resolution of the image under test. Subsequent pairs of columns give the execution (wall clock) time measured in seconds and number of detections of a face object by each implementations. For better presentation the test images were separated into two groups based on their resolution. The geometric mean of the execution time and the speed up were computed separately for the two groups of images. There are three implementations compared in the table: the one labelled **OpenCV CPU** is the multi-threaded implementation provided in OpenCV [18] version 3.1 package and is run on the CPU, the one labelled **MyCPU** is the naive serial implementation (described above) running on the mobile CPU and the one labelled **MyGPU Copied** is the above naive implementation running on the Adreno 420 Mobile GPU. The term *Copied* is in reference to the explicit copying of data that was done in this implementation between the host CPU and the GPU memories.

It is apparent from Table 4.1 that the single-threaded **MyCPU** is about 5 times slower than the parallel **OpenCV CPU** implementation. **MyGPU Copied**, our naive implementation on the mobile GPU, was a little faster than the **MyCPU** implementation, however, it was still 3.8 and 4.3 times slower than the **OpenCV CPU** on average for the Full HD and the smaller sized images respectively. Taking the **OpenCV CPU** detection results as the baseline, it can be observed there are slight differences in the number of detections of objects in the table. This behaviour is most likely caused by the rearrangements made in the algorithm during its implementation as well as floating point representations on the different devices. We can also infer from the results in Table 4.1 that more improvements can and need be done on the naive implementation for better results.

TABLE 4.1: Runtime Measurement for the Naive Implementation

	OpenCV CPU		MyCPU			MyGPU Copied		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	3.18	1	0.20
Full HD - 2	1.15	2	6.85	3	0.17	5.53	4	0.23
Full HD - 3	1.10	3	6.12	3	0.18	3.80	3	0.34
Full HD - 9	1.33	9	7.26	9	0.18	5.04	8	0.26
Full HD - 19	1.36	19	7.50	20	0.18	5.05	20	0.27
Full HD - 72	1.74	72	8.24	72	0.21	6.19	74	0.30
Geometric Mean	**1.17**		**6.38**		**0.19**	**4.68**		**0.26**
512x512 - 1	0.16	1	0.75	1	0.21	0.55	1	0.29
450x326 - 2	0.09	2	0.41	2	0.22	0.37	2	0.22
647x650 - 31	0.31	31	1.59	31	0.19	1.52	31	0.20
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.68**		**0.23**

Using Local Memory

The way the feature rectangles are accessed and evaluated depends on the current window under scrutiny in the Viola-Jones [21] object detection algorithm. Depending on the values of the current feature evaluation results the next feature to be evaluated is chosen. This next feature to be evaluated may not be stored in contiguous memory as the previous one. This shows the algorithm has non-uniform memory access. As it is discussed in Chapter 3 that if an application has non-uniform global memory access it can benefit from the caching of data in the local memory which has a much lower latency than the global memory in the GPU. Hence, the naive implementation was modified so as image data associated with each work-item in a work-group is copied to the local memory. As each work-item is to operate with one window, it requires access to $[window.width\ x\ window.height]$ pixels that are associated with the specific window. As a result, a $[window.width\ x\ window.height]\ X\ [8x8]$ pixels need copying to be processed by a work-group as the work-items were organized in an $[8x8]$ work-group. However, all the windows that are evaluated in a work-group are found adjacent to each other. Therefore, we only need to copy $[(window.width\ +\ 8)\ x\ (window.height\ +\ 8)]$ pixels per work group. We refer to this modified implementation as **MyGPU Copied LM**, where the *LM* signifies the use of local memory.

The results collected after applying this modification on the same test images and using the same experiment settings stated earlier are given in Table 4.2. We keep the results for the **OpenCV CPU** and **MyCPU** from Table 4.1 for comparison. The third major column gives run time, number of detections and the speed up for **MyGPU Copied LM** implementation. It can be seen that the **MyGPU Copied LM** is about 4.17 and 4.76 times slower than the **OpenCV CPU** implementation for the different classes of images respectively. When the current implementation is compared to the **MyGPU Copied** implementation from the previous section, it was found to be about 7.9% and 8.6% slower for the different classes of test images, which means the use of local memory did not improve performance.

The cause of such performance degradation can be attributed to the redundant copying of image data done to the local memory by each work-group. As each work-group is a composed of an $[8x8]$ block of work-items and each work-item processes a window with a resolution of $[20x20]$ pixels, and these candidate windows are contiguous, a $[28x28]$ image block needs to be copied to the local memory per work-group. However, data copied to the local memory is only visible to the specific work-group. This leads each work work-group make copies of significant portions of the image data that could have been shared. In fact, 91.8% of the data copied by each work-group was copied by other work-groups as well. Hence, **MyGPU Copied LM** is an inefficient implementation that can benefit from further improvements.

TABLE 4.2: Runtime Measurement for the Naive Implementation with Local Memory

	OpenCV CPU		MyCPU			MyGPU Copied LM		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	3.55	1	0.18
Full HD - 2	1.15	2	6.85	3	0.17	6.17	4	0.21
Full HD - 3	1.10	3	6.12	3	0.18	4.22	3	0.31
Full HD - 9	1.33	9	7.26	9	0.18	5.60	8	0.23
Full HD - 19	1.36	19	7.50	20	0.18	5.55	20	0.27
Full HD - 72	1.74	72	8.24	72	0.21	6.79	74	0.26
Geometric Mean	**1.17**		**6.38**		**0.19**	**5.19**		**0.24**
512x512 - 1	0.16	1	0.75	1	0.21	0.60	1	0.27
450x326 - 2	0.09	2	0.41	2	0.22	0.40	2	0.20
647x650 - 31	0.31	31	1.59	31	0.19	1.65	31	0.18
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.73**		**0.21**

Data Transfer Reduction

In the previous GPU implementations the input image was copied from the host side to the device side explicitly. This explicit copy of data between the CPU and GPU is slow and wasteful in the mobile system used for experimentation in this work. The reason is that both the CPU and GPU are connected to the same memory and can share the same memory space. In the next modification of the naive implementation, we allocate buffers on the GPU memory (recall it is the same memory as the CPU memory) in a way that allows them to be shared. These allocated buffers are then mapped to pointers on the CPU side. After the mapping process, the pointers can be used as any other pointer used to access memory on the host CPU. This new implementation is referred to as **MyGPU Mapped**, where the term *Mapped* shows this method avoids explicit copying of data between the host CPU and the GPU.

Table 4.3 gives the results of the measurements done for this particular modification. The measurements were done with the experiment device and experiment setup discussed earlier. We again keep the results collected for the **OpenCV CPU** and **MyCPU** implementations for comparison purposes. In the third major column in Table 4.3, the runtime, number of detections and speed up obtained in the current implementation are provided. The average speed up computed for **MyGPU Mapped** is around 0.3 and 0.27 as compared to the OpenCV CPU implementation for the two groups of test images. However, when compared to the naive **MyGPU Copied** implementation there is a 15% and 17% improvement respectively. This indicates that avoiding the direct copying of data between the CPU and GPU memory can help with improving performance on such heterogeneous architectures.

Work-item Organization

The cascade classifier used in the object detection is usually trained with a smaller window size than normal input images. Most of the area of the input image is background or part of the object of interest in the case the object is large enough to fill

TABLE 4.3: Runtime Measurement for the Naive Implementation with Mapped Memory

	OpenCV CPU		MyCPU			MyGPU Mapped		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	2.80	1	0.23
Full HD - 2	1.15	2	6.85	3	0.17	4.66	4	0.28
Full HD - 3	1.10	3	6.12	3	0.18	3.34	3	0.39
Full HD - 9	1.33	9	7.26	9	0.18	4.33	8	0.30
Full HD - 19	1.36	19	7.50	20	0.18	4.30	20	0.31
Full HD - 72	1.74	72	8.24	72	0.21	5.32	74	0.34
Geometric Mean	**1.17**		**6.38**		**0.19**	**4.04**		**0.30**
512x512 - 1	0.16	1	0.75	1	0.21	0.47	1	0.34
450x326 - 2	0.09	2	0.41	2	0.22	0.31	2	0.26
647x650 - 31	0.31	31	1.59	31	0.19	1.30	31	0.24
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.57**		**0.27**

significant portions of the input image. This means during the search for the object in the input image, most windows would not contain the object being searched for. Hence, most of the work-items that were instantiated in the naive implementation will be idle after the application of the first few stages of the classifier. In order to mitigate the effect of idle work-items, the number of work-items were reduced to $[8\ x\ H]$ as used in the case of the integral image computation as shown in Section 4.1. In this modification, a row of work-items in a work-group are used to process a row of windows. We refer to this implementation as **MyGPU Mapped RTC**. The *RTC* stands for reduced thread count which signifies the reduced number of threads (a short-hand term for work-items) used.

TABLE 4.4: Runtime Measurement for MyGPU Mapped Reduced Thread (Work-item) Count

	OpenCV CPU		MyCPU			MyGPU Mapped RTC		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	3.12	1	0.20
Full HD - 2	1.15	2	6.85	3	0.17	5.51	4	0.23
Full HD - 3	1.10	3	6.12	3	0.18	3.78	3	0.34
Full HD - 9	1.33	9	7.26	9	0.18	5.02	8	0.26
Full HD - 19	1.36	19	7.50	20	0.18	4.99	20	0.27
Full HD - 72	1.74	72	8.24	72	0.21	6.20	74	0.30
Geometric Mean	**1.17**		**6.38**		**0.19**	**4.65**		**0.26**
512x512 - 1	0.16	1	0.75	1	0.21	0.55	1	0.29
450x326 - 2	0.09	2	0.41	2	0.22	0.37	2	0.22
647x650 - 31	0.31	31	1.59	31	0.19	1.55	31	0.19
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.68**		**0.23**

In this reduced thread implementation we build on the performance gained with the **MyGPU Mapped** version. The buffers used in the object detection algorithm are mapped for shared use between the CPU and the GPU. The same experiment device and setup are used for measurement in this case as well. Table 4.4 as usual keeps the measurement data collected for**OpenCV CPU** and **MyCPU** as reference. It also provides the measurements collected for the new **MyGPU mapped RTC**. The average speed up calculated from the results given in Table 4.4 with respect to the **OpenCV CPU** implementation is 0.26 and 0.23 for the larger and smaller set of images respectively. The amount of performance reduction compared to **MYGPU Mapped**, which is 13.3% and 14.8%, and the identical performance as **MyGPU Copied** observed indicates that the amount of time most work-items spend being idle in the

previous implementations. Hence, all further improvements in the object detection implementation in this work will be based on this **MyGPU Mapped RTC** implementation. With regard to the number of objects detected, there are slight discrepancies from the other implementations on the GPU.

4.2.2 Modified Classifier Representation

The most frequently accessed data in the object detection algorithm is the cascade classifier as it is applied to all windows tested. In the naive implementation, the data structure that holds the cascade classifier is stored in the global memory. It was ascertained earlier that accessing data from the global memory is slow as compared to the other memory types present in the GPU. We also have found out that accessing data in a coalesced pattern can hide the global memory latency. However, the pattern of accessing the cascade classifier data stored in the global memory depends on the current window under test.This makes the access pattern variable from work-item to work-item. On the other hand, the cascade classifier data is used by all the work-items which makes it frequently used data that may or may not be cached by the system. A more direct way is to store this frequently used data in a faster memory that is visible to all work-items. Such a memory , as suggested in [17], is the constant memory. Storing the cascade classifier in the constant memory is ideal as the data is constant and does not change. However, the cascade classifier in its current representation is very large to fit inside the limited constant memory available.

Our original representation of the cascade classifier on the GPU is given in Listing 4.4. It is based on the representation of the cascade classifier on the CPU as shown in Listings 2.4 and 2.5. As can be seen, there is a visible difference between the representations on the CPU and the GPU. This difference appeared only because of the difference in data types used in the CPU and GPU. The changes that are made are described as follows. The feature representation on the GPU does not include a boolean component used to check for the presence of tilted features as our implementation considers upright features only. The ***Rect*** data structure, which is composed of four integers, used to represent the feature rectangles is replaced with ***cl_int4*** vector data type. The associated feature weights are stored in ***cl_float4*** type. We have used the ***cl_float4*** instead of ***cl_float3*** type to keep a proper alignment of data. The feature rectangles, weak classifiers and alpha values are stored in arrays respectively. Our main goal was to represent the cascade classifier with minimum number of bytes that lead to the use of aligned vector data types. This use of aligned vector data types has additional benefits as their use is efficient on the GPU [58].

LISTING 4.4: Representation of the Cascade Classifier on the GPU

```
typedef struct {

    cl_int4 f[3];       //feature rectangles
    cl_float4 weights;  //feature weights

}HAAR_Feature_GPU;
typedef struct{
    cl_char2 lr; //where l - left(1 byte), r- right (1 byte)
    cl_int featureIndex;        //pointer to feature
    float threshold;     //weak classifier threshold
}HAAR_TreeNode_GPU;

typedef struct {
    cl_short2 countTPtr;
    //count - number of tree nodes
        //TPtr - pointer to tree node
    cl_short alphaStart; //pointer to  associated alpha values
}HAAR_WeakClassifier_GPU;

typedef struct{
    cl_short2 countWPtr;
    //count - stores the number of weak Classifiers
    //WPtr -  the beginning of the weak classifier
    float threshold;    //stage threshold
}HAAR_Stage_classifier_GPU;

typedef struct{
    int count;
    Size orig_window_size;
    Size real_window_size;
    float scale;
    float window_area;
    HAAR_Stage_classifier_GPU* stageClassifier;
}HAAR_ClassifierCascade_GPU;
```

As an example to compute how much memory is required to represent a cascade classifier, we have used one of files containing a cascade classifier for detection of the face of a person from the OpenCV library. We have made sure the selected cascade classifier file does not contain tilted features as our focus is a classifier trained with upright features. In this particular file, there are 2135 features associated with the weak classifiers present in the file. Each weak classifier is composed of a stump tree node which means there is exactly one feature per weak classifier. Hence, the total number of weak classifiers in this example cascade classifier becomes 2135. The weak classifiers each also contain two alpha values (α) that are selected based on the evaluation of the feature it contains. The weak classifiers are organized in 22 stages in the example file. These stages contain varying number of weak classifiers between 3 to 213 per stage. Table 4.5 gives the amount of memory needed to store

each component of the cascade classifier for the face detection example in OpenCV based on the representation given in Listing 4.4. It can be seen that a total of $188,088$ bytes are needed to store the cascade classifier in the GPU memory.

TABLE 4.5: Representation of the Cascade Classifier on the GPU

Data	Bytes/Feature	Instance	Total Bytes
HAAR_Feature_GPU	64	2135	136640
HAAR_TreeNode_GPU	10	2135	21350
HAAR_WeakClassifier_GPU	6	2135	12810
HAAR_Stage_classifier_GPU	8	22	176
Alpha Values	4	4270	17080
HAAR_ClassifierCascade_GPU	28	1	28

In order for the cascade classifier to fit inside the available constant memory, it needs to be represented in an alternative way. A close look at the cascade classifier file revealed that the representation of the feature rectangles assumes the top-left corner of the window rectangle as the origin. The window rectangle is the size of the window the classifier is trained for. In our case, the window has a resolution of $[20\ x\ 20]$ pixels. Hence, the dimensions used to represent the feature rectangles can never have a value greater than 20. This means the feature rectangles values can be represented with just 5 bits. However, the smallest number of bits that can be addressed by modern computers is a group of 8 bits or 1 byte. In OpenCL, the data type that uses one byte to store values is the ***cl_char*** data type. Since there are four values to be stored for each feature rectangle we use ***cl_char4*** instead as shown in Listing 4.5. Another revelation from the scrutiny of the cascade classifier file is that the weights associated with each feature rectangle are small signed integers ($<< 127$) rather than large real numbers. This means each weight can be represented with ***cl_char*** data type. Because there can be upto three rectangles per feature, the weights can be represented with***cl_char4*** collectively. ***cl_char4*** is used in this case to keep the alignment of data as stated earlier.

LISTING 4.5: Alternate Represenation of the Cascade Classifier on the GPU

```
typedef struct {

    cl_char4 f[3];      //feature rectangles
    cl_char4 weights;   //feature weights

}HAAR_Feature_GPU_Packed;

typedef struct{
    cl_char2 lr; //where l - left(1 byte), r- right (1 byte)
    cl_short featureIndex; //pointer to feature
    float threshold;    //weak classifier threshold
}HAAR_treeNode_GPU_Packed;
```

The inspection of the file also revealed that the left child and right child pointers of the feature tree in the file only store values of 0 or 1 and can be represented with ***cl_char*** each. The number of features inside the example file were limited to 2135 which means a data type that can store this with the minimum number of bytes can be selected. Hence, the ***cl_short*** data type is used to represent the $featureIndex$. The $alpha$ values are stored using ***cl_float***data type and further reduction of storage space was not made because the alpha values utilize significant number of decimal places. The other data structure shown earlier are already in optimized state and are not considered for modification here. The memory space saving done because of this modification is given in Table 4.6.

TABLE 4.6: Representation of the Cascade Classifier on the GPU

Data	**Bytes/Feature**	**Instance**	**Total Bytes**
HAAR_Feature_GPU_Packed	16	2135	34160
HAAR_TreeNode_GPU_Packed	8	2135	21350
HAAR_WeakClassifier_GPU	6	2135	12810
HAAR_Stage_classifier_GPU	8	22	176
Alpha Values	4	4270	17080
HAAR_ClassifierCascade_GPU	28	1	28

As can be seen in Table 4.6,the new required total $81,338$ bytes size is still larger than the 64KB constant memory reported to be present in our system by querying the device tested. In our implementation, the features were stored in a separate array structure that can be made to reside in the constant memory. These features are one of the most frequently accessed data types as such can benefit from being stored on the constant memory. The HAAR_TreeNode_GPU_Packed data structure is also stored on the constant memory. Together these two modified data structures will only need about $55,510$ bytes which is well below the size of the constant memory in the system.

The experiments done for the measurement of this implementation used the same set of test images and experiment settings as used earlier. It can be seen from Table 4.7 that the 0.26 and 0.23 average speed ups are the same as the **MyGPU Mapped RTC** implementation. This suggests that utilization of the constant memory is not as important as indicated in [17]. This behaviour can be attributed to the fact that there is no separate constant memory but the caches in the system are used for storing buffers flagged as constant memory [32]. In our implementation we flagged the most commonly used data to be stored in the constant memory which could have been stored in the caches by the caching system anyway.

TABLE 4.7: Runtime Measurement for the Packed Implementation with Reduced Thread/Work-item Count

	OpenCV CPU		MyCPU			MyGPU Packed RTC		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	3.19	1	0.20
Full HD - 2	1.15	2	6.85	3	0.17	5.53	4	0.23
Full HD - 3	1.10	3	6.12	3	0.18	3.82	3	0.34
Full HD - 9	1.33	9	7.26	9	0.18	5.05	8	0.26
Full HD - 19	1.36	19	7.50	20	0.18	4.95	20	0.27
Full HD - 72	1.74	72	8.24	72	0.21	6.10	74	0.30
Geometric Mean	**1.17**		**6.38**		**0.19**	**4.66**		**0.26**
512x512 - 1	0.16	1	0.75	1	0.21	0.56	1	0.29
450x326 - 2	0.09	2	0.41	2	0.22	0.38	2	0.21
647x650 - 31	0.31	31	1.59	31	0.19	1.50	31	0.20
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.68**		**0.23**

4.2.3 Work Size Reduction

In section 4.2.1, different approaches used to improve the naive implementation by using techniques of improving performance on the GPU were discussed. In this section, we will describe the application of work size reduction for performance improvement as suggested in [59, 60]. We will build on the reduced thread (work-item) count implementation discussed earlier. The idea is that most candidate windows would be of background type or a portion of the object in a typical image under test. This means, the test on these types of candidate windows will fail at the earliest stages of the cascade classifier. The effect of this behaviour is that the workload distribution among the work-items will not be even. Most work-items will finish early and stay idle while some of the candidate windows are being tested for the higher stages of the cascade classifier.

To reduce this idle time, we can apply the stages of the cascade classifier one at a time or grouped in stages on the candidate windows. In the first approach, the first stage can be applied to all the candidate windows and windows that pass the stage are collected to be put on a queue. Then in the second round, the second stage will be applied to all the candidate windows that were on the work-queue and only those that pass the second stage are put back on the work-queue and so on. In the second approach, instead of computing stage by stage, the stages were grouped into multiple bands and applied on candidate windows as reported in [59]. However, we have observed this approach didn't work well as there was drastic changes in the number of candidate windows that passed to the next stage after the computation on the first few stages. This meant grouping together stages resulted in the same effect as the naive implementation where most work-items stayed idle. Therefore, the first approach is used in this work as well as the OpenCL implementation of object detection in OpenCV [18].

The procedure described in the above paragraph can be computed in two phases based on whether a work-queue is used. In the first phase the work-queue is not yet initialized and is empty. The first stage of the cascade classifier is then applied

to all candidate windows and windows that passed the first stage are put on the work-queue. The work-queue stores information about the location of the candidate window and important values associated with it. After applying the first stage on all candidate windows, the work-queue potentially contains candidate windows for further processing, which means the next phase can begin. In this next phase, work-items will be assigned work from the work-queue, apply the next classifier stage and put back candidate windows that pass this stage on the work-queue. Repeat this process until the work-queue is empty or all the stages are applied on the candidate windows that remained in the work queue.

The kernel for this implementation is processed with $[8\ x\ H]$ number of work items in a $[8\ x\ 8]$ work-group. In the unoptimized version of this implementation, 64 adjacent windows i.e. one window per work item was processed in the first phase. It is apparent that unless this group of work-items is processing near the location where the object may reside in the input image,very few windows pass to the next stage. This makes most of the work-items in the work-group idle. In our implementation, we have empirically found that it is efficient to process a block of $16\ x\ 64$ candidate windows per work-group in the first phase. As there are more candidates, the work-queue size was also increased to accommodate the possible number of windows that pass the first phase. However, the work-queue size can't increase significantly as it is located on the local memory of the GPU which is a limited resource. This implementation is refered to as **MyGPU Mapped RTC WC** where the term *WC* refers to the work compaction signifying the further reduction of work-items staying idle.

TABLE 4.8: Runtime Measurement for the Reduced Thread/Work-item Count with Work Compaction

	OpenCV CPU		MyCPU			MyGPU Mapped RTC WC		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	0.27	1	2.33
Full HD - 2	1.15	2	6.85	3	0.17	1.54	1	0.84
Full HD - 3	1.10	3	6.12	3	0.18	0.59	3	2.19
Full HD - 9	1.33	9	7.26	9	0.18	1.05	9	1.24
Full HD - 19	1.36	19	7.50	20	0.18	1.27	19	1.06
Full HD - 72	1.74	72	8.24	72	0.21	1.75	72	1.05
Geometric Mean	**1.17**		**6.38**		**0.19**	**0.91**		**1.35**
512x512 - 1	0.16	1	0.75	1	0.21	0.21	1	0.76
450x326 - 2	0.09	2	0.41	2	0.22	0.14	2	0.57
647x650 - 31	0.31	31	1.59	31	0.19	0.53	31	0.57
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.25**		**0.63**

As in the previous sections the same experiment set up and test images were used for measuring the performance of **MyGPU Mapped RTC WC**. Table 4.8 lists the run times, number of detections and speed up measured for the current implementation. The **MyGPU Mapped RTC WC** implementation shows a 35% average speed over the base line for the Full HD image category.This was the first time a positive performance enhancement was achieved. In fact, for the test image *Full HD - 1*, we recorded a 2.33 speed up over the baseline. This test image hand only one face object in the scene and because of the nature of the algorithm used most

of the candidate windows were rejected earlier that lead to the reduction of idle work-items. The work-items were kept busy processing only from the work-queue which is work efficient. However, for the smaller images the performance was lower than the baseline by 37% on average. Even though **MyGPU Mapped RTC WC** was slower for the smaller images with respect to the baseline, it was more than two-fold faster compared to our previous GPU implementations. The number of detections with the exception of one test image were the same as the baseline in this implementation. This was achieved because enough local memory was allocated to store the work-queue.

Using Local Memory

The use of local memory was employed to see if further improvements can be gained. Recall that in section 4.2.1 that a block of $[8x8]$ windows was copied to the local memory. Each window is in turn $20x20$ pixels which means a $28x28$ block of image has to be copied to the local memory for every set of work-group. This lead to a huge amount (about 92%) of redundant memory copies resulting in significant performance loss. In this implementation, we have tried to reduce this overhead by making copies of sixteen blocks of windows or a $52x52$ block of image per work-group. In this way pixel data is shared between adjacent windows reducing redundant copies to 62%.

Even though the redundant memory copies were reduced to 62% with this approach, performance was still reduced as the 62% redundant copy is still significant. This conclusion can be reached from the data presented in Table 4.9. It can be seen that the average speed up reduced by 36% for the larger images when local memory was used. However, for the smaller images the average speed up was similar. An other observation was that the number of detections were significantly affected for most of the images. The reduction of memory used for storing the work queue was found to be the reason for the difference in the number of detection. The work queue size was reduced to half it size because now there is a need to accommodate the $52x52$ block of image data that need to be copied to the local memory.

As a proof that decreasing the work-queue size impacts the number of detection, one can compare the results reported in the work by Andargie, F.A., et. al[29] with the result reported in Table 4.8. The size of the work queues were set to be same (at half the size of what is used in section 4.2.3 for both the **MyGPU RTC** and the **MyGPU RTC LM** in [29], which are similar to **MyGPU Mapped RTC WC** and same as **MyGPU Mapped RTC WC LM** respectively. It can be seen, the resulting number of detections for these two implementations deviated from the baseline **OpenCV CPU** implementation in a significant way for most images tested. However, **MyGPU**

Mapped RTC WC implementation reported above had an increased size of work queue which resulted in comparatively similar detections with the baseline.

TABLE 4.9: Runtime Measurement for the Reduced Thread/Work-item Count with Work Compaction and Local Memory Use

	OpenCV CPU		MyCPU			MyGPU Mapped RTC WC LM		
Images Res - #Object	Runtime(S)	Detections	Runtime(S)	Detections	Speed Up	Runtime(S)	Detections	Speed Up
Full HD - 1	0.64	1	3.58	1	0.18	1.07	1	0.59
Full HD - 2	1.15	2	6.85	3	0.17	1.39	1	0.93
Full HD - 3	1.10	3	6.12	3	0.18	1.33	2	0.97
Full HD - 9	1.33	9	7.26	9	0.18	1.56	6	0.83
Full HD - 19	1.36	19	7.50	20	0.18	1.62	13	0.83
Full HD - 72	1.74	72	8.24	72	0.21	1.63	111	1.12
Geometric Mean	**1.17**		**6.38**		**0.19**	**1.42**		**0.86**
512x512 - 1	0.16	1	0.75	1	0.21	0.20	1	0.80
450x326 - 2	0.09	2	0.41	2	0.22	0.18	1	0.44
647x650 - 31	0.31	31	1.59	31	0.19	0.46	33	0.65
Geometric Mean	**0.16**		**0.79**		**0.20**	**0.25**		**0.61**

It is to be remembered that the run-time measurements were taken as the average of 20 consecutive runs of each implementation. All implementations were run atleast once before measurement commenced for warming the caches. The standard deviation of each measurement session was also computed. It was found that the standard deviation for the timing measurement for multiple runs is with in 9% of the average run-times overall and less than 1% for **MyGPU Mapped RTC WC** implementation which is our best performing implementation.

4.2.4 Energy Efficiency Measurement

In this subsection, we will describe the experiments and measurements that were done to measure the energy consumption of our select implementations and compare them with the baseline **OpenCV CPU** implementation. We have measured the energy consumptions of **OpenCV CPU**, **MyCPU**, **MyGPU Mapped RTC WC**, and **MyGPU Mapped RTC WC LM**. In order to measure the energy consumption, we have used a data logger from National Instruments to measure the current drawn by mobile development platform used in this study while the implementations mentioned earlier were running. The screen of the tablet was kept turned off at all times and the tablet has been left idle for sometime before measurements were taken.

We measured the energy consumption of the different algorithms in the following way: the current drawn by the tablet was measured while the implementations listed were being run on the different test images. In order to have reliable measurements, each implementation was run for multiple times as in the case of the run time measurement. However, in this case the number of runs was varied depending on the run times associated with the test images. We used 20 runs for images that took longer to process and 50 runs for images with lower runtimes. This was done to have reasonable current measurement samples from the data logger. Then the root mean square power usage was computed from the collected data to arrive at the energy

consumption values (note that these values are the overall energy consumed by the tablet for each algorithm).

Table 4.10 presents the energy consumption measured in Joules and the improvements achieved in percentage as compared to the baseline, which is the **OpenCV CPU** implementation. An immediate observation from the data in Table 4.10 is that serial and multi-threaded implementations on the CPU consistently consumed more energy compared to the implementations on the GPU. The serial implementation **MyCPU** consumed almost only twice as much as the multithreaded OpenCV CPU implementation. This is a surprise because **OpenCV CPU** was almost five times faster than **MyCPU**. The reason for this behaviour may arise from the fact that all four cores of the CPU are activated for the **OpenCV CPU** implementation and the device draws more current from the supply for the OpenCV version. It was observed that the multi-threaded **OpenCV CPU** drew around $500mA$ while **MyCPU** drew about $200mA$.

TABLE 4.10: Energy Performance Measurement

	OpenCV CPU	MyCPU		MyGPU Mapped RTC WC		MyGPU Mapped RTC WC LM	
Images Res - #Object	Energy(J)	Energy(J)	%Improvement	Energy(J)	%Improvement	Energy(J)	%Improvement
Full HD Object - 1	3.81	8.42	-121.00	0.61	83.99	2.23	41.47
Full HD Object - 2	7.66	16.15	-110.84	3.38	55.87	2.85	62.79
Full HD Object - 3	7.71	14.30	-85.47	1.33	82.75	2.65	65.63
Full HD Object - 9	7.89	17.29	-119.14	2.41	69.46	3.24	58.94
Full HD Object - 19	8.08	17.62	-118.07	2.88	64.36	3.35	58.54
Full HD Object - 72	9.78	19.56	-100.00	4.07	58.38	3.92	59.92
Geometric Mean	**7.21**	**15.04**	**-108.29**	**2.06**	**68.28**	**2.99**	**57.28**
512x512 - 1	0.80	1.76	-120.00	0.44	45.00	0.42	47.50
450x326 - 2	0.43	0.95	-120.93	0.28	34.88	0.38	11.63
647x650 - 31	1.68	3.68	-119.05	1.07	36.31	0.92	45.24
Geometric Mean	**0.83**	**1.83**	**-119.99**	**0.51**	**38.48**	**0.53**	**29.24**

Both the GPU based implementations measured consistently showed energy efficiency improvement over the **OpenCV CPU** version. In particular, the best energy efficiency (about 84%) improvement was achieved for **MyGPU Mapped RTC WC** when run with a Full HD image with only one object in the scene. This can be attributed to the nature of the algorithm that rejected most of the candidate windows at the early stages of the cascade classifier. Lower but positive energy efficiency improvements were measured for the smallest resolution images although they are still significantly better than the **OpenCV CPU** implementation.

The **MyGPU Mapped RTC WC LM** implementation had a maximum of 66% and a minimum of 11% energy consumption improvement over the baseline. Recall from the previous section that **MyGPU Mapped RTC WC LM** was mostly slower in runtime compared to the **OpenCV CPU** implementation. This shows that even when an implementation is under performing in runtime on the mobile GPU, there is a higher chance that energy can be saved by pushing some computation to the GPU. This suggests that for non-real time applications using the mobile GPU will definitely be beneficial in conserving battery charge levels.

4.2.5 Comparison with other works

A summary of speed ups and energy consumption reductions reported in the literature reviewed in Section 2.5 is given in Table 4.11. The first column lists applications on mobile GPUs while in the second column the type of GPU used with the applications is given. The maximum resolution of images in the dataset used with experiments for the particular applications is given in the third column. The fourth column shows what type of programming language is used for the implementation of the corresponding application. In column five the speed up is given as reported in the literature while the energy reduction reported is given in column six. Energy consumption reduction data is not provided by almost half of the literature reviewed as can be seen in Table 4.11.

It is difficult from Table 4.11 to directly compare our work with others as most of the applications in reviewed literature are different with differing algorithmic behaviour and different data sets used. Also, the GPUs used in the experiments in the literature are from different vendors and different generations which is another reason that makes the comparison difficult. If one must compare the different applications, the resolution of data set can be used as comparison metric. In other words, is there performance gain while processing the largest resolution images. One can observe from the table the speed ups reported for the other applications are higher than for our application. However, the resolution of images used in the other applications is significantly lower than what our application can processes. Unlike the others, our application has an average speed up of 35% with an average 68% energy usage reduction while processing full high definition images. Also, our reported speed up is not against a serial version of the respective algorithm on the CPU. Our average 35 speed gain reported is when compared against a parallel version of the same application running on a quad core mobile CPU.

TABLE 4.11: Comparison with other implementations on the mobile GPU

Application	GPU Type	Data Set (Max. Res.)	Programming Language	Speed up	Energy Reduction (%)
Face Recognition [6]	NVIDIA Tegra	64x84	OpenGL ES	2	45.3
SIFT [39]	NVIDIA Tegra	320x280	OpenGL ES	4-7	87
SIFT [45]	Adreno 320	320x256	OpenCL	1.69	41
uSURF-ES [43]	Multiple Systems	512x384	OpenGL ES	2-14	Not Given
Cartoon Style NPR [44]	PowerVR SGX 540	800x480	OpenGL ES	5	Not Given
SURF [44]	PowerVR SGX 540	800x480	OpenGL ES	1.7	Not Given
Stereo Matching [44]	PowerVR SGX 540	384x288	OpenGL ES	5-7	Not Given
Object Removal [46]	Adreno 330	1024x550	OpenCL	8.44 - 28.3	Not Given
Embodied Robot Simulation [47]	ARM MALI T604 & T628 MP6	N/A	OpenCL	1.82	30
Viola-Jones Object Detection [29]	Adreno 420	1920x1080	OpenCL	1.35	68

4.3 Summary

In this chapter, we discussed how the Viola-Jones [21] object detection can be implemented on a mobile GPU. The computation of the integral image needed before objects can be searched from the images was first discussed. It was then followed by discussion of how the actual object search can be conducted on an input image. First a serial (single threaded) version of the Viola-Jones [21] object detection algorithm, that has similar computations as this algorithm's parallel implementation in the OpenCV [18] library, was implemented. This was done to understand the algorithm's complexity and performance need. Then this serial implementation was ported into a naive parallel implementation on a mobile GPU. Further enhancements of the naive implementation that utilized performance enhancing aspects of the GPU were shown.

It was indeed found out that running the object detector on the mobile GPU used for testing had more than two folds speed up for some full high definition images and is 35% faster on average than the parallel OpenCV [18] implementation on the mobile CPU. Our energy consumption measurements showed that our best implementation has a better energy consumption trend than OpenCV [18] on the CPU even when our implementation is slower in runtime.

Chapter 5

Application to Medical Image Classification

In Chapter 1 Section 1.3 we discussed the relevance of this work has to our country, Ethiopia. To this effect, this chapter discuss our efforts to apply the Viola-Jones [21] object detection for medical image classification. In particular, we focus on the detection of the ringworm skin disease from image samples. We describe what ring-worm disease is, the procedures followed to train a new classifier for ring-worm detection and our test results in the subsequent sections.

5.1 Ringworm

The ringworm is an infection of the skin and nail caused by dermatophyte fungi which are pathogenic [61]. There are about 40 species of dermatopytes that can cause the ringworm infection. The ringworm infection is also referred to as *tinea* or *dermatophytosis*. Ringworm infection can occur on different parts of a person's body and has different names accordingly. According to the naming convention described in [61], if a name for ringworm disease starts with *tinea*, then the body part on which the disease occurs will be referred in Latin. On the other hand, if naming starts with *dermatophytosis*, then the body part it occurs on is referred in English. The list below gives the specialised ringworm infections according to the definitions found from Center for Disease Control [62]

- *tinea capitis* or *dermatophytosis of the scalp* is ringworm infection found on the scalp
- *tinea faciei* or *dermatophytosis of the face* is ringworm infection found on a person's face
- *tinea barbae* or *dermatophytosis of the beard* is ringworm infection found on a person's beard and moustache area usually found in adult men

- *tinea manuum* or *dermatophytosis of the hands* is ringworm infection found on the hands
- *tinea pedis* or *dermatophytosis of the feet* is ringworm infection found on the feet of a person and is commonly referred to as *athlete's foot*
- *tinea cruris* or *dermatophytosis of the groin and perianal area* is ringworm infection found on the inner thighs, the groin and the buttocks
- *tinea unguium* or *dermatophytosis of the fingernails* is ringworm infection found on the fingernails of the hand or toenails
- *tinea corporis* or *dermatophytosis of the body* is ringworm infection found on the other parts of the body

The ringworm infection is usually characterized by itchy, red circular rash. Figure 5.1 shows two sample pictures of Ringworm infection. In Figure 5.1a, the type of the Ringworm infection is *tinea corporis* as the infection is located by the shoulder blades. In Figure 5.1b, the ringworm infection is of the type *tinea faciei*. The ringworm disease can occur both on humans and animals, however, in this work only ringworm disease occurring on human skin is considered.

(a) Tinea Corporis (b) Tinea Faciei

FIGURE 5.1: Sample pictures of Ringworm Skin Disease

According to [30], the estimated incidence of ringworm in sub-saharan Africa was about 78 million in 2005. The research conducted in [30], [63], [64] reported that the incidence of ringworm disease is mainly present with pre-teen children in select regions of Ethiopia, Egypt and Nigeria. In [65], the type and causing strain of the ringworm (dermaphytosis) are identified in a hospital that caters to dermatological diseases in Addis Ababa, Ethiopia. In [66], the risk factors and communicability of ringworm disease of the scalp in south-western region of Ethiopia is studied.

5.2 Related Work

In this section, we discuss research that has tried to automatically classify ringworm disease. In the work by Kundu et al. [67], the local binary pattern (LBP) [68] features extracted from skin images are used as an input to machine learning algorithms for ringworm detection. The dataset used in this experiment contains 70 positive and 70 negative images. The images in the dataset have been resized to have a uniform size and converted to gray scale as well. 50% of the positive and negative images are used for training and the remaining 50% are used for testing. The extracted LBP features are used for training three classifiers; namely, the Bayesian classifier[69], the Support Vector Machine (SVM) [70], and the Multi-layer Perceptron (MLP) [71]. After training, 72.85%, 90%, and 94.28% accuracy were reported for the Bayesian classifier, SVM and MLP respectively. The researchers combined the three classifiers into a majority voting scheme which resulted in a detection accuracy of 91.42%.

In [72] Saha et al., extracted wavelet energy signatures of the skin and used it to detect ringworm disease. Their approach is based on the nature of the ringworm infection having a circular scaly outer bound that contrasts with the internal and surrounding skin. Hence, the input image is 3-level decomposed for the extraction of Daubechies (DB), Coiflet (CF), Biorthogonal (BO) and Discrete Meyer (DM) wavelets energy signatures of the skin. The extracted wavelet energy signatures are then used as features for the Support Vector Machine (SVM) for classification. They used 35 positive and 35 negative samples. These images were converted into gray scale and resized so as to have a uniform size. The images were divided in training and testing although the testing set has only 3 images from each class. After training, the reported accuracy on the testing data set is 86.66%.

In [73], the ringworm and scabies diseases are segmented using K-means and Fuzzy clustering means (FCM) methods. It is reported that the K-means clustering method out performed the FCM the reason being that FCM uses grayscale images. In [74], fungal infections of the scalp are automatically detected from Optical Coherent Tomography (OCT) images. One of the targeted fungal infections in this work is the tinea capitis which is ringworm of the scalp. The dataset used in this work had 30 positive and 30 negative OCT images. These were used with a multi-level ensemble machine learning model that has fused decision tree, extreme learning machine (ELM), neural network, support vector machine (SVM), random forest, average neural network (avNNet). The training was conducted with 60% of the dataset and testing was done on the rest. They were able to achieve binary classification for the presence of fungal infections with an accuracy of 91.66%.

The researches discussed earlier focused on how to automatically classify the existence of fungal infections specifically ringworm infection on human skin. Machine learning methods were used as the classifiers. The processing platform used

in these researches were desktop computer environments. We were not able to find automatic detection of the ringworm skin disease on mobile computing platforms such as smart phones in our research of the literature. As discussed in Chapter 1, smart phones are ubiquitously available in most parts of the world. Also given the large incidence of ringworm skin disease in school age students [30, 63, 64], implementing a ringworm skin disease detector on a mobile platform will allow the early detection and diagnosis of the disease.

5.3 Training the Ringworm Cascade Classifier

In order to use the Viola-Jones object detector [21] for detection of any object of interest a classifier has to be trained first for the particular object. In this section we detail the process we followed to train a classifier for the ringworm skin disease detection.

5.3.1 Data Collection

The precursor for the training process is the collection of a suitable dataset. The dataset should include positive samples which can be representatives of the the object of interest which in our case is a picture of a ringworm infection. The negative samples can be any image that strictly does not contain the object of interest. However, it is preferred for the negative samples to be images of the natural background where the object of interest may appear usually i.e. images of normal skin for our case.

The image samples in our dataset were collected from various sources such as the Internet. We have collected about 63 positive samples that contain images of different ringworm infections from sources on the internet (a complete list of sources can be found in Table B.1 and Table B.2 of Appendix B). These samples are then used to generate training and testing positive dataset. In our case, 52 of the 63 positive samples were used to generate more than 1000 positive training samples. The rest 11 were used to generate around 300 positive testing samples. This generation of many positive samples from fewer originals is called data augmentation [75]. Negative samples were prepared from our personal collection of images composed of skin from peoples' faces and other body parts. The number of images in the negative samples dataset is a little more than 1000.

5.3.2 Training process

In this section, we discuss the process that is followed to train a classifier that can classify the ringworm infection. We describe the tools that are used in the training process and give descriptions of the different classifier trainings done using the dataset described in Section 5.3.1.

Tools used for training

The cascade classifier training process described in [76] and [77] is followed in this work where the OpenCV [18] library is used for the training process. The OpenCV libary contains three tools that are used for training a cascade classifier that detects the object of interest in an image. These are

- ***opencv_annotation:*** is a tool used for annotating (selecting) portions of an input image as the object of interest. This tool produces a file that has a list of coordinates of the rectangles containing the object of interest. These annotated areas are considered to be truly the positive samples.
- ***opencv_createsamples:*** this tool is used for creating more positive samples from the already annotated positive samples through data augmentation[75]. The new training positive samples are created by transforming the positive samples by rotating the rectangles within a specified bound of rotation angles and variation of lighting condition of the positive samples.The test samples are generated in a similar way and then are inserted inside a set of given negative samples.This way the generated testing will be used for testing the quality of the classifier trained.
- ***opencv_traincascade:*** is a tool that actually does the training of the cascade classifier given the input positive and negative samples of the object of interest. Users can specify if they wanted to use upright features only or a combination of upright features with 45° rotated features.

The user is allowed to set different parameters that may lead to a good classifier. The main parameter that needs to be used with all the tools discussed above is the size of the rectangle containing the object of interest. The value of this parameter has to be the same when used with all three tools. In our case the value of this rectangle is set to $20\ by\ 20$ pixels.

Classifier Training

The *opencv_annotation* tool was used to annotate the 63 positive samples mentioned in section 5.3.2. Of the 63 positive samples, the 52 samples were used to generate more than 1000 new positive training samples using the *opencv_createsamples* tool. The *opencv_createsamples* tools was also used to generate positive samples for testing. We have trained many classifiers as detectors for the ringworm infection based on these generated samples. The different classifiers were trained by varying the false alarm rate used with the training tool. The training process was conducted on a desktop computer system.

The different classifiers were trained with 1001 positive samples and 999 skin based negative samples. The *maxFalseAlarmRate* was varied from 0.3 to 0.95 in steps 0.05. The *maxWeakCount* was kept constant at 200. Most of the parameters of the training tool were left to their default values. The following list shows parameter values that are changed in this work

- **Feature mode:** upright features only
- **object image width:** 20 pixels
- **object image height:** 20 pixels
- **# of positive samples:** 1001
- **# of negative samples:** 999
- **# of stages:** 20 stages in the classifier
- **maxFalseAlarmRate:** was varied between $0.3 - 0.95$ in 0.05 step
- **maxWeakCount:** 200

Test Results

After the training process was done, the trained classifiers were tested on a test set of 307 generated positive test samples and 307 negative samples. The performance of each classifier on the test dataset was measured using the performance metrics *accuracy*, *precision*, *recall* and the *F1-Score*[78]. Accuracy determines the accurate prediction of both the positive and negative samples from the total test dataset. Precision measures the quality of the classifier by computing the percentage of true positives from all positive predictions. Recall measures the sensitivity of the classifier by determining the percentage of true positives predicted from the actual positive samples. These three metrics separately do not necessarily reflect how good a classifier is. Such a meteric that is usually used for determining the quality of a classifier is

the F1-score. The F1-score combines precision and recall of a classifier by computing their harmonic average. The higher the F1-Score the better the classifier is expected to be.

Accuracy, precision and recall are computed based on the confusion matrix given in Table 5.1. The first row of the confusion matrix shows how many of the input positive samples the classifier predicted as positive and negative. The second row shows the negative samples predicted as positive and negative. Equations 5.1, 5.2, 5.3, 5.4 show how accuracy, precision, recall and F1-score are computed respectively.

TABLE 5.1: Confusion Matrix

	Predicted Positive	Predicted Negative
Actual Positive	#True Positive (TP)	#False Negative (FN)
Actual Negatives	#False Positive (FP)	#True Negative (TN)

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

$$Precision = \frac{TP}{TP + FP} \tag{5.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.3}$$

$$F1 - score = \frac{2 * (Precision * Recall)}{Precision + Recall} \tag{5.4}$$

We named the trained classifiers after the *maxFalseAlarmRate* used to train them. For example, a classifier trained with a false alarm rate of 0.3 is called c0.3 where the letter "c" indicates that it is a classifier. Table 5.2 gives the results for the trained classifiers. The first column in the table gives the classifier names. The next four columns give the number of True Postives, False Postives, True Negatives and False Negatives measured for each classifier, respectively. The computed accuracy, precision, recall and F1-Score are given in the next successive columns.

In Table 5.2, there are four good candidates based on accuracy while there are four good candidates based on the F1-score (with > 0.7 value). In the first set are classifiers **c0.35**,**c0.4**,**c0.45** and **c0.5** which have respective accuracies of 0.72, 0.73,0.76 and 0.75. In the second set, classifiers **c0.45**, **c0.5**,**c0.55** and **c0.6** with 0.73, 0.74, 0.72 and 0.71 F1-Scores respectively are grouped. Two classifiers from the two sets described earlier overlap. These are classifier **c0.45** which has an accuracy of 0.76 (76%) and F1-Score of 0.73, and classifier **c0.5** with accuracy of 0.75 (75%) and F1-Score of

TABLE 5.2: Ringworm Classifiers Testing Results

classifer	maxFalseAlarmRate	True Positive	False Positive	True Negative	False Negative	Accuracy	Precision	Recall	F1-Score
c0.3	0.3	103	5	302	204	0.66	0.95	0.34	0.0.50
c0.35	0.35	143	9	298	164	**0.72**	0.94	0.47	0.62
c0.4	0.4	149	8	299	158	**0.73**	0.95	0.49	0.64
c0.45	0.45	195	33	274	112	**0.76**	0.86	0.64	**0.73**
c0.5	0.5	215	62	245	92	**0.75**	0.78	0.70	**0.74**
c0.55	0.55	252	140	167	55	0.68	0.64	0.82	**0.72**
c0.6	0.6	269	185	122	38	0.64	0.59	0.88	**0.71**
c0.65	0.65	276	225	82	31	0.58	0.55	0.90	0.68
c0.7	0.7	289	258	49	18	0.55	0.53	0.94	0.68
c0.75	0.75	291	168	39	16	0.54	0.52	0.95	0.67
c0.8	0.8	296	288	19	11	0.51	0.51	0.968	0.66
c0.85	0.85	287	296	11	20	0.49	0.49	0.93	0.64
c0.9	0.9	239	301	6	68	0.40	0.44	0.78	0.56
c0.95	0.95	190	305	2	117	0.31	0.38	0.61	0.47

0.74. The desirable value of the metrics accuracy and F1-score is the higher the better. It seems there is no clear winner as **c0.45** has more accuracy than **c0.5** while **c0.5** has more F1-Score than **c0.45**. However, the F1-Score will have more weight as it is a combination of two other metrics. Therefore, the classifier c0.5 is selected as the best classifier with good qualities from the trained classifiers in this work.

We were able to show that with limited number of positive samples, it is possible to train a classifier for the ringworm skin disease using the Viola-Jones [21] based object detection. Although, the accuracy of our classifier compared to the work by others discussed earlier is lower, it is first of its kind that can work on the mobile GPU. Also, there is a great chance that the accuracy of the classifier can be improved with access to clinical ringworm skin disease image dataset.

5.4 Summary

In this chapter, the nature and types of the ringworm skin disease were briefly discussed. Researches that have been done to automatically detect the presence of the ringworm skin disease from images were presented. It was observed from our literature review that all the attempts at automatic detection of ringworm skin disease were done on the desktop system. This presented an opportunity for our object detector implemented on mobile GPU to be used as detector for the ringworm skin disease on the mobile platform. The process of training a classifier to be used for this purpose was presented. Classifiers were trained and the best one was chosen to be used with our object detector.

Chapter 6

Conclusion and Future Work

This chapter is organized in two sections. In the first section we give a brief summary of the contributions made in this work and conclude. In the second section we give recommendations for future work.

6.1 Contributions

In this work, three tasks were conducted, namely; the performance characterization of a set of mobile GPUs [28], the implementation and optimization of the Viola-Jones [21] object detector on a mobile GPU [29] and the training of a cascade classifier for the detection of the ringworm skin disease from an image of a person's skin.

- In the first task measurements were done to understand the mobile GPU's architecture and its related performance. The lessons learned are listed below
  - Data transfer throughput between the host and the device was measured to be very low. Hence, its use should be reduced or completely avoided in systems that have shared memory such as mobile (smartphones) systems.
  - Data transfer throughput of different global memory access patterns were measured. The coalesced memory access had the best performance. The worst performing pattern was the strided access whose effect can be reduced by moving data to the local memory.
  - Memory latencies were measured for the different levels of cache, local memory and global memory for the mobile GPU. Also, this technique was used to measure the effect of bank conflicts in the local memory. Understanding this helps with choosing where to store data that is frequently accessed.
  - Computational latencies were also measured for the basic arithmetic operations. This method was adapted to measure the available parallelism in a GPU. The insights gained from these measurements were used for

example to reduce the use of divisions or replace it with multiplication operations and to limit the number work-items used in a work group to utilize the available parallelism.

- The second task was the implementation and optimization of the Viola-Jones [21] object detection on a specific mobile GPU. The following steps were implemented on the GPU
  - The input gray image was successively reduced (scaled down) using the bilinear interpolation method to create an image pyramid.
  - The integral and square integral images were computed for each scale in the image pyramid and stored in memory.
  - Different approaches were used to implement the Viola-Jones[21] algorithm on the mobile GPU
  - The implemented object detector was tested with sample images and the runtime performance and energy consumption of the different approaches were measured.

  The object detector on the mobile GPU was measured to be 35% faster on average and more than two folds faster on the best case on Full HD images as compared to an optimized parallel implementation on the CPU of the same mobile device. In addition the energy consumption measurement revealed that an energy consumption reduction of 35% to 84% can be achieved on the entire test dataset. It was also observed that our best object detector has a better energy consumption trend than OpenCV's object detector [18] implementation on the mobile CPU even when our implementation is slower in runtime.

- Our third task was using the object detector for a medical image classification problem. The ringworm skin disease was selected as it has large prevalence in sub-saharan Africa[30] and as it easy for a person to take a picture of a skin lesion. The following steps were taken to train a classifier for the ringworm disease
  - Positive and negative sample were prepared where positive sample means an image that contains a ringworm while a negative sample doesn't. As the number of positive samples collected was smaller than expected data augmentation was applied to get a larger positive data set.
  - Many classifiers were trained on the prepared dataset by varying the hyper parameters of the classifier.
  - The trained classifiers were applied on the test dataset to select the best classifier. The best classifier was selected based on its accuracy and F-1 Score which are 75% and 0.74 respectively.

From the results discussed earlier, we can conclude that a better understanding of a mobile GPU architecture is achieved. Using this understanding an energy efficient object detector with enhanced runtime performance was implemented on the mobile GPU. We have also shown that the object detector can be used for a medical image classification to detect the ringworm skin disease on the mobile GPU.

6.2 Future work

Although we have shown good results can be achieved by using the mobile GPU, improvements can still be made if the following approaches are taken for the object detector implementation and the ringworm detection. Accordingly, we make the following recommendations

- *Heterogeneous (CPU-GPU) computing*: our best implementation of the object detector on the GPU (**MyGPU Mapped RTC WC**) was implemented with the premise of a fixed set of work-items retrieving work from a work-queue. It is to be remembered from Section 4.2.3 that the amount of work reduces as the stage number increases during the application of the cascade classifier. Hence, as work gets depleted from the work-queue, more and more work-items become idle. This means load balancing will be an issue at later stages. However, if computation is transferred to the CPU the load balancing issue would be reduced. We recommend further work to figure out at which stage of the classifier to move computation from the GPU to the host CPU.

- *Accuracy of Ringworm Detection*: the accuracy of our ringworm classifier is lower than what is expected for a clinical level detector. Also, our classifier doesn't differentiate between different classes of ringworm disease. The qualities of our classifier are limited because of the small positive samples used for training. Therefore, we recommend further work to collect more positive samples and retrain the classifier.

- *Deep learning*: deep learning approaches have become more mature and a lot of research is being done in this area. We think the use of deep learning can be used for multi-class ringworm classification.

- *Specialized Neural Cores*: specialized hardware cores that are used for neural processing are coming into the picture such as the Tensor cores that are present in recent NVidia GPUs and in some smart phones. These cores enable energy efficient faster training and classification for deep learning algorithms. Therefore, we recommend the adoption of these cores for future medical image classification.

Bibliography

[1] Robert R Schaller. "Moore's law: past, present and future". In: *IEEE spectrum* 34.6 (1997), pp. 52–59.

[2] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. "A survey of general-purpose computation on graphics hardware". In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[3] Cristobal A Navarro, Nancy Hitschfeld-Kahler, and Luis Mateu. "A survey on parallel computing and its applications in data-parallel problems using GPU architectures". In: *Communications in Computational Physics* 15.2 (2014), pp. 285–329.

[4] Eric Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. *The TOP 500 list*. URL: http://www.Top500.org/lists (visited on 08/05/2017).

[5] Song Huang, Shucai Xiao, and Wu-chun Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–8.

[6] Kwang-Ting Cheng and Yi-Chu Wang. "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones". In: *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*. IEEE. 2011, pp. 1–4.

[7] Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R Cavallaro. "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU-a case study". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 2629–2633.

[8] Karim Arabi. *Low power Design Techniques in Mobile Processors*. 2004.

[9] Android Developers. "Android, the world's most popular mobile platform". In: *Google, USA* (2013).

[10] David Ehringer. "The dalvik virtual machine architecture". In: *Techn. report (March 2010)* 4 (2010), p. 8.

[11] Google. *ART and Dalvik*. URL: https://source.android.com/devices/tech/dalvik/ (visited on 07/19/2018).

[12] *Introducing Neon*. Development Article. Arm Ltd. 2009.

[13] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU". In: *ACM SIGARCH computer architecture news* 38.3 (2010), pp. 451–460.

[14] James Fung and Steve Mann. "OpenVIDIA: parallel GPU computer vision". In: *Proceedings of the 13th annual ACM international conference on Multimedia*. ACM. 2005, pp. 849–852.

[15] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. "Gpucv: an opensource gpu-accelerated framework forimage processing and computer vision". In: *Proceedings of the 16th ACM international conference on Multimedia*. ACM. 2008, pp. 1089–1092.

[16] James Fung and Steve Mann. "Using graphics devices in reverse: GPU-based image processing and computer vision". In: *Multimedia and Expo, 2008 IEEE International Conference on*. IEEE. 2008, pp. 9–12.

[17] W Hwu Wen-Mei. *GPU computing gems emerald edition*. Elsevier, 2011, pp. 439–546.

[18] *The Open Computer Vision Library*. URL: http://www.opencv.org/about.html (visited on 08/05/2017).

[19] *NVIDIA CUDA*. Programming Guide. NVIDIA. 2008.

[20] Khronos OpenCL Working Group. *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*. URL: http://www.khronos.org/opencl (visited on 08/05/2017).

[21] Paul Viola and Michael Jones. "Rapid object detection using a boosted cascade of simple features". In: *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2001, pp. I–I.

[22] Michael Jones and Paul Viola. "Fast multi-view face detection". In: *Mitsubishi Electric Research Lab TR-20003-96* 3.14 (2003), p. 2.

[23] Theo Ephraim, Tristan Himmelman, and Kaleem Siddiqi. "Real-time viola-jones face detection in a web browser". In: *Computer and Robot Vision, 2009. CRV'09. Canadian Conference on*. IEEE. 2009, pp. 321–328.

[24] Daniel Hefenbrock, Jason Oberg, Nhat Tan Nguyen Thanh, Ryan Kastner, and Scott B Baden. "Accelerating Viola-Jones face detection to FPGA-level using GPUs". In: *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*. IEEE. 2010, pp. 11–18.

[25] Modesto Castrillón, Oscar Déniz, Daniel Hernández, and Javier Lorenzo. "A comparison of face and facial feature detectors based on the Viola–Jones general object detection framework". In: *Machine Vision and Applications* 22.3 (2011), pp. 481–494.

[26] Qian Li, Usman Niaz, and Bernard Merialdo. "An improved algorithm on Viola-Jones object detector". In: *Content-Based Multimedia Indexing (CBMI), 2012 10th International Workshop on*. IEEE. 2012, pp. 1–6.

[27] Yi-Qing Wang. "An analysis of the Viola-Jones face detection algorithm". In: *Image Processing On Line* 4 (2014), pp. 128–148.

[28] Fitsum Assamnew Andargie and Jonathan Rose. "Performance characterization of mobile GP-GPUs". In: *AFRICON, 2015*. IEEE. 2015, pp. 1–6.

[29] Fitsum Assamnew Andargie, Jonathan Rose, Todd Austin, and Valeria Bertacco. "Energy efficient object detection on the mobile GP-GPU". In: *AFRICON, 2017 IEEE*. IEEE. 2017, pp. 945–950.

[30] Alem Alemayehu, Gebremedhin Minwuyelet, and Gizachew Andualem. "Prevalence and Etiologic Agents of Dermatophytosis among Primary School Children in Harari Regional State, Ethiopia". In: *Journal of Mycology* 2016 (2016).

[31] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". In: *Parallel Computing* 38.8 (2012), pp. 391–407.

[32] Lee Howes and Aaftab Munshi. *The OpenCL Specification*. Version 2.0. Khronos OpenCL Working Group. 2015.

[33] Michael Sung. "Simd parallel processing". In: *Architectures Anonymous* 6 (2000), p. 11.

[34] Edgar Osuna, Robert Freund, and Federico Girosit. "Training support vector machines: an application to face detection". In: *Computer vision and pattern recognition, 1997. Proceedings., 1997 IEEE computer society conference on*. IEEE. 1997, pp. 130–136.

[35] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. Vol. 1. IEEE. 2005, pp. 886–893.

[36] Yoav Freund and Robert E Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting". In: *Journal of computer and system sciences* 55.1 (1997), pp. 119–139.

[37] Rainer Lienhart and Jochen Maydt. "An extended set of haar-like features for rapid object detection". In: *Image Processing. 2002. Proceedings. 2002 International Conference on*. Vol. 1. IEEE. 2002, pp. I–I.

[38] Kari Pulli, Wei-Chao Chen, Natasha Gelfand, Radek Grzeszczuk, Marius Tico, Ramakrishna Vedantham, Xianglin Wang, and Yingen Xiong. "Mobile visual computing". In: *Ubiquitous Virtual Reality, 2009. ISUVR'09. International Symposium on*. IEEE. 2009, pp. 3–6.

[39] Blaine Rister, Guohui Wang, Michael Wu, and Joseph R Cavallaro. "A fast and efficient SIFT detector using the mobile GPU". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 2674–2678.

[40] David G Lowe. "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[41] Wonwoo Lee, Youngmin Park, Vincent Lepetit, and Woontack Woo. "Point-and-shoot for ubiquitous tagging on mobile phones". In: *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on*. IEEE. 2010, pp. 57–64.

[42] Andrew Ensor and Seth Hall. "GPU-based image analysis on mobile devices". In: *arXiv preprint arXiv:1112.3110* (2011).

[43] Robert Hofmann, Hartmut Seichter, and Gerhard Reitmayr. "A GPGPU accelerated descriptor for mobile devices". In: *Mixed and Augmented Reality (ISMAR), 2012 IEEE International Symposium on*. IEEE. 2012, pp. 289–290.

[44] Nitin Singhal, Jin Woo Yoo, Ho Yeol Choi, and In Kyu Park. "Implementation and optimization of image processing algorithms on embedded GPU". In: *IEICE TRANSACTIONS on Information and Systems* 95.5 (2012), pp. 1475–1484.

[45] Guohui Wang, Blaine Rister, and Joseph R Cavallaro. "Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone". In: *Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE*. IEEE. 2013, pp. 759–762.

[46] Guohui Wang, Yingen Xiong, Jay Yun, and Joseph R Cavallaro. "Computer vision accelerators for mobile systems based on opencl gpgpu co-processing". In: *Journal of Signal Processing Systems* 76.3 (2014), pp. 283–299.

[47] Simon Jones, Matthew Studley, and Alan Winfield. "Mobile GPGPU acceleration of embodied robot simulation". In: *Artificial Life and Intelligent Agents Symposium*. Springer. 2014, pp. 97–109.

[48] James A Ross, David A Richie, Song J Park, Dale R Shires, and Lori L Pollock. "A case study of OpenCL on an Android mobile GPU". In: *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*. IEEE. 2014, pp. 1–6.

[49] Jeremiah van Oosten. *Optimizing CUDA Applications*. URL: https://www.3dgep.com/optimizing-cuda-applications/ (visited on 02/12/2019).

[50] CUDA NVidia. "C best practices guide". In: *NVIDIA, Santa Clara, CA* (2018).

[51] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. "Demystifying GPU microarchitecture through microbenchmarking". In: *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE. 2010, pp. 235–246.

[52] *MDP Tablet based on the Qualcomm® Snapdragon™ 805 Processor*. Qualcomm Technologies, Inc. URL: https://www.qualcomm.com/documents/snapdragon-805-processor-product-brief (visited on 11/07/2017).

[53] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. "GPU computing". In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[54] Zhiyi Yang, Yating Zhu, and Yong Pu. "Parallel image processing based on CUDA". In: *Computer Science and Software Engineering, 2008 International Conference on*. Vol. 3. IEEE. 2008, pp. 198–201.

[55] *Bilinear Image Scaling*. URL: http://tech-algorithm.com/articles/bilinear-image-scaling/ (visited on 02/11/2017).

[56] Mark Harris, Shubhabrata Sengupta, and John D Owens. "Parallel prefix sum (scan) with CUDA". In: *GPU gems* 3.39 (2007), pp. 851–876.

[57] Guy E Blelloch. "Prefix sums and their applications". In: (1990).

[58] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1*. Newnes, 2012.

[59] Paulius Micikevicius. "Maximizing Face Detection Performance". GPU Technology Conference. 2015.

[60] Haipeng Jia, Yunquan Zhang, Weiyan Wang, and Jianliang Xu. "Accelerating viola-jones facce detection algorithm on gpus". In: *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*. IEEE. 2012, pp. 396–403.

[61] FC Odds, T Arai, AF Disalvo, EGV Evans, RJ Hay, HS Randhawa, MG Rinaldi, and TJ Walsh. "Nomenclature of fungal diseases: a report and recommendations from a Sub-Committee of the International Society for Human and Animal Mycology (ISHAM)". In: *Journal of Medical and Veterinary Mycology* 30.1 (1992), pp. 1–10.

[62] *Ringworm Definition*. URL: https://www.cdc.gov/fungal/diseases/ringworm/definition.html (visited on 05/14/2018).

[63] Azza GA Farag, Mostafa A Hammam, Reda A Ibrahim, Reda Z Mahfouz, Nada F Elnaidany, Masroor Qutubuddin, and Rehab RE Tolba. "Epidemiology of Dermatophyte Infections among School Children in Menoufia Governorate, Egypt". In: *Mycoses* (2018).

[64] Eziyi Iche Kalu, Victoria Wagbatsoma, Ephraim Ogbaini-Emovon, Victor Ugochukwu Nwadike, and Chiedozie Kingsley Ojide. "Age and sex prevalence of infectious dermatoses among primary school children in a rural South-Eastern Nigerian community". In: *Pan African Medical Journal* 20.1 (2015).

[65] Y Woldeamanuel, R Leekassa, E Chryssanthou, Y Mengistu, and B Petrini. "Clinico-mycological profile of dermatophytosis in a reference centre for leprosy and dermatological diseases in Addis Ababa". In: *Mycopathologia* 161.3 (2006), pp. 167–172.

[66] Jose Ignacio Figueroa, Thomas Hawranek, Aynalem Abraha, and Roderick James Hay. "Tinea capitis in south-western Ethiopia: a study of risk factors for infection and carriage". In: *International journal of dermatology* 36.9 (1997), pp. 661–666.

[67] Srimanta Kundu, Nibaran Das, and Mita Nasipuri. "Automatic detection of ringworm using local binary pattern (LBP)". In: *arXiv preprint arXiv:1103.0120* (2011).

[68] Timo Ojala, Matti Pietikäinen, and David Harwood. "A comparative study of texture measures with classification based on featured distributions". In: *Pattern recognition* 29.1 (1996), pp. 51–59.

[69] Pedro Domingos and Michael Pazzani. "On the optimality of the simple Bayesian classifier under zero-one loss". In: *Machine learning* 29.2-3 (1997), pp. 103–130.

[70] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. "Support vector machines". In: *IEEE Intelligent Systems and their applications* 13.4 (1998), pp. 18–28.

[71] Simon S Haykin. *Neural networks : a comprehensive foundation*. 3ed. Prentice Hall, 2008, pp. 122–229. ISBN: 0-13-147139-2,978-0-13-147139-9.

[72] Manas Saha, Mrinal Kanti Naskar, and Biswa Nath Chatterji. "Human ringworm detection using wavelet energy signature". In: *Recent Trends in Information Systems (ReTIS), 2015 IEEE 2nd International Conference on*. IEEE. 2015, pp. 178–182.

[73] Olusayo Deborah Fenwa, OO ALO, and AS FALOHUN. "Coloured Image Segmentation Using K-Means Algorithm". In: *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY* 15.3 (2016), pp. 6555–6562.

[74] Kavita Dubey, Vishal Srivastava, and Dalip Singh Mehta. "Automated in vivo identification of fungal infection on human scalp using optical coherence tomography and machine learning". In: *Laser Physics* 28.4 (2018), p. 045602.

[75] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[76] *Cascade Classifier Training*. URL: https://docs.opencv.org/3.1.0/dc/d88/tutorial_traincascade.html (visited on 04/04/2018).

[77] Thorsten Ball. *TRAIN YOUR OWN OPENCV HAAR CLASSIFIER*. URL: https://coding-robin.de/2013/07/22/train-your-own-opencv-haar-classifier.html (visited on 04/04/2018).

[78] David Martin Powers. "Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation". In: (2011).

Appendix A

OpenCL Application Example: Matrix Multiplication

The following code demonstrates how to program in OpenCL using matrix multiplication as an example. The program is organized in three components where the first component is the main program that initializes the OpenCL environment, stages data and submits the data for computation on the OpenCL device. The second component is the matrix multiplication implementation in C programming language. The third component is the matrix multiplication kernel in OpenCL. In the end of the main program results of the matrix multiplication computed by the different methods is compared.

LISTING A.1: Matrix multiplication in OpenCL - staging

```
1  //  main.cpp
2  //  MatrixMultiplication
3
4  #include <iostream>   //header file console input output
5  #include "MatMul.h"   //header file for C based matrix multiplication
6
7  /* Header for OpenCL - has different names for different platforms */
8  #ifdef __APPLE__
9  #include <OpenCL/OpenCL.h>
10 #else
11 #include<CL/cl.h>
12 #endif
13
14 using namespace std;
15 /*
16  @Function: queryPlatform
17  Given a platform, retrievies information about the platform such has
        Name, Vendor
18  and Supported OpenCL Version.
19  */
20 void queryPlatform(cl_platform_id platform)
21 {
22     const int bufferSize=2048;
```

```
23      char buffer[bufferSize];  //stores information
24
25      //get platform info
26      clGetPlatformInfo(platform,CL_PLATFORM_NAME,bufferSize,buffer,NULL)
            ;
27      printf("Name:%s\n",buffer);
28      clGetPlatformInfo(platform,CL_PLATFORM_VENDOR,bufferSize,buffer,
            NULL);
29      printf("Vendor:%s\n",buffer);
30      clGetPlatformInfo(platform,CL_PLATFORM_PROFILE,bufferSize,buffer,
            NULL);
31      printf("Profile:%s\n",buffer);
32      clGetPlatformInfo(platform,CL_PLATFORM_VERSION,bufferSize,buffer,
            NULL);
33      printf("Version:%s\n",buffer);
34      //... more information can be retrieved
35
36  }
37  /*
38   @Function: queryDevice
39   Given a device, retrievies information about the device such as Name,
         Vendor,
40   Device Version, Driver Version, #Compute Units, Frequency,Global
         Memory Size, ...
41   */
42  void queryDevice(cl_device_id device_id)
43  {
44      const int bufferSize=2048;
45      char buffer[bufferSize];
46      cl_uint max_work_group_size=1;
47
48      cl_uint computeUnits;
49      cl_ulong globalMemSize;
50
51      clGetDeviceInfo(device_id,CL_DEVICE_NAME,bufferSize,buffer,NULL);
52      printf("\tDevice Name:%s\n",buffer);
53      clGetDeviceInfo(device_id,CL_DEVICE_VENDOR,bufferSize,buffer,NULL);
54      printf("\tDevice Vendor:%s\n",buffer);
55      clGetDeviceInfo(device_id,CL_DEVICE_VERSION,bufferSize,buffer,NULL)
            ;
56      printf("\tDevice Version:%s\n",buffer);
57      clGetDeviceInfo(device_id,CL_DRIVER_VERSION,bufferSize,buffer,NULL)
            ;
58      printf("\tDriver Name:%s\n",buffer);
59      clGetDeviceInfo(device_id,CL_DEVICE_MAX_COMPUTE_UNITS,sizeof(
            computeUnits),
60      &computeUnits,NULL);
61      printf("\tDevice Max Compute Units:%u\n",computeUnits);
62      clGetDeviceInfo(device_id,CL_DEVICE_MAX_CLOCK_FREQUENCY,sizeof(
            computeUnits),
63      &computeUnits,NULL);
64      printf("\tDevice Max Clock Frequency:%u MHz\n",computeUnits);
```

```
65      clGetDeviceInfo(device_id,CL_DEVICE_GLOBAL_MEM_SIZE,sizeof(
            globalMemSize),
66      &globalMemSize,NULL);
67      printf("\tDevice Global Mem Size:%f Mbytes\n",(float)globalMemSize
            /(1024*1024));
68
69      //... more information can be retrieved
70  }
71  /*
72   @Function: readProgram
73   Given a file name, this function reads the contents of the file in to
         memory
74   and returns a pointer to the memory location.
75   */
76  const char* readProgram(char *fileName)
77  {
78      char *source;
79      FILE *file;
80      file=fopen(fileName,"r"); //open the file in read mode
81      printf("Trying to open:%s \n",fileName);
82      //check if the file is open properly otherwise return NULL.
83      if(file!=NULL)
84      {
85          fseek(file,0L,SEEK_END);
86          long size=ftell(file);
87          rewind(file);
88          source=new char[size+1];
89          fread(source,1,size,file);
90          source[size]='\0';
91          fclose(file);
92      }
93      else
94          return NULL;
95      return source;
96  }
97  /*
98   @Function: main
99    This is the main function.
100  */
101 int main(int argc, const char * argv[]) {
102     //1. Initialize OpenCL Device
103     cl_int error;       // stores opencl error codes
104     cl_uint num_platforms=0,num_devices=0;
105     cl_platform_id platforms[10];
106     cl_device_id devices[10],device;
107     cl_context context; //to identify which context is associated with
            a deice
108         //holds list of kernels to be executed on the context on the
                device
109     cl_command_queue commandQueue;
110     int platformChoice=0,deviceChoice=0;
111
```

```
112     //retrieve opencl platforms in the system
113     error=clGetPlatformIDs(1,platforms,&num_platforms);
114     printf("%d platforms found.\n",num_platforms);
115     if(num_platforms!=0)
116     {
117         //print the OpenCL platforms found in the system
118         for(int i=0;i<num_platforms;i++){
119             printf("Platform %d: \n",i);
120             queryPlatform(platforms[i]);
121         }
122     }
123     else {
124         printf("OpenCL Platform not found\n");
125         printf("OpenCL is not supported by the system\n");
126         exit(1);
127     }
128     printf("Choose which platform to use (default 0): \n");
129     scanf("%d ",&platformChoice);
130 
131     //retrieve opencl devices found in the platform choice of the user
132     if(platformChoice>=0 && num_platforms!=0)
133         error=clGetDeviceIDs(platforms[platformChoice],
                CL_DEVICE_TYPE_ALL,10,devices,&num_devices);
134 
135     if(num_devices!=0){
136         for(int i=0;i<num_devices;i++)
137         {
138             printf("Device %d: \n",i);
139             queryDevice(devices[i]);
140         }
141     }
142     else {
143         printf("No OpenCL supporting devices found in selected Platform
                .\n");
144         exit(1);
145     }
146 
147     printf("Choose which device to use (default 0): \n");
148     scanf("%d",&deviceChoice);
149     if(deviceChoice>=0 &&deviceChoice<num_devices)
150         device=devices[deviceChoice];
151     else
152     {
153         printf("Invalid device choice! Exiting ...");
154         exit(1);
155     }
156 
157     //create a context on the chosen opencl device
158     context=clCreateContext(NULL,1,&device,NULL,NULL,&error);
159     //checkError
160 
161     //create a command queue on the selected device
```

```
162     commandQueue=clCreateCommandQueue(context,device,
            CL_QUEUE_PROFILING_ENABLE,
163     &error);
164     //checkError
165
166     //2. Building the OpenCL program
167     cl_program program;
168
169     //read the source
170     //( the source file should be in the same folder as the application
            )
171     const char* source=readProgram("matMul_kernel.cl");
172
173     //create a program - compile the opencl program
174     program=clCreateProgramWithSource(context,1,&source,NULL,&error);
175
176     //build the program
177     error=clBuildProgram(program,0,NULL,NULL,NULL,NULL);
178     //check if the program building failed
179     if(error!=CL_SUCCESS)
180     {
181         size_t len;
182         //measure the size of the build report
183         clGetProgramBuildInfo(program,device,CL_PROGRAM_BUILD_LOG,0,
                NULL,&len);
184         char* buffer=new char[len+1];
185         //retrieve the build report
186         error=clGetProgramBuildInfo(program,device,CL_PROGRAM_BUILD_LOG
                ,len,
187         buffer,&len);
188         buffer[len]='\0';
189         //print the build report to the screen
190         printf("%s\n","Program failed:");
191         printf("%s\n",buffer);
192         exit(1);
193     }
194     /*
195     3. create buffers and manage data
196        Data first needs to be prepared on the host side first.
197        Then buffers should be allocated on the OpenCL Device to hold
               the data.
198        Data needs to be copied from host memory to host memory.
199      */
200     //the 'h' in the variables indicates host side
201     //the sizes for the Matrices is randomly assigned
202     int hRowA=157,hRowB=115,hColA=115,hColB=204;
203     float *hA,*hB,*hC;
204     //allocate buffers on the host side
205     hA=(float*)malloc(hRowA*hColA*sizeof(float));
206     hB=(float*)malloc(hRowB*hColB*sizeof(float));
207     hC=(float*)malloc(hRowA*hColB*sizeof(float));
208     //fill matrix A and matrix B with random numbers
```

```
209     for(int i=0;i<hRowA;i++)
210         for(int j=0;j<hColA;j++)
211         {
212             hA[i*hColA+j]=1;//rand();
213         }
214
215     for(int i=0;i<hRowB;i++)
216         for(int j=0;j<hColB;j++)
217         {
218             hB[i*hColB+j]=1;//rand();
219         }
220
221     /* Allocate buffers on the OpenCL device and copy data
222      from the host device to the OpenCL Device */
223     cl_mem devA=clCreateBuffer(context,CL_MEM_READ_ONLY|
224           CL_MEM_COPY_HOST_PTR,hRowA*hColA*sizeof(float) , hA,NULL);
225     cl_mem devB=clCreateBuffer(context,CL_MEM_READ_ONLY|
226           CL_MEM_COPY_HOST_PTR,hRowB*hColB*sizeof(float) , hB,NULL);
227     cl_mem devC=clCreateBuffer(context,CL_MEM_WRITE_ONLY,
228                 hRowA*hColB*sizeof(float), NULL,NULL);
229
230     /* 4. Set arguments and Enqueue Kernels
231      */
232     cl_kernel kernel=clCreateKernel(program, "OpenCL_MatMul", &error);
233     error=clSetKernelArg(kernel, 0, sizeof(cl_mem), &devA);
234     error|=clSetKernelArg(kernel, 1, sizeof(cl_mem), &devB);
235     error|=clSetKernelArg(kernel, 2, sizeof(cl_mem), &devC);
236     error|=clSetKernelArg(kernel, 3, sizeof(int), &hRowA);
237     error|=clSetKernelArg(kernel, 4, sizeof(int), &hRowB);
238     error|=clSetKernelArg(kernel, 5, sizeof(int), &hColB);
239     if(error!=CL_SUCCESS)
240     {
241         printf("Failed to set arguments\n");
242         exit(1);
243     }
244
245     size_t global_ws[2];
246     size_t local_ws[2]={8,8};
247
248     //instatiate enough number of work-items to handle the computation
249     //add padding where it is needed
250     size_t temp = (hRowA/ 8)*8;
251     global_ws[0]= (hRowA - temp) == 0 ? hRowA : hRowA + 8 - (hRowA -
            temp);
252
253     temp = (hColB/ 8)*8;
254     global_ws[1]= (hColB - temp) == 0 ? hColB : hColB + 8 - (hColB -
            temp);
255
256     //execute the kernel on the OpenCL Device
257     error=clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL ,
258               global_ws,local_ws, 0, NULL, NULL);
```

```
259     clFinish(commandQueue);
260     if(error!=CL_SUCCESS)
261     {
262         printf("Failed to excute kernel on the device:%d\n",error);
263         exit(1);
264     }
265 
266     /* 5. Get results back from the OpenCL Device */
267     error=clEnqueueReadBuffer(commandQueue, devC, CL_TRUE, 0,
268           hRowA * hColB* sizeof(float), hC, 0, NULL,NULL);
269     clFinish(commandQueue);
270     if(error!=CL_SUCCESS)
271     {
272         printf("Failed to read data from the OpenCL device:%d\n",error)
                ;
273         exit(1);
274     }
275     //allocate buffers on the host side
276     //to store computed results on the host side
277     float *gC;
278     gC=(float*)malloc(hRowA*hColA*sizeof(float));
279     C_MatMul(hA, hB, gC, hRowA, hRowB, hColB);
280 
281     //compare the results from host side and the device side
282     printf("The matrix multiplication on the OpenCL Device and
283      the Host is %s\n",compareResults(hC,gC,hRowA,hColB)?
284      "identical.":" not identical.");
285 
286     return 0;
287 }
```

Listing A.2 gives the C implementation of the matrix multiplication in a function. Also, the a function that checks the similarity of two Matrices is given in the same listing. This comparison function is added to show that computing on the host side and the OpenCL device side will give similar if not identical results.

LISTING A.2: Matrix multiplication function in C

```
1  //  MatMul.c
2  //  MatrixMultiplication
3 
4  #include "MatMul.h"  //contains the prototypes of the functions in this
       file
5  #include <math.h>
6 
7  /*
8   @Function: C_MatMul
9   Accepts two Matrices (A and B) and stores their product on Matrix C.
10  Assumptions:
11  - The number of columns of A are equal to the number of rows of B.
12  - Matrix C is of size RowA x ColB
13  */
```

```
14 void C_MatMul(float* A,float* B, float* C,
15               int RowA,int ColRowAB,int ColB )
16 {
17     float sum=0.0;
18     for(int i=0;i<RowA;i++)
19         for(int j=0;j<ColRowAB;j++)
20         {
21             sum=0.0;
22             for(int k=0;k<ColB;k++)
23                 sum+=A[i*ColRowAB+k]*B[k*ColRowAB+j];
24             C[i*ColRowAB+j]=sum;
25         }
26 }
27
28 /*
29  @Function: compareResults
30  Accepts two matrices and compares if they are identical
31  to a certain degree of error.
32  */
33 bool compareResults(float* A,float *B, int row, int col)
34 {
35     for(int i=0; i<row; i++ )
36         for(int j=0;j<col;j++)
37         if(fabs(A[i*col+j]-B[i*col+j])>1e-5)
38         {
39             //show where the error occured
40             printf("(%d,%d) - (%f,%f)\n",i,j,A[i*col+j],B[i*col+j]);
41             return false;
42         }
43
44     return true;
45 }
```

The implementation of the matrix multiplication in an OpenCL kernel is given in Listing A.3.

LISTING A.3: Matrix multiplication OpenCL kernel

```
1  //  matMul_kernel.cl
2  //  MatrixMultiplication
3
4  /*
5   @kernel: OpenCL_MatMul
6   This kernel accepts two floating point Matrices A and B.
7   It computes and stores the product of Matrix A and
8   Matrix B in Matrix C.
9   */
10
11 __kernel void OpenCL_MatMul(global float* A,global float* B,
12                              global float* C,int RowA,int ColRowAB,int ColB)
13 {
14
```

```
15      int i = get_global_id(0);    //work-item id in dimension 1
16      int j = get_global_id(1);    //work-item id in dimension 2
17      float sum = 0.0;
18
19      if ((i < RowA) && (j < ColRowAB)) {
20          sum = 0.0;
21          for(int k = 0; k < ColB; k++)
22          {
23              sum += A[i*ColRowAB + k]*B[k*ColRowAB + j] ;
24          }
25          C[i*ColRowAB + j] = sum;
26      }
27  }
```

Appendix B

List of Source of Ringworm Images

All ringworm images used in this work are collected from sources on the internet. These images are used as positive samples for training and testing of a ringworm classifier as discussed in Chapter 5. Hence, the following table gives the sources of the images, how many images are taken form that source and a brief description of the sources website. Table B.1 gives the list of source websites for ringworm images used in the training. Table B.2 gives list of sources for images used for testing the ringworm detector. Note that the number of images from these sources does not match the number of ringworm images used during training. This is because some of the source images had more than one ringworm samples in them.

TABLE B.1: Ringworm Image Sources for Training

S/N	Number of Images	Website	Description
1	1	www.nhsdirect.wales.nhs.uk	NHS Direct Wales is a health advice and information service available 24 hours a day, every day
2	2	www.healthline.com	
3	1	www.babycentre.co.uk	BabyCentre provides support for parents at every stage of their child's development, from preconception to age five.
4	4	educationtoday.eu/a-ring-worm.html	
5	1	www.skincarearticles.com/	
6	1	www.babyrashhq.com	
7	2	www.rvrhs.com/	
8	1	www.medicalimages.com	
9	2	emedicine.medscape.com	

TABLE B.1: *Continued*

S/N	Number of Images	Website	Description
10	2	www.webmd.com	
11	1	ringwormtreatmentips.wordpress.com	
12	1	medical-dictionary.thefreedictionary.com	
13	1	mdhairmixtress.com	
14	1	Rosemary Shy - DOI: 10.1542/pir.28-5-164	Tinea Corporis and Tinea Capitis
15	1	Neerja Puri, Asha Puri - DOI: 10.7241/ourd.20132.36	A Study of Tinea Capitis on preschool and school going children
16	1	www.hairtx.com	Hair transplant center
17	1	www.researchgate.net	Tinea corporis over abdomen showing erythematous scaly lesions, annular, sharply marginated plaques with raised border and central clearing.
18	1	5minuteconsult.com	
19	1	www.infinitypath.com.au	Pathology Service
20	1	www.studyblue.com	Study flash card service
21	1	www.pharmacy.gov.my	Pharmaceutical Services Programme Ministry of Health Malaysia
22	1	www.globalskinatlas.com	The site provides useful to people across the globe looking for information and help with their skin problems irrespective of skin colour or race
23	1	africanskindiseases.org	An educational website designed to provide access to information and high quality dermatology images of common African skin diseases
24	1	www.cortesedermatology.com	
25	1	www.cram.com	Dermatology Flashcards
26	1	skintreat.net	Education
27	2	www.omicsonline.org	Dermatology Flashcards
28	1	www.aocd.org	tinajero.pdf

TABLE B.2: Ringworm Image Sources for Testing

S/N	**Number of Images**	**Website**	**Description**
1	2	patientslounge.com	A site created by patients, caregivers, and loved ones sharing knowledge on personal medical experiences.
2	1	www.news-medical.net	A tight-knit community of scientific, medical, and life sciences experts that produce and share the latest information, in a readable, understandable way.
3	1	dermatoweb.udl.es	
4	1	www.ringworm-treatment.net	
5	1	www.huffingtonpost.co.uk	
6	2	allskinrashes.blogspot.com	
7	1	www.babycentre.co.uk	
8	1	www.rytir.info	
9	1	www.kidskunst.info	