University of Southern Queensland

Faculty of Health, Engineering and Sciences

Optimisation of Multicore Processor and GPU for use in Embedded Systems

A dissertation submitted by

Chloe Mansell

In fulfilment of the requirements of

ENG4112 Research Project

towards the degree of

Bachelor of Computer Systems Engineering (Honours)

Submitted October, 2015

Abstract

The advancement in technology continues to consume an increasing part of our lives and as we watch the slowing of Moore's Law as Integrated Circuits approach physical limitations, we will continue to search for faster execution of programs.

The advancement in robotics and machine vision will see them become part of our daily lives and the need for real time machine vision algorithms will increase.

This dissertation will investigate optimisation options when executing machine vision algorithms on a multi-core processor and provide a guide for programmers to use when writing similar machine vision algorithms on Arm A7 or A15 processors containing a Mali T628 Graphics processing unit.

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

University of Southern Queensland

Faculty of Health, Engineering and Sciences

ENG4111/ENG4112 Research Project

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.

I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

C. Mansell

0061024275

Acknowledgments

Firstly I would like to extend my thanks and appreciation to my supervisor Stephen Rees who has always been available when needed and has offered valuable input and feedback.

I would also like to acknowledge the massive undertaking a university degree is and the large effects it has had on my family. I would like to say a huge thankyou to my immediate family especially to my supportive husband who has had to pick up the slack and endure many weekends and holidays without his wife. To my beautiful boys who are my inspiration and keep me going.

I would also like to acknowledge the assistance I have received from extended family and friends. I have not gotten this far on my own, from babysitting, cooking, washing, cleaning, staying away so I can study as well as a friendly ear to listen to me complain about the many assignments, exams and projects I needed to complete. This has all helped me get to this stage.

Contents

Abstract **i**

Acknowledgments **iv**

List of Figures

List of Tables

Chapter 1 Introduction **1**

1.1 Research Aim 2

1.2 Objectives 2

1.3 Overview of Dissertation 2

1.4 Background Information 4

1.4.1 Common Embedded Systems. 4

1.4.2 What is Optimisation 19

1.4.3 Why Optimise 19

Chapter 2 Literature Review **21**

2.1 Software Optimisation techniques 21

2.1.1 Software techniques 21

2.1.2 Compiler Optimisations 22

2.1.2.1 Function inlining. 22

2.1.2.2 Eliminating common sub-expressions. . . . 22

2.1.2.3 Loop unrolling 23

2.1.2.4 GCC optimisation options 24

2.1.3 Source code modifications 26

2.1.3.1 Loop termination 26

2.1.3.2 Loop Fusion 26

2.1.3.3 Variable Selection 27

2.1.3.4 Pointer aliasing 27

2.1.3.5 Division and modulo 27

2.1.4 Software Optimisation Conclusion 28

2.2 Optimising with Hardware 29

2.2.1 Neon Data Engine 29

2.2.2 Graphics Processing Unit 31

Chapter 3 Methodology 33

3.1 Research and Development methods 33

3.2 Task analysis 34

3.3 Risk Assessment 35

3.3.1 Risk Identification 35

3.3.2 Risk Evaluation 35

3.3.3 Risk Control 36

3.4 Resource Analysis 37

3.4.1 Research and reporting 37

3.4.2 Software development and Hardware testing 37

3.5 Project Timeline 39

Chapter 4 Software Development 40

4.1 Optimisation Techniques 41

4.2 Image Processing 42

4.3 Grey Scale Thresholding 41

4.4 Average Smoothing Filter 52

4.5 Testing 60

Chapter 5 Results and Discussion 61

5.1 Grey Scale Thresholding 61

5.11 Results 62

5.2 Average Smoothing Filter 71

5.2.1 Results 72

5.3 Conclusion 80

Chapter 6 Conclusion and Further Work 82

References 83

Appendix A Project Specification 87

Appendix B Code for Grey Scale Threshold Image Processing 89

Appendix C Average Smoothing Filter code for Image Processing 107

Appendix D Output photos from Grey Scale Algorithm 132

Appendix E Output photos for Average Smoothing Filter 137

List of Figures

1.1 Harvard architecture 5

1.2 Von Neumann architecture 5

1.3 Arm Cortex A7 7

1.4 Cortex A7 Pipeline 7

1.5 Arm Cortex A15 8

1.6 Cortex A15 Pipeline 8

1.7 Scalar add vs SIMD parallel add 9

1.8 Neon registers 10

1.9 Single Precision floating-point format 11

1.10 Circular buffer operation 13

1.11 Typical DSP architecture 14

1.12 FIR filter steps 15

1.13 FPGA 16

1.14 Comparison between CPU and GPU cores 17

1.15 Computer vision vs computer graphics 18

1.16 Arm Mali – T628 18

2.1 Example of loop 24

2.2 Example of loop unrolled 24

2.3 Example of unmerged and merged loops 27

3.1 ODROID XU3 Development Board 38

3.2 Project Timeline 39

4.1 (a) Input Image 43

4.2 Grey Scale threshold development code. 45

4.3 Loop reversal for greyscale threshold 46

4.4 Loop unrolling excerpt 47

4.5 For loop within grey scale threshold program 48

4.6 Block of code directing process through single core 48

4.7 Loop for Neon data engine 50

4.8 Convert greyscale image to RGB code 51

4.9 Smoothing filter algorithm 52

4.10 Original image used for image processing 53

4.11 Image after alterations from average smoothing filter 53

4.12 Average Smoothing Filter program 54

4.13 Loop for average smoothing filter with loop unrolling 56

4.14 Load and sum functions in average smoothing filter 58

4.15 Copy of program section which calculates execution time 60

5.1 Image prior to image processing 61

5.2 Output image from Grey Scale thresholding 62

5.3 Grey Scale Thresholding Graph 69

5.4 Grey Scale Thresholding Graph without GPU 70

5.5 Output image from Average Smoothing Filter 71

5.6 Average Smoothing Filter Graph 78

5.7 Average smoothing filter graph without GPU 79

List of Tables

5.1 Results from Grey Scale Thresholding Program 62

5.2 Results from Grey Scale Thresholding with loop reversal 63

5.3 Results from Grey Scale Thresholding with loop unrolling 64

5.4 Results from Grey Scale Thresholding with pointer Optimisation. 64

5.5 Results from Grey Scale Thresholding pointer and loop unrolling 65

5.6 Results from Grey Scale Thresholding Pointer and loop reversal.. 66

5.7 Results from Grey Scale Thresholding directed through A7 Core. 66

5.8 Results from Grey Scale Thresholding directed through A15 core..67

5.9 Results from Grey Scale Thresholding directed through Neon . . . 68

5.10 Results from Grey Scale Thresholding directed through GPU . . . 69

5.11 Results from Average Smoothing Filter 72

5.12 Results from Average Smoothing Filter with loop reversal 72

5.13 Results from average smoothing filter with loop unrolling 73

5.14 Results from average smoothing filter with pointer optimisation . 74

5.15 Results from average smoothing filter with pointer and reversal. . 74

5.16 Results from average smothing filter with pointer and unrolling. . .75

5.17 Results from average smoothing filter directed through A7. 76

5.18 Results from average smoothing filter directed through A15 76

5.19 Results from average smoothing filter through Neon 77

5.20 Results from average smoothing filter through GPU 78

Abbreviations

DSP	Digital Signal Processor
FPU	Floating Point Unit
SIMD	Single instruction stream multiple data stream
GPU	Graphics processing unit
RISC	Reduced instruction set computer
CISC	Complex instruction set computer
CPU	Central Processing Unit

Chapter 1

Introduction

The advancement of technology has been continuing at a rate predicted by Moore's Law half a century ago. Continued advancements including the decrease in the size of integrated circuit components allowing more components on a single chip has decreased the cost of processors and greatly improved performance including clock speeds and power efficiency.

We are now seeing the slowing of Moore's Law as advancement's in Integrated Circuits approach physical limitations. This slowing has contributed in the birth of multicore processors and parallel processing to allow for continued increase in processing power. As the cost of multicore processors and development boards become lower this will allow multicore processors to become more of a viable option for use in embedded systems.

This dissertation will investigate the use of multicore processors and Graphics Processing Unit to execute optimised embedded system algorithms.

.

1.1 Research Aim

The aim of this project is to investigate how to optimise a multicore processor and its ancillary computational hardware, for use in an embedded system. The project will determine if there is an advantage in using a range of optimisation software techniques combined with manually directing data through the Central Processing Unit, Neon unit and the Graphics Processing unit, in comparison to allowing the compiler to schedule the optimise code and direct the data.

1.2 Objectives

1. To develop a software program which will benchmark computational speeds of different components on the ODROID XU3 development board.
2. Test the development board with software programs containing typical machine vision algorithms and record execution speeds.
3. Analyse and benchmark the data obtained.

1.3 Overview of Dissertation

The dissertation has been broken up into the following chapters.

Chapter 1: Introduction

This chapter will outline the research aim and objectives as well as discuss background information required to understand current processes used in embedded systems.

Chapter 2: Literature Review

The literature review will review current literature on software optimisation techniques which will assist in the decision on what techniques will be used on the software code for the purpose of this dissertation. The review will also investigate the current literature covering the use of hardware components including the Neon Data engine and Graphics processing unit to further improve on execution speeds of the software program.

Chapter 3: Methodology

Chapter three contains research and development methods as well as a task analysis outlining milestones for the project. This is followed by a Risk assessment, Resource analysis and Project timeline.

Chapter 4: Software Development

Chapter four will outline the development of each software program and how the algorithms were approached.

Chapter 5: Results and Discussion

Chapter five will show the results from the execution of the software and analysis of the results.

Chapter 6: Conclusion and further work

Chapter six will discuss the achievements of the project’s objectives and outline any future work in regards to this dissertation

1.4 Background Information

This section will cover the background information required to obtain an in-depth understanding of current processors used in embedded systems. Optimisation and why optimisation is required will also be examined.

1.4.1 Common embedded systems

Microcontroller

A microcontroller can be described as a computer on a single chip containing simplified elements which are able to perform less complex applications. The microcontroller is a smaller more economical options which has been typically used in the use of embedded systems (Bates 2011, p.17). A microcontroller will contain a central processing unit, memory and access to input data and the ability to output data. The following background information will cover a range of processing units available to be used to perform the tasks required in a real time system.

Central Processing Unit (CPU)

To understand how different CPU's works an understanding of the typical architectures is required. There are two major types of physical architectures available and the first architecture to be examined is the Harvard architecture. The Harvard architecture consists of two separate storages, one for the data memory and one for the instruction memory. These storage spaces are connected to the main control unit through a bus, and as there is two separate

storage spaces and two buses this allows simultaneous transfer of instructions and data as well as allowing the instruction and data sizes to be different (USQ, 2013).

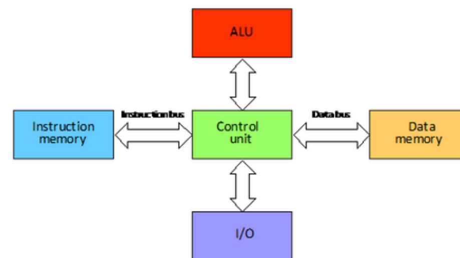


Figure 1.1 Harvard architecture (USQ 2013, p.137)

The second major architecture available is the Von Neumann architecture which has one place for storage hence the data and instructions are stored within the same memory. As there is only one storage unit available there is also only one bus for the information to move from memory to the control unit not allowing for simultaneous transfer of instructions and data. Movement between these units will consequently need to be scheduled and the size of the data and instruction will need to be the same length (USQ, 2013).

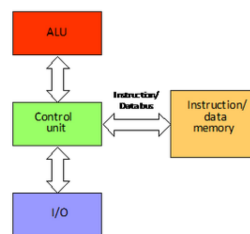


Figure 1.2 Von Neumann architecture (USQ 2013, p.136)

There are other variations on these architectures available, one of these being the modified Harvard architecture. This architecture takes the best of both of the architectures and combines them for a more enhanced design. The modified Harvard architecture consists of a Von Nuemann architecture with

two on board memory caches added. There is a cache available for instruction memory and one available for data memory and each has a common address space that share the one memory. The modified Harvard architecture allows the processor to perform similar to a Von Neumann when accessing memory and similar to a Harvard when accessing the caches.

Other terms which are common when discussing different forms of CPU's are the type of instructions sets which are used when programming. The two distinct types are CISC architecture which stands for a complex instruction set computer and RISC which stands for a reduced instruction set computer.

A CISC is a complex set of instructions and therefore there are more instruction available to use then the RISC set, this allows the programmer to write less code to perform the same task when compared to a RISC (USQ, 2013). This however is not always appropriate in an embedded system as more transistors are required on the chip and there is also a higher power consumption and heat dissipation when using this form of architecture. Therefore a RISC architecture would be a more appropriate choice for microcontrollers when used in embedded or mobile applications. This is seen in the ARM architectures which is a popular set of RISC microcontrollers which can be found in 95% of smart phones, 80% of digital cameras and 35% of all electronic devices (ARM, 2014).

The Exynos 5422 the processor on the ODROID XU3 development board contains a Cortex A7 quad core alongside the Cortex A15 quad core

A7 – CPU

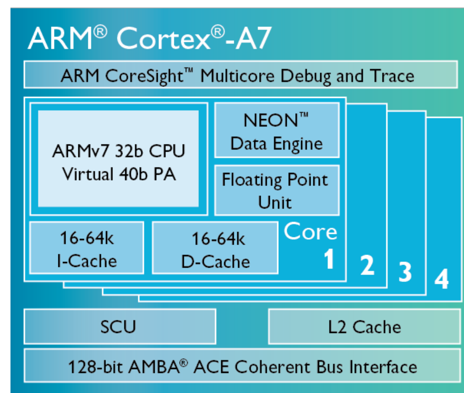


Figure 1.3 Arm Cortex A7 (ARM 2015)

The Cortex A7 has an in order, non-symmetric dual issue processor with a pipeline length between 8 to 10 stages (ARM 2015). The chip contains hardware for SIMD in the Neon data engine, a floating point unit and two on board cache.

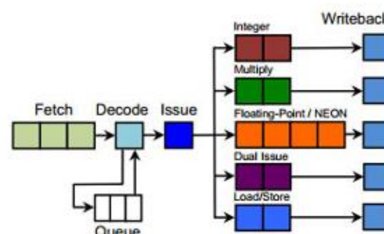


Figure 1.4 Cortex A7 Pipeline (Electronic Design 2011).

The A7 processor main advantage is its power efficiency. It is used in conjunction with the A15 to run smaller less intensive applications whilst the A15 sits idle until more powerful processing is required.

A15 – CPU

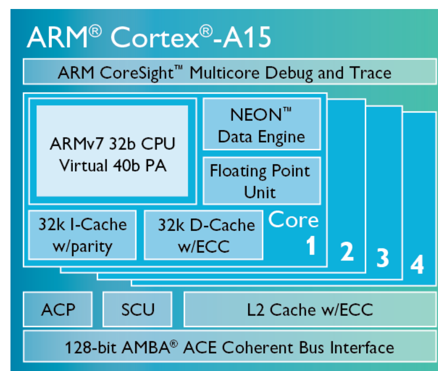


Figure 1.5 ARM Cortex A15 (ARM 2015)

The Cortex A-15 is a high end triple issue, out of order processor core. The A15 has the ability to implement virtualisation instructions, hardware-accelerated integer division and also a 40 bit virtual memory addressing extensions (ARM 2015). The A15 also contains a SIMD unit in the Neon data engine and floating point unit as well as two on board cache.

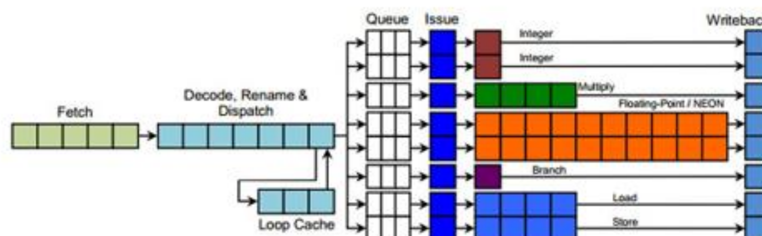


Figure 1.6 Cortex A15 Pipeline (Electronic Design 2011).

Comparing the two processors the assumption is that the A15 core will perform the test calculations at a faster rate than the A7 core.

SIMD – Single Input Multiple Data

SIMD or Single Instruction Stream Multiple Data Stream allows the same operation to be performed on Multiple Data at the same time (Zhou & Shi 2008).

SIMD will allow a single instruction to split a register into multiple data elements which will allow multiple identical operations on these elements. The following figure 2.1 illustrates the difference between a scalar add and a SIMD parallel add (ARM 2013).

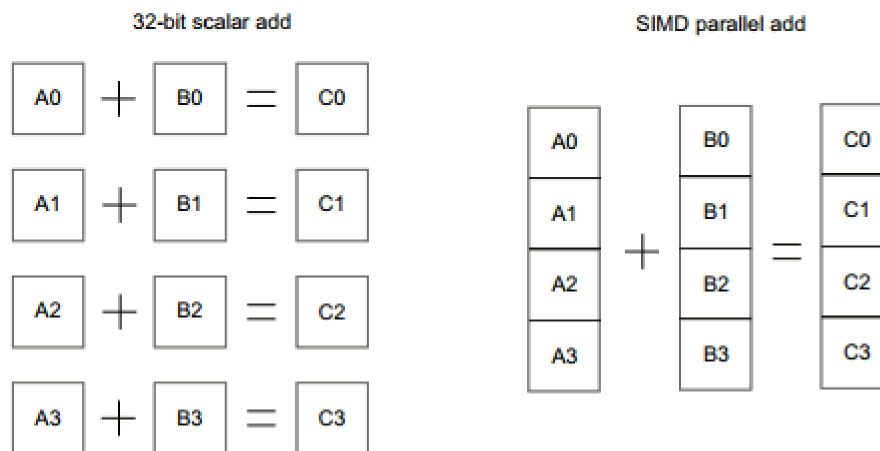


Figure 1.7 Scalar add vs SIMD parallel add (ARM 2013).

SIMD operations are available to use on a variety of CPU's through specialised instructions or hardware units. The Arm architecture has a Neon Data engine to carry out vectored operations. The Neon is able to accelerate repetitive operations on large data sets which is an advantage when used in Digital filters, Pixel processing and Matrix operations (ARM 2013).

Neon uses this concept outlined in figure 2.1 using a 64 bit and 128 bit instruction set. This instruction set will provide a 128 bit wide vector

operations. The registers can either be made up of 16 x 128 bit or 32 x 64 bit registers (ARM 2013).

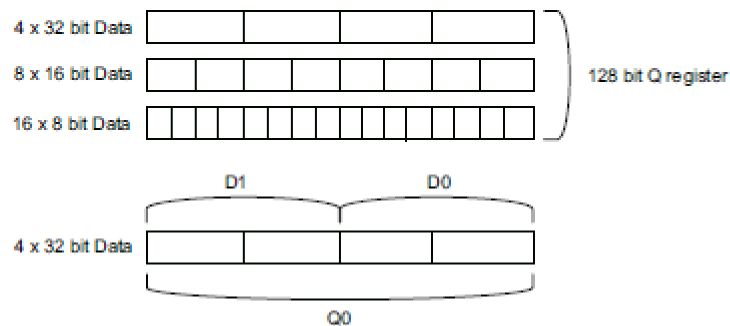


Figure 1.8 Neon registers (ARM 2015)

NEON has the following data types available:

- Unsigned integer U8 U16 U32 U64.
- Signed integer S8 S16 S32 S64
- Integer of unspecified type I8 I16 I32 I64
- Floating-point number F16 F32
- Polynomial over {0,1} PS

(ARM 2013)

Processing data in NEON can be done in either Normal, Long, Wide, Narrow and saturating variants. Processing may also be done with scalars and vectors where the scalar may be 8,16,32 or 64 bit (ARM 2013).

Floating Point Unit (FPU)

Each core in both the Arm A7 and A15 processors contain a Floating Point Unit which can be used when it has been enabled otherwise any Floating point calculations will be done through the use of library functions. (ARM 2013).

As per the IEEE-754 standard floating point numbers are represented within the Arm Cortex A series hardware as follows:

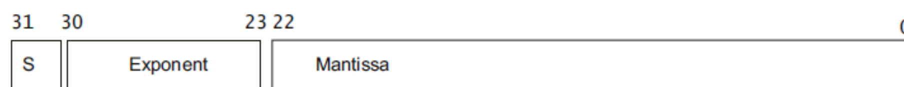


Figure 1.9 Single Precision floating-point format (ARM 2013)

S = Sign Bit which indicates if the number is positive or negative

Exponent gives the order of the magnitude of the number and the Mantissa is the fractional binary digits of the number. If the number is a single precision float the number is stored as per figure 2.10. The conversion to a single 32 bit float may cause a loss of precision if the number being stored cannot be represented wholly within the 23 bit mantissa. In this case the use of a double-precision floating point number may be appropriated as it has an exponent field with 11 bits and a mantissa with 52 bits (ARM 2013).

Both the Cortex A7 and A15 have the VFPv4 Floating point micro-architecture and the following registers:

- Thirty two or sixteen double-word registers (ARM 2013).
- Floating point system ID register (FPSID) which is used to allow the system to read and to determine which features are available within the hardware (ARM 2013)

- Floating point status and control registers (FPSCR) which are used to hold comparison results, flags for exceptions, select rounding options and enable floating point exception trapping (ARM 2013).
- Floating point Exception register (FPEXC) which is used to enable system software which controls exceptions which determine what has happened (ARM 2013).
- Media and VFP feature register 0 and 1 (MVFR0 and MVFR1) which enable software which determines what features from Floating point or SIMD are implemented on the processor (ARM 2013).

Digital Signal Processor (DSP)

A Digital signal processor or DSP is a programmable microcontroller which has been designed to manipulate a stream of real time digital data usually in the form of a signal. A DSP is often used in processing audio, video and graphics processing (Thompson, 2001).

A DSP is designed to perform data manipulation and mathematical calculations at a rate fast enough to allow usage in a real time system and will have the following characteristics.

- Specialised high speed arithmetic
- A form of data transfer from and to the real world
- A memory architecture which will allow multiple access

(Smith, 1997)

A DSP will often be used in digital signals to apply a digital filter to a signal to allow the signal to be free from distortion, interference or it may be used to separate two signals. This process involves taking samples of a signal, performing arithmetic to this sample and outputting the modified signal all in real time, therefore the DSP needs to perform large calculations fast. The signal will be represented by an equation for example the input will be x(n) and the output will be equal to y(n). The following is an example of an equation to find the output signal from a DSP with a FIR digital filter applied.

$$y(n) = a_0 x(\mathrm{n}) + a_1 x(n-1) + a_2 x(n-2) + \ldots\ldots\ldots \tag{1.1}$$

The number of coefficients in this equation could be a number in the thousands and the DSP may be sampling a large amount of samples of the signal every second. To enable the DSP to do this it uses a circular buffer where it will store each coefficient for example x(n-1) where n-1 represents a past input separately within the buffer. As the time moves forward only one coefficient needs to be added and therefore only one value needs to be updated and not every coefficient within the equation (Smith, 1997).

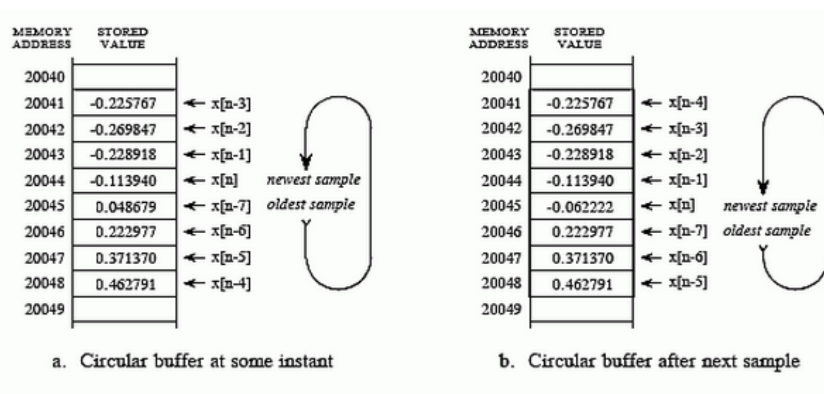


Figure 1.10 Circular buffer operation (Smith 1997)

In Figure 1.1 smith shows an example of how the circular buffer is used and how only one value x(n) is added which replaces the oldest value x(n-7). This allows the DSP to perform these calculations in real time and allows there to be no obvious delay between the readings of the information from the input signal to the output of the filtered signal (Smith 1997).

A DSP has a modified Harvard architecture which contains two data busses, one for the instructions and one for data. The DSP also has an instruction cache which will allow the storage of all the recent program instructions as seen in Figure 1.2. As the DSP performs the same repetitive instructions when sampling data, the DSP will retrieve the instructions from the cache after the first loop. Once the DSP is receiving instructions from the on board cache the memory transfer can be achieved within a single loop.

The DSP is now capable of receiving the sample from the signal from the data memory bus, the coefficients from the program memory bus and the instructions from the cache in parallel (Smith 1997).

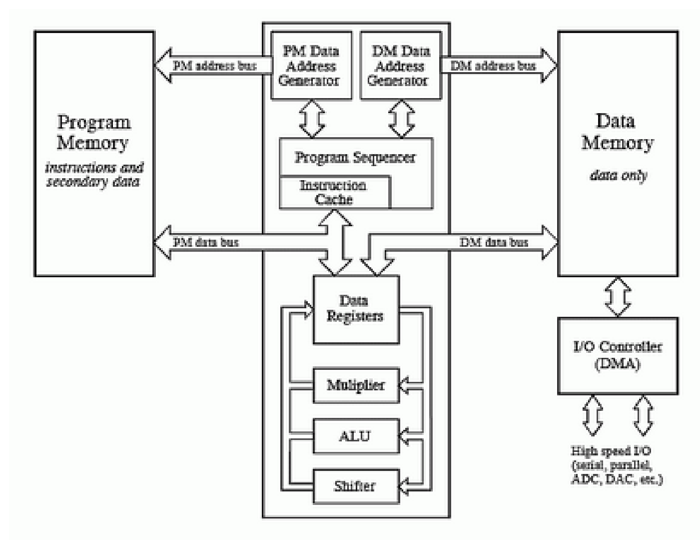
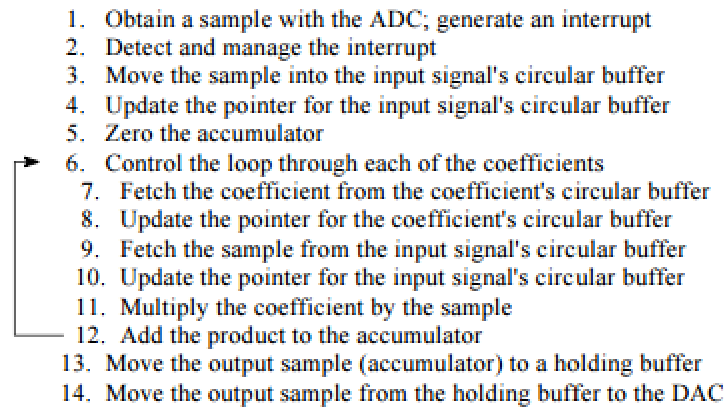


Figure 1.11 Typical DSP architecture (Smith 1997)

A DSP will also allow access to the Multiplier, ALU and shifter to be accessed in parallel. (Smith 1997). In the following figure 1.3 Smith outlines the following steps in a DSP cycle when applying an FIR filter.



1. Obtain a sample with the ADC; generate an interrupt
2. Detect and manage the interrupt
3. Move the sample into the input signal's circular buffer
4. Update the pointer for the input signal's circular buffer
5. Zero the accumulator
6. Control the loop through each of the coefficients
7. Fetch the coefficient from the coefficient's circular buffer
8. Update the pointer for the coefficient's circular buffer
9. Fetch the sample from the input signal's circular buffer
10. Update the pointer for the input signal's circular buffer
11. Multiply the coefficient by the sample
12. Add the product to the accumulator
13. Move the output sample (accumulator) to a holding buffer
14. Move the output sample from the holding buffer to the DAC

Figure 1.12 FIR filter steps (Smith 1997)

Once the beginning steps 1 – 5 are carried out Steps 6-14 may be carried out simultaneously within a single clock cycle which will allow a calculation for 100 coefficients to be performed in approximately 105-110 clock cycles.

Programmable Logic Devices

A logic device is a circuit which will accept either a logic 1 or logic 0 or a combination and return an output of either a 1 or a 0 (USQ 2013). A programmable logic device is a block of logic devices which are organised in a way to perform a particular task. They can be known as a Complex programmable logic device (CPLD), Field programmable gate array (FPGA) as well as many others.

A complex programmable logic device is a device with contains re-programmable functional blocks. These blocks are all connected via inputs

and outputs through a global interconnection matrix. This matrix can be changed as necessary to allow different connections between blocks. There is the ability to connect input and output to these devices. The blocks can be programmed to perform logic functions such as OR, AND, NAND, NOR etc.

A field programmable gate array (FPGA) is a controller which consists of an array of logic blocks surrounded by input and output blocks. The logic blocks in an FPGA will implement the logic whilst programmable interconnect wires connect the inputs and outputs to the logic blocks as seen in Figure 1.3. Combined with the ability to be reprogrammed an FPGA can be very flexible and is often used in systems such as software-defined radio, aerospace and defence systems as well as medical imaging, computer vision and speech recognition (National Instruments 2014).

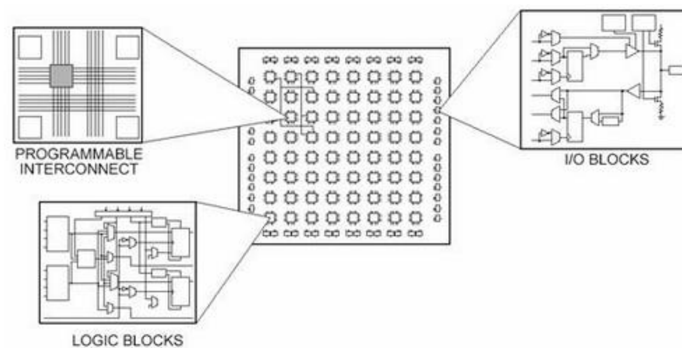


Figure 1.13 FPGA (National Instruments 2014)

GPU's – Graphics processing unit

A graphics processing unit is a multi-processor which has been designed for use in graphics processing. As a GPU is designed to process graphics which can include the process of thousands of pixels at one time they have a parallel architecture with thousands of small cores which are designed to all work simultaneously. This massive parallel processing is now being identified to be useful in applications other than graphics processing including scientific computation (NVidia 2015).

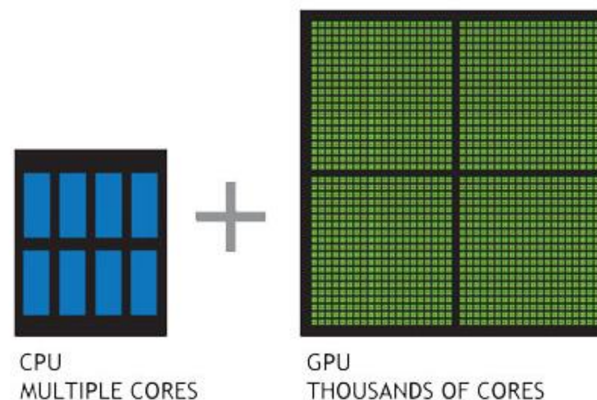


Figure 1.14 Comparison between CPU and GPU cores (NVidia 2015)

The use of the Graphics Processing unit has become useful for computer vision algorithms in particular as these algorithms are the inverse of the GPU's primary use (Pulli et al. 2012). For example the GPU was designed to transfer data of an object into pixels and output it onto the screen and a typical computer vision algorithm will take the pixel and transform it into data to be manipulated (Pulli et al. 2012). Figure 1.15 shows the relationship between the two processes.

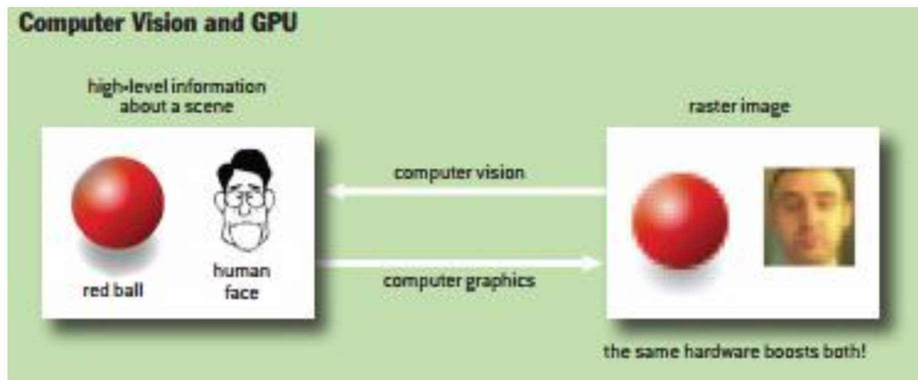


Figure 1.15 Computer vision vs computer graphics (Pulli et al. 2012)

The ability to partially program the GPU was created by the addition of shaders which allows data to be shared with the CPU and not sent directly to the display via a fixed-function pipeline (Pulli et al. 2012).

Writing parallel programs to utilise the GPU can be extremely difficult and complex which has seen the development of specific languages such as OpenCL, OpenGL and CUDA to assist programmers.

The GPU on the ODROID XU3 is the Arm Mali –T628 as shown in figure 1.15.

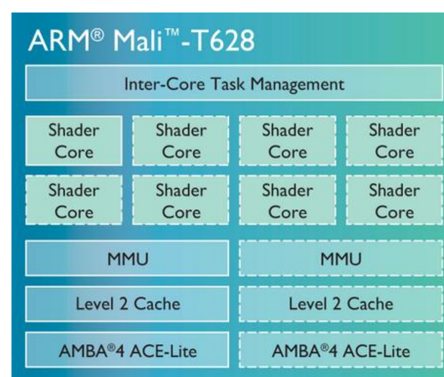


Figure 1.16. Arm Mali –T628 (ARM 2015)

The Mali has 8 shader cores enabling them to be partially programmed.

1.4.2 What is optimisation?

Optimisation is when a process or task is designed to be the most efficient it can be. In computing terms this can be done through various techniques either by optimising the software code or through the use of various hardware units depending on the task at hand. For example a Digital signal processor would be used to apply a filter to a signal as this is the most efficient hardware available for this task.

Software optimisation is optimising the process through programming techniques such as; the use of registers, the removal of any dead code, the use of pointers and unrolling of loops to name a few. Appropriate software techniques will be looked at in more detail in the next chapter. In an embedded system optimisation can be used to increase the speed of the execution of code, increase the performance of battery life, code density or reducing the memory footprint of the code (Arm 2013).

1.4.3 Why optimise

Optimisation is important when in terms of embedded systems as it is typically required to operate in a real time environment, therefore the process will need to seem instantaneous for it to be affective.

As we see the slowing of moore's law as the amount of transistors which can be put on a single chip is slowing due to physical limitations we can no longer expect a continued large increase in processing power with the next generation of chip being developed. This slowing has led to other options being exploited to increase computing power such as parallel computing, multi core processors, optimisation of code to increase execution speeds and

the use of specialised units such as the Neon data engine, Floating point units and Graphics processing units in combination with the CPU.

Chapter 2

Literature Review

In an embedded system optimisation is typically used to reduce the execution time of a program, reduce battery life, code density or the memory footprint (Arm 2013). For the purpose of this dissertation optimisation will be used to increase the speed of the code being executed. This section will cover possible optimisation techniques available to be used in conjunction with the ARM A series processors A7 and A15 as well as the ARM Mali T-628 GPU, this will also include the use of the NEON and FPU units included on the A7 and A15 processors.

2.1 Software Optimisation techniques

2.1.1 Profiling

The technique known as profiling is typically used to help identify parts of code which are consuming large amounts of execution time and is used in embedded applications to assist with the optimisation of the code. Cao et al. (2010) study on Loop-Centric Profiling method for embedded applications highlights that code written for an embedded application will typically follow the 90-10 rule which states that 10 percent of the code in an embedded application will account for 90 percent of the execution time. Profiling is used to identify the 10 percent of code to allow the programmer to optimise the section of code highlighted and therefore reduce the execution time (Cao et al. 2010). As each code that will be developed for the use in this

dissertation will be limited to a single set of calculations representing the 10 percent of code, profiling will therefore not be necessary.

2.1.2 Compiler Optimisations

GCC-4.8 will be the compiler which will be used to compile code on the development board. The Arm A-series Programmers guide outlines the following compiler optimizations to be used with the GCC Compiler to assist with optimisation of code to be run on an ARM A series Processor.

2.1.2.1 Function inlining

Function inlining is a technique that can be performed by the GCC compiler using a specific keyword inline. Inlining is the process of creating a copy of the function code and placing it where the function is being called within the main program. Function inlining eliminates the overhead created when calling a function and is useful in applications with small functions which are going to be called a large amount of times (ARM 2013). Mahalingam & Asokan (2012) study on Optimising GCC for ARM architecture highlights function inlining as being non advantageous with the ARM architecture due to interruptions to the pipeline and therefore we would not expect to see any improvement in execution speed using function inlining through the compiler.

2.1.2.2 Eliminating common sub-expressions

Eliminating common sub-expressions is the process where the compiler uses an already computed result in a later expression instead of re calculating the value again, however the compiler may not pick up on all cases and this

should be done manually if possible by the programmer (ARM 2013). Eliminating common sub-expressions was used as part of software optimisation in a study conducted by Park et al (2013) Software Optimisation for Embedded Communication systems alongside other optimisation techniques and an improvement in code execution time was found. Although eliminating common sub-expressions was not tested independently, the literature does suggest that an improvement was found and therefore this will be a technique used to optimise code for this dissertation.

2.1.2.3 Loop unrolling

Loop unrolling is a method which increases the programs performance by increasing pipeline efficiency and decreasing branching penalties and pointer arithmetic's associated when using loops (Velkoski et al. 2014). Velkoski et al (2014) study on the performance impact analysis of loop unrolling results found using loop unrolling techniques on a matrix multiplication algorithm had improved performances of the execution speed however this improvement varied depending on the size of the matrix and also on the architecture of the CPU. Park et al study on software optimisation for embedded communication system also found through the use of loop optimisations they were able to improve execution time of code with an improvement between 11.9 and 79.2 percent depending on the message type they were optimizing.

The following code is an example of loop unrolling.

Prior to loop unrolled

```c
for (j = 0; j < 8; j++)
{

 answer[j] = data[j] * three[j];

}
```

Figure 2.1 Example of loop written in C

Loop unrolled

```c
int answer[8];

answer[1] = data[1] * three[1];
answer[2] = data[2] * three[2];
answer[3] = data[3] * three[3];
answer[4] = data[4] * three[4];
answer[5] = data[5] * three[5];
answer[6] = data[6] * three[6];
answer[7] = data[7] * three[7];
answer[8] = data[8] * three[8];
```

Figure 2.2 Example of loop unrolled written in C

As the literature supports the conclusion that loop unrolling will be advantageous this technique will be used for the purpose of this dissertation when optimising code.

2.1.2.4 GCC optimisation options

There are various GCC optimisation levels which can be chosen to increase the performance of code when compiling (Arm 2013). Optimisation levels available are as follows:

- 00 – No optimisation
- 01 – uses common optimisation techniques to reduces the size of the program whilst increasing the performance

- 02 – Additional optimisation still ensuring speed without increase in size of program.
- 03 – optimisations which may increase the speed of the program and increase the size of the program
    - Can add to this level –ftree-vectorize which will attempt to generate NEON code
- –funroll-loops – will enable loop unrolling
- -0s. This will minimise the size of the program which may cause decrease in speed

(ARM 2013)

Available also are armcc compiler optimisation options, however during testing the gcc compiler will be used and therefore these optimisations will therefore not be relevant.

Park et al (2013) study on software optimisation had a justification that compilers typically focused on high performance computers and not on embedded systems and therefore the best way to optimise software code for embedded systems is to manually transform code. Simunic et al (2000) study on source code optimisation and profiling of energy consumption in embedded systems also states that manual code rewriting is more efficient than using compiler optimisations as they tend to be for higher level usage. Therefore it would be expected that a better result would be achieved through manually altering code alongside the use of compiler optimisation levels. As execution speed is the focus of this dissertation compiler options 01, 02 and 03 will be tested.

2.1.3 Source code modifications

The ARM Cortex –A Series Programmer's Guide recommends the following source code modifications to optimize code to run on an ARM A series processor.

2.1.3.1 Loop termination

Loop termination is the process of finishing loops within a program at zero and involves decrementing loops opposed to incrementing the loops after starting at a value of zero (ARM 2013). Joshi and Gurumurthy (2014) study analysing and improving the performance of software code for real time embedded systems found a reduction in execution time using a process known as Loop Reversal which is the same process known as loop termination. Joshi and Gurmurthy concluded that they had a 30% improvement in the speed of execution of code using this technique.

2.1.3.2 Loop Fusion

Loop Fusion is the technique a programmer will use to optimise code which will merge loops together and result in a single loop over multiple loops. This is typically used when loops have the same count as per figure 2.3 (ARM 2013). Joshi and Gurnmurthy tested loop fusion as part of their study and found an increase of 60% in execution speed.

```
for (i = 0; i < 10; i++)
{
  x[i] = 1;
}
for (j = 0; j < 10; j++)
{
  y[j] = j;
}
```

```
for (i = 0; i < 10; i++)
{
  x[i] = 1;
  y[i] = i;
```

Figure 2.3 Example of unmerged and merged loops (ARM 2013).

2.1.3.3 Variable Selection

As the arm registers are 32 bit, variables should be 32 bit in size. This will prevent overflows if the variable are 8 or 16 bit in size which can slow the execution of the code (Arm 2013). Variable selection will be a continued consideration during the software development.

2.1.3.4 Pointer aliasing

Will become an issue if more than one pointer is used within the code and both pointers point to the same memory location. As with variable selection this will also be a constant consideration during software development

2.1.3.5 Division and modulo

If the hardware being used does not have support for division it can slow the code down as library functions will need to be used to divide and use the modulo function. As both the A7 and A15 have hardware available to divide this will not be a consideration when optimising code (ARM 2013).

Software Optimisation Conclusion

Simunic et al (2000) study used techniques in critical loops to optimise code such as loop merging, loop unrolling, software pipelining and loop invariant extraction and found an 87 percent increase in performance.

Joshi and Gurumurthy (2014) study analysing and improving the performance of software code for real time embedded systems found a reduction in execution time with the following techniques. Loop Reversal also knows as loop termination had a 30 percent increase in speed, Loop fusion had a 60 percent increase in speed and Loop unswitching also had a 60 percent increase in speed.

Velkoski et al (2014) study on the performance impact analysis of loop unrolling found using loop unrolling techniques could have a performance increase on execution speeds.

After review of the current literature available the Software optimisation techniques which will be used for the purpose of this dissertation will be as follows:

- Loop unrolling
- Loop reversal

Alongside the above three techniques, elimination of common subexpressions, variable selection and pointer aliasing will be constant considerations during the development of software although they will not be independently tested for this dissertation.

It has also been decided to test the difference in execution times when pointers are allocated to image arrays to see if a decrease in execution time will be seen.

2.2 Optimising with Hardware

This section will review the capabilities of the units on the ODROID XU3 and the ability to manually use these units to optimise the calculation time of the algorithm.

To help the compiler compile optimal code there are available identifiers to tell the compiler which platform is being targeted.

- –march=<arch> - where arch is the architecture wanting to compile for
- –mtune=<cpu> - tunes the code for specific cpu
- –mfpu=<fpu> - targets specific hardware for example the floating point unit or the NEON unit. (ARM 2013)

2.2.1 Neon Data Engine

The Neon data engine is a Single input multiple data stream unit which allows multiple data to be performed on simultaneously. The neon unit allows for vectored operation which has been shown to accelerate repetitive operations as previously discussed in chapter 1.4 Background information under the heading of SIMD.

When writing software for the NEON unit it has been found to be efficient when written in Assembly although this has been known to be hard to use and also bug prone (Jo et al. 2014). Another option to access the Neon unit is through the use of auto vectorisation which is implemented through the GCC compiler. The last option available is through instrinisc function which when called are replaced with a NEON instruction (Jo et al. 2014).

Jang et al (2011) study on the performance analysis of Arm Neon technology for mobile platforms found an increase in execution speeds when the Neon unit was used in the processing, however as this study was based on the use of auto vectorisation they found that the use of the Neon unit was not always used.

Mitra et al. (2013) study on the Use of SIMD Vector operations to accelerate application code performance on lower-powered ARM and intel platforms found using neon intrinsic with hand written code with intrinsic functions was between 1.05x and 13.88x faster than that of the auto vectorization through the GCC compiler.

The literature shows an increased improvement in execution speeds when the Neon unit is used with a big advantage being through the use of intrinsic functions over the auto vectorisation option. The software code written for the Neon unit will therefore be written with the use of intriniscs as this appears to be a more efficient process when accessing the Neon data unit.

The NEON unit can be specified when compiling code using GCC with the following instructions –mfpu=neon-vfpv4.

2.2.2 Graphics Processing Unit (GPU)

Grasso et al. (2014) study on energy efficient HPC on Embedded SoC's: Optimization techniques for Mali GPU found an increase in speed by 8.7x over the cortex-A15 on the Arm Mali-T604 GPU. These increased speeds where found by optimising code for the Arm Mali architecture using OpenCl. The following techniques where used:

Memory allocation and mapping – unlike typical GPU – CPU combinations on a desktop system, the Mali GPU has a memory system which is unified with the CPU and therefore copying operations are not required and the GPU cannot access memory buffers created with the malloc function. Grasso et al therefore suggest that buffers be allocated using the clCreateBuffer function with the CL_MEM_ALLOC_HOST_PTR flag and the clEnqueueMapBuffer and clEnqueueUnmanpMemObject. This will enable both the application processor and the Mali GPU to access the data (Grasso et al. 2014).

Load distributions – Grasso et al. (2014) recommend to manually tune the local_work_size parameter after they noticed performance degradation.

Yi et al. (2014) study on real-time integrated face detection and recognition on Embedded GPGPUs has a look at optimization techniques for the local binary pattern integrated face detection and recognition algorithms. They were able to achieve increased execution speeds 2.9 times using the Mali T604 GPU in comparison to using the CPU.

The current literature suggests that an increase in execution speeds should be able to be achieved in comparison to the CPU as long as certain optimisations and architecture considerations are taken into consideration.

The Arm mali- T628 has Open GL ES 3.0/2.0/1.1 and OpenCL 1.1 Full profile available for programming, in the current literature OpenCL and CUDA are the libraries which have been popular choices for programing GPU's and therefore for the purpose of this dissertation the Mali GPU will be programmed using the OpenCL 1.1.

Chapter 3

Methodology

This chapter will cover the Research and development methods required to successfully complete the development of a software program to optimise the ODROID XU3 development board for the use in an embedded system including a task analysis. This will be followed by a risk and a resource requirement analysis.

3.1 Research and Development methods

The Research component of the project will be to attain which software optimisation techniques will be advantageous in the development of the software. The Neon data engine and Graphics processing unit will also be reviewed to assist in the optimisation of the execution time of the software.

The Software written for this project will be used to show an understanding of how the ODROID XU3 multi core processor and development board can be optimised when performing two typical machine vision algorithms as follows:

- Global thresholding of a grey scale image.
- Average Smoothing filter

3.2 Task analysis

The task analysis will outline the milestones in the project which will be required to reach to successfully complete the project

- Identify components and current practices of embedded systems including optimisation techniques.
- Investigate the ODROID XU3 development board in particular the board's capabilities for the use in embedded systems. Including the following Hardware components
    - NEON data engine
    - Graphics processing unit
- Investigate GCC compiler and how to compile code to use different hardware components on the development board.
- Develop software for two typical machine vision algorithms as per the advice of the NCEA. Each algorithm will be optimised through various software techniques as well as modified to allow for them to be executed through the Neon data engine and the graphics processing unit.
- Implement developed software and obtain execution times through the use of the opencv gettickcount function to calculate execution time. Analyse and compare results to determine where improvements in execution time was found.

3.3 Risk Assessment

The Risk Assessment section of this dissertation will identify the risks involved in the project, evaluate these risks and decide on what control measures will be implemented.

3.3.1 Risk Identification

The following risks have been identified within this project.

a) Energy Source – Electrical shock
b) Storage – Loss of Documents stored on computer
c) Hardware – Computer failure
d) Hardware – Damage to ODROID XU3
e) Sickness – Failure to complete work due to illness or other unforeseen circumstances

3.3.2 Risk Evaluation

a) There may be a small risk of electrocution when plugging ODROID into power source, as this may need to occur many times to set and reset ODROID during testing.
b) A slight risk of computer files being corrupted resulting in the loss of data and files.
c) A very slight risk that damage to current computer resulting in the loss of availability to computer and internet resources.
d) A slight risk of damage to the ODROID XU3 development board.

e) Very slight risk for personal illness occurring and or other unforeseen circumstances in which may result in failing to complete dissertation.

3.3.3 Risk Control

a) When turning the ODROID off either use the power down options within the operating system or turn power off from the source. Do not plug and unplug chord from the ODROID when power is still supplied to chord.
b) When saving files on computer also regularly back up files either on a cloud based service such as sky drive or onto a USB.
c) Access to another computer will be required if there is damage resulting in a computer which no longer works. This can be achieved immediately as there is access to numerous computers.
d) If the ODROID XU3 is damaged the need to order a new one from the company will be required. This will result in a charge of approximately $180 USD and an approximate wait of two weeks for delivery.
e) If personal illness or other unforeseen circumstance is to arise documentation from a medical professional to assist in request for extension if required. Otherwise remaining ahead of schedule will assist to ensure delivery of dissertation in a timely manner.

3.4 Resource Analysis

The following section will outline the required resources needed at each stage of the project to be able to successfully complete the required tasks.

3.4.1 Research and reporting

There will be a large research and reporting component of this project which will begin at the start and will not be completed until the project is complete. This will require the following resources to be available for the duration.

- Computer access
    - A computer with access to the internet to assist with research and background information.
    - Word processor for reporting requirements.

3.4.2 Software development and Hardware testing

The process of developing software will be made more efficient if access to the ODROID XU3 is possible. The software can be can be written directly onto the ODROID through the use of its Linux based operating system lubuntu. Lubuntu is a light version of the more popular operating system Ubuntu. This will also allow testing to be completed as the program develops to allow for a more efficient debugging process. The ODROID development board will be paramount to obtain results and complete dissertation. The following components are required and have been supplied by the National Centre for Engineering in Agriculture.

- ODROID XU3
    - Including access to the operating system on board the ODROID XU3 and including access to a tool to write and compile the software such as a Linux based txt editor.

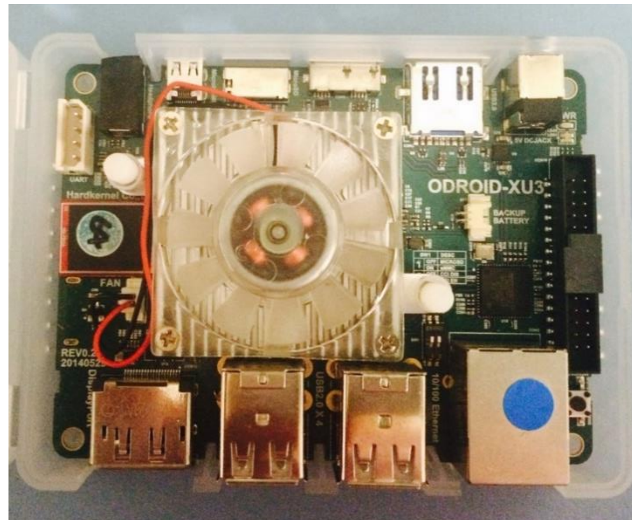


Figure 3.1 ODROID XU3 Development Board

- Power Supply
    - A 5V Power supply for the ODROID XU3
- EMMC
    - Memory card containing lubuntu operating system
- Micro HDMI to Large HDMI cable
    - Cable required to attach ODROID to monitor or screen
- Monitor
    - Monitor or screen to attach the development board to
- Mouse and Keyboard
    - Mouse and Keyboard will be required to operate the development board effectively.

3.5 Project Timeline

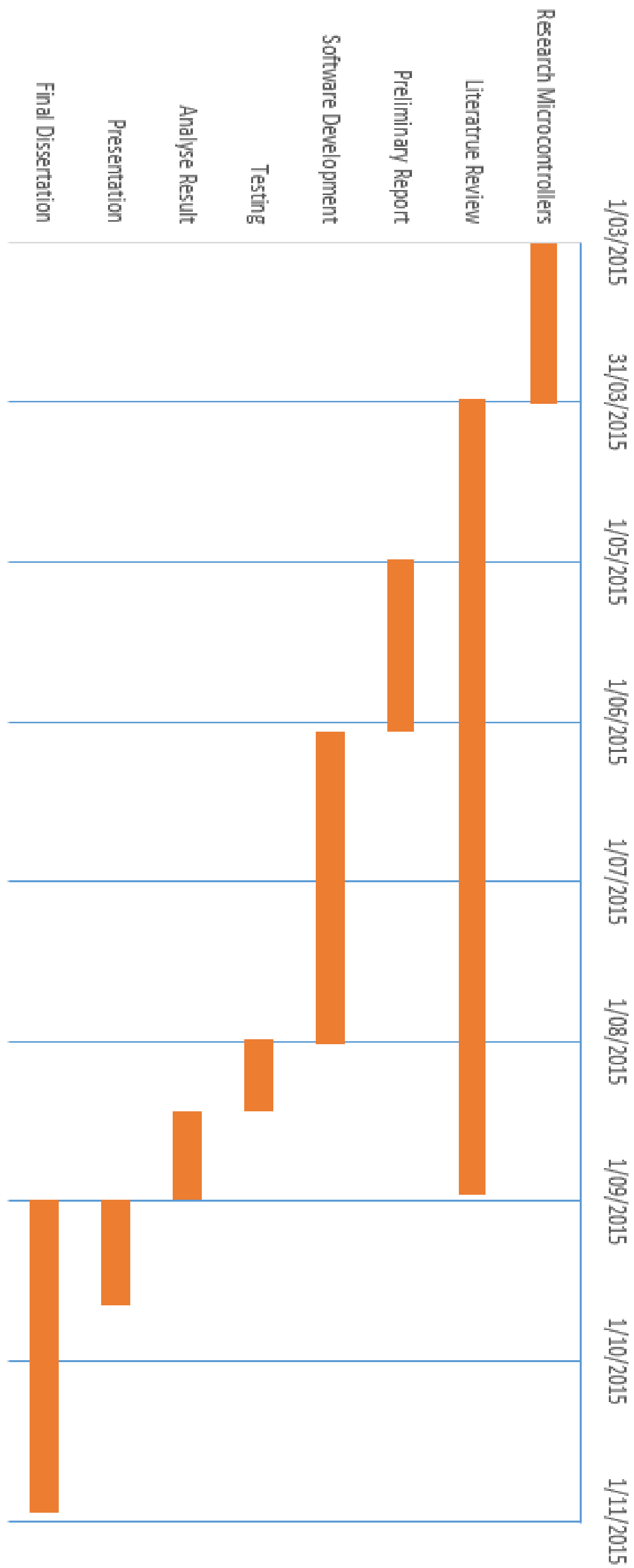


Figure 3.2 Project Timeline

Chapter 4

Software Development

This chapter will outline the stages undertaken during the software development. This will include the software optimisation techniques as well as the process of accessing the separate hardware units contained on the development board through the software. Software libraries which have been accessed to assist with software development and image processing will also be reviewed.

The process of development of each program will be examined including the output which was required from the two machine vision algorithms. Samples of code have been included to assist with understanding of each program.

Compiler options used for testing and how the timing of execution during testing was calculated will also be reviewed.

4.1 Optimisation Techniques

The literature review identified the following software optimisation techniques as being appropriate techniques to reduce execution time within software programs.

1. Loop unrolling
2. Loop reversal
3. Pointers

To test which software techniques will improve execution time the code was written and tested for two machine vision algorithms as follows:

1. Without any optimisation
2. Loop Reversal
3. Containing one loop unrolled.
4. Optimised through the use of pointers
5. Optimised with pointers and loop reversal.
6. Optimised with pointers and a loop unrolled.
7. Any calculations which can be manually vectorised to be calculated by the Neon data engine
8. Calculations done through programming the GPU

4.2 Image Processing

Image processing was required for both programs and was achieved through the use of the following libraries.

- OpenCV - was used to import image pixel values allowing alterations to images to be made as required as well as other functions such as displaying and saving altered images. OpenCV was also used to assist with timing of execution through the getTickCount() and getTickFrequency() functions.
- OpenCL has been used to access the Graphics processing unit for the execution of kernels.
- Neon Intrinsic library was used to develop program which required access to the Neon data engine.

4.3 Grey Scale Thresholding

Greyscale thresholding is a machine vision algorithm which comprises of changing the intensity levels of individual pixels within a greyscale image. A threshold was chosen to identify at which value pixel colours or intensity were changed. By altering the intensity value of a pixel for example to a value of 0, the pixel will change to the colour black. If the intensity is changed to the top value of 255 this will cause the pixel to change to white.

The photo chosen to be used when testing the two machine vision algorithms was found within the sample codes in the openCV library. This image was chosen as it was a greyscale image and appropriate for testing of the programs.

Figure 4.1 (a) Input image 4.1 (b) Output image

For the purpose of this dissertation a threshold value of 128 was chosen and any value below this threshold was changed to an intensity of 0. The result of this change is shown in figure 4.1.

The first program to be developed was the grey scale threshold program containing no optimisations. The program was developed to loop through each pixel value in the image and change any values below the threshold. The OpenCV data type Mat was used to hold the data values of the image and the image was imported via the imread function. The Mat data type is a matrix, and as the image used was a greyscale image the matrix stored the intensity values of each pixel ranging from 0 to 255.

The program contains two nested for loops which is where the pixel level is checked and changed. The first loop, loops through until it reaches the value of the height of the image matrix and the second nested loop, loops until

reaching the value of the width. The height and width values were found using OpenCV methods of height and width. These two loops allow the program to compare each individual pixel of the image with the threshold. The program developed is show in figure 4.2 and can also be found in Appendix B.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
        // Get start time
        int64 tickCount = getTickCount();

        // Read in image using mat function
        Mat image;
        image = imread("scene_r.bmp", 0);
        Size s = image.size();
        int intensity = 0;
        int T = 128;

        // Loop through matrix and change intensity accordingly

        for (int i = 0; i < s.height; i++)
        {
                for(int j = 0; j < s.width; j++)
                {
                        unsigned char bgrPixel = image.at<uchar>(i,j);
                        intensity = int(bgrPixel);

                        if (intensity < T){

                                        image.at<uchar>(i,j) = 0;
                        }

                }
        }

        // Get finish time and find difference devided by frequency
        double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
        cout << time << endl;


//      namedWindow("Display window", WINDOW_NORMAL);
//      imshow( "Display window", image);
//      imwrite("Output.bmp",image);
        waitKey(0);

        return 0;
        }
```

Figure 4.2 Grey scale threshold development code.

The second program developed and tested is the Grey Scale threshold with loop reversal. Developing this program only required a small change from the original, which included changing the loops to decrementing in place of incrementing loops. This is shown in figure 4.3 and a copy of the full program can be found in Appendix B.2.

```
for (int i = s.height; i>0; i--)
{
        for(int j = s.width; j>0; j--)
        {
                unsigned char bgrPixel = image.at<uchar>(i,j);
                intensity = int(bgrPixel);

                if (intensity < T){
                        image.at<uchar>(i,j) = 0;
                }

        }
}
```

Figure 4.3 Loop reversal for greyscale threshold.

The second software optimisation used was loop unrolling. Development of this code was done by unrolling the inner loop. To achieve loop unrolling the width of the image matrix was required and found. This width of 384 was the amount of times the inside loop needed to be written. An excerpt of the loop unrolling program can be found in figure 4.4 and the full program is shown in Appendix B.3.

```
for (int i = 0; i < s.height; i++)
{

        unsigned char bgrPixel = image.at<uchar>(i,1);
        intensity = int(bgrPixel);
        if (intensity < T){
                image.at<uchar>(i,1) = 0;
        }

         bgrPixel = image.at<uchar>(i,2);
        intensity = int(bgrPixel);
        if (intensity < T){
                image.at<uchar>(i,2) = 0;
        }

         bgrPixel = image.at<uchar>(i,3);
        intensity = int(bgrPixel);
        if (intensity < T){
                image.at<uchar>(i,3) = 0;
        }

         bgrPixel = image.at<uchar>(i,4);
        intensity = int(bgrPixel);
        if (intensity < T){
                image.at<uchar>(i, 4) = 0;
        }
```

Figure 4.4 Loop unrolling excerpt.

The third and final software optimisation technique used was the use of pointers in the program development. Pointer Optimisation was achieved through using the IplImage data type contained within the OpenCV library. IplImage data type enables a declaration of a pointer to the image data. By using the pointer in the IplImage, the program is able to loop through the size of the image within one loop opposed to the two loops previously used. This is demonstrated in figure 4.5.

```
// Loop through matrix and change intensity
for(int x = 0; x < size; x++)
{
        if (*ptr < T){
        *ptr = 0;
        ptr++;
        }
        else {ptr++;}
}
```

Figure 4.5 For loop within the Grey scale threshold algorithm with use of pointer.

The pointer optimised program was then changed to allow for loop reversal and loop unrolling and each program individually tested.

The next section of the software development comprised altering the program to allow control over which hardware units were to process the data.

The first changes made to the code was achieved by adding to the Grey scale algorithm with the use of pointers the code shown in figure 4.6.

```
cpu_set_t my_set;
CPU_ZERO(&my_set);
CPU_SET(7, &my_set);
sched_setaffinity(0, sizeof(cpu_set_t), &my_set);
```

Figure 4.6 Block of code directing process through single core

Using sched_setaffinity allows the programmer to direct the data through the chosen CPU core. Use of this function with the ODROID XU3 allows the use of the eight cores with the following numbers:

A7 Core 1 = 0	A15 Core 1 = 4
A7 Core 2 = 1	A15 Core 2 = 5
A7 Core 3 = 2	A15 Core 3 = 6
A7 Core 4 = 3	A15 Core 4 = 7

The addition of the block of code to the next two programs to be tested allowed the data to be directed through a single A7 core by setting the CPU to 0 and through a single A15 core by setting the CPU to 7. These two programs can be found in Appendix B.7 and B.8.

The next program to be developed was a Grey scale threshold program which was to be directed through the Neon data engine. As explained previously in chapter 2 section 2.2.1. The Neon data engine works with SIMD or single input multiple data stream and the software was written using neon intrinsic functions and data types. The arm_neon.h header was added to allow use of the neon intrinsic functions.

To use the neon data engine the program needed to be vectorised. Vectorisation was achieved through the use of vector data types available within the neon intrinsic library. The image pointer was declared as vector type unin8x16_t. This contains 16, 8 bit unsigned intergers. To use the neon data engine a new image was created and another pointer allocated for the altered output image data.

Two additional vectors were declared, one was named threshold and another named mask, both vectors were of data type unin8x16_t. The threshold

vector was loaded with the threshold value of 128 into each slot of the vector. This was done using the vdupq_n_u8(128).

The mask vector was used for the output of the compare of the ptr and threshold. If the ptr was greater than the threshold a value of 1 is recorded in the mask and if it is less a value of 0 is saved in the mask. An and operation is then performed on the ptr and mask and the result is then pointed to by the output image pointer. These calculations were done in a loop to ensure each pixel in the image was compared with the threshold and recorded. Figure 4.7 has an excerpt of the program containing the loop and the full program can be found in appendix B.9.

```
for (; size > 0; size -=16, ptr++, optr++){
mask = *ptr > threshold;
*optr = *ptr & mask;
}
```

Figure 4.7 Loop for Neon data engine

The last program to be developed for the greyscale thresholding algorithm was to direct the processing through the graphics processing unit. To access the graphics processing unit OpenCL libraries where needed alongside OpenCV.

After following tutorials in the Mali OpenCL SDK v1.1.0 the image_scaling sample was found to be similar to the program required and with a few alterations the grey sale algorithm was able to be achieved. This folder was copied and changed as follows:

#include <opencv2/opencv.hpp> was added to allow access to OpenCV functions and the Tick count functions were added to enable timing in .line with previous programs. The original program continued, initialising variables and setting up the OpenCL environment including creating context, command queue, program and kernel.

A pointer to the input image was then initialised and the height and width of the image found. As the program was set up to take an RGB or Red Green Blue image the greyscale image was converted to RGB with the code block contained in figure 4.8.

```
// Convert image to RGB added CM 25/10/2015
 unsigned char *rgbIn = new unsigned char[width * height * 3];
 luminanceToRGB(input, rgbIn, width, height);
 saveToBitmap("input.bmp", width, height, rgbIn);
```

Figure 4.8 Convert greyscale image to RGB code

A 2D Image is now created and held in memory alongside a memory object for the new output image to be written to. The image is then transferred to a RGBA format which essentially adds an extra empty container in the RGB array.

Global memory of width and height are set to allow access to these within the kernel. The kernel is now called which contains the grey scale algorithm as per the code in appendix B.10. The kernel compares each value in the first position of the RGBA array with the threshold and the output image data is written into memory. The memory is now accessed by the host code, transferred back into the RGB format and saved as a bitmap file.

4.4 Average Smoothing Filter

The Average smoothing filter is a typical machine vision algorithm which smoothes sharp edges in images. This is calculated by adding neighbouring pixel values and replacing the pixel value with the average of these neighbours. This is demonstrated in figure 4.9. This calculation is carried out on each pixel in the image. There are issues with this type of algorithm when considering outside pixels and therefore edges of the diagram have the same value then prior to the image processing.

$$M = 14/9$$
$$M = 1.55$$

1	2	2
1	1	2
1	2	2

Figure 4.9 smoothing filter algorithm.

Figure 4.11 shows the result of an image being processed by a smoothing filter algorithm and figure 4.10 contains the original image.

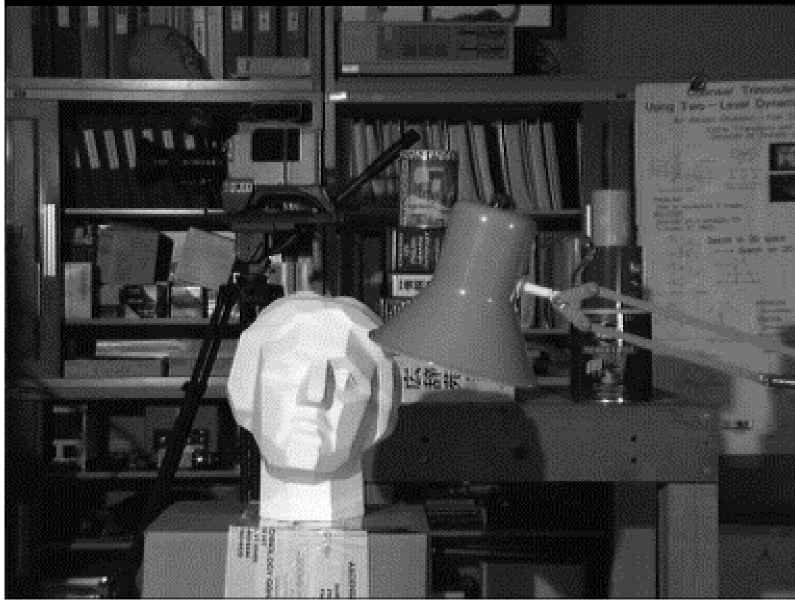


Figure 4.10 Original image used for image processing

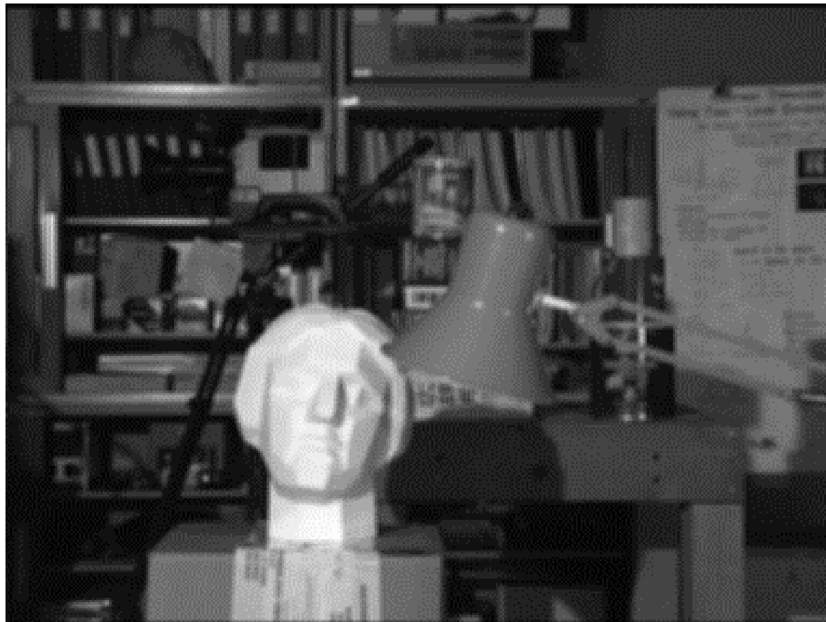


Figure 4.11 Image after alterations from average smoothing filter

Figure 4.12 contains a full copy of the average smoothing filter.

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main ()
{
        // Get start time
        int64 tickCount = getTickCount();

        // Read in image using mat function
        Mat image;
        image = imread("scene_r.bmp", 0);
        Size s = image.size();
        unsigned char bgrPixel;
        int avg = 0;

        // Loop through matrix and change intesity accordingly
        for (int i = 1; i < s.height; i++)
        {
                for(int j = 1; j < s.width; j++)
                {

                        for( int k = 0; k<=2; k++){
                            for (int l = 0; l<=2; l++){
                                avg = avg + image.at<uchar>(i+k,j+l);
                        }
                        }
                        image.at<uchar>(i,j) = avg/9;
                        avg = 0;
                }
        }

        // Get finish time and find difference devided by frequency
        double time = ((getTickCount() -
        tickCount)/getTickFrequency())*1000;
        cout << time << endl;

        //namedWindow("Display window", WINDOW_NORMAL);
        //imshow( "Display window", image);
        //imwrite("Output.bmp",image);
        //waitKey(0);

        return 0;
        }
```

Figure 4.12 Average Smoothing Filter Program

The main algorithm for this program contains four loops, the first two loops are identical to the greyscale image and allow the program to loop through the width, times the height of the image. The inner two loops allow the program to find the sum of the values of the 3x3 matrix required to find the pixels average of the neighbouring values. This process continues through every pixel in the image and changes values of pixels accordingly.

The next two programs developed for the smoothing filter has the same changes that where made previously to the greyscale threshold programs. The first change being Loop reversal and as done previously this involves changing the loops to a decrement replacing the incrementing loop. A full copy of the average smoothing filter with loop reversal can be found in appendix C.2.

The next change was to unroll one inner loop in the code to create the average smoothing filter with loop unrolling. This has been a smaller change in comparison to the greyscale threshold as the inner loops only contain 9 lines in total as shown in figure 4.13. The full code for this software program can be found in appendix C.3.

```
// Loop through matrix and change intensity accordingly
for (int i = 1; i < s.height; i++)
{
        for(int j = 1; j < s.width; j++)
        {

                avg = image.at<uchar>(i,j);
                avg = avg + image.at<uchar>(i+2,j+2);
                avg = avg + image.at<uchar>(i+1,j+2);
                avg = avg + image.at<uchar>(i+0,j+2);
                avg = avg + image.at<uchar>(i+2,j+1);
                avg = avg + image.at<uchar>(i+1,j+1);
                avg = avg + image.at<uchar>(i+0,j+1);
                avg = avg + image.at<uchar>(i+2,j+0);
                avg = avg + image.at<uchar>(i+1,j+0);

                image.at<uchar>(i,j) = avg/9;
                avg = 0;
        }
}
```

Figure 4.13 Loop for average smoothing filter with loop unrolling

The next software optimisation to be added to the smoothing filter is the use of pointers. This has again been achieved through the use of the IplImage data structure allowing a pointer to be pointed to the image data. The full code can be found at appendix C.4.

The pointer optimisation smoothing filter code has then had loop reversal added and as with the greyscale threshold involves decrementing loops. The full code for this software development can be found in Appendix C.5.

The pointer optimisation was then altered by unrolling the inner loop as was done previously and can be seen in figure 4.13 with the only change being the use of pointers opposed to the mat image function. The full code for this program can be found in Appendix C.6.

To direct the processing through selected cores in both the A7 and A15 processes the sched_setaffinity() function has been used as was done in the grey scale threshold and CPU values set accordingly. This has been used to direct data through an individual A7 and A15 core.

Until this point all the changes made to the original smoothing filter program have been very similar to the changes made in the grey scale thresholding programs, however the next changes were to allow the data to be vectorised and sent through the data engine unit and have very different changes.

As this algorithm requires 3x3 matrix values, this has been a challenge to vectorise. The neon data engine allows a 128 bit wide vector length and this was found to be difficult to use when trying to split data into a 3x3 matrix.

The pointers to the image where stored in uint8x8_t data types which only allows one value at a time. As we are now taking image data and creating a new image opposed to changing values in the original image, the outside values of the image need to be set to the original values.

The program was then developed to load 3 unit8x8_t (contains 8 unit values) with 8 image values from three lines. This giving the appearance of an 8x3 matrix. Line 1 and line 2 are summed together but now need to be stored in a 16x8_t data type to allow for a number larger than 255. Line 3 is then added to this value. Figure 4.14 is the excerpt of code of which load and sum through the use of intrinsic functions.

```
line1 = vld1_u8(ptr);
line2 = vld1_u8(ptr + width);
line3 = vld1_u8(ptr + (width * 2));

sum = vaddl_u8(line1, line2);
sum = vaddw_u8(sum, line3);
```

Figure 4.14 Load and sum functions in average smoothing filter directed through neon data engine.

The program adds the first three values together contained in the sum variable and divides the total by 9. This value is now stored into the output pointer, and the output pointer is incremented. This continues six times which does cause the last value in the vector to remain empty. The program continues to loop through these calculations and each time increments the ptr by a value of six. This program will only work for an image which has a width which is divisible by 6. The full code for this program can be found in Appendix C.9.

The last program written for this dissertation was the average smoothing filter which was directed through the graphics processing unit. As with the grey scale thresholding algorithm a sample program was chosen which had similar requirements to the average smoothing filter. The program chose was the fir_float.cpp and kernel.

Appropriate OpenCL headers and getTickCount functions were added to the program. The image was loaded through the IplImage data structure, however this time it was saved in the buffer and not in a 2D image. The image array values are converted to floats and the kernel for the smoothing filter algorithm is queued.

Four values from the first line of the image are loaded into a float4 vector structure through the vload4 instruction. The next container has the four values loaded from one point forward and the last container has another four values loaded from the next point forward. This would give the following values where the values contained in the data vectors are the position of the first line in the image data.

data 0 = 0,1,2,3

data 1 = 1,2,3,4

data 2 = 2,3,4,5

The first values in each array are added together giving the sum of the first three values in line 1. This is continued for line 2 and line 3, thus giving the summed value of a 3x3 matrix. This value is then divided by 9 and outputted to the output image pointer. Once all pixel values have been changed the program returns to the host and retrieves the output image values. These values are converted from a float to and uchar and stored as a bitmap image. The full code for this program can be found in appendix C.10.

4.5 Testing

The calculation of execution time was found with the use of the OpenCV function getTickCount(). GetTickCount returns the number of ticks after an event (OpenCV 2014). This function were placed at the beginning of the program and again at the end. The tick count from the beginning of the program was subtracted from the end count which gave an overall tick count for the execution of the program. To turn the tick count into time the getTickFrequency() function was used. The getTickFrequency() fuction returned the number of ticks per second (OpenCV 2014). The overall tick count was then divided by the tick frequency to obtain the time in seconds and multiplied by 1000 to obtain the time in milliseconds. The code used to calculate execution time is in figure 4.1.

```
// Get finish time and find difference devided by frequency
double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
cout << time << endl;
```

Figure 4.15 Copy of program section which calculates execution time.

To assist with getting an accurate representation of execution time 10 results were taken and the average used as the result.

All programs were compiled using the G++ compiler v4.8. Each program was tested using optimisation values O0, O1, O2, and O3 and execution time recorded.

Chapter 5

Results and Discussion

This chapter will display and discuss test results from the execution of the developed programs. When programs were tested they were executed ten times and the average was calculated and used as the final result. All ten test results for each program have been displayed throughout this chapter. All output images from testing both the greyscale thresholding programs and the average smoothing filter programs can be found in appendix D and appendix E.

5.1 Grey Scale Thresholding

Figure 5.1 and 5.2 is an example of the effects of the greyscale thresholding image processing.

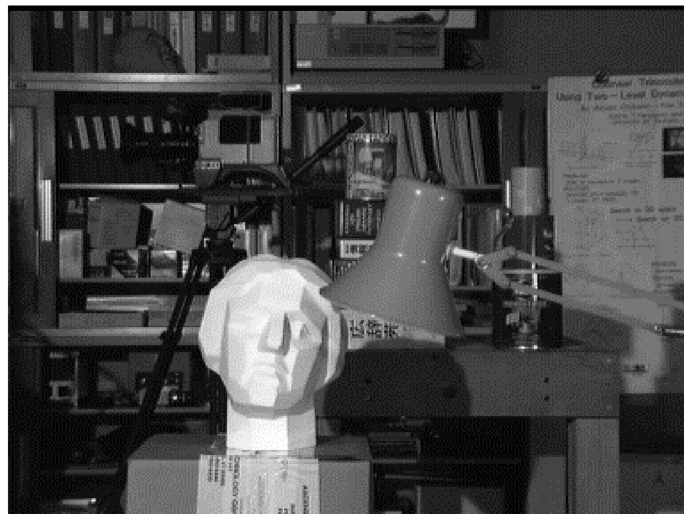


Figure 5.1 Image prior to image processing.

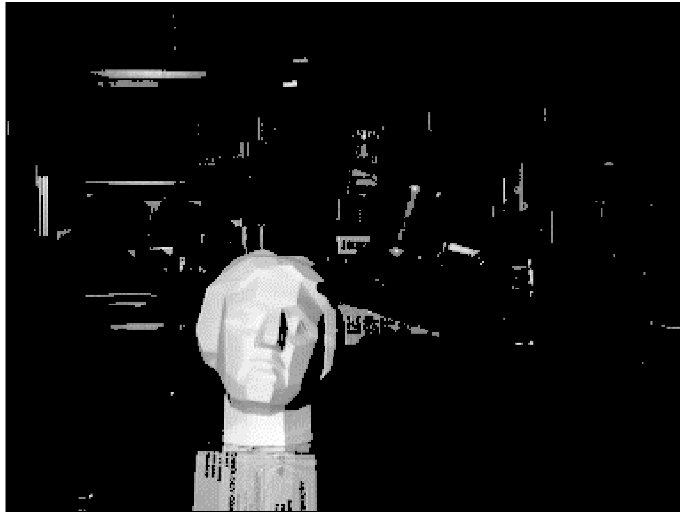


Figure 5.2 Output image from Grey Scale Thresholding

5.1.1 Results

No Optimisation

The first program tested was the grey scale thresholding program which contained no optimisations and would be used as a base time for all other tests to be compared. As seen in table 5.1 the compiler optimisations made a difference to the execution time and it can be noted that the fastest time was using the 03 compiler option.

	Grey scale thresholding			
Compiler Option	**0**	**1**	**2**	**3**
test 1	3.764	1.971	1.926	1.987
test 2	3.851	1.984	1.929	1.924
test 3	3.754	1.924	1.927	1.917
test 4	3.737	1.924	1.971	1.952
test 5	3.748	1.925	1.972	1.923
test 6	3.700	2.039	1.971	1.921
test 7	3.754	1.917	1.925	1.977
test 8	3.730	1.982	1.977	1.924
test 9	3.747	1.919	1.945	1.924
test 10	3.722	1.925	1.930	1.927
Average (ms)	**3.751**	**1.951**	**1.947**	**1.938**

Table 5.1 Results from grey scale thresholding program

Loop Reversal

The next test performed was on the first of the optimised programs which was grey scale thresholding with loop reversal. A decrease in execution time was expected and comparing the results in table 5.2 with table 5.1 there is a small decrease in time when no compiler optimisation was chosen. When the G++ compiler is optimised using 01, 02 and 03 is used 03 is still the fastest options.

	Grey scale thresholding with loop reversal			
Compiler Option	**0**	**1**	**2**	**3**
test 1	3.684	1.987	1.989	1.930
test 2	3.680	1.988	1.937	1.936
test 3	3.772	1.977	1.939	1.927
test 4	3.719	1.973	1.934	1.967
test 5	3.679	1.941	1.943	1.929
test 6	3.687	1.940	1.933	1.928
test 7	3.655	1.939	1.942	1.985
test 8	3.679	1.940	1.940	1.986
test 9	3.694	1.939	1.944	1.937
test 10	3.685	1.975	1.929	1.926
Average (ms)	**3.693**	**1.960**	**1.943**	**1.945**

Table 5.2 Results from grey scale thresholding with loop reversal

Loop Unrolling

The next program to be tested is the grey scale thresholding with a loop unrolled. The expectations of unrolling the loop were not high as it increased the size of the execution code by approximately 2000 lines of code. The results shown in table 5.3 show that there was not a decrease in execution time and in fact there was a significant increase.

	Grey scale thresholding with loop unrolling			
Compiler Option	**0**	**1**	**2**	**3**
test 1	4.030	2.096	2.092	2.061
test 2	3.871	2.097	2.218	2.140
test 3	3.925	2.131	2.033	2.008
test 4	3.923	2.118	2.089	2.133
test 5	3.964	2.011	2.028	2.138
test 6	4.070	2.068	2.117	2.004
test 7	4.013	2.079	2.007	2.007
test 8	4.001	2.167	2.054	2.067
test 9	4.066	2.007	2.058	2.272
test 10	4.036	2.132	2.139	2.224
Average (ms)	**3.990**	**2.091**	**2.083**	**2.105**

Table 5.3 Results from grey scale thresholding program with loop unrolling

Pointer Optimisation

Pointer optimisation was the next optimisation technique chosen to see if the execution time can be decreased. This was the first large decrease in execution time, going from 1.938 ms to 1.201 ms using g++ compiler option 02.

	Grey scale pointer optimisation			
Compiler Option	**0**	**1**	**2**	**3**
test 1	1.466	1.205	1.188	1.192
test 2	1.465	1.184	1.273	1.192
test 3	1.481	1.199	1.185	1.198
test 4	1.465	1.190	1.192	1.202
test 5	1.463	1.186	1.190	1.201
test 6	1.462	1.189	1.190	1.194
test 7	1.463	1.186	1.202	1.198
test 8	1.524	1.187	1.196	1.190
test 9	1.464	1.299	1.209	1.334
test 10	1.465	1.221	1.189	1.190
Average (ms)	**1.472**	**1.205**	**1.201**	**1.209**

Table 5.4 Results from grey scale thresholding with pointer optimisation

Pointer and Loop unrolling optimisation

Using the pointer optimisation and adding loop unrolling did not show a significant increase in code size and this is reflected in the results, however there has been a small decrease in execution time under the gcc 03 compiler option in comparison to the 03 column in table 5.4, however the result is equal to the 02 compiler option and therefore there is only an advantage using the gcc 03 compiler option.

	Grey scale pointer optimisation loop unrolling			
Compiler Option	**0**	**1**	**2**	**3**
test 1	1.583	1.204	1.201	1.257
test 2	1.587	1.200	1.211	1.261
test 3	1.575	1.243	1.205	1.379
test 4	1.642	1.202	1.198	1.205
test 5	1.581	1.198	1.212	1.201
test 6	1.581	1.198	1.246	1.257
test 7	1.584	1.229	1.260	1.200
test 8	1.576	1.205	1.201	1.201
test 9	1.574	1.205	1.203	1.209
test 10	1.579	1.205	1.203	1.290
Average (ms)	**1.586**	**1.209**	**1.214**	**1.246**

Table 5.5 Results from grey scale thresholding with pointer optimisation and loop unrolling

Pointer and Loop reversal Optimisation

Loop unrolling was then added to the pointer optimisation program and tested, a small decrease in execution time was found with no compiler optimisation, however no decrease in execution time was found with the g++ compiler optimisation options, as shown in figure 5.6.

	Grey scale pointer optimisation loop reversal			
Compiler Option	**0**	**1**	**2**	**3**
test 1	1.462	1.256	1.219	1.231
test 2	1.468	1.260	1.222	1.223
test 3	1.507	1.274	1.221	1.230
test 4	1.498	1.282	1.223	1.228
test 5	1.469	1.225	1.277	1.361
test 6	1.467	1.222	1.229	1.230
test 7	1.460	1.265	1.225	1.231
test 8	1.515	1.235	1.224	1.227
test 9	1.472	1.222	1.223	1.227
test 10	1.495	1.224	1.263	1.236
Average (ms)	**1.481**	**1.247**	**1.233**	**1.242**

Table 5.6 Results from grey scale thresholding with pointer optimisation and loop reversal

Arm A7

The next section of testing involved using different hardware units on the development board. The first test was to use a single Arm A7 core, the results were expected to see an increase in execution time due to clock speeds in comparison to the A15. As shown in table 5.7 the expected results were seen, however this did confirm that the scheduler is processing the grey scale thresholding program on an A15 core, when it is not being manually selected.

	Grey scale threshold A7			
Compiler Option	**0**	**1**	**2**	**3**
test 1	4.113	2.496	2.409	2.372
test 2	4.087	2.424	2.475	2.421
test 3	4.072	2.482	2.309	2.344
test 4	4.090	2.488	2.581	2.420
test 5	4.163	2.740	2.362	2.456
test 6	3.971	2.397	2.548	2.422
test 7	4.094	2.445	2.433	2.443
test 8	4.127	2.454	2.384	2.524
test 9	4.073	2.334	2.471	2.472
test 10	4.428	2.390	2.393	2.378
Average (ms)	**4.122**	**2.465**	**2.436**	**2.425**

Table 5.7 Results from grey scale thresholding directed through A7 core

Arm A15

Testing on the Arm A15 confirmed what was suspected from A7 testing results. The execution time when manually directing data through the A15 is similar to when no core is chosen, thus confirming that the scheduler is scheduling the greyscale threshold algorithm through the A15.

	Grey scale threshold A15			
Compiler Option	**0**	**1**	**2**	**3**
test 1	1.505	1.213	1.260	1.217
test 2	1.453	1.207	1.200	1.206
test 3	1.497	1.301	1.201	1.262
test 4	1.512	1.204	1.236	1.211
test 5	1.507	1.210	1.203	1.215
test 6	1.460	1.303	1.207	1.200
test 7	1.628	1.247	1.214	1.199
test 8	1.460	1.261	1.195	1.213
test 9	1.459	1.204	1.256	1.201
test 10	1.456	1.314	1.208	1.207
Average (ms)	**1.494**	**1.246**	**1.218**	**1.213**

Table 5.8 Results from grey scale thresholding directed through A15 core

Neon Data Engine

The next hardware unit the data was to be directed through was the Neon Data Engine. This test shows the biggest decrease in execution time so far. The best time being 1.051 ms using the g++ compiler optimisation 01.

	Grey scale threshold Neon			
Compiler Option	**0**	**1**	**2**	**3**
test 1	1.103	1.044	1.061	1.060
test 2	1.127	1.056	1.044	1.047
test 3	1.174	1.053	1.045	1.158
test 4	1.138	1.045	1.047	1.042
test 5	1.105	1.088	1.050	1.048
test 6	1.116	1.044	1.052	1.049
test 7	1.115	1.049	1.039	1.046
test 8	1.122	1.036	1.103	1.140
test 9	1.118	1.045	1.043	1.084
test 10	1.120	1.052	1.042	1.059
Average (ms)	**1.124**	**1.051**	**1.053**	**1.073**

Table 5.9 Results from Grey Scale Thresholding directed through Neon

Graphics Processing Unit

The last test performed on the grey scale threshold algorithm was using the graphics process unit. Unfortunately there is a large increase in execution time found when using the GPU. When only testing the execution time of the kernel there is only a small increase, however to keep timing in line with other programs the timing is calculated from the beginning of the program to the end. There may be more use for the GPU when continually running image processing functions as this will reduce the effect of the overhead due to the complexity and size of the program.

	Grey scale threshold GPU			
Compiler Option	**0**	**1**	**2**	**3**
test 1	196.482	185.425	184.421	184.108
test 2	194.219	183.977	184.450	183.520
test 3	193.843	184.489	183.435	184.147
test 4	194.104	185.935	184.185	183.828
test 5	194.005	184.838	184.066	185.166
test 6	194.750	184.106	184.765	184.584
test 7	194.073	184.656	183.670	185.410
test 8	194.753	184.122	184.443	183.809
test 9	195.198	183.789	183.692	183.447
test 10	193.974	184.939	184.289	184.055
Average (ms)	**194.540**	**184.628**	**184.142**	**184.207**

Table 5.10 Results from Grey scale program directed through GPU

Figure 5.3 contains all results from the grey scale thresholding testing.

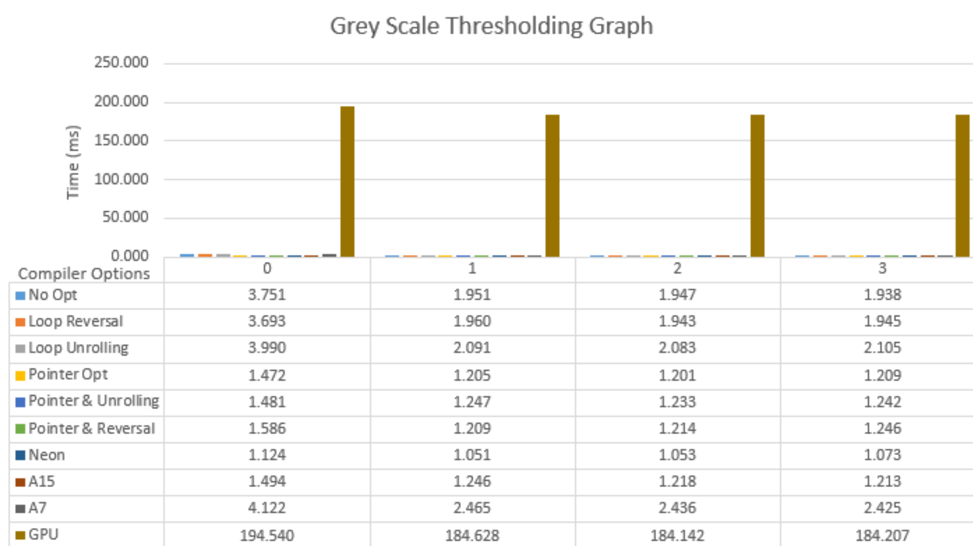
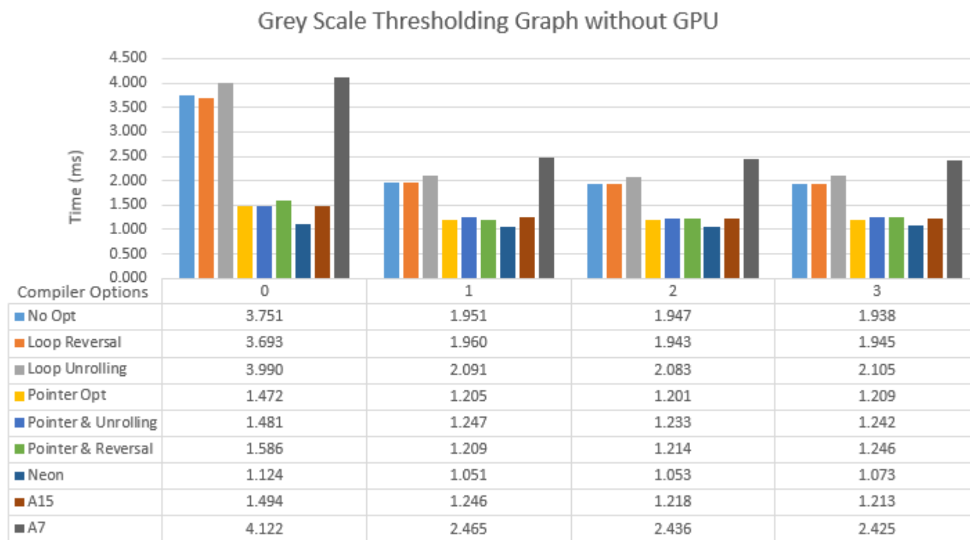


Figure 5.3 Grey Scale Thresholding Graph

Due to the large outlier from the execution speeds when using the GPU the graph in figure 5.3 does not give a clear indication of other results and therefore Figure 5.4 has a graph containing the results without the GPU results.



Compiler Options	0	1	2	3
No Opt	3.751	1.951	1.947	1.938
Loop Reversal	3.693	1.960	1.943	1.945
Loop Unrolling	3.990	2.091	2.083	2.105
Pointer Opt	1.472	1.205	1.201	1.209
Pointer & Unrolling	1.481	1.247	1.233	1.242
Pointer & Reversal	1.586	1.209	1.214	1.246
Neon	1.124	1.051	1.053	1.073
A15	1.494	1.246	1.218	1.213
A7	4.122	2.465	2.436	2.425

Figure 5.4 Grey Scale Thresholding Graph without GPU results

As shown in figure 5.4 the best result from making changes to the grey scale thresholding algorithm has been through the automatic compiler optimisations, pointer optimisations and through the use of the Neon data engine. There was no significant decrease in execution times using the two software optimisation techniques of loop unrolling and loop reversal after the program was compiled using compiler optimisation options.

5.2 Average Smoothing Filter

The same image that was used for the greyscale filter was used again for the average smoothing filter. The output picture from the average smoothing filter is shown in figure 5.5.

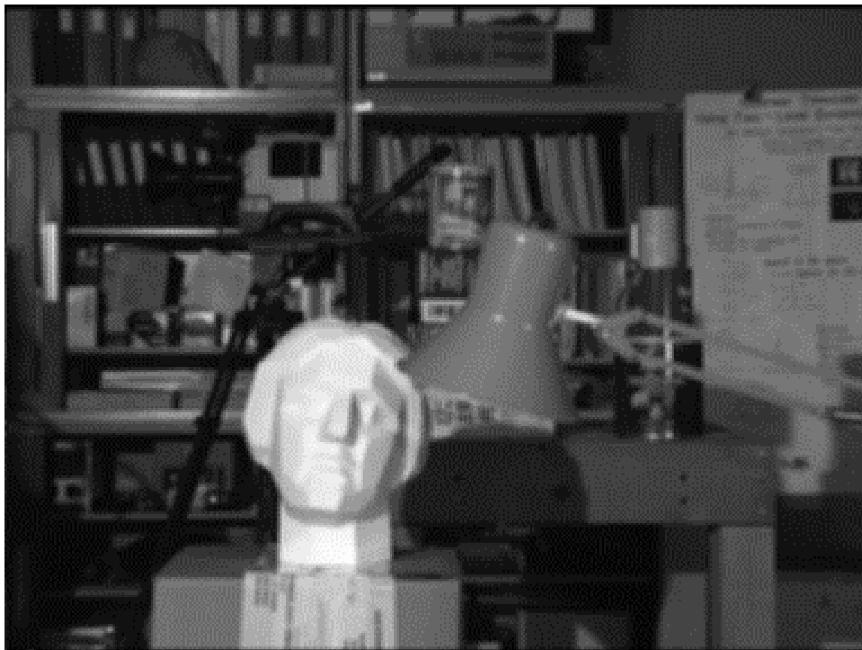


Figure 5.5 Output image from average smoothing filter

The smoothing of any edges in the image can be seen, giving the image an almost blurred effect.

5.2.1 Results

No Optimisation

The first test completed on the Smoothing filter program was with no software optimisations and this will again be used as the base of execution times for all other programs to be compared.

	Smoothing Filter			
Compiler Option	**0**	**1**	**2**	**3**
test 1	16.134	5.566	5.209	2.044
test 2	16.230	5.645	5.305	2.077
test 3	16.160	5.694	5.345	2.083
test 4	16.134	5.636	5.255	2.024
test 5	16.209	5.704	5.342	2.026
test 6	16.254	5.734	5.199	2.023
test 7	16.163	5.582	5.363	2.064
test 8	16.154	5.645	5.314	2.030
test 9	16.177	5.806	5.187	2.090
test 10	16.183	5.523	5.367	2.133
Average (ms)	**16.180**	**5.654**	**5.289**	**2.059**

Table 5.11 Results from Average Smoothing Filter

Loop Reversal

The next test performed on the smoothing filter algorithm was using the software which had loop reversal optimisation added. Once again there was no advantages gained from using lop reversal.

	Smoothing with Loop reversal			
Compiler Option	**0**	**1**	**2**	**3**
test 1	16.134	5.566	5.209	2.044
test 2	16.230	5.645	5.305	2.077
test 3	16.160	5.694	5.345	2.083
test 4	16.134	5.636	5.255	2.024
test 5	16.209	5.704	5.342	2.026
test 6	16.254	5.734	5.199	2.023
test 7	16.163	5.582	5.363	2.064
test 8	16.154	5.645	5.314	2.030
test 9	16.177	5.806	5.187	2.090
test 10	16.183	5.523	5.367	2.133
Average (ms)	**16.180**	**5.654**	**5.289**	**2.059**

Table 5.12 Results from average smoothing filter with loop reversal

Loop unrolling

Adding Loop unrolling to the average smoothing filter has shown a large decrease in execution time when using compiler optimisation options of 0, 1 and 2. These gains were lost when compiler optimisation option 3 was tested.

	Smoothing with Loop unrolling			
Compiler Option	**0**	**1**	**2**	**3**
test 1	11.457	2.170	2.136	2.079
test 2	11.459	2.082	2.144	2.065
test 3	11.506	2.112	2.141	2.079
test 4	11.523	2.127	2.138	2.032
test 5	11.471	2.137	2.173	2.033
test 6	11.596	2.133	2.140	2.097
test 7	11.526	2.080	2.151	2.093
test 8	11.574	2.122	2.141	2.072
test 9	11.499	2.117	2.191	2.078
test 10	11.570	2.124	2.139	2.034
Average (ms)	**11.518**	**2.120**	**2.149**	**2.066**

Table 5.13 Results from average smoothing filter with loop unrolling

Pointer Optimisations

Similar to the grey scale threshold program a large decrease in execution time was found by adding pointers to the image structure. A new pattern started to emerge where there was a notable slowing of the program using compiler optimisation 02 and then a large increase in execution speeds when compiler option 03 was chosen.

	Smoothing Optimised			
Compiler Option	**0**	**1**	**2**	**3**
test 1	9.480	4.185	6.179	1.223
test 2	9.503	4.650	6.880	1.233
test 3	9.523	4.127	6.940	1.229
test 4	9.535	4.559	8.308	1.245
test 5	9.499	4.188	6.974	1.233
test 6	9.554	3.967	6.259	1.225
test 7	9.528	4.088	6.203	1.229
test 8	9.502	4.171	8.208	1.229
test 9	9.517	4.113	6.549	1.284
test 10	9.508	4.150	6.819	1.230
Average (ms)	**9.515**	**4.220**	**6.932**	**1.236**

Table 5.14 Results from average smoothing filter with pointer optimisation

Pointer Optimisations with loop reversal

Adding loop reversal eliminated the decrease in speeds under compiler optimisation 02. This did not see any other significant gains as shown in table 5.15.

	Smoothing Optimised with loop reversal			
Compiler Option	**0**	**1**	**2**	**3**
test 1	9.422	4.167	2.810	1.282
test 2	9.433	5.006	2.803	1.243
test 3	9.517	4.874	2.805	1.250
test 4	9.394	4.140	2.809	1.278
test 5	9.455	4.987	2.812	1.257
test 6	9.433	4.208	2.798	1.245
test 7	9.407	4.459	2.933	1.241
test 8	9.493	4.880	2.860	1.308
test 9	9.470	5.058	2.811	1.249
test 10	9.436	4.719	2.944	1.283
Average (ms)	**9.446**	**4.650**	**2.839**	**1.264**

Table 5.15 Results from average smoothing filter with pointer optimisation and loop reversal

Pointer Optimisations with loop unrolling

The next test was completed on a smoothing filter program using Pointer Optimisation and loop unrolling as the software optimisation. Loop unrolling once again has seen a slowing of the program for compiler option 03 in comparison to table 5.14. There was however an increase in speed under optimisation option 01 and 02 which were still almost one millisecond slower than the best time we have received so far.

	Smoothing Optimised with loop unrolling			
Compiler Option	**0**	**1**	**2**	**3**
test 1	11.457	2.170	2.136	2.079
test 2	11.459	2.082	2.144	2.065
test 3	11.506	2.112	2.141	2.079
test 4	11.523	2.127	2.138	2.032
test 5	11.471	2.137	2.173	2.033
test 6	11.596	2.133	2.140	2.097
test 7	11.526	2.080	2.151	2.093
test 8	11.574	2.122	2.141	2.072
test 9	11.499	2.117	2.191	2.078
test 10	11.570	2.124	2.139	2.034
Average (ms)	**11.518**	**2.120**	**2.149**	**2.066**

Table 5.16 Results from average smoothing filter with pointer optimisation and loop unrolling

Arm A7

The next test performed involved testing the arm A7 hardware. The results were similar to the grey scale threshold program with the A7 being slower.

	Smoothing Opt A7			
Compiler Option	**0**	**1**	**2**	**3**
test 1	31.595	7.758	6.105	3.025
test 2	31.784	7.728	6.079	3.207
test 3	31.595	7.559	5.986	3.108
test 4	31.709	7.645	6.207	3.000
test 5	31.688	7.611	6.102	2.889
test 6	31.722	7.506	6.136	2.946
test 7	31.679	7.814	6.088	2.955
test 8	31.373	7.715	6.010	3.029
test 9	31.569	7.684	6.146	2.914
test 10	31.276	7.724	6.127	2.913
Average (ms)	**31.599**	**7.674**	**6.099**	**2.999**

Table 5.17 Results from average smoothing filter directed through A7

Arm A15

Testing the A15 unit also confirmed that it is likely that the scheduler is automatically scheduling any machine vision algorithms through the A15 core as timing is similar without manually choosing.

	Smoothing Optimised A15			
Compiler Option	**0**	**1**	**2**	**3**
test 1	9.518	4.852	6.764	1.311
test 2	9.531	4.176	6.877	1.238
test 3	9.572	4.144	6.272	1.237
test 4	9.588	5.069	7.077	1.337
test 5	9.500	5.055	7.544	1.239
test 6	9.543	5.051	7.006	1.251
test 7	9.510	4.135	6.872	1.237
test 8	9.572	4.947	6.884	1.230
test 9	9.519	4.994	6.763	1.233
test 10	9.534	4.063	7.042	1.250
Average (ms)	**9.539**	**4.649**	**6.910**	**1.256**

Table 5.18 Results from average smoothing filter directed through A15

Neon Data Engine

Testing the software code for the average smoothing filter through the Neon data unit, did not see an increase in execution speeds. When using the Neon data engine it is important code can be vectorised neatly for it to be an advantage. This has shown with the smoothing filter as it was difficult to vectorise the 3x3 moving window.

	Smoothing Neon Data Engine			
Compiler Option	**0**	**1**	**2**	**3**
test 1	3.373	1.397	1.426	1.449
test 2	3.280	1.403	1.478	1.424
test 3	3.263	1.400	1.419	1.413
test 4	3.255	1.403	1.417	1.412
test 5	3.191	1.403	1.475	1.475
test 6	3.212	1.398	1.459	1.419
test 7	3.239	1.395	1.414	1.421
test 8	3.267	1.394	1.415	1.423
test 9	3.265	1.396	1.472	1.464
test 10	3.223	1.394	1.413	1.417
Average (ms)	**3.257**	**1.398**	**1.439**	**1.432**

Table 5.19 Results from average smoothing filter directed through Neon

Graphics Processing Unit

The next test performed involved testing the Mali T628 Graphics Processing and as it was seen with the Grey Scale threshold program no advantage was found.

	Smoothing GPU			
Compiler Option	**0**	**1**	**2**	**3**
test 1	112.381	109.983	109.631	109.512
test 2	113.123	109.605	109.103	109.482
test 3	112.399	109.362	110.154	109.478
test 4	112.629	109.417	109.612	110.147
test 5	111.835	109.619	109.411	110.050
test 6	112.483	110.226	109.448	110.100
test 7	112.129	109.673	110.472	109.605
test 8	112.426	109.371	109.258	110.682
test 9	113.188	109.561	109.730	109.765
test 10	112.539	109.431	109.888	109.528
Average (ms)	**112.513**	**109.625**	**109.671**	**109.835**

Table 5.20 Results from average smoothing filter directed through GPU

The following graph in figure 5.6 contains all results for the average smoothing filter and again the GPU processing has caused the graph to be hard to read.

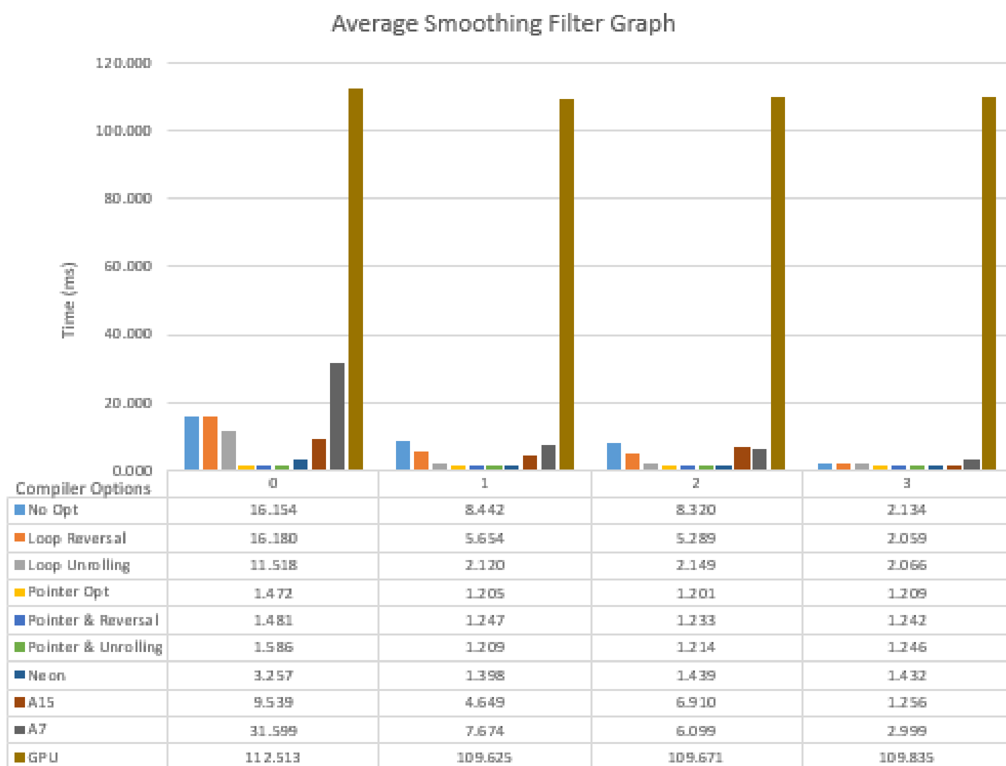
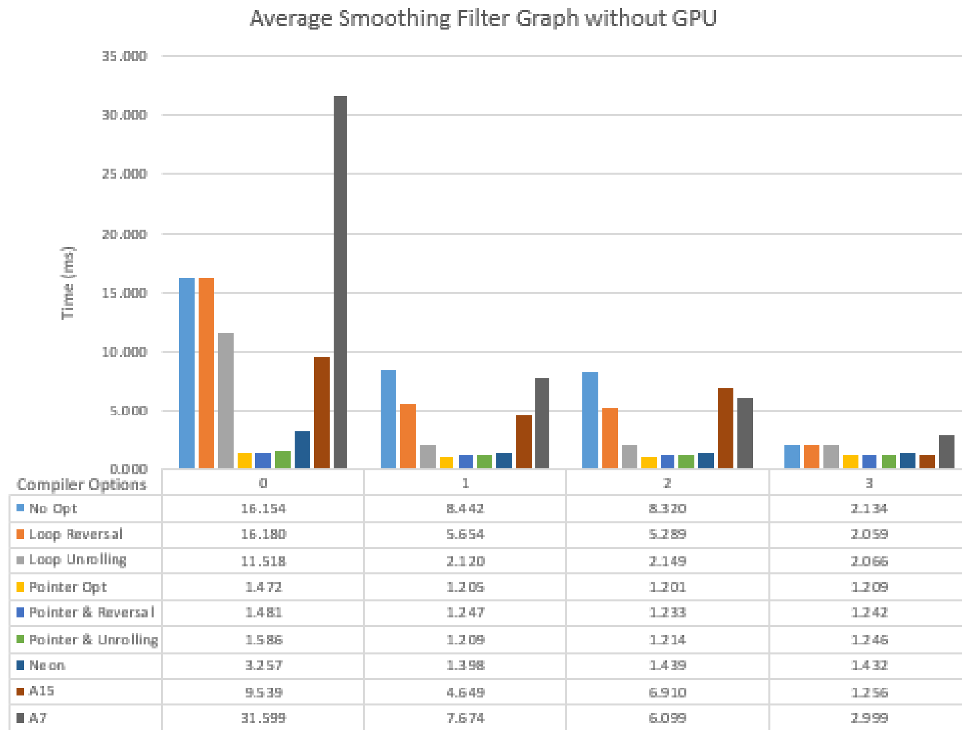


Figure 5.6 Average Smoothing Filter Graph

Figure 5.7 contains all results without the outliers from the GPU and therefore gives a good indication of the result from testing.



Compiler Options	0	1	2	3
No Opt	16.154	8.442	8.320	2.134
Loop Reversal	16.180	5.654	5.289	2.059
Loop Unrolling	11.518	2.120	2.149	2.066
Pointer Opt	1.472	1.205	1.201	1.209
Pointer & Reversal	1.481	1.247	1.233	1.242
Pointer & Unrolling	1.586	1.209	1.214	1.246
Neon	3.257	1.398	1.439	1.432
A15	9.539	4.649	6.910	1.256
A7	31.599	7.674	6.099	2.999

Figure 5.7 Average Smoothing Filter Graph without GPU results

5.3 Conclusion

Following analysis of the results, the following conclusions and recommendations have been drawn.

1. Compiler optimisation option 03 using g++ would be the recommendation when compiling machine vision algorithms
2. When code can be easily vectorised the Neon data engine is the fastest way of executing machine vision algorithms
3. When using compiler optimisations loops unrolling and loop reversal do not make a significant impact on the execution times.
4. The Graphics Processing Unit is not a viable option when running single image processing.
5. Automatic scheduling by the scheduler is a faster option when running single image machine vision algorithms.

Chapter 6

Conclusion and Further Work

6.1 Conclusion

This dissertation has successfully completed all 8 program objectives as outlined in the project specification contained in appendix A. The following points as outlined at the end of chapter five were determined.

1. Compiler optimisation option 03 using g++ would be the recommendation when compiling machine vision algorithms
2. When code can be easily vectorised the Neon data engine is the fastest way of executing machine vision algorithms
3. When using compiler optimisations loops unrolling and loop reversal do not make a significant impact on the execution times.
4. The Graphics Processing Unit is not a viable option when running single image processing.
5. Automatic scheduling by the scheduler is the faster option when running single image machine vision algorithms.

6.2 Further Work

If time constraints were not an issue it would be advantageous to re-test the programs processing more than one image. This may increase the viability of using the graphics processing unit as overheads become less of an issue.

References

ARM 2015, ARM Limited viewed 8th July 2015,

<http://www.arm.com/products/processors/cortex-a/cortex-a7.php.>

ARM 2015, ARM Limited viewed 8th July 2015,

<http://www.arm.com/products/processors/cortex-a/cortex-a15.php.>

ARM 2015, ARM Limited viewed 8th July 2015,

< http://www.arm.com/products/multimedia/mali-performance-efficient-graphics/mali-t628.php>

ARM 2014, ARM Limited viewed 18th November 2014,

<http://www.arm.com/about/company-profile/index.php>

ARM 2013, *ARM Cortex –A Series Programmer's Guide,* Version 4.0, Cambridge, England.

Arm Connected Community, ARM Limited viewed 10th July 2015,

<http://community.arm.com/groups/processors/blog/2014/03/28/arm-biglittle-making-waves>

Bates, M 2011, *PIC MICROCONTROLLERS,* Elsevier, Oxford.

Cao, C Naifeng, J, Weifeng, H & Yuzhuo, F 2010,'A loop-centric profiling method for embedded applications', *Microelectronics and Electronics (PrimeAsia), 2010 Asia Pacific Conference on Postgraduate Research in* , vol., no., pp.45,48, 22-24.

Electronic design, electronic design viewed 9th July 2014, <http://electronicdesign.com/microcontrollers/little-core-shares-big-core-architecture>

ELE 1301 Computer Engineering: study book 2013, University of Southern Queensland, Toowoomba

FPGA Fundamentals 2012, National Instruments Corporation, Austin, viewed 8th July 2015 < http://www.ni.com/white-paper/6983/en/>.

Grasso, I Radojkovic, P Rajovic, N Gelado, I & Ramirez, A 2014,'Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU',*2014 IEEE 28th International Parallel & Distributed Processing Symposium,* pp.123-132.

Jang, M Kim, K & Kim, K 2011, 'The performance analysis of ARM NEON technology for mobile platforms', *ACM Symposium on Research in Applied computation (RACS '11)*, pp 104,106.

Jo, G Jeon, W Jung, W Taft, G & Lee, J 2014,'OpenCL framework for ARM processors with NEON support', *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing*, pp. 33-40.

Joshi, P & Gurumurthy, K 2014 'Analysing and Improving the Performance of Software Code for Real Time Embedded Systems' *Devices, Circuits and Systems (ICDCS), 2014 2nd International Conference,* March 6-8, pp.1-5.

Mahalingam, P. R. & Asokan, S 2012, 'A framework for optimizing GCC for ARM architecture', *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, pp. 337-3423

Mitra, G Johnston, B Rendell, A McCreath, E & Zhou, J 2013,'Use of SIMD vector operation to accelerate application code performance on lower-powered ARM and intel platforms',*Parallel and Distributed Processing Symposium Workshops & PHD Forum, 2013 IEEE 7th International, pp 1107-1116.*

OpenCV 2014, *OpenCv 2.4.11.0 documentation*, viewed 25th October 2015, <http://docs.opencv.org/modules/core/doc/utility_and_system_functions_and_macros.html#gettickcount>.

Park, I Lee, H & Lee, H 2013 'Software Optimization for Embedded Communication system', *Information Networking (ICOIN) International Conference,* Jan 28-30, pp. 676-679.

Pulli, K Baksheev, A Kornyakov, K & Eruhimov. V 2012, ' Realtime Computer Vision with OpenCV', *Queue* 10, 4, pp 40, 57.

Nvidia 2015, NVIDIA Corporation CA, USA viewed 8th July 2015, <http://www.nvidia.com/object/what-is-gpu-computing.html>.

Simunic, T Benini, L De Micheli, G & Hans, M 2000, 'Source code optimization and profiling of energy consumption in embedded systems', *Proceedings of the 13th international symposium on system synthesis (ISSS '00),* pp. 193-198.

Smith, S 1997, *The scientist and engineer's guide to digital signal processing,* California Technical Pub., California.

Thompson, T2001,'Digital Signal Processor', *Computerworld,* vol.35, no. 11, p. 66.

Velkoski, G Gusev, M. & Ristov, S 2014, 'The performance impact analysis of loop unrolling', *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention*, vol., no., pp.307-312.

Yi, S Yoon, I Oh, C & Yi, Y 2014,'Real-time integrated face detection and recognition on embedded GPGPUs", *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium,* Oct 16-17, pp, 98-107.

Zhou, J & Shi, C 2008, 'Efficient SIMD optimization for media processors', *Journal of Zhejiang University SCIENCE A,* Vol. 9, April 20, pp. 524-530.

Appendix A

Project Specification

University of Southern Queensland

FACULTY OF ENGINEERING AND SURVEYING

ENG 4111/4112 Research Project
PROJECT SPECIFICATION

FOR:	Chloe Mansell
TOPIC:	OPTIMISATION OF MULTICORE PROCESSOR AND GPU FOR USE IN EMBEDDED SYSTEMS
SUPERVISOR:	Mr Steven Rees, National Centre of Engineering in Agriculture
ENROLMENT:	ENG 4111 – S1, 2015 ENG 4112 – S2, 2015
PROJECT AIM:	The aim of this project is to investigate the Exynos 5422 multicore processor and ancillary computational hardware on the ODROID XU3 development board. Develop software to optimise the processor and ancillary devices on the development board. Compare the operation of the optimised software and automated compiler generated software for benefits to embedded systems.

PROGRAMME: **Issue A. 6th March 2015**

1. Research Background information in relation to operation and current practices of:
   - embedded microcontrollers, DSPs, Logic devices and GPUs;
   - multicore processors;
   - optimisation techniques
2. Investigate the ODROID XU3 development board hardware capacity (CPU cores, FPU, NEON and GPU).
3. Design and develop software to optimise the operation of the CPU cores, FPU, NEON and GPU for application of an algorithm for benchmarking.
4. Implement and record operational data for optimised software. Implement and run automated code generated by the compiler for the ODROID XU3.
5. Compare and analyse the results of the optimised verse automated software to determine the value of manually optimising embedded systems for the additional on-board hardware.
6. Investigate the benefits of optimising machine vision algorithms (e.g. stereo vision) for on-board hardware.
7. Write dissertation.
8. Submit Dissertation on the above

AGREED:

Chloe Mansell (Student)	Steven Rees (Supervisor)
17/03/2015	17/03/2015

Appendix B

Code for Grey Scale Threshold Image Processing

B.1 Grey Scale Threshold

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	Mat image;
	image = imread("scene_r.bmp", 0);
	Size s = image.size();
	int intensity = 0;
	int T = 128;

	// Loop through matrix and change intensity accordingly

	for (int i = 0; i < s.height; i++)
	{
		for(int j = 0; j < s.width; j++)
		{
			unsigned char bgrPixel = image.at<uchar>(i,j);
			intensity = int(bgrPixel);

			if (intensity < T){

					image.at<uchar>(i,j) = 0;
			}

		}
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;


//	namedWindow("Display window", WINDOW_NORMAL);
//	imshow( "Display window", image);
//	imwrite("Output.bmp",image);
	waitKey(0);

	return 0;
	}
```

B.2 Grey Scale Threshold with Loop Reversal

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;


int main ()

{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	Mat image;
	image = imread("scene_r.bmp", 0);
	Size s = image.size();
	int intensity = 0;
	int T = 128;


	for (int i = s.height; i>0; i--)
	{
		for(int j = s.width; j>0; j--)
		{
			unsigned char bgrPixel = image.at<uchar>(i,j);
			intensity = int(bgrPixel);

			if (intensity < T){
				image.at<uchar>(i,j) = 0;
			}

		}
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;


//	namedWindow("Display window", WINDOW_AUTOSIZE);
//	imshow( "Display window", image);
//	imwrite("Output.bmp",image);
//	waitKey(0);
	return 0;
	}
```

B.3 Grey Scale Threshold with Loop Unrolling

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;

int main ()

{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	Mat image;
	image = imread("scene_r.bmp", 0);
	Size s = image.size();
	int intensity = 0;
	int T = 128;

	// use s.width to discover how many times to unroll loop
//	cout << s.width << endl;

	for (int i = 0; i < s.height; i++)
	{

			unsigned char bgrPixel = image.at<uchar>(i,1);
			intensity = int(bgrPixel);
			if (intensity < T){
				image.at<uchar>(i,1) = 0;
			}

			 bgrPixel = image.at<uchar>(i,2);
			intensity = int(bgrPixel);
			if (intensity < T){
				image.at<uchar>(i,2) = 0;
			}

			 bgrPixel = image.at<uchar>(i,3);
			intensity = int(bgrPixel);
			if (intensity < T){
				image.at<uchar>(i,3) = 0;
			}

			 bgrPixel = image.at<uchar>(i,4);
			intensity = int(bgrPixel);
			if (intensity < T){
				image.at<uchar>(i, 4) = 0;
			}
```

```
                         bgrPixel = image.at<uchar>(i,5);
                        intensity = int(bgrPixel);
                        if (intensity < T){
                                image.at<uchar>(i,5) = 0;
                        }

                         bgrPixel = image.at<uchar>(i,6);
                        intensity = int(bgrPixel);
                        if (intensity < T){
                                image.at<uchar>(i,6) = 0;
                        }


………………………………………………CONTINUED TO THE VALUE 384

                        bgrPixel = image.at<uchar>(i,383);
                        intensity = int(bgrPixel);
                        if (intensity < T){
                                image.at<uchar>(i,383) = 0;
                        }

                         bgrPixel = image.at<uchar>(i,384);
                        intensity = int(bgrPixel);
                        if (intensity < T){
                                image.at<uchar>(i,384) = 0;
                        }



        }

        // Get finish time and find difference devided by frequency
        double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
        cout << time << endl;

        // Display and store image to check corectness
        //namedWindow("Display window", WINDOW_AUTOSIZE);
        //imshow( "Display window", image);
        //imwrite("Output.bmp",image);
        //waitKey(0);
        return 0;
        }
```

B.4 Grey Scale Threshold with Pointer Optimisation

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;

int main ()

{
	// Get start time
	int64 tickCount = getTickCount();

	// Load image and assign pointer
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;

	int T = 128;
	int size = image->imageSize;


	// Loop through matrix and change intensity
	for(int x = 0; x < size; x++)
	{
		if (*ptr < T){
		*ptr = 0;
		ptr++;
		}
		else {ptr++;}
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;

	// Used to test image output and save
	//cvNamedWindow("Display window", WINDOW_NORMAL);
	//cvShowImage( "Display window", image);
	//cvSaveImage("GreyScaleOpt.jpg",image);
	//cvWaitKey(0);

	cvReleaseImage(&image);
	return 0;
}
```

B.5 Grey Scale Threshold with Pointer Optimisation and Loop Unrolling

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;

int main ()

{
	// Get start time
	int64 tickCount = getTickCount();

	// Load image and assign pointer
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int T = 128;
	int size = image->imageSize;

	// Loop through matrix and change intensity
	for(int x = 0; x < size; x=x+6)
	{
		if (*ptr < T){
		*ptr = 0;
		ptr ++;
		}
		else {ptr ++;}

		if (*ptr < T){
		*ptr = 0;
		ptr ++;
		}
		else {ptr ++;}

		if (*ptr < T){
		*ptr = 0;
		ptr ++;
		}
		else {ptr ++;}

		if (*ptr < T){
		*ptr = 0;
		ptr ++;
		}
		else {ptr ++;}

		if (*ptr < T){
		*ptr = 0;
		ptr ++;
```

```
            }
            else {ptr ++;}

            if (*ptr < T){
            *ptr = 0;
            ptr ++;
            }
            else {ptr ++;}

        }

        // Get finish time and find difference devided by frequency
        double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
        cout << time << endl;

        //cvNamedWindow("Display window", WINDOW_NORMAL);
        //cvShowImage( "Display window", image);
        //cvWaitKey(0);
        //cvSaveImage("GreyScaleOptloop.bmp",image);
        cvReleaseImage(&image);
        return 0;
}
```

B.6 Grey Scale Threshold with Pointer Optimisation and Loop Reversal

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;

int main ()

{
	// Get start time
	int64 tickCount = getTickCount();

	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int T = 128;
	int size = image->imageSize;

	// Loop through matrix and change intensity
	for(int x = size; x >0; x--)
	{
		if (*ptr < T){
		*ptr = 0;
		ptr++;
		}
		else ptr++;
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;

	//cvNamedWindow("Display window", WINDOW_NORMAL);
	//cvShowImage( "Display window", image);
	//cvSaveImage("GreyScaleOptdec.bmp",image);
	//cvWaitKey(0);
	cvReleaseImage(&image);
	return 0;
}
```

B.7 Grey Scale Threshold with Pointer Optimisation directed through A7 core

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;

int main ()

{
	cpu_set_t my_set;
	CPU_ZERO(&my_set);
	CPU_SET(0, &my_set);
	sched_setaffinity(0, sizeof(cpu_set_t), &my_set);

	int64 tickCount = getTickCount();

	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int T = 128;
	int size = image->imageSize;

	for(int x = 0; x < size; x++)
	{
		if (*ptr < T){
		*ptr = 0;
		ptr++;
		}
		else {ptr++;}
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;

	//cvNamedWindow("Display window", WINDOW_NORMAL);
	//cvShowImage( "Display window", image);
	//cvSaveImage("GreyScaleOptA7.bmp",image);
	//cvWaitKey(0);
	cvReleaseImage(&image);
	return 0;
}
```

B.8 Grey Scale Threshold with Pointer Optimisation directed through A15 core

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>


using namespace cv;
using namespace std;

int main ()

{
        cpu_set_t my_set;
        CPU_ZERO(&my_set);
        CPU_SET(7, &my_set);
        sched_setaffinity(0, sizeof(cpu_set_t), &my_set);

        int64 tickCount = getTickCount();

        IplImage* image = cvLoadImage("scene_r.bmp",0);
        unsigned char *ptr= (unsigned char*)image ->imageData;
        int T = 128;
        int size = image->imageSize;

        for(int x = 0; x < size; x++)
        {
                if (*ptr < T){
                *ptr = 0;
                ptr++;
                }
                else {ptr++;}
        }

        // Get finish time and find difference devided by frequency
        double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
        cout << time << endl;

        //cvNamedWindow("Display window", WINDOW_NORMAL);
        //cvShowImage( "Display window", image);
        //cvSaveImage("GreyScaleOptA15.bmp",image);
        //cvWaitKey(0);
        cvReleaseImage(&image);
        return 0;
}
```

B.9 Grey Scale Threshold directed through Neon Data Engine

```
 #include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>
#include <arm_neon.h>

using namespace cv;
using namespace std;


int main ()

{
	// Get start time
	int64 tickCount = getTickCount();

	//  Download and store image
	IplImage* image = cvLoadImage("scene_r.bmp",0);

	// Assign pointer to image data
	uint8x16_t* __restrict ptr= (uint8x16_t *)image ->imageData;

	int size = image->imageSize;

	// Create Image to store result
	IplImage* result = cvCreateImage(cvGetSize(image),IPL_DEPTH_8U,1);
	uint8x16_t*__restrict optr = (uint8x16_t *)result ->imageData;

	uint8x16_t threshold = vdupq_n_u8(128);
	uint8x16_t mask;
	uint8x16_t output;

	for (; size > 0; size -=16, ptr++, optr++){
	mask = *ptr > threshold;
	*optr = *ptr & mask;
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;


	//namedWindow("Display window", WINDOW_NORMAL);
	//cvShowImage( "Display window", result);
	//cvSaveImage("GreyScaleOptNeon2.bmp",image);
	//waitKey(0);

	cvReleaseImage(&image);
	return 0;
	}
```

B.10 Grey Scale Threshold directed through Graphics Processing Unit

Image_scaling.cpp and image_scaling.cl from Mali OpenCL SDK v1.1.0 samples were altered to perform grey scale thresholding. Program has been marked with CM initials and dated where program code has been altered. Please note that comments have also been changed to reflect new purpose of code.

```
/*This confidential and proprietary software may be used only as
 * authorised by a licensing agreement from ARM Limited
 *    (C) COPYRIGHT 2013 ARM Limited
 *        ALL RIGHTS RESERVED
 * The entire notice above must be reproduced on all authorised
 * copies and copies may only be made to the extent permitted
 * by a licensing agreement from ARM Limited.
 */

#include "common.h"
#include "image.h"
#include "common.cpp"         //added CM 25/10/2015
#include "image.cpp"  //added CM 25/10/2015

#include <CL/cl.h>
#include <iostream>
#include <opencv2/opencv.hpp> //added CM 25/10/2015
#include <stdio.h>
#include <stdlib.h>

using namespace std;
using namespace cv;


int main(void)
{

    // Get start time added CM 25/10/2015
    int64 tickCount = getTickCount();

    cl_context context = 0;
    cl_command_queue commandQueue = 0;
    cl_program program = 0;
    cl_device_id device = 0;
```

```
    cl_kernel kernel = 0;
    const int numMemoryObjects = 2;
    cl_mem memoryObjects[numMemoryObjects] = {0, 0};
    cl_int errorNumber;

    // Set up OpenCL environment: create context, command queue, program
and kernel
    // Create context
    if (!createContext(&context))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed to create an OpenCL context. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }
    // Create Command Queue
    if (!createCommandQueue(context, &commandQueue, &device))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed to create the OpenCL command queue. " << __FILE__
<< ":"<< __LINE__ << endl;
        return 1;
    }
    // Create Program
    if (!createProgram(context, device, "assets/image_scaling.cl",
&program))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed to create OpenCL program." << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }
    // Create Kernel
    kernel = clCreateKernel(program, "image_scaling", &errorNumber);
    if (!checkSuccess(errorNumber))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed to create OpenCL kernel. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    // Load the input image data
        // accessing greyscale image through opencv added CM 25/10/2015
    IplImage* image = cvLoadImage("assets/input.bmp",0);
    cl_int height = image->height;
```

```
    cl_int width = image->width;
    unsigned char* input=(unsigned char*)image ->imageData;

   // Convert image to RGB added CM 25/10/2015
    unsigned char *rgbIn = new unsigned char[width * height * 3];
    luminanceToRGB(input, rgbIn, width, height);
    saveToBitmap("input.bmp", width, height, rgbIn);

    cl_image_format format;
    format.image_channel_data_type = CL_UNORM_INT8;
    format.image_channel_order = CL_RGBA;

    //Allocate memory for the input image that can be accessed by the CPU
and GPU.
    bool createMemoryObjectsSuccess = true;

    memoryObjects[0] = clCreateImage2D(context,
CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR, &format,
width, height, 0, NULL, &errorNumber);
    createMemoryObjectsSuccess &= checkSuccess(errorNumber);

    memoryObjects[1] = clCreateImage2D(context,
CL_MEM_WRITE_ONLY | CL_MEM_ALLOC_HOST_PTR, &format,
width, height, 0, NULL, &errorNumber);
    createMemoryObjectsSuccess &= checkSuccess(errorNumber);

    if (!createMemoryObjectsSuccess)
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed creating the image. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    size_t origin[3] = {0, 0, 0};
    size_t region[3] = {width, height, 1};
    size_t rowPitch;

    unsigned char* inputImageRGBA = (unsigned
char*)clEnqueueMapImage(commandQueue,  memoryObjects[0],
CL_TRUE, CL_MAP_WRITE, origin, region, &rowPitch, NULL, 0,
NULL, NULL, &errorNumber);
    if (!checkSuccess(errorNumber))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed mapping the input image. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
```

```
    }
    //Convert the input data from RGB to RGBA
    RGBToRGBA(rgbIn, inputImageRGBA, width, height);
    delete[] rgbIn;

    //Unmap the image from the host
    if (!checkSuccess(clEnqueueUnmapMemObject(commandQueue,
memoryObjects[0], inputImageRGBA, 0, NULL, NULL)))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed unmapping the input image. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    cl_float widthNormalizationFactor = 1.0f / width;
    cl_float heightNormalizationFactor = 1.0f / height;

    // Setup kernel argument
    bool setKernelArgumentsSuccess = true;
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 0,
sizeof(cl_mem), &memoryObjects[0]));
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 1,
sizeof(cl_mem), &memoryObjects[1]));
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 2,
sizeof(cl_float), &widthNormalizationFactor));
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 3,
sizeof(cl_float), &heightNormalizationFactor));
    if (!setKernelArgumentsSuccess)
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, 3);
        cerr << "Failed setting OpenCL kernel arguments. " << __FILE__ <<
":"<< __LINE__ << endl;
        return 1;
    }

            // set workDimensions
    const int workDimensions = 2;
    size_t globalWorkSize[workDimensions] = {width, height};

    // An event to associate with the kernel
    cl_event event = 0;

    // Enqueue the kernel
    if (!checkSuccess(clEnqueueNDRangeKernel(commandQueue, kernel,
workDimensions, NULL, globalWorkSize, NULL, 0, NULL, &event)))
    {
```

```
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed enqueuing the kernel. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    // Wait for kernel execution completion.
    if (!checkSuccess(clFinish(commandQueue)))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed waiting for kernel execution to finish. " << __FILE__
<< ":"<< __LINE__ << endl;
        return 1;
    }

    printProfilingInfo(event);
    // Release event object
    if (!checkSuccess(clReleaseEvent(event)))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed releasing the event object. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    size_t newRegion[3] = {width, height, 1};

    unsigned char* outputImage = (unsigned
char*)clEnqueueMapImage(commandQueue,  memoryObjects[1],
CL_TRUE, CL_MAP_READ, origin, newRegion, &rowPitch, NULL, 0,
NULL, NULL, &errorNumber);
    if (!checkSuccess(errorNumber))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);
        cerr << "Failed mapping the input image. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    unsigned char* outputImageRGB = new unsigned char[width * height *
3];
    RGBAToRGB(outputImage, outputImageRGB, width, height);

    saveToBitmap("output.bmp", width, height, outputImageRGB);

    delete[] outputImageRGB;
```

```
    cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numMemoryObjects);

    // Get finish time and find difference divided by frequency
    double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
//altered CM 25/10/2015
    cout << time << endl;        //altered CM 25/10/2015
    return 0;
}



// Kernel  Program

// Define a sampler
const sampler_t sampler = CLK_NORMALIZED_COORDS_TRUE |
CLK_ADDRESS_CLAMP | CLK_FILTER_LINEAR;


__kernel void image_scaling(__read_only image2d_t sourceImage,
                        __write_only image2d_t destinationImage,
                        const float widthNormalizationFactor,
                        const float heightNormalizationFactor)
{
    //Calculate the coordinates
    int2 coordinate = (int2)(get_global_id(0), get_global_id(1));
    float2 normalizedCoordinate = convert_float2(coordinate) *
(float2)(widthNormalizationFactor, heightNormalizationFactor);

    //Read from the source image
    float4 colour = read_imagef(sourceImage, sampler,
normalizedCoordinate);

     //Greyscale Threshold algorithm
            // added CM 26/10/2015

     if (colour.x < 0.50f){
     colour = (0.0f);
     }
     else {colour = colour;}

     //Write to the destination image
     write_imagef(destinationImage, coordinate, colour);

}
```

Appendix C

Average Smoothing Filter code for Image Processing

C.1 Average Smoothing Filter

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	Mat image;
	image = imread("scene_r.bmp", 0);
	Size s = image.size();
	unsigned char bgrPixel;
	int avg = 0;

	// Loop through matrix and change intesity accordingly
	for (int i = 1; i < s.height; i++)
	{
		for(int j = 1; j < s.width; j++)
		{

			for( int k = 0; k<=2; k++){
			    for (int l = 0; l<=2; l++){
			        avg = avg + image.at<uchar>(i+k,j+l);
			}
			}
			image.at<uchar>(i,j) = avg/9;
			avg = 0;
		}
	}

	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() -
	tickCount)/getTickFrequency())*1000;
	cout << time << endl;

	//namedWindow("Display window", WINDOW_NORMAL);
	//imshow( "Display window", image);
	//imwrite("Output.bmp",image);
	//waitKey(0);

	return 0;
	}
```

C.2 Average Smoothing Filter with Loop Reversal

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	Mat image;
	image = imread("scene_r.bmp", 0);
	Size s = image.size();
	unsigned char bgrPixel;
	int avg = 0;

	// Loop through matrix and change intesity accordingly
	for (int i = (s.height-1); i>0; i--)
	{
		for(int j = (s.width-1); j>0; j--)
		{

			for( int k = 2; k >= 0; k--){
			   for (int l = 2; l >= 0; l--){
			      avg = avg + image.at<uchar>(i-k,j-l);
			}
			}
			image.at<uchar>(i,j) = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;

	namedWindow("Display window", WINDOW_NORMAL);
	imshow( "Display window", image);
	imwrite("Output.bmp",image);
	waitKey(0);

	return 0;
	}
```

C.3 Average Smoothing Filter with Loop Unrolling

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	Mat image;
	image = imread("scene_r.bmp", 0);
	Size s = image.size();
	unsigned char bgrPixel;
	int avg = 0;

	// Loop through matrix and change intensity accordingly
	for (int i = 1; i < s.height; i++)
	{
		for(int j = 1; j < s.width; j++)
		{

			avg = image.at<uchar>(i,j);
			avg = avg + image.at<uchar>(i+2,j+2);
			avg = avg + image.at<uchar>(i+1,j+2);
			avg = avg + image.at<uchar>(i+0,j+2);
			avg = avg + image.at<uchar>(i+2,j+1);
			avg = avg + image.at<uchar>(i+1,j+1);
			avg = avg + image.at<uchar>(i+0,j+1);
			avg = avg + image.at<uchar>(i+2,j+0);
			avg = avg + image.at<uchar>(i+1,j+0);

			image.at<uchar>(i,j) = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() -
	tickCount)/getTickFrequency())*1000;
	cout << time << endl;
```

```
namedWindow("Display window", WINDOW_NORMAL);
imshow( "Display window", image);
imwrite("Output.bmp",image);
waitKey(0);

return 0;
}
```

C.4 Average Smoothing Filter with Pointer Optimisation

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int size = image->imageSize;
	int widthstep = image->widthStep;
	int height = image->height;
	int width = image->width;
	int step = widthstep/sizeof(uchar);
	int avg = 0;

	// Loop through matrix and change intesity accordingly
	for (int i = 1; i < height; i++)
	{
		for(int j = 1; j < width; j++)
		{
			for( int k = 0; k<=2; k++){
				for (int l = 0; l<=2; l++){
					avg = avg +
					(int)ptr[((i*step)+k)+(j+l)];
				}
			}
			ptr[i*step + j] = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() -
	tickCount)/getTickFrequency())*1000;
	cout << time << endl;


	//cvNamedWindow("Display window", WINDOW_NORMAL);
```

```
//cvShowImage( "Display window", image);
//cvSaveImage("GreyScaleOpt.bmp",image);
//cvWaitKey(0);

return 0;
}
```

C.5 Average Smoothing Filter with Pointer Optimisation and Loop Reversal

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int size = image->imageSize;
	int widthstep = image->widthStep;
	int height = image->height;
	int width = image->width;
	int step = widthstep/sizeof(uchar);
	int avg = 0;


	// Loop through matrix and change intesity accordingly
	for (int i = height-1; i >0; i--)
	{
		for(int j = width-1; j >0; j--)
		{
			for( int k = 2; k>=0; k--){
				for (int l = 2; l>=0;l--){
					avg = avg + (int)ptr[((i*step)-k)+(j-l)];
				}
			}
			ptr[i*step + j] = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() -
	tickCount)/getTickFrequency())*1000;
	cout << time << endl;
```

```
//cvNamedWindow("Display window", WINDOW_NORMAL);
//cvShowImage( "Display window", image);
//cvSaveImage("GreyScaleOpt.bmp",image);
//cvWaitKey(0);

cvReleaseImage(&image);

return 0;
}
```

C.6 Average Smoothing Filter with Pointer Optimisation and Loop Unrolling

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{
	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int size = image->imageSize;
	int widthstep = image->widthStep;
	int height = image->height;
	int width = image->width;
	int step = widthstep/sizeof(uchar);
	int avg = 0;


	// Loop through matrix and change intesity accordingly
	for (int i = 1; i < height; i++)
	{
		for(int j = 1; j < width; j++)
		{

			avg = avg + (int)ptr[((i*step))+(j)];
			avg = avg + (int)ptr[((i*step)+0)+(j+1)];
			avg = avg + (int)ptr[((i*step)+0)+(j+2)];
			avg = avg + (int)ptr[((i*step)+1)+(j+0)];
			avg = avg + (int)ptr[((i*step)+1)+(j+1)];
			avg = avg + (int)ptr[((i*step)+1)+(j+2)];
			avg = avg + (int)ptr[((i*step)+2)+(j+0)];
			avg = avg + (int)ptr[((i*step)+2)+(j+1)];
			avg = avg + (int)ptr[((i*step)+2)+(j+2)];
			ptr[i*step + j] = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;


	//cvNamedWindow("Display window", WINDOW_NORMAL);
```

```
//cvShowImage( "Display window", image);
//cvSaveImage("GreyScaleOpt.bmp",image);
//cvWaitKey(0);

cvReleaseImage(&image);

return 0;
}
```

C.7 Average Smoothing Filter with Pointer Optimisation directed through A7 Core

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{

	cpu_set_t my_set;
	CPU_ZERO(&my_set);
	CPU_SET(0, &my_set);
	sched_setaffinity(0, sizeof(cpu_set_t), &my_set);

	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int size = image->imageSize;
	int widthstep = image->widthStep;
	int height = image->height;
	int width = image->width;
	int step = widthstep/sizeof(uchar);
	int avg = 0;


	// Loop through matrix and change intesity accordingly
	for (int i = 1; i < height; i++)
	{
		for(int j = 1; j < width; j++)
		{
			for( int k = 0; k<=2; k++){
				for (int l = 0; l<=2; l++){
					avg = avg + (int)ptr[((i*step)+k)+(j+l)];
				}
			}
			ptr[i*step + j] = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;
```

```
//cvNamedWindow("Display window", WINDOW_NORMAL);
//cvShowImage( "Display window", image);
//cvSaveImage("GreyScaleOpt.bmp",image);
//cvWaitKey(0);

cvReleaseImage(&image);

return 0;
}
```

C.8 Average Smoothing Filter with Pointer Optimisation directed through A15 Core

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;


int main ()
{

	cpu_set_t my_set;
	CPU_ZERO(&my_set);
	CPU_SET(7, &my_set);
	sched_setaffinity(0, sizeof(cpu_set_t), &my_set);

	// Get start time
	int64 tickCount = getTickCount();

	// Read in image using mat function
	IplImage* image = cvLoadImage("scene_r.bmp",0);
	unsigned char *ptr= (unsigned char*)image ->imageData;
	int size = image->imageSize;
	int widthstep = image->widthStep;
	int height = image->height;
	int width = image->width;
	int step = widthstep/sizeof(uchar);
	int avg = 0;


	// Loop through matrix and change intesity accordingly
	for (int i = 1; i < height; i++)
	{
		for(int j = 1; j < width; j++)
		{
			for( int k = 0; k<=2; k++){
				for (int l = 0; l<=2; l++){
					avg = avg + (int)ptr[((i*step)+k)+(j+l)];
				}
			}
			ptr[i*step + j] = avg/9;
			avg = 0;
		}
	}


	// Get finish time and find difference devided by frequency
	double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
	cout << time << endl;
```

```
//cvNamedWindow("Display window", WINDOW_NORMAL);
//cvShowImage( "Display window", image);
//cvSaveImage("GreyScaleOpt.bmp",image);
//cvWaitKey(0);

cvReleaseImage(&image);

return 0;
}
```

C.9 Average Smoothing Filter directed through Neon Data Engine

```
#include <stdio.h>
#include <stdlib.h>
#include <opencv2/opencv.hpp>
#include <arm_neon.h>


using namespace cv;
using namespace std;



int main ()

{

        // Get start time
        int64 tickCount = getTickCount();

        //  Download and store image
        IplImage* image = cvLoadImage("scene_r.bmp",0);

        // Assign pointer to image data
        uint8_t * __restrict ptr= (uint8_t*)image ->imageData;
        uint8_t *start = ptr;

        int size = image->imageSize;
        int height = image->height;
        int width = image->width;

        uint16x8_t sum;

        // Create Image to store result
        IplImage* result =
cvCreateImage(cvGetSize(image),IPL_DEPTH_8U,1);
        uint8_t*  optr = (uint8_t *)result ->imageData;
        uint8_t *start2 = optr;

        // Initialise line3
        uint8x8_t line1;
        uint8x8_t line2;
        uint8x8_t line3;
        uint8x8_t final;


        // Fill first row;
        for ( int i = width;  i > 0;  i--,optr++, ptr++){
```

```
*optr = *ptr;
}

// Fill sides
for (int i = height; i > 0; i--, optr++, ptr++){
*optr = *ptr;
optr += (width-1);
ptr +=(width-1);
*optr = *ptr;
}

optr = optr+=width;
ptr = ptr+=width;

// Fill last row;
for ( int i = width;  i > 0;  i--,optr--, ptr--){
*optr = *ptr;
}

ptr = start;
optr = start2 + width;

// Calculate new pixel values
for (; (size-(width*2)) > 0; size -=6, ptr+=6){

line1 = vld1_u8(ptr);
line2 = vld1_u8(ptr + width);
line3 = vld1_u8(ptr + (width * 2));

sum = vaddl_u8(line1, line2);
sum = vaddw_u8(sum, line3);

optr++;
*optr = ((vgetq_lane_u16(sum, 0) + vgetq_lane_u16(sum, 1) +
vgetq_lane_u16(sum, 2))/9);
optr++;
*optr = ((vgetq_lane_u16(sum, 1) + vgetq_lane_u16(sum, 2) +
vgetq_lane_u16(sum, 3))/9);
optr++;
*optr = ((vgetq_lane_u16(sum, 2) + vgetq_lane_u16(sum, 3) +
vgetq_lane_u16(sum, 4))/9);
optr++;
*optr = ((vgetq_lane_u16(sum, 3) + vgetq_lane_u16(sum, 4) +
vgetq_lane_u16(sum, 5))/9);
optr++;
*optr = ((vgetq_lane_u16(sum, 4) + vgetq_lane_u16(sum, 5) +
vgetq_lane_u16(sum, 6))/9);
optr++;
*optr = ((vgetq_lane_u16(sum, 5) + vgetq_lane_u16(sum, 6) +
vgetq_lane_u16(sum, 7))/9);
```

```
}

// Get finish time and find difference devided by frequency
double time = ((getTickCount() -
tickCount)/getTickFrequency())*1000;
cout << time << endl;

// Display and check image
//namedWindow("Display window", WINDOW_NORMAL);
//cvShowImage( "Display window", result);
//cvSaveImage("GreyScaleOptNeon2.bmp",result);
//waitKey(0);

cvReleaseImage(&image);
return 0;
}
```

C.10 Average Smoothing Filter directed through Graphics Processing Unit

Fir_float.cpp and fir_float.cl from Mali OpenCL SDK v1.1.0 samples were altered to perform Average Smoothing filter function. Program has been marked with CM initials and dated where program code has been altered. Please note that comments have also been changed to reflect new purpose of code.

```
/*
 * This confidential and proprietary software may be used only as
 * authorised by a licensing agreement from ARM Limited
 *   (C) COPYRIGHT 2013 ARM Limited
 *       ALL RIGHTS RESERVED
 * The entire notice above must be reproduced on all authorised
 * copies and copies may only be made to the extent permitted
 * by a licensing agreement from ARM Limited.
 */

#include "common.h"
#include "image.h"
#include "common.cpp"                  // added CM 27/10/2015
#include "image.cpp"           // added CM 27/10/2015

#include <CL/cl.h>
#include <iostream>
#include <fstream>
#include <sstream>
#include <cstddef>
#include <cmath>
#include <opencv2/opencv.hpp>       // added CM 27/10/2015



using namespace std;
using namespace cv;            // added CM 27/10/2015

int main(void)
{
    // Get start time
    int64 tickCount = getTickCount(); //added CM 27/10/2015

    cl_context context = 0;
    cl_command_queue commandQueue = 0;
```

```
   cl_program program = 0;
   cl_device_id device = 0;
   cl_kernel kernel = 0;
   const unsigned int numberOfMemoryObjects = 2;                    //altered
CM 27/10/2015
   cl_mem memoryObjects[numberOfMemoryObjects] = {0, 0};    //altered
CM 27/10/2015
   cl_int errorNumber;

    // Set up OpenCL environment: create context, command queue, program
and kernel
   // Create context
   if (!createContext(&context))
   {
      cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
      cerr << "Failed to create an OpenCL context. " << __FILE__ << ":"<<
__LINE__ << endl;
      return 1;
   }
   // Create Command Queue
   if (!createCommandQueue(context, &commandQueue, &device))
   {
      cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
      cerr << "Failed to create the OpenCL command queue. " << __FILE__
<< ":"<< __LINE__ << endl;
      return 1;
   }
   // Create Program
   if (!createProgram(context, device, "assets/fir_float.cl", &program))
   {
      cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
      cerr << "Failed to create OpenCL program." << __FILE__ << ":"<<
__LINE__ << endl;
      return 1;
   }
   // Create Kernel
   kernel = clCreateKernel(program, "fir_float", &errorNumber);
   if (!checkSuccess(errorNumber))
   {
      cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
      cerr << "Failed to create OpenCL kernel. " << __FILE__ << ":"<<
__LINE__ << endl;
      return 1;
   }

   // Load the input image data
```

```
    // added to import greyscale image through opencv  CM 27/10/2015
    IplImage* image = cvLoadImage("assets/scene_r.bmp",0);
    cl_int height = image->height;
    cl_int width = image->width;

    unsigned char* input=(unsigned char*)image ->imageData;

    // buffersize
    size_t bufferSize = width * height * sizeof(float);


    bool createMemoryObjectsSuccess = true;
    memoryObjects[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
CL_MEM_ALLOC_HOST_PTR, bufferSize, NULL, &errorNumber);
    createMemoryObjectsSuccess &= checkSuccess(errorNumber);
    memoryObjects[1] = clCreateBuffer(context, CL_MEM_WRITE_ONLY
| CL_MEM_ALLOC_HOST_PTR, bufferSize, NULL, &errorNumber);
    createMemoryObjectsSuccess &= checkSuccess(errorNumber);
    if (!createMemoryObjectsSuccess)
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
        cerr << "Failed to create OpenCL buffers. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    cl_float* inputImageData =
(cl_float*)clEnqueueMapBuffer(commandQueue, memoryObjects[0],
CL_TRUE, CL_MAP_WRITE, 0, bufferSize, 0, NULL, NULL,
&errorNumber);
    if (!checkSuccess(errorNumber))
    {
       cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
       cerr << "Mapping memory objects failed " << __FILE__ << ":"<<
__LINE__ << endl;
       return 1;
    }

    //Converting data into float
    for(int i = 0; i < width * height; i++)
    {
        inputImageData[i] = (float)input[i] / 255.0f;
    }

    //Unmap memory
    if (!checkSuccess(clEnqueueUnmapMemObject(commandQueue,
memoryObjects[0], inputImageData, 0, NULL, NULL)))
    {
```

```
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
        cerr << "Unmapping memory objects failed " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    // Kernel
    bool setKernelArgumentsSuccess = true;
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 0,
sizeof(cl_mem), &memoryObjects[0]));
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 1,
sizeof(cl_mem), &memoryObjects[1]));
    setKernelArgumentsSuccess &= checkSuccess(clSetKernelArg(kernel, 2,
sizeof(cl_int), &width));
    if (!setKernelArgumentsSuccess)
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
        cerr << "Failed setting OpenCL kernel arguments. " << __FILE__ <<
":"<< __LINE__ << endl;
        return 1;
    }

    // even information
    cl_event event = 0;

    //Kernel size
    size_t globalWorksize[2] = {width / 4, height / 1};


    // Enque the kernel
    if (!checkSuccess(clEnqueueNDRangeKernel(commandQueue, kernel, 2,
NULL, globalWorksize, NULL, 0, NULL, &event)))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
        cerr << "Failed enqueuing the kernel. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    // Wait for kernel execution completion
    if (!checkSuccess(clFinish(commandQueue)))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
        cerr << "Failed waiting for kernel execution to finish. " << __FILE__
<< ":"<< __LINE__ << endl;
        return 1;
```

```
    }

    // Get finish time and find difference devided by frequency

    /* Print the profiling information for the event. */
    printProfilingInfo(event);
    /* Release the event object. */
    if (!checkSuccess(clReleaseEvent(event)))
    {
        cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
        cerr << "Failed releasing the event object. " << __FILE__ << ":"<<
__LINE__ << endl;
        return 1;
    }

    /* Map the output memory to a host side pointer. */
    cl_float* output = (cl_float*)clEnqueueMapBuffer(commandQueue,
memoryObjects[1], CL_TRUE, CL_MAP_READ, 0, bufferSize, 0, NULL,
NULL, &errorNumber);
    if (!checkSuccess(errorNumber))
    {
       cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
       cerr << "Mapping memory objects failed " << __FILE__ << ":"<<
__LINE__ << endl;
       return 1;
    }

    /* Convert the float output to unsigned char for saving to bitmap. */
    unsigned char *outputData= new unsigned char[width * height];
    for(int i = 0; i< width * height; i++)
    {
        outputData[i] = (unsigned char)(output[i] * 255.0f);
    }

    /* Unmap the output. */
    if (!checkSuccess(clEnqueueUnmapMemObject(commandQueue,
memoryObjects[1], output, 0, NULL, NULL)))
    {
       cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
       cerr << "Unmapping memory objects failed " << __FILE__ << ":"<<
__LINE__ << endl;
       return 1;
    }

    /* Release OpenCL objects. */
    cleanUpOpenCL(context, commandQueue, program, kernel,
memoryObjects, numberOfMemoryObjects);
```

```
   /* Convert the output luminance array to RGB and save it out to a file. */
   unsigned char *rgbOut = new unsigned char[width * height * 3];
   luminanceToRGB(outputData, rgbOut, width, height);
   delete [] outputData;

         // Get execution time in ms and output
   double time = ((getTickCount() - tickCount)/getTickFrequency())*1000;
                  //added CM 27/10/2015
   cout << time << endl;
                                                                  //added CM 27/10/2015

   saveToBitmap("output.bmp", width, height, rgbOut);
   delete [] rgbOut;


   return 0;
}



// Kernel__kernel void fir_float(__global const float* restrict input,
                  __global float* restrict output,
                  const int width)
{

   const int column = get_global_id(0) * 4;
   const int row = get_global_id(1);

   /* Offset calculates the position in the linear data for the row and the
column. */
   const int offset = row * width + column;


   /* Accumulator array of 4 floats. */
   float4 accumulator = (float4)0.0f;


   // Load first row
   /*
    * Access the first row in the 6x3 window
    * data1 can be constructed from the other vectors without doing an
additional load.
    */
   float4 data0 = vload4(0, input + offset);    //altered CM 27/10/2015

   float4 data2 = vload4(0, input + offset + 2);        //altered CM
27/10/2015
   float4 data1 = (float4)(data0.s12, data2.s12);       //altered CM
27/10/2015
```

```
    accumulator += data0;
    accumulator += data1;
    accumulator += data2;
    /* [Filter first row] */

    /* [Load and filter second and third row] */
    // Access the second row in the 6x3 window and repeat the process
    data0 = vload4(0, input + offset + width);             //altered CM
27/10/2015
    data2 = vload4(0, input + offset + width + 2);        //altered CM
27/10/2015
    data1 = (float4)(data0.s12, data2.s12);                            //altered CM
27/10/2015

    accumulator += data0;
    accumulator += data1;
    accumulator += data2;

    //Access the third row in the 6x3 window and repeat the process
    data0 = vload4(0, input + offset + width * 2);        //altered CM
27/10/2015
    data2 = vload4(0, input + offset + width * 2 + 2); //altered CM
27/10/2015
    data1 = (float4)(data0.s12, data2.s12);               //altered CM
27/10/2015

    accumulator += data0;
    accumulator += data1;
    accumulator += data2;

   //altered CM 27/10/2015
   // divide accumulator by 9
    accumulator = accumulator/9;

          /* [Store] */
    /* Store the accumulator. */
    vstore4(accumulator, 0, output + offset);
    /* [Store] */
}
```

Appendix D

Output photos for Grey scale algorithm

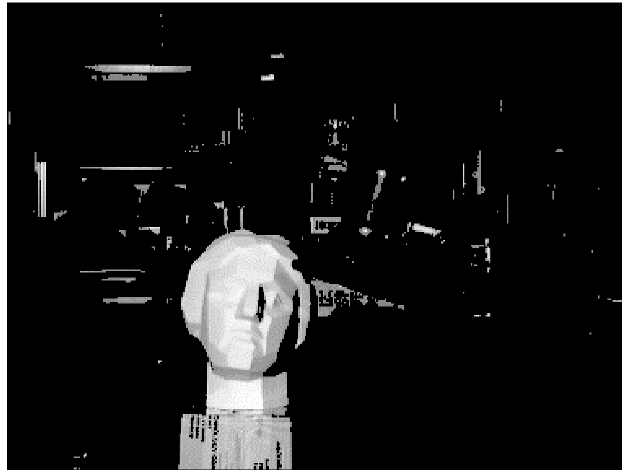


Figure D.1 Grey scale program output photo

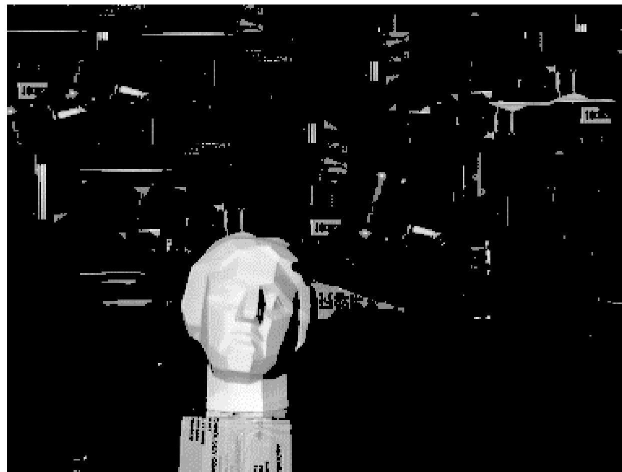


Figure D.2 Grey scale program with loop reversal output photo

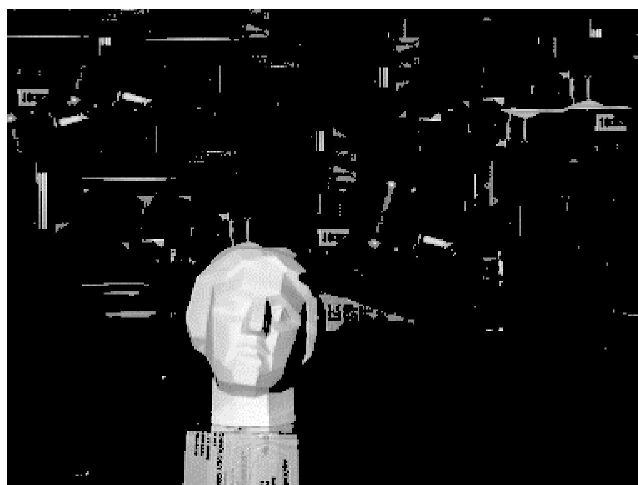


Figure D.3 Grey scale program with loop unrolling output photo

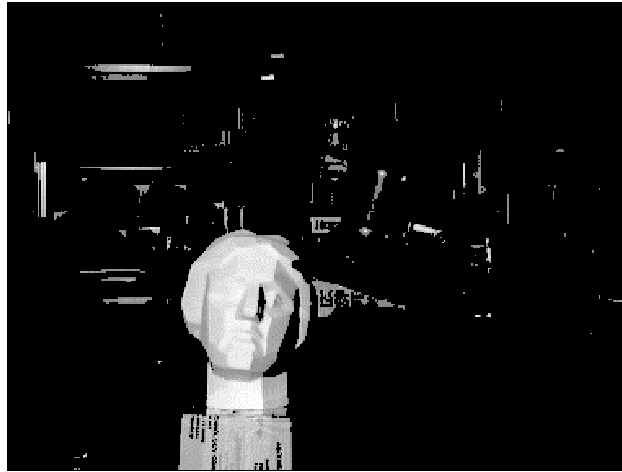


Figure D.4 Grey scale program with pointer optimisation output photo

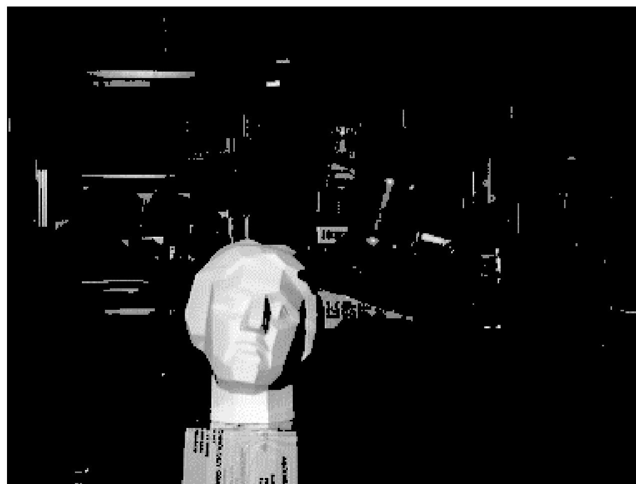


Figure D.5 Grey scale program with pointer optimisation and loop reversal output photo

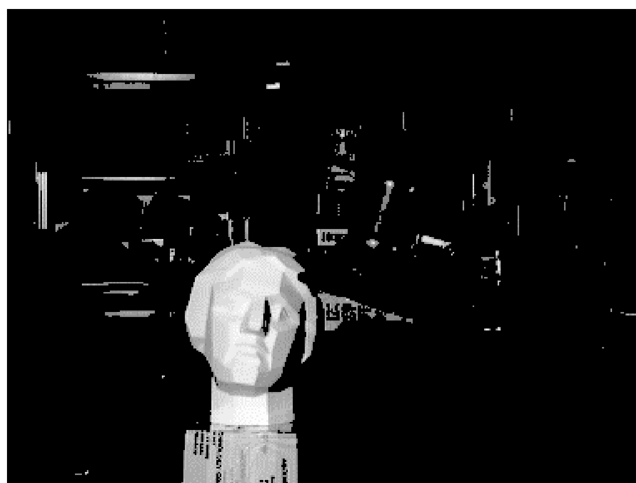


Figure D.6 Grey scale program with pointer optimisation and loop unrolling output photo

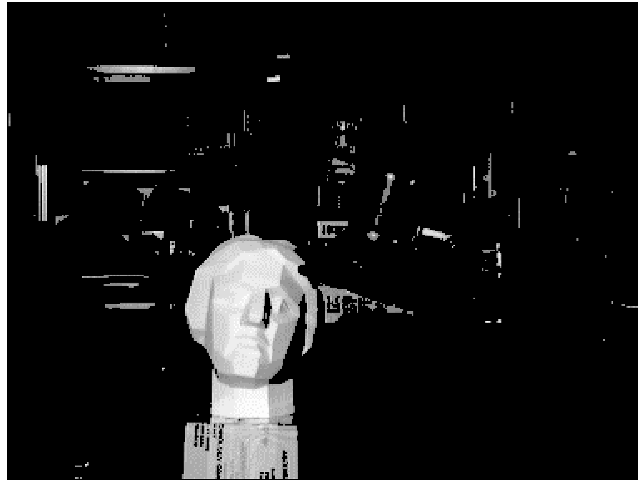


Figure D.7 Grey scale program with pointer optimisation and directed through A7 core output photo

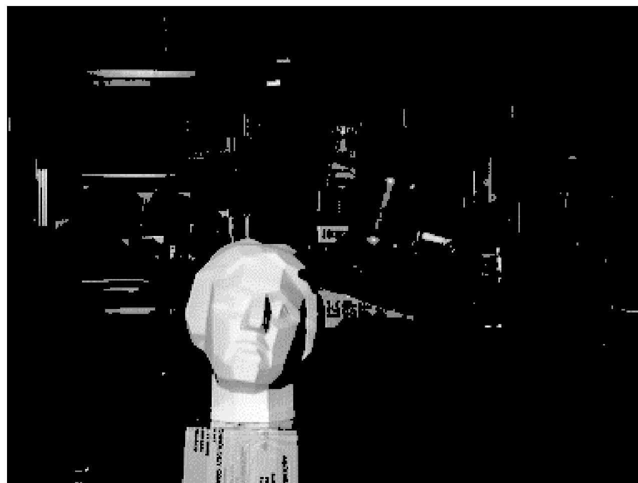


Figure D.8 Grey scale program with pointer optimisation and directed through A15 core output photo

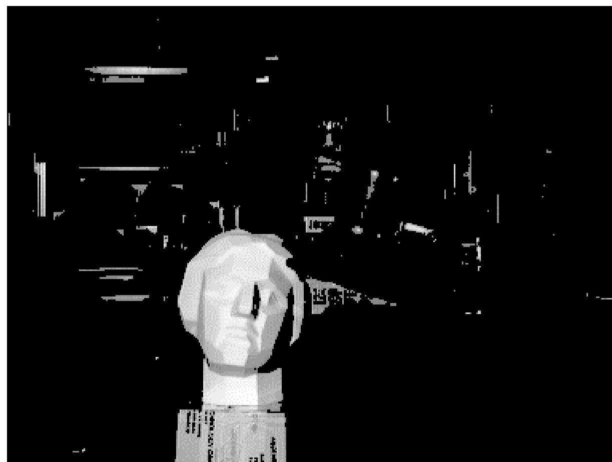


Figure D.9 Grey scale program directed through Neon data engine output photo

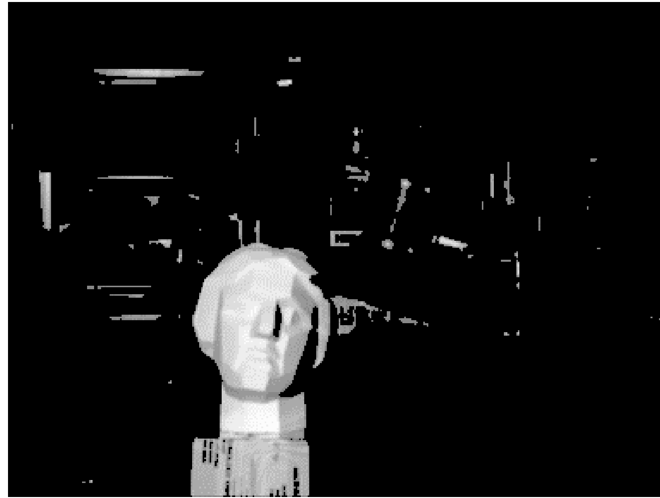


Figure D.10 Grey scale program directed through Graphics processing unit output photo

Appendix E

Output photos for Averaging Smoothing Filter

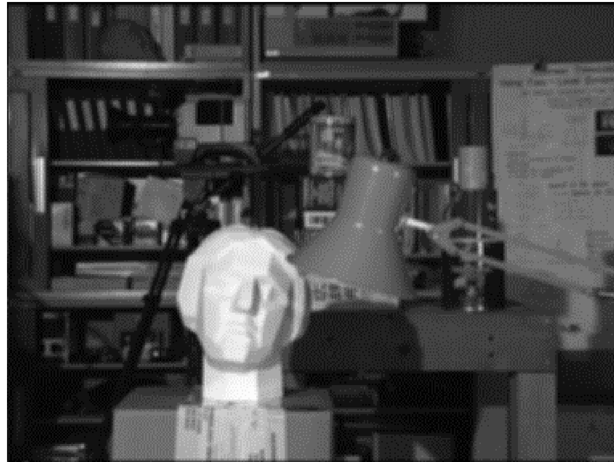


Figure E.1 Smoothing Filter program output photo

Figure E.2 Smoothing Filter program with loop reversal output photo

Figure E.3 Smoothing filter program with loop unrolling output photo

Figure E.4 Smoothing filter program with pointer optimisation output photo

Figure E.5 Smoothing filter program with pointer optimisation and loop reversal output photo

Figure E.6 Smoothing filter program with pointer optimisation and loop unrolling output photo

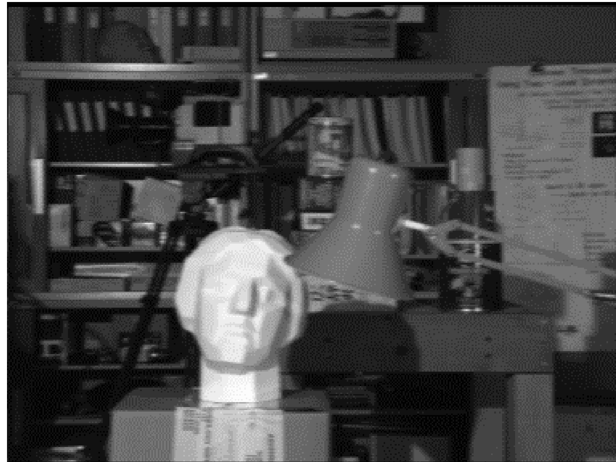


Figure E.7 Smoothing filter program with pointer optimisation and directed through A7 output photo

Figure E.8 Smoothing filter program with pointer optimisation and directed through A15 output photo

Figure E.9 Smoothing filter program directed through Neon data engine output photo

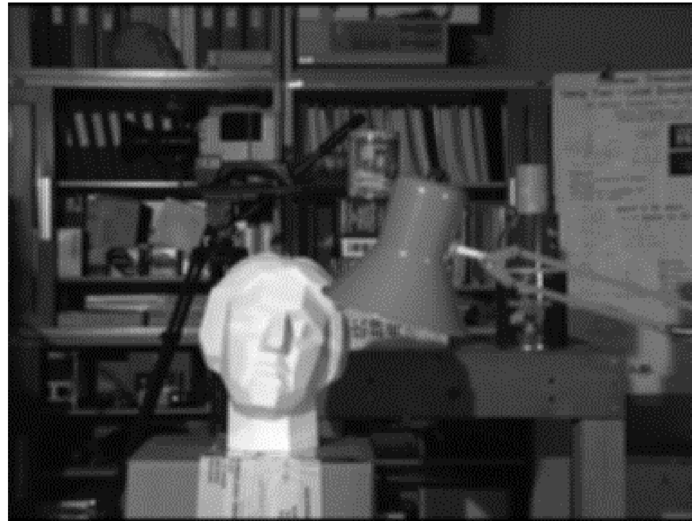


Figure W.10 Smoothing filter program directed through Graphics processing unit output photo