University of
HUDDERSFIELD

University of Huddersfield Repository

Brotherton, Mark

GPU Accelerated X-Ray Image Enhancement

Original Citation

Brotherton, Mark (2011) GPU Accelerated X-Ray Image Enhancement. Masters thesis, University of Huddersfield.

This version is available at http://eprints.hud.ac.uk/11044/

The University Repository is a digital collection of the research output of the University, available on Open Access. Copyright and Moral Rights for the items on this site are retained by the individual author and/or other copyright owners. Users may access full items free of charge; copies of full text items generally can be reproduced, displayed or performed and given to third parties in any format or medium for personal research or study, educational or not-for-profit purposes without prior permission or charge, provided:

- The authors, title and full bibliographic details is credited in any copy;
- A hyperlink and/or URL is included for the original metadata page; and
- The content is not changed in any way.

For more information, including our policy and submission procedure, please contact the Repository Team at: E.mailbox@hud.ac.uk.

http://eprints.hud.ac.uk/

GPU Accelerated X-Ray Image Enhancement

Mark James Brotherton

A thesis submitted to the University of Huddersfield
in partial fulfilment of the requirements for
the degree of Masters of Science by Research

The University of Huddersfield

April 2011

Copyright Statement

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Huddersfield the right to use such copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions

Abstract

This paper presents an automated method for preparing digital X-rays for use by a procedural mesh generator. This process will facilitate the generation of a 3D polygon mesh depicting the bones contained within the X-ray image. The process of preparing the image involves identifying and retaining bone elements whilst removing any superfluous aspects contained within the image, such as text and orientation markers. This will allow a virtual podiatric surgical simulator, VirtuOrtho to feature patient specific bone models generated from the processed X-rays. The algorithm additionally employs parallel processing techniques that are capable of utilizing either multi-core CPUs or a GPU to help to reduce the computational time required to process an image. The relative performance of the multi-core CPU implementation will be compared and contrasted to the performance of the GPU version. As part of the image processing algorithm, two GPGPU algorithms for median filter are proposed: a caching method improves upon the fast, small-radius median filter (McGuire, 2008) and a second method which uses a histogram based technique (Huang et al., 1979) to facilitate filters with large-radius masks to be processed using the GPU.

Keywords: Image Processing, GPGPU, Parallel Processing, X-Ray, Median Filtering

Acknowledgements

The author would like to thank to Andrew Boothroyd, Damian De Luca, Daniel Fitchie, Dr Duke Gledhill, Dr Jim Picard and Ruth Taylor for their help and support with this project.

Table of Contents

Copyright Statement 2

Abstract 3

Acknowledgements 4

List of Figures 9

List of Abbreviations 13

Glossary 15

1. Introduction 21

1.1. Overview 21

1.2. VirtuOrtho 22

1.3. Problem Statement 23

1.4. Project Aims 26

1.4.1. Algorithm Objectives 26

1.4.2. Implementation Objectives 27

1.4.3. Median Filter Objectives 28

1.4.4. Expected Results 28

2. Literature Review 30

2.1. Research Objectives 30

2.1.1. Parallel Processing 30

2.1.2. Image Processing and Analysis 31

2.2. Parallel Processing 32

2.2.1. Background 32

2.2.2. General-Purpose Computing on Graphics Processing Units 34

2.2.3. Parallel Processor Architectures 35

2.2.4. Parallel Algorithm Design 40

2.2.5. Parallel Processing APIs 45

2.3. Image Processing 51

2.3.1. Medical Imaging 51
2.3.2. Applicable Image Processing Techniques 54
2.4. Conclusions 59
3. Methodology 61
3.1. Overview 61
3.2. Implementation 61
3.3. Data Gathering 61
3.3.1. Algorithm Experiments 61
3.4. Results Analysis 63
4. Implementation 64
4.1. Overview 64
4.2. General Algorithm Description 64
4.2.1. Median Filter 65
4.2.2. Histogram and Image Thresholding 66
4.2.3. Sobel Edge Detection 68
4.2.4. Active Contour Model Feature Extraction 68
4.2.5. Threshold Mask 68
4.3. SISD Implementation 68
4.4. MIMD Implementation 68
4.4.1. Median Filtering 69
4.4.2. Histogram and Thresholding 70
4.4.3. Sobel 75
4.4.4. Active Contour Model Feature Extraction 75
4.4.5. Threshold Mask 76
4.5. SIMD Implementation 76
4.5.1. Median Filtering 76
4.5.2. Histogram and Thresholding 81
4.5.3. Sobel 82

4.5.4. Active Contour Model Feature Extraction 82

4.5.5. Threshold Mask 83

4.6. Development Issues 83

4.6.1. Limited Data Types Available on GPU 83

4.6.2. Reduced Functionality in CS4.0 83

4.6.3. Other Issues 84

4.6.4. Hardware Specific Optimisations 85

5. Results and Analysis 87

5.1. Overall Performance 87

5.2. Individual Component Performance 88

5.3. Data Format Performance 90

5.4. Median Filtering Performance 91

5.5. Optimum Number of Threads 94

5.6. Average Image Error 96

5.7. Image Comparison 97

6. Conclusions 103

6.1. Overview 103

6.1.1. Individual Project Aims 103

6.2. Parallel Processing 104

6.2.1. Multi-core CPU 104

6.2.2. GPGPU 105

6.3. Algorithm 107

6.4. Summary 107

7. Future Work 108

7.1. Active Contour Model Feature Extraction 108

7.2. GPU Median Filter 108

7.3. X-Ray Header Information 109

7.4. Integration 110

7.5. Generating Skin Mesh 110

7.6. Parallel Mesh Generation Pipeline. 110

7.7. Benchmarking 110

7.8. Alternative GPGPU Implementations 111

8. References 112

9. Appendices 119

9.1. Appendix A: Hardware Statistics 119

9.2. Appendix B: Test Hardware 121

9.3. Appendix C: YEF Proposal 122

9.4. Appendix D: Performance Results Data 125

9.4.1. Overall Execution Time 125

9.4.2. Individual Component Execution Time 125

9.4.3. Data Format Performance 126

9.4.4. Median Filter Performance 126

9.4.5. Optimum Number of Threads 127

9.5. Appendix E: Source Code 128

9.5.1. DirectCompute Fast, Small-Radius Median Filter 128

9.5.2. GPU Histogram Median Filter 129

9.5.3. GPU Sobel Filter 130

9.6. Appendix F: Compute Shader Functionality 131

Word Count: 24,968

List of Figures

Figure 1 : VirtuOrtho 22
Figure 2 : SensAble PHANTOM® Omni 23
Figure 3 : Lateral Foot X-ray (Original) 24
Figure 4 : Lateral Foot X-ray with Unwanted Elements Highlighted 25
Figure 5 : Planar Foot X-Ray (Original) 25
Figure 6 : Planar Foot X-Ray with Unwanted Elements Highlighted 26
Figure 7 : Intel Core i7 Multi-core CPU Architecture 36
Figure 8 : AMD HD 5870 GPU Architecture 38
Figure 9 : Nvidia GeForce GTX 580 GPU Architecture 39
Figure 10 : GPU Branch Operation 40
Figure 11 : Amdahl's Law 43
Figure 12 : Dispatch Call 49
Figure 13 : Proposed Method for Processing an X-ray Image 64
Figure 14 : Anticipated Algorithm Output 65
Figure 15 : Typical X-Ray Image Histogram 67
Figure 16 : X-ray Image Histogram with Secondary Background 67
Figure 17 : Mask Overlaps between Neighbouring Pixels 78
Figure 18 : Caching 3x3 Median Filter Diagram 79
Figure 19 : Algorithm Performance Results 87
Figure 20 : Individual Component Performance Results (Overall) 89
Figure 21 : Individual Component Performance Results (In Detail) 89
Figure 22 : Data Format Performance 91
Figure 23 : Median Filter Performance Results 92
Figure 24 : Median Filter Performance Results (Large-radius) 93
Figure 25 : Optimum Number of Threads 95
Figure 26 : Algorithm Final Output - Integer (Lateral) 97
Figure 27 : Algorithm Final Output - Float (Lateral) 98
Figure 28 : Image Differences Integer CPU – Integer GPU (Lateral) 98
Figure 29 : Image Differences Integer – Floating Point (Lateral) 99
Figure 30 : Algorithm Final Output - Integer (Planar) 99
Figure 31 : Algorithm Final Output - Float (Planar) 100
Figure 32 : Image Differences Integer – Floating Point (Planar) 100
Figure 33 : Image Differences Integer CPU – Integer GPU (Planar) 100

Figure 34 : Improved Caching Median Filter..109

List of Tables

Table 1 : Source Code Formatting ... 12

Table 2 : Flynn's Taxonomy ... 33

Table 3 : Overall Algorithm Speedup ... 87

Table 4 : Individual Component Speedup ... 90

Table 5 : Median Filter Speedup ... 94

Table 6 : Number of Errors ... 96

Table 7 : Steam Hardware Survey, Number of CPU cores ... 119

Table 8 : Steam Hardware Survey, Advanced CPU Feature Support ... 119

Table 9 : Steam Hardware Survey, DirectX 11 Graphics Cards ... 120

Table 10 : Test Hardware Configuration ... 121

Table 11 : Visual Studio 2010 C++ Compiler Settings ... 121

Table 12 : YEF Funding Application Form ... 122

Table 13 : Overall Execution Time ... 125

Table 14 : Individual Component Execution Time ... 125

Table 15 : Data Format Performance ... 126

Table 16 : Median Filter (3 x 3) Performance ... 126

Table 17 : Median Filter (19 x 19) Performance ... 126

Table 18 : Optimum Number of Threads (Median) ... 127

Table 19 : Optimum Number of Threads (Sobel) ... 127

Table 20 : Compute Shader Functionaility ... 131

List of Equations

Equation 1 : AMD HD 5770 Processing Cores 39
Equation 2 : nVidia GTX 580 Processing Cores 39
Equation 3 : Amdahl's Law – Overall Speedup 42
Equation 4 : Amdahl's Law - Speedup 42
Equation 5 : Gustafson's Law 43
Equation 6 : Parallel Speedup 44
Equation 7 : Maximum Speedup 44
Equation 8 : Parallel Algorithm Efficiency 44
Equation 9 : Calculating Parallel Portion 53
Equation 10 : Amdahl’s Law for HyperThreading 53
Equation 11 : Work per SIMD Engine 95

Code Listings

Code Listing 1 : OpenMP Loop Declaration 46
Code Listing 2 : Example DirectCompute Shader..... 48
Code Listing 3 : DirectCompute Dispatch Function 50
Code Listing 4 : Histogram Based Median Filter Algorithm 55
Code Listing 5 : Sorting Based Median Filter Algorithm 56
Code Listing 6 : OpenMP Synchronisation Constructs..... 69
Code Listing 7 : OpenMP Median Filter 70
Code Listing 8 : OpenMP Reduction Operation 71
Code Listing 9 : OpenMP Histogram Calculation 72
Code Listing 10 : OpenMP Histogram Calculation via Atomic Operations 73
Code Listing 11 : OpenMP Threshold Operation 74
Code Listing 12 : OpenMP Sobel Filter..... 75
Code Listing 13 : Compute Shader 5.0 Caching Median Filter..... 80
Code Listing 14 : HLSL Median Calculation 81
Code Listing 15 : GPU Threshold Filter..... 82
Code Listing 16 : Compute Shader 4.0 Caching Median Filter..... 85
Code Listing 17 : GPU Fast, Small-Radius Median Filter 128
Code Listing 18 : GPU Histogram Median Filter..... 129
Code Listing 19 : GPU Sobel Filter..... 130

Source Code Formatting

Note: The source is written in C++ or HLSL.

Table 1 : Source Code Formatting

	Format	Notes
Line Number	`1:`	If a line of code is longer that the available space, it will wrap beneath and not have a line number.
Data Types	`int variable`	
Intrinsic Functions	`if(value)`	
Compiler Directives	`#define`	
Comments	`//Comment`	
Array Operator	`array[index]`	
End of Instruction	`;`	A single line of code may have multiple instructions.
Function Call	`Function();`	

List of Abbreviations

2D: Two Dimensional

3D: Three Dimensional

AAM: Active Appearance Model

ACM: Active Contour Model

API: Application Programming Interface

ALU: Arithmetic Logic Unit

ASM: Active Shape Model

AVX: Advanced Vector Extensions

CPU: Central Processing Unit

CS4.0: Compute Shader 4.0

CS5.0: Compute Shader 5.0

CT: Computed Tomography

CUDA: Compute Unified Device Architecture

DICOM: Digital Imaging and Communications in Medicine

Float: Single Precision Floating Point

GPU: Graphics Processing Unit

GPGPU: General-Purpose computing on Graphics Processing Units

HLSL: High Level Shading Language

HT: HyperThreading

half: Half Precision Floating Point

int: Signed Integer

L1: Level 1

L2: Level 2

L3: Level 3

MIMD: Multiple Instruction Multiple Data

MISD: Multiple Instruction Single Data

MRI: Magnetic Resonance Imaging

OS: Operating System

PS: Pixel Shader

R16F: Red 16bit Float

R32F: Red 32bit Float

RAM: Random Access Memory

SIMD: Single Instruction Multiple Data

SISD: Single Instruction Single Data

SSE: Streaming SIMD Extensions

uint: Unsigned Integer

ushort: Unsigned Short Integer

VS: Vertex Shader

Glossary

Active Appearance Model (AAM): Active Appearance Models are an improvement ASM method of feature extraction. Like ASMs they track contours but also use the texture surrounding the contours to reduce the number of landmarks required.

Active Contour Model (ACM): Active Contour Models or “Snakes” is a method of feature extraction that tracks contours contained in an image by moving a set of points until they enclose the target feature.

Active Shape Model (ASM): Active Shape Models are an improvement of the popular ACM method of feature extraction. ASMs locate objects by being trained using a number of images with marked landmark points covering the possible shape variations.

Advanced Vector Extensions (AVX): Advanced Vector Extensions is an improved version of the SSE instruction set, capable of processing larger amounts of data (256 bits instead of 128 bits) and instructions with 3-operands rather than 2-operand instructions available with SSE.

Application Programming Interface (API): An API is an interface which a program can use to obtain access to functionality provided by procedures, functions and classes from either the operating system or another application.

Arithmetic Logic Unit (ALU): The Arithmetic Logic Unit is the part of a processor that performs arithmetic and logic operations on data. The individual processing cores of a GPU’s SIMD engine are typically referred to as ALUs.

Atomic Operations: Is an operation that is executed in such a manner that only a single thread can alter the item of data at a time.

Benchmarking: Benchmarks are designed to measure the real-world computational performance of a given system by mimicking the typical workload a system is expected to perform.

C: Is a procedural, high level programming language.

C++: Is an object oriented, high level programming language based on the C language.

Cache Memory: Cache memory is a type of high speed memory that can be used to store frequently used data, so that it can be retrieved without having to access main memory. In consumer hardware cache memory is available in three types: Level 1 (L1), Level 2 (L2) and Level 3 (L3).

Note: For the purposes of this report, the term cache will be used to refer to cache memory contained in each SIMD engine in the case of a GPU and Shared Level 3 Cache Memory in CPUs, unless otherwise stated.

Central Processing Unit (CPU): The CPU is a general purpose processor in a computer which controls all other systems in the computer.

Compute Shader: A GPU program for performing calculations on general data rather than vertices or pixels. It is currently available as two main versions, CS4.0 for DirectX 10 compliant hardware and CS5.0 which contains additional functionality and other enhancements for DirectX 11 GPUs.

Compute Unified Device Architecture (CUDA): CUDA is a GPGPU API developed by nVidia; it is programmed in a C Style language. It currently only supports certain models of nVidia Graphics Cards (NVIDIA, 2010b).

Computed Tomography (CT): A technique that is capable of generating a volumetric (3D) image of an object using a series of two-dimensional X-ray images taken about a single axis.

Core: A core is the part of the processor that is capable of reading and executing instructions on data. A single-core processor can process and execute only one instruction at a time, whereas a multi-core processor can execute one instruction per core.

Note: For the purposes of this report, the term core or processing core will be used to refer to a single physical processing unit in the context of CPUs. When used in the context of GPUs it will be used to refer to a single processor within a SIMD engine.

Digital Imaging and Communications in Medicine (DICOM): DICOM is a file format for storing Digital Medical Images including X-rays, MRI and CT scans.

DirectCompute: Microsoft's DirectCompute is a GPGPU API available as part of the DirectX 11 package of APIs. Programs written with the API can be executed by DirectX 10 and 11 compliant graphics processors.

DirectX: DirectX is a collection of APIs for handling game programming and other related tasks. It provides capabilities for 3D graphics rendering, audio and video amongst others. The latest version of DirectX is version 11 but version 9.0c is still very popular amongst game developers.

General-Purpose computing on Graphics Processing Units (GPGPU): A technique for utilising the processing capabilities of the Graphics Card to process general computing problems rather than the typical graphics related ones.

Global Memory: This term is used to distinguish between the "main" random access memory incorporated into a GPU and the cache memory available to each thread group. The term "global memory" is used specifically in the context of GPUs and does not refer to the "main" random access memory of the computer.

Graphics Card: See Graphics Processing Unit.

Graphics Processing Unit (GPU): The GPU is a specialised processor primarily designed for accelerating graphics related data processing to provide real-time 3D rendering.

Haptics: Is a technology used provide to tactile feedback to a user by the application of forces, vibrations, and/or motions.

High Level Shading Language (HLSL): HLSL is a C style programming language used to create shaders for use with Microsoft's DirectX API, including Compute Shaders.

Histogram: A histogram is an array of numbers (bins), where each bin corresponds to the frequency count of a range of values contained within an image associated with that particular bin.

Instruction Set: An instruction set is a list containing all the instructions a processor compliant with that particular instruction set can execute.

Integer (int): A data type available in C++ which is used to store integer values using 32bits of memory. It is capable of storing a whole number value between -2,147,483,648 and 2,147,483,647.

Magnetic Resonance Imaging (MRI): Is a medical imaging method used to visualize internal structures in great detail. It especially useful in producing images of the brain, muscles, heart, etc. compared with other medical imaging techniques such as X-rays.

Mesh: A polygon mesh is a collection of vertices, edges and faces that define a polyhedral object in 3D computer graphics.

Multi-core: Multi-core is a term typically used to describe CPUs which contain more than one processing core.

Multiple Instruction Multiple Data (MIMD): Is a parallel processing architecture which has a number of processors that are capable of functioning asynchronously and independently of each other. The data and instructions being processed by a particular processor does not necessarily have to be related in any way to those being processed by different processor.

Multiple Instruction Single Data (MISD): Is an uncommon parallel processing architecture which executes multiple instructions simultaneously on a single item of data, typically it is used for fault tolerant computing.

Open Computing Language (OpenCL): OpenCL is designed to be a platform independent implementation of a GPGPU API.

Open Graphics Language (OpenGL): Is a cross-language and cross-platform API for writing applications that produce 2D and 3D computer graphics. It fulfils the same roles as the Direct3D API does in DirectX

Open Multi-Processor (OpenMP): OpenMP is designed to be a platform independent API which provides a lightweight method of processing data in parallel on multi-processor systems.

Pixel Shader: A GPU program for performing calculations on pixels. It is typically used to calculate the colour and any lighting effects on a 3D mesh.

Polygon: A polygon is the basic building block for constructing 3D meshes in computer graphics. It is a triangular surface and contains three vertices describing the location of its corners.

R16F: A texture format that has a single 16bit floating point value red colour channel.

R32F: A texture format that has a single 32bit floating point value red colour channel.

Random Access Memory (RAM): Is a type of memory that can retrieve a piece of data in a constant time, regardless of its physical location in memory. RAM is also referred to as "Main Memory"

Salt and Pepper Noise: Salt and Pepper noise manifests itself as random speckles within an image.

Scalar Processor: A processor architecture can execute a single instruction on a single item of data at once.

Shader: A program designed to be executed on a GPU, typically used for lighting calculations.

Shader Model: The shader model specifies what HLSL instruction set is supported by the GPU. The lowest and most restrictive model being Shader Model 1, with Shader Model 5 currently being highest and most feature rich.

SIMD Engine: SIMD engine refers to the SIMD processors utilised by GPU architectures. A SIMD engine consists of a number of processing units which process data using a SIMD architecture.

Hardware vendors use different terminology when referring to SIMD processors. AMD use the term SIMD engine whilst Nvidia uses CUDA Cores.

Note: For the purposes of this report, the term "SIMD engine" will be used when referring to SIMD processors for both hardware vendors, as AMD GPUs are the main focus of this report.

Single Instruction Multiple Data (SIMD): A parallel processing architecture that performs the same operation on multiple items of data at the same time.

Single Instruction Single Data (SISD): A parallel processor architecture that can execute a single instruction on a single item of data at a time.

Single Precision Floating Point (float): Is a data type available in C++ which is used to represent fractional values. It is capable of storing a fractional value between $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$ with 7 digits of precision.

Streaming SIMD Extensions (SSE): Is a SIMD instruction set used to expedite certain operations on a CPU.

Texture: In computer graphics a texture is an image that is mapped onto a polygon mesh.

Thread: A thread is a part of a computer program comprised of a single sequence of instructions which can be execute independently to other parts of a computer program. A group of threads can be executed in parallel by a multi-core processor.

Thread Group: The term refers to a group of threads which are executed by the same SIMD engine in parallel on a GPU. Threads within a thread group can be synchronised and have access to a shared region of cache memory.

Three Dimensional (3D): A Three Dimensional object in computer graphics has depth information. Typical 3D objects in computer graphics are polygon meshes and volumetric models.

Two Dimensional (2D): A Two Dimensional object in computer graphics is flat and contains no depth information. Typical 2D objects in computer graphics are images and fonts.

Unified Shader Architecture: The unified shader architecture or stream processing replaced separate vertex and pixel shader processors with a processor that was capable of processing both data types.

Unsigned Integer (uint): A data type which can store a whole number value between 0 and 4,294,967,295 using 32bits of memory.

Unsigned Short Integer (ushort): A data type which can store a whole number value between 0 and 65,535 using 16bits of memory.

Vector Processor: Is a processor architecture that is capable of executing a single instruction on multiple items of data in a single operation.

Vertex/Vertices: In computer graphics a vertex describes one of the three corners of a triangular surface (or polygon).

Vertex Buffer: A Vertex Buffer is a memory location for storing mesh vertices relating to a particular mesh. The vertices are stored in the most efficient manner for the GPU to process them.

Vertex Shader: A GPU program for performing calculations on vertices. It is typically used to transform vertices to their correct location and orientation in a 3D environment.

Video Memory: Video memory is a term used to differentiate between the "main" random access memory of a computer and the specialised RAM incorporated into a GPU.

Voxel: A Voxel is a cube shaped object which represents a point in 3D space, similar to a pixel in 2D space.

X-Ray: A "plain" X-ray image is a flat 2D image which is produced using X-ray radiation.

1. Introduction

1.1. Overview

This report details an algorithm that will be capable of providing an automated method for preparing digital X-rays for use by a procedural mesh generator. The mesh generator will construct a three dimensional (3D) polygon mesh from a number of X-ray images processed using the algorithm. This mesh will subsequently be used to provide a patient specific bone model in VirtuOrtho, a virtual surgical simulator. The algorithm will prepare the X-ray image by identifying and retaining areas of interest (bone) whilst removing any superfluous aspects contained within the image including text, orientation markers and backgrounds. Noise reduction techniques will be employed to suppress any noise present in the image. This will ensure that the mesh generated from the processed X-rays will have sufficient accuracy for the purposes of the simulation.

The proposed algorithm requires the processing of a considerable quantity of data and as a result this will have an impact on the time required to execute the algorithm. A significant portion of this computation could potentially be performed in parallel, minimising the time required to process an image. Therefore techniques which allow the algorithm to exploit the parallel processing capabilities of either a multi-core CPU or a GPU will be employed. Additionally the relative performance of the multi-core CPU implementation will be compared and contrasted to the performance of the GPU version.

As part of the GPU version of the algorithm, two methods for accelerating the calculation of a median filter using a GPU are suggested. The caching method improves upon the fast, small-radius median filter (McGuire, 2008) by utilising the cache memory provided in the latest generation of consumer graphics cards. The second technique uses the cache memory to implement a histogram based (Huang et al., 1979) method for applying median filters this enables the GPU to process median filters with a large-radius mask unlike previous GPU accelerated methods.

1.2. VirtuOrtho

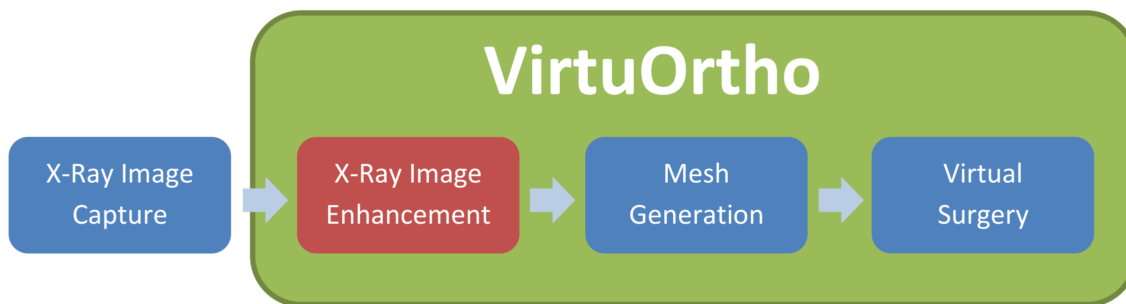

Figure 1 : VirtuOrtho

The functionality provided by VirtuOrtho can be broken down into a number of components. The algorithm proposed in this report relates specifically to the enhancement of X-ray images in preparation for their use by the mesh generator, which is being developed separately.

VirtuOrtho is intended to be a virtual surgical simulator for use as part of a training regime for trainee podiatric surgeons. The software will utilise a SensAble PHANTOM® Omni haptic input device as the primary input method during the virtual surgery. The device's capabilities (SensAble Technologies, 2010) enable the software to feature accurate, real time tracking of user input in three dimensions. It also provides haptic force feedback effects, which will be used to mimic the forces felt by the surgeon whilst performing a surgical procedure. The software will procedurally generate a 3D polygon mesh of the bones within a patient's foot by analysing a set of digital X-ray images taken at a number of predefined angles[1]. The use of a procedural mesh generator enables the incorporation of patient specific bone models into the virtual environment, allowing the training to be tailored to suit a particular operation. An additional benefit is that this also will greatly reduce the complexity of reproducing a range of afflictions, enabling VirtuOrtho to replicate the corresponding surgical procedure. To further enhance the realism of the training environment provided by VirtuOrtho, it will incorporate support for a 3D stereoscopic monitor which offers the user improved depth perception within the virtual environment.

[1] Currently only X-rays taken at lateral and planar orientations are used.

Figure 2 : SensAble PHANTOM® Omni

The SensAble PHANTOM® Omni is a haptic input device capable of tracking user input and providing force feedback in 3 dimensions.

1.3. Problem Statement

VirtuOrtho requires a number of polygon meshes depicting a variety of afflictions, so that it is capable of simulating a broad range of surgical procedures which occur in Podiatric Surgery. Typically these meshes are created by a digital artist using 3D modelling software such as 3Ds Max. This can be a time consuming and expensive process. Due to the intricate and highly detailed nature of the meshes required, it is realistic to assume that they may contain a number of inaccuracies, which would in turn affect the usefulness of the simulation. The inclusion of a procedural mesh generator into VirtuOrtho would eliminate the need for a digital artist to create the mesh, negating the cost issues mentioned. Procedurally generated meshes should contain fewer inaccuracies because they can utilise the maximum precision of the X-ray image and it reduces the chance that errors may be introduced. However, a process of determining the accuracy of both a procedurally generated and a handmade mesh would have to be devised. An additional benefit is that this would allow the software to incorporate patient specific meshes rather than a generic mesh for each affliction. Consequently this would make the software a more comprehensive and therefore useful training tool.

The mesh generator must construct a polygon mesh using 2D digital X-rays rather than a 3D scan such as those obtained using MRI or CT imaging. This is because 3D imaging techniques are rarely used in Podiatric Surgery, primarily due to X-rays being the most cost effective method of creating an image which contains all the pertinent information required to perform a surgical procedure. From a software development standpoint a CT scan or similar 3D imaging technique would provide a much simpler basis from which to generate a 3D polygon mesh, as it contains three dimensional data (FAKULTI KEJURUTERAAN ELEKTRIK, n.d., p.2). A CT scan is able to achieve this because unlike a conventional X-ray which scans from a single direction, it scans from multiple directions. Three

dimensional data would allow the implementation of either an iso-surface extraction technique (JaJa et al., 2008) or a voxel based system (Jeong et al., 2007). A voxel based representation as an alternative to a polygon mesh greatly simplifies the generation process, however there is a considerable increase in the computational cost for the visualization of the 3D model compared to polygon meshes.

The images contained within digital X-rays are monochrome, typically stored as 12bit brightness values. The properties of X-ray radiation which is used to capture X-ray images means that the density of a material affect its brightness, with high density materials like bone being brighter than materials with a low density such as soft tissue. This is not a uniform brightness, as the relative distance of the material from the capture device also affects its brightness[2]. Salt-and-pepper noise (Myler and Weeks, 1993, p.202) occurs in X-ray images due to the imprecise nature of the radiation which is used to capture an X-ray. X-ray images may also contain the following superfluous elements which need to be removed before the X-ray can be used by the mesh generator [Figure 4, Figure 6]: text, orientation markers, rulers and secondary backgrounds. Removing these would result in a more accurate mesh being generated.

Figure 3 : Lateral Foot X-ray (Original)

The above image shows the original lateral image encoded in the DICOM file.

[2] For the same density, objects further away of the same material appear darker.

Figure 4 : Lateral Foot X-ray with Unwanted Elements Highlighted

This image highlights those unwanted elements contained within the X-ray image which need to be removed by the algorithm. ***Blue*** *is the primary background,* ***Red*** *is descriptive text and* ***Green*** *is the orientation marker.*

Figure 5 : Planar Foot X-Ray (Original)

The above image shows the original planar image encoded in the DICOM file.

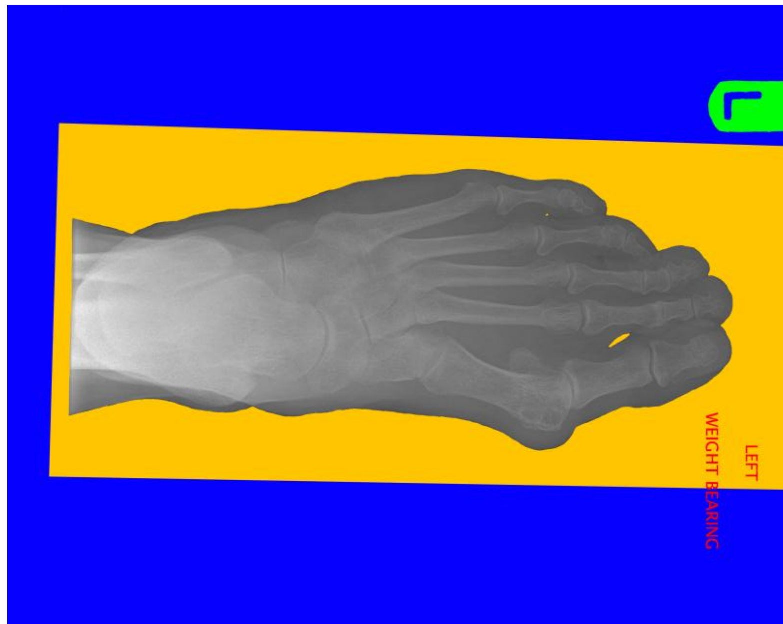

Figure 6 : Planar Foot X-Ray with Unwanted Elements Highlighted

This image highlights those unwanted elements contained within the X-ray image which the algorithm needs to remove. ***Blue*** *is the primary background,* ***Yellow*** *the secondary background,* ***Red*** *is descriptive text and* ***Green*** *is an orientation marker*

1.4. Project Aims

It is the aim of this project to produce an algorithm that is capable of removing unwanted features such as text and orientation markers from a digital X-ray whilst retaining areas of interest such as the various bones contained within the image. This algorithm should be developed so that it is capable of making optimal use of a parallel processor such as a multi-core CPU or a GPU. It is hoped that by adopting parallel processing techniques, the algorithm will have a considerably reduced execution time in comparison to a serial CPU implementation. The images processed by the algorithm will subsequently be used to generate a 3D mesh replicating the bones depicted within the X-ray images. The mesh generation and virtual surgery functionality of VirtuOrtho are being developed separately.

The objectives for this project have been split into two distinct categories, those objectives which directly relate to the capabilities of the algorithm and those which affect its implementation.

1.4.1. Algorithm Objectives

1. **Separate Soft Tissue and Bone**: The algorithm requires the ability to accurately identify and subsequently isolate soft tissue and bone within an X-ray image. Ideally the algorithm will also be able to separate them from each other.

2. **Remove Unwanted Features**: Generally X-rays contain unwanted features [Figure 4, Figure 6] such as text, various backgrounds and orientation markers. These need to be removed from the image to simplify the mesh generation process.
3. **Noise Reduction:** Appropriate techniques should be employed to diminish the amount of noise present within the X-ray image, on the condition that it does not negatively impact the fine details stored in useful areas of the image.
4. **Planar and Lateral X-Rays**: The algorithm must be capable of adapting to compensate for the differences between X-rays taken at lateral and planar orientations. This will allow it to correctly process images irrespective of their capture orientation. The mesh generator is currently limited to constructing polygon meshes utilising only X-rays captured in planar and lateral orientations. However it is envisioned that additional orientations will be used to increase the accuracy of the generated mesh. Therefore, if possible it should also be capable of processing X-rays taken at additional intermediate angles.
5. **Non-Uniform Brightness**: Due to the method by which X-ray images are created, some bones, particularly those at the extremities, are of a significantly lower brightness than other areas of bone. The algorithm must therefore be able to recognise this and adapt accordingly.

1.4.2. Implementation Objectives

1. **DICOM X-Rays**: In-order to help ensure that the mesh generation process is as straightforward and seamless as possible for the end user; the algorithm should be capable of processing digital X-ray images in their native file format (DICOM file). This will allow VirtuOrtho to generate a patient specific model in the simplest manner possible, requiring only the location where the X-rays are stored on the computer.
2. **Fully Automated**: VirtuOrtho is designed as a training tool; the algorithm should therefore require minimal human input wherever possible.
3. **Consumer Hardware**: The algorithm must be restricted to utilising only parallel processing technologies that are available in current consumer hardware. This will help to minimise the cost of purchasing hardware for the client.
4. **Single Instruction Single Data Implementation**: The algorithm will need to be implemented as a serial process for the CPU in addition to the proposed parallel versions. This implementation will be used to provide a control group in the performance comparison experiments and therefore will serve as the baseline execution time for the algorithm on a given X-ray image. This will be used in the calculation of the potential speed-up derived from parallelising the process.

   This version of the algorithm will be referred to as the SISD algorithm.

5. **Multiple Instruction Multiple Data Algorithm**: A version of the algorithm that can be executed on a multi-core CPU should be implemented. This will allow a more comprehensive analysis when determining how substantial the reduction of the algorithm's execution time a GPGPU implementation achieves compared to an optimised CPU version.
   This version of the algorithm will be referred to as the MIMD algorithm.
6. **Single Instruction Multiple Data Algorithm**: A GPGPU version of the algorithm will be implemented using the DirectCompute API. If possible the algorithm will be designed so that it can target hardware supporting the minimum version of the Compute Shader (CS 4.0). This will enable the algorithm to be executed on the lowest class GPU hardware supported by the DirectCompute API, Shader Model 4.0 (DirectX 10.0) compliant hardware[3] rather than the Shader Model 5.0 (DirectX 11.0) GPUs[4] required by CS 5.0.
   This version of the algorithm will be referred to as the SIMD algorithm.
7. **Native Resolution:** All implementations of the algorithm must be capable of processing the X-ray images at their native resolution. Additionally, if the data format used to encode the image is not available on a particular parallel processing architecture then it must be converted to a data type capable of equal or greater precision than the original. These requirements are to prevent image quality from being sacrificed to expedite processing on a particular architecture.

1.4.3. Median Filter Objectives

1. **Caching Median Filter:** A version of the fast, small-radius median filtering algorithm should be implemented which utilises the cache memory accessible via DirectCompute to improve performance.
2. **GPU Histogram Median Filter:** A GPU accelerated implementation of the standard histogram median filter should be developed. It will employ the cache memory accessible via DirectCompute to allow large-radius median filters to be processed without incurring the considerable performance penalties associated with using the graphics card's global video memory.

1.4.4. Expected Results

The results of this project are expected to show that parallel processing offers a considerable reduction in the execution time, with GPGPU producing the greatest reduction by a sizeable margin. The data will be processed in two formats, its native integer format and floating point. It is anticipated that the former should give the CPU an advantage as integers traditionally take longer to process on the GPU than floating point values. The latter data format should reverse this trend

[3] Nvidia 8000, AMD HD 2000 series or above.
[4] Nvidia 400, AMD HD 5000 series or above.

giving the GPU a clear advantage as it is a dedicated floating point processor. In spite of this there will be an associated penalty arising from converting the data from its native format to floating point values.

It is anticipated that the caching median filter will yield considerably improved performance compared to the fast, small-radius median filter. This assumption is based on the filters applying the same radius mask and both being implemented in DirectCompute. The GPU accelerated histogram median filter is expected to allow large-radius median filters to be computed by a GPU.

2. Literature Review

2.1. Research Objectives

The following literature review encompasses a number of current research topics, including methods for parallel processing and image processing techniques. The investigation will discuss techniques which either are currently used or have the potential to be beneficial for medical imaging. The parallel processing research will concentrate on methods of parallelisation which are suitable for use with parallel processors which are available in current consumer hardware, such as multi-core CPUs or GPUs. Research into the architecture of multi-core CPUs and GPUs will be conducted, to gain an understanding of how best to exploit their parallel processing capabilities. The process by which parallel algorithms are designed differs significantly to those used in serial algorithm design; therefore an appropriate parallel algorithm design methodology will be investigated.

The research conducted for this literature review will attempt to answer the following questions to help with design and implementation of the algorithm:

2.1.1. Parallel Processing

2.1.1.1. Background

1. What are the advantages and disadvantages of parallel processing?
2. What computer architectures utilise parallel processing?
3. How do these architectures differ?

2.1.1.2. General-Purpose Computing on Graphics Processing Units

1. What is General-Purpose Computing on Graphics Processing Units (GPGPU)?
2. How does GPGPU differ from conventional parallel processing?

2.1.1.3. Parallel Processing Architectures

Multi-core CPU Architecture

1. How do CPUs implement a MIMD architecture?
2. What are the advantages of using a MIMD architecture?

GPU Architecture

1. How do GPUs implement a SIMD architecture?
2. What are the advantages of using a SIMD architecture?

2.1.1.4. Parallel Algorithm Design

1. How are parallel algorithms constructed?

2. What methodologies are used to design parallel algorithms?
3. Can the speedup derived from parallel processing be predicted?

Performance Profiling Parallel Algorithms

1. How can the efficiency and maximum potential speedup due to parallel processing of an algorithm be calculated?
2. How can the execution time of a parallel algorithm be accurately measured?

2.1.1.5. Parallel Processing APIs

OpenMP

1. What is OpenMP?
2. What are the advantages of OpenMP compared to other threading APIs?

DirectCompute

1. What is DirectCompute?
2. What are the advantages of DirectCompute compared to competing GPGPU APIs?

2.1.2. Image Processing and Analysis

2.1.2.1. Medical Imaging

1. What are the common techniques used to process and analyse images in medicine?
2. Have any medicinal image processing techniques utilised parallel processors?

2.1.2.2. Applicable Image Processing Techniques

Median Filter

1. What algorithms exist for calculating median filters, particularly for median filters which use large-radius masks?
2. Do any parallel or GPGPU implementations of a median filter exist?

Histogram Calculation

1. Do methods exist for calculating a histogram using a GPU?

Thresholding

1. Do methods exist for calculating a threshold value and applying a threshold operation?
2. Could the process of calculating the threshold value be automated?
3. Do parallel or GPGPU implementations exist for automated thresholding?

Edge Detection

1. What methods are there for performing Edge Detection?
2. Do any parallel or GPGPU implementations exist?

Feature Extraction

1. What methods are commonly used for feature extraction?
2. What are the differences between the various methods for feature extraction?
3. Do any parallel or GPGPU implementations exist?

2.2. Parallel Processing

The term parallel processing covers a range of technologies including: Shared memory, distributed, and hybrid parallel processors in addition to GPGPU. For the purposes of this project only shared memory systems in the form of a single multi-core CPU and GPGPU will be investigated.

2.2.1. Background

Parallel processing is a technique whereby the computation of an algorithm is divided into portions or "threads" which can be executed simultaneously by multiple processors. The result of this process is that those algorithms which can be effectively parallelised exhibit reduced execution times compared to those which are processed sequentially by a single processor. Whilst parallel processing can significantly reduce the time required to execute an algorithm, care has to be taken as a number of issues can occur which have no equivalent in sequential programming. These include: race conditions, deadlocks, parallel slowdown and synchronisation. These issues can reduce the performance of an algorithm or more seriously lead to erroneous data and even prevent it from finishing altogether. Another peculiarity and a potentially serious but subtle issue that can occur in parallel programs is that when operating on floating point data an algorithm may be unstable (Mattson and Strandberg, 2008). This is not readily apparent in a serial implementation because this instability manifests itself by calculating different results depending on the number of threads used in the algorithm's execution.

A number of classifications for computer architectures have been proposed (Flynn, 1966). Almost all incorporate some form of parallel processing.

Table 2 : Flynn's Taxonomy

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

The architectures suggested [Table 2] are:

- **SISD**: Single Instruction, Single Data
- **SIMD**: Single Instruction, Multiple Data
- **MISD**: Multiple Instruction, Single Data
- **MIMD**: Multiple Instruction, Multiple Data

SISD or serial processor architectures have been in widespread use in consumer hardware for a considerable period of time and are therefore well understood by software developers. Processing is executed sequentially by a single processing core on a single item of data.

The availability of SIMD processors in consumer hardware is a relatively recent development, with the introduction of the graphics processing unit (GPU). Initially limited to processing visualisation related calculations, research has removed this restriction and facilitated the use of GPUs for the computation of general problems rather than those exclusively relating to graphics. SIMD architectures accomplish parallel processing by distributing the computation across a large number of processing cores (Roosta, 1999, p.6), with each core executing the same operation simultaneously on different data elements. A SIMD based architecture has the potential for significant performance bottlenecks to occur because it cannot short-cut execution. Processing cores cannot proceed with new work until all cores have completed the previously assigned work (Danielsson, 1984).

MISD is a rarely used architecture, typically used in fault tolerant computing and is therefore not normally employed in consumer hardware. Since the algorithm is restricted to consumer hardware [Section 1.4.2-3], the research will concentrate on SIMD and MIMD approaches to parallel processing and the associated hardware. A SISD implementation will be used to provide a control group for the experiments.

MIMD architectures are less restrictive than SIMD, with each processing core being completely independent. Therefore each core is able to execute different operations on unique data simultaneously (Roosta, 1999, p.23). MIMD based parallel processors have come to prominence in consumer hardware with the introduction and now widespread adoption of multi-core[5] CPUs [Table 7]. Multi-processor[6] based MIMD architectures have been available for longer than their multi-core

[5] Multi-core CPUs are a single physical chip with multiple independent processing cores.
[6] Multi-processor systems contain a number of physical separate CPUs. Each CPU may however be a multi-core processor.

counterparts, but are generally restricted to commercial environments due to their prohibitive cost compared to multi-core solutions.

2.2.2. General-Purpose Computing on Graphics Processing Units

General-purpose computing on graphics processing units (GPGPU) is the term used to describe "general"[7] computational problems that can be processed utilising the graphics rendering pipeline by a graphics processing unit (GPU). GPGPU is an active area of research covering a broad variety of subjects ranging from analysis of financial markets (Preis et al., 2009) to the simulation of molecular dynamics (Davis et al., 2009). The main attraction of GPGPU is the "massively parallel" nature of GPUs, which can potentially yield significant reductions in the execution time required to perform computations on large quantities of data. The High Level Shading Language (HLSL) used to create shader programs in DirectX is credited with being the most widely used programming language for the parallel processing of data (Boyd, 2008, p.23); this has no doubt encouraged the use of GPUs for general purpose computation.

The performance benefits of GPU processing are derived from its SIMD based architecture and the sheer quantity of processing cores a GPU has at its disposal. The result of this is that algorithms which can be effectively parallelised to suit SIMD processing can yield dramatic increases in performance compared to serial and parallel CPU implementations. The magnitude of the performance increase derived from GPGPU processing is disputed (Lee et al., 2010) and can be disingenuous considering that not all computational algorithms are suitable for processing using SIMD architectures. Typically algorithms which exhibit a large amount of data parallelism are those which benefit most from SIMD processing. Research (Bordawekar et al., 2010b) has also indicated that GPGPU applications take longer to develop than parallel CPU implementations and when this is taken into account with the performance increase, GPGPU becomes a much less compelling prospect. A study examining the performance and productivity of parallel processing using GPGPU and OpenMP reflected these findings (Christadler, 2010), with GPGPU offering superior performance, but being more time consuming to develop for. It must be acknowledged however that it is difficult to conduct these tests entirely scientifically as typically programmers are introduced to SISD and MIMD processing models much earlier that the SIMD model used by GPGPU.

Traditionally GPGPU programs have leveraged the processing power of the GPU by utilising the graphics pipeline via graphics APIs such as DirectX. Shader programs[8] are used to perform the processing and textures provide equivalent functionality to an array (Davidson, 2006, p.3) in

[7] i.e. non graphics related.
[8] Typically pixel shaders.

conventional programming. One significant difference is that individual memory locations are addressed using floating point values (Lönroth, 2009) with a texture rather than absolute values like conventional arrays. This makes precise memory indexing more problematic. Using the graphics pipeline is not an ideal solution; it is designed to maximize the performance of graphics operations and therefore has numerous restrictions on its use in place. This is not an ideal approach because the graphics pipeline is restrictive and requires the developer to be familiar with its intricacies to achieve optimum performance. Constraints such as the format data must be presented to the graphics card and the lack of thread synchronisation constructs further restrict the computational problems that can be processed using a GPU.

These limitations and the requirement for the developer to be familiar with the graphics rendering pipeline have led to the development of several GPGPU APIs including DirectCompute. These APIs are far more flexible in terms of how GPGPU algorithms can be implemented and provide far greater control over how data is subsequently processed. Specifically they provide facilities to perform thread synchronisation during the execution of the algorithm and allow thread allocation to be explicitly stipulated.

2.2.3. Parallel Processor Architectures

This section of the literature review investigates and details the architecture of two parallel processors available in current consumer hardware, which can be utilised by the proposed algorithm. Gaining a detailed understanding of the different approaches taken to parallel processing by multi-core CPUs and GPUs is fundamental in leveraging the maximum potential performance from the particular processor architecture. There are considerable architectural distinctions between CPUs and GPUs arising from their differing approaches to parallel processing. Developments such as Intel's Larabee which uses multiple in-order x86 CPU cores (Seiler et al., 2008) for graphics and general purpose processing and AMD's new Accelerated Processing Units which "combine general-purpose x86 CPU cores with programmable vector processing engines[9] on a single silicon die" (Brookwood, 2010) blur these distinctions.

[9] These are based on the SIMD Engines contained in AMD's HD 5000 Series discrete graphics processors.

2.2.3.1. Multi-core CPU Architecture

This section details the architecture used by the Intel Nehalem family of multi-core CPUs.

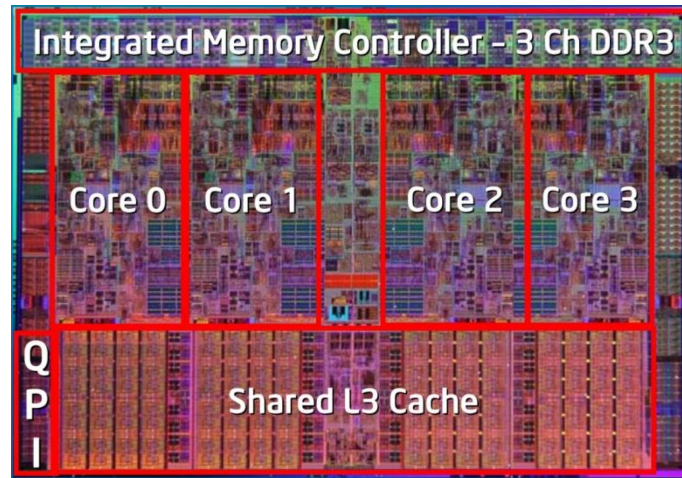

Figure 7 : Intel Core i7 Multi-core CPU Architecture

This diagram (Gelsinger, 2008) *highlights the significant components of an i7 CPU. Note how a significant portion of the die is encompassed by the processing cores and the shared cache memory.*

Prior to the introduction of MIMD multi-core CPUs, consumer CPU designs tended to use SISD architectures with a single processing core. These single core designs made use of the increased number of transistors available with each improvement in the manufacturing processor of integrated circuits, primarily by increasing the clock frequency and thereby the available computational power. Whilst the rate of increase in number of transistors which can be placed on an integrated circuit continues to be accurately predicted by Moore's Law (1965), the physical limitations of transistors have made it unviable to continue increasing clock frequency as the primary means of increasing computational performance. The introduction of multi-core CPUs has negated this problem by utilising the increased number of transistors to provided additional processing cores, allowing the computational power of a CPU to continue to increase. The additional computational power gained from multiple processing cores is more difficult to exploit as software needs to be explicitly designed for parallel processing unlike increases in clock speed.

The Nehalem architecture employs a MIMD approach to processing, relying on fewer, more complex processing cores compared to a current generation GPU. The majority of transistors and therefore physical space on the die of a Nehalem CPU [Figure 7] is occupied by either processing cores or the shared cache memory. These cores are capable of out-of-order execution meaning that they can re-order instructions to minimise the potential for stalls. Current Nehalem processors feature between 2 and 6 physical cores and some feature HyperThreading (Magro et al., 2002) which provides an

additional logical processing core per physical core. HyperThreading functions by duplicating certain portions of processing hardware whilst sharing others including the execution unit, the idea being that this will minimise amount of time that the execution unit is idle. Whilst these technologies could potentially give the multi-core algorithm an increase in performance, it is also possible that it may degrade performance by increasing the number of cache misses (Dawson, 2010). Furthermore it is exclusively supported by a limited number of Intel CPUs [Table 8].

The most significant differences between the Nehalem architecture and that of a GPU are the comparatively large amounts of shared cache memory[10] and branch predictors which prevent stalls caused by logic operations [Figure 10]. Each core on the CPU can also perform SIMD vector based processing by making use of SSE instructions.

2.2.3.2. GPU Architecture

This section will focus on describing the SIMD architectures utilised in current GPUs. The discussion will focus on the architecture used by the AMD HD 5000 series graphics processors. An overview of the differences between architectural approaches taken by AMD and Nvidia will also be included.

The architecture of graphics processors has altered substantially over the past decade, transitioning from performing simple “Transform and Lighting” (NVIDIA, 1999) operations via fixed function processing units (Chu, 2010) to a more flexible and capable programmable shader architecture. The initial generations of programmable shaders were split into pixel and vertex shader each requiring a specialised processor to execute a particular type of shader, the former using pixel processors and the latter vertex processors. This processing model could however result in some of the processors being underutilised and therefore idle depending on whether the workload was pixel shader or vertex shader intensive. The introduction of a unified shader architecture improved the efficiency of the programmable pipeline by forcing all shader processors to incorporate the same basic functionality (MSDN, 2010a), replacing the separate vertex and pixel shader processors with a shader processor capable of executing any type of shader and thereby maximising resource usage.

[10] A Nehalem CPU has 4MB to 12MB of cache memory which is shared between all cores, compared to 32KB per SIMD Engine for a DirectX 11 GPU.

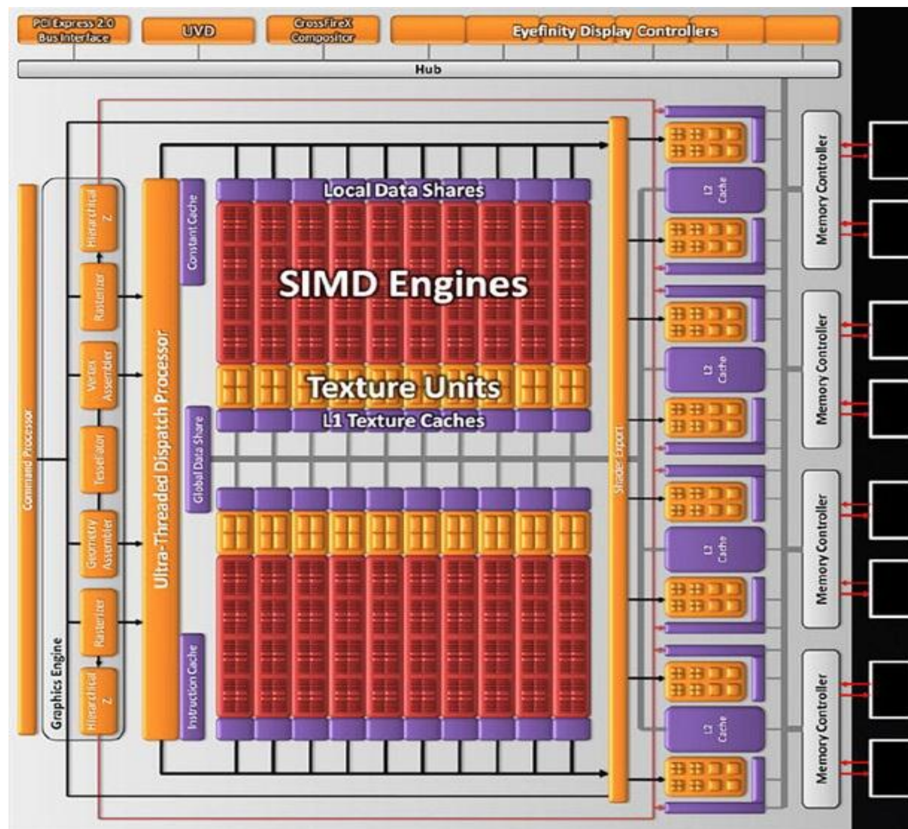

Figure 8 : AMD HD 5870 GPU Architecture

This diagram (Bit-tech.net, 2009) *highlights the various components present in a GPU (AMD HD 5000 Series). In the context of GPGPU the relevant components are SIMD Engines (**Red**), Local Data Shares (**Purple**, Above SIMD Engines), Texture Units (**Orange**), Instruction (**Purple**, Bottom Left) and Constant Caches (**Purple**, Top Left).*

GPUs utilise a SIMD approach to parallel processing; this processing model requires a large quantity of processing cores compared to a CPU. Fitting the required quantity of processing cores onto a die necessitates that they are far less complex than their CPU counterparts, lacking features such as branch predictors. The nature of graphics processing requires that each of the processing cores are specialised floating point processors. The processing cores are grouped into SIMD Engines, with each engine containing a small amount of cache memory[11] (Fried, 2010, p.5). This cache memory can be used by all processors in a SIMD engine to share data between themselves. Each SIMD engine is totally independent and cannot communicate or synchronise with other SIMD engines (Bleiweiss, 2008, p.4) unlike a multi-core CPU. The total number of cores is more difficult to calculate [Equation 1, Equation 2] compared to a CPU because each GPU may have multiple SIMD engines, each of which contain a number of scalar or vector processors. AMD utilises vector processors in its GPU architecture whereas Nvidia [Figure 9] GPUs feature a scalar architecture.

[11] 16KB for DirectX 10 and 32KB for DirectX 11 compliant graphics cards.

$$\text{Total Cores} = SE \times P \times V$$

$$\text{AMD HD 5770 Total Cores} = 10 \times 16 \times 5 = 800$$

Equation 1 : AMD HD 5770 Processing Cores

The number of processing cores available in a particular model of AMD's 5000 Series is calculated with the above formula. SE *represents the number of SIMD Engines,* P *the number of processors per SIMD Engine and each processor is a vector unit capable of simultaneously processing* V *instructions.*

$$\text{Total Cores} = GPC \times SE \times P$$

$$\text{Nvidia GTX 580 Total Cores} = 4 \times 4 \times 32 = 512$$

Equation 2 : nVidia GTX 580 Processing Cores

The number of processing cores available in Nvidia's GTX 500 Series is calculated differently because it is a scalar rather than vector processor architecture. SE *represents the number of SIMD Engines*[12]*, each containing* P *processors. The SIMD Engines are however grouped into Graphics Processing Clusters* (GPC)

Figure 9 : Nvidia GeForce GTX 580 GPU Architecture

[12] Nvidia calls SIMD Engines CUDA cores to highlight that they can be used for GPGPU applications which use Nvidia's CUDA API.

This diagram (Bit-tech.net, 2010) *highlights the architectural design of the Nvidia GTX 580. The main difference between this design and that utilised by AMD* [Figure 11] *is that the SIMD engines (**Green**) are grouped into Graphics Processing Clusters (GPC).*

Branching operations pose a particular problem for the SIMD architecture of a GPU, when a branch operation is encountered it is capable of processing only a single branch at a time [Figure 10]. This impairs performance because all the cores that require the alternative branch are stalled.

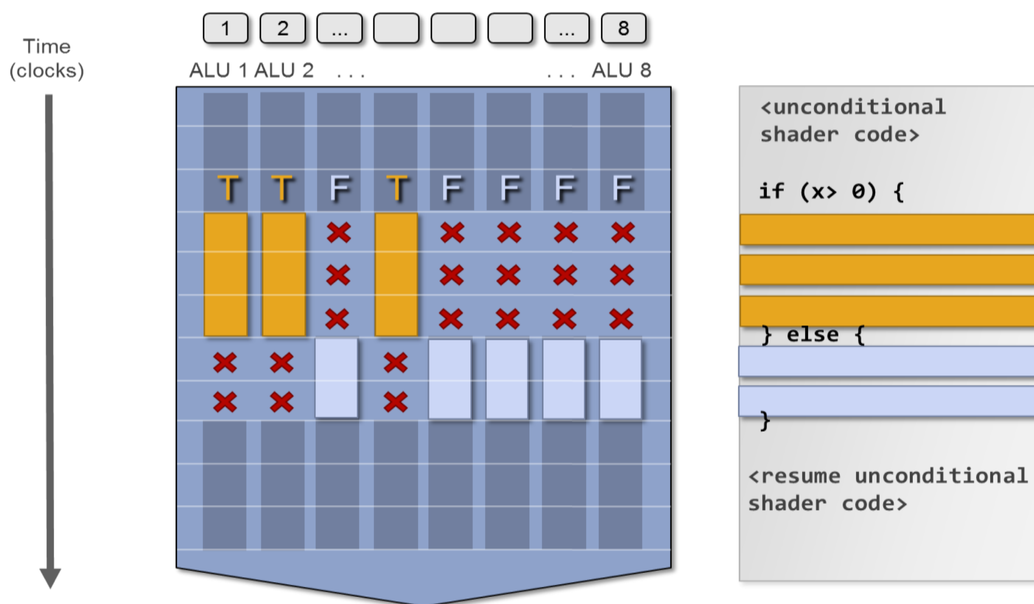

Figure 10 : GPU Branch Operation

This diagram (Pfister, n.d., p.26) *demonstrates how conditional logic operations on a GPU can greatly impair performance by stalling a number of processing cores (ALU). X represents when the ALU is stalled and therefore idle. **Yellow** and **Blue** boxes represent when each ALU is actively processing instructions related to a particular branch of the logic operation.*

2.2.4. Parallel Algorithm Design

Algorithms which are capable of utilising parallel processor architectures require a different design methodology to those developed for a serial implementation. Four discrete stages have been identified for designing parallel algorithms (Roosta, 1999, p.223):

1. **Partitioning**
2. **Communication**
3. **Agglomeration**
4. **Mapping**

The partitioning stage is used to determine whether Functional or Domain decomposition is the most suitable method for deriving parallelism from a particular problem. Functional decomposition involves breaking the algorithm down into its constituent tasks and subsequently identifying which of these tasks can be executed simultaneously. Domain decomposition attempts to isolate where in the algorithm multiple items of data have the same calculation applied and can therefore be executed in parallel.

The communication stage is used to ascertain the amount and type of communications that will be required between individual threads. Algorithms which require a large amount of inter-thread communication tend to exhibit reduced performance, especially when “Blocking” type communication is required.

Agglomeration is the process whereby the design produced using the previous two steps is evaluated and processing is amalgamated wherever possible. This can improve performance by reducing the number of potential bottlenecks and amount of inter-thread communication necessary. It is also used to minimise code complexity and thereby development costs.

The final stage in the design process is mapping. This is where the most applicable parallel processing architecture is selected for the particular problem being computed. When the architecture has been chosen, the concurrent tasks are “mapped” to the specific hardware being employed so that the work load is balanced for all processing cores thereby maintaining optimal resource utilisation and performance.

Ian Foster (1995, pp.27-28,42,49-50,56-57) has produced a checklist for the design process described above. The basic criteria for each stage are listed below:

1. **Partitioning**
    a. Is the partitioning at least an order of magnitude more than number of cores?
    b. Are there significant amounts of redundant computation and memory usage?
    c. Are tasks of comparable size?
    d. Does the task scale with problem size?
2. **Communication**
    a. Do the tasks require similar amounts of communication?
    b. Do the tasks communicate with a small number of neighbours?
    c. Is the communication non-blocking?
3. **Agglomeration**
    a. Has agglomeration reduced the amount of communication?

b. Does replicating computing impact performance?
c. Does replicating data affect scalability?
d. Does agglomeration leave tasks with similar computation time requirements?
e. Do tasks scale with problem size?
f. Has agglomeration removed too much concurrency?
g. Could the number of tasks be reduced without penalty?
h. Is parallelising the algorithm worth the development cost?

4. **Mapping**
   a. Is a SIMD algorithm better than MIMD algorithm for this problem?
   b. Is centralised load balancing a problem[13]?
   c. Is dynamic load balancing too costly?

Amdahl's Law (1967) [Figure 11] is commonly used to predict the potential speedup which can be achieved with an algorithm by using parallel processing. It states that the speedup of a particular program due to the parallelisation of its processing is constrained by the portion of the program which cannot be executed in parallel and therefore has to be processed sequentially. The law indicates that as the number of processors working in parallel is increased; the serial portion of the program becomes more relevant, limiting the maximum speedup achievable with parallel processing and reducing the efficiency. Algorithms which require minimal sequential processing are capable of scaling to effectively utilise large numbers of processing cores. The law is based on the assumption that the problem is of a fixed size and it does not scale with the number of processors.

$$S_{TOTAL} = \frac{1}{(1-P)+\frac{P}{S}}$$

Equation 3 : Amdahl's Law – Overall Speedup

This equation calculates the overall speedup S_{TOTAL} of an algorithm when the parallel section P has a speedup of S applied.

$$S = \frac{1}{(1-P)+\frac{P}{N}}$$

Equation 4 : Amdahl's Law - Speedup

This equation allows the potential speedup from using N processors simultaneously to be calculated for an algorithm with a parallelisable portion P.

[13] This type of load balancing only occurs in SIMD architectures.

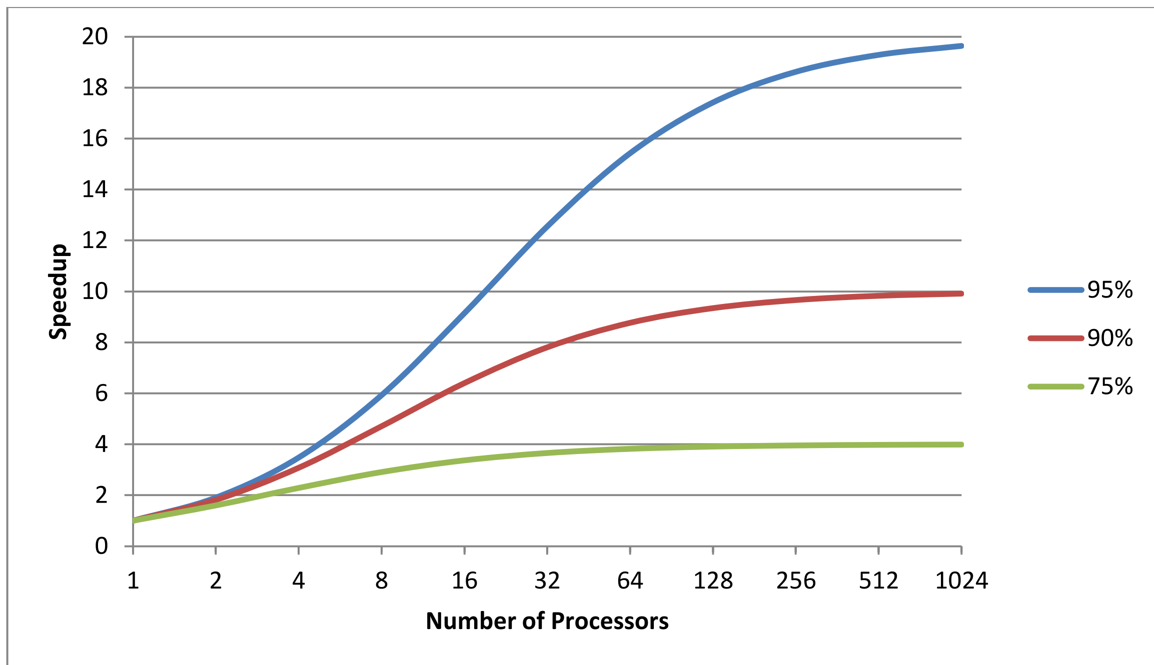

Figure 11 : Amdahl's Law

This graph depicts the predictions of the potential speedup an algorithm can gain from parallel processing for a given portion of parallelisable code. It demonstrates how many processors that Amdahl's Law predicts can be efficiently utilised in parallel before the serial portions of the algorithm restrict the amount of speedup. The coloured lines depict how the potential speedup that can be achieved by parallel processing is limited by the amount of parallelisable code contained within an algorithm: ***Blue*** *95% parallelisable code,* ***Red*** *90% parallelisable code and* ***Green*** *75% parallelisable code.*

Gustafson's Law (1988) rectifies this shortcoming by proposing that software developers tend to scale the size of the problem to the available processing power, so that it can be executed in an acceptable timeframe. Crucially it considers the influence that the serial portions of an algorithm have on the total execution time remains constant whilst the number of processors grows, rather than increasing as is the assumption made in Amdahl's law. This hypothesis however only remains true for problems with "large" repetitive data sets. Any problem that fits this profile is particularly suited for processing with SIMD type processors and by extension GPGPU processing. Gustafson's law explains why "massively" parallel processors can achieve speedups well in excess of those Amdahl's law suggests are possible.

Both laws clearly indicate the necessity of minimising the serial components of an algorithm, even at the cost of increasing the amount of computation the parallel sections have to perform.

$$S(P) = P - \alpha \cdot (P - 1)$$

Equation 5 : Gustafson's Law

Gustafson's Law states that the problem size and thus the potential speedup S scale relative to the available processing power given by P processors. The serial component of the algorithm α is constant no matter the problem size.

2.2.4.1. Parallel Algorithm Performance Analysis

In order to assess the scaling characteristics of the parallel algorithms developed for this project, formulae for calculating speedup and algorithm efficiency are required. The following equations (Grama and Kumar, 2008) facilitate this. The efficiency of a parallel algorithm describes the portion of the computation that can be effectively processed in parallel. Those algorithms with a high efficiency are capable of scaling to effectively utilise more cores before becoming restricted by the serial portions of the computation. The speedup value is the factor by which the execution time was reduced by parallelising the algorithm compared to a serial implementation.

The majority of the equations identified for calculating the efficiency of a parallel algorithm are not useful when determining the efficiency of GPGPU implementations. This is because these equations rely on the assumption that the serial version of an algorithm utilises an identical processor to the parallel algorithm, albeit using only a single core. This is because the amount of computation a single CPU processing core can perform is not equivalent to what a single processing core on a GPU can achieve. A serial GPU version of the algorithm cannot be implemented because the GPGPU APIs do not provide this functionality; they are designed strictly for parallel processing.

$$S = \frac{T_s}{T_p}$$

Equation 6 : Parallel Speedup

This equation states that the speedup S due to parallel processing can be calculated by dividing the total execution time of the serial implementation of the algorithm T_s by the total execution time of the parallel implementation T_p.

$$S = \frac{1}{(1 - P)}$$

Equation 7 : Maximum Speedup

The theoretical maximum speedup S that an algorithm can achieve is equal to the fraction of the program which is executed serially and therefore not part of the parallel portion P.

$$E = \frac{S}{N}$$

Equation 8 : Parallel Algorithm Efficiency

The efficiency E of a parallel algorithm is proportional to the speedup S achieved by utilising N processors. An efficient algorithm has a speedup value approaching the number of processors used to process the algorithm.

High performance CPU timers (Walbourn, 2005) can be used to measure the elapsed time for all versions of the algorithm, including the GPGPU implementation. Particular care has to be taken when profiling the GPGPU programs because GPUs are asynchronous devices. Without a blocking "Dispatch()" call [Code Listing 3, Line 39] to assign processing to the GPU, the timer measures the length of time it took the CPU to perform the call rather than how long the GPU took to perform the computation. OpenCL provides functionality to gather performance timings directly from the GPU (NVIDIA, 2009) using its internal timing mechanism. Unfortunately, DirectCompute does not provide equivalent functionality. Profilers such as Intel VTune (Lindberg, 2009, p.3) are capable of more detailed analysis, but there are currently no profilers available which are capable of analysing both CPU and GPU programs. Profilers will not be used in the experiments to measure and record the performance timings; this is because using different profilers to record the performance timings for each particular parallel processor architecture could introduce inconsistencies in to the results.

Due to the architectural differences between CPUs and GPUs it is difficult to accurately predict the potential performance increase due to parallelisation based solely on these equations. Therefore benchmarking using actual hardware is the only method available which allows accurate recommendations to be made.

2.2.5. Parallel Processing APIs

This section details two APIs: OpenMP and DirectCompute, that can be used to facilitate parallel processing. The former can be used with shared memory processors and the latter is exclusively for use with GPU hardware.

2.2.5.1. OpenMP

OpenMP is a directive based[14], platform independent API designed to facilitate parallel processing on shared memory processors, such as a multi-core CPU. This approach uses compiler directives to indicate to the compiler where the developer wishes parallel processing to occur [Code Listing 1, Lines 1, 3]. The advantage of this approach is that the compiler can provide automatic optimisations or ignore directives altogether if the compiler does not support OpenMP (Chandra et al., 2000, p.12). It is suggested to be a good alternative to low level threading libraries, such as PThreads, because it is programmed at a much higher level and is suited to processing data parallel tasks.

[14] Some OpenMP functionality is provided by libraries to minimise the problems arising from a purely declarative approach.

OpenMP was chosen because it has a small onetime setup cost (Lindberg, 2009, p.14) compared to other threading APIs. It features automatic thread pooling and functionality for performing load balancing. Thread pooling is very useful (Saccone, 2007) as it reduces the amount of time a thread spends idle; it also reduces the number of threads which have to be created and the associated cost of doing so. OpenMP uses "thread teams" (Isensee, 2006) which automatically allocate a number of threads to suit the available number of processing cores. Each time a parallel section is encountered [Code Listing 1, Lines 1 - 3] the master thread distributes the processing amongst the threads contained within the thread team. Once all threads within the team have completed their particular portion of the processing the master thread is permitted to continue with the sequential processing until the next parallel section is encountered. The thread team concept and the directive based implementation of OpenMP has the effect of reducing code complexity compared to other threading libraries.

```
1:      #pragma omp parallel
2:      {
3:              #pragma omp for
4:              for(int i = 0; i < 1000; i++)
5:              {
6:                      int value = data[i] * multiplier;
7:              }
8:      }
```

Code Listing 1 : OpenMP Loop Declaration

The OpenMP instruction "`#pragma omp for`*" [Line 3] is a compiler directive instructing it to parallelise the subsequent for loop. The number of iterations is distributed evenly to all processing cores.*

2.2.5.2. DirectCompute

DirectCompute is a dedicated GPGPU API provided as part of Microsoft's DirectX package of APIs. It is a platform specific API, currently restricted to computers which incorporate a graphics card compliant with either the Shader Model 4 or 5 specifications and use Microsoft Windows Vista/7 operating systems. It is hardware agnostic, meaning it can utilise GPUs from any hardware vendor[15], unlike Nvidia's CUDA GPGPU API (NVIDIA, 2010a) which is restricted to Nvidia GPUs. OpenCL (Khronos Group, 2010) is another alternative GPGPU API that is both platform and hardware vendor agnostic, but it lacks DirectCompute's efficiency for sharing data between the GPGPU processing and the graphics rendering elements of the DirectX API (Howes, 2010).

[15] The graphics processor and its associated drivers must support DirectCompute.

DirectCompute currently consists of two main[16] instructions sets, CS4.0 and CS5.0, reflecting the capabilities of Shader Model 4 and 5 compliant graphics processors respectively. CS5.0 is the preferred instructions set for DirectCompute as it features a host of improvements, the most notable of which are: increased cache memory, improved performance and atomic synchronisation constructs (Bilodeau, 2009, p.32). DirectCompute also has the following advantages over traditional GPGPU methods:

- Data does not have to be converted to conform to texture formats in order to be processed.
- A screen aligned quad is not required to process data.
- It has access to high speed cache memory.
- It is able to randomly read from and write to memory locations in video memory (Sandy, 2010, p.20). This technique is known as scattering.

A GPGPU program or "compute shader" is written in HLSL when using the DirectCompute API. The computation which is to be performed by the GPU is specified in the main method [Code Listing 2, Lines 17 - 27]. The input and output data is transferred from main memory to the GPU's video memory using buffers [Code Listing 2, Lines 11 - 12]. Each thread has a unique identification number, which can be calculated from its thread group and thread identification numbers [Code Listing 2, Line 19]. The number of threads each thread group consists of is specified in the shader using the "`numthreads`" command [Code Listing 2, Line 16] and cache memory is allocated using the "`groupshared`" function [Code Listing 2, Line 14].

[16] CS4.1 is a subset of CS4.0, offering some improvements but has minimal hardware support compared to CS4.0.

```
1:      cbuffer Constants : register(b0)
2:      {
3:              uint multiplier;
4:      };
5:
6:      struct BufType
7:      {
8:              uint integer;
9:      };
10:
11:     StructuredBuffer<BufType> BufferIn : register(t0);
12:     RWStructuredBuffer<BufType> BufferOut : register(u0);
13:
14:     groupshared uint cache[40];
15:
16:     [numthreads(40,1,1)]
17:     void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
18:     {
19:             int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
        yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
20:             cache[g.x] = BufferIn[i].integer;
21:
22:              // Comment
23:              GroupMemoryBarrierWithGroupSync();
24:
25:             value = cache[g.x];
26:             BufferOut[i].integer = value * multiplier;
27:     }
```

Code Listing 2 : Example DirectCompute Shader

This code demonstrates a simple Compute Shader. Using structured data input [Line 11] and output [Line 12] buffers require their data format(s) to be specified in the shader [Lines 6 - 9]. The allocation of cache memory [Line 14] and the number of threads per SIMD engine [Line 16] is also explicitly defined in the shader. Synchronisation of all threads within a thread group can be performed using the "`GroupMemoryBarrierWithGroupSync()`*" function [Line 23]. The identification number of the thread within its particular thread group can be calculated using the "*`uint3 g : SV_GroupID`*" parameter and the identification number of the thread group can be determined using the "*`uint3 gt : SV_GroupThreadID`*". The identification number of the actual thread in relation to all threads across all thread groups can be calculated using these values [Line 19]. Specifically for our implementation this allows the location of the actual pixel to be determined. Constant buffers allow useful data values to be passed into the shader without having to append them to the actual data buffer [Lines 1 - 4].*

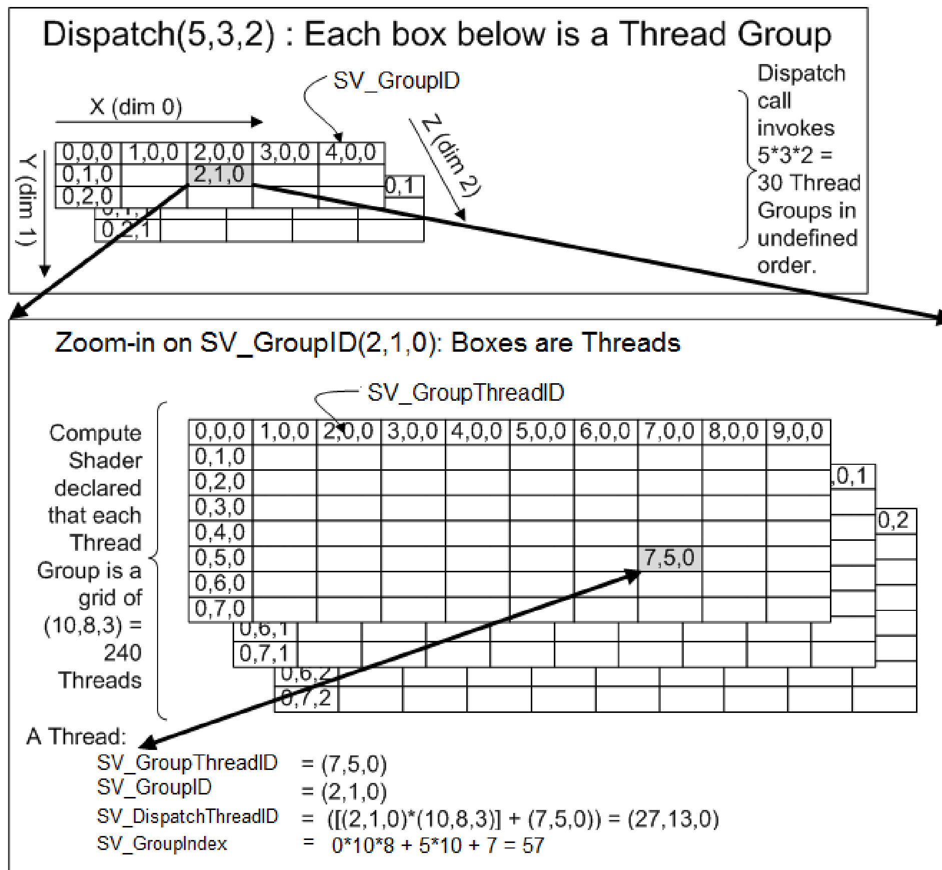

Figure 12 : Dispatch Call

The diagram (MSDN, 2010d) *illustrates how a Dispatch Call assigns work to the GPU. The dispatch call issues work as a 3 dimensional array of thread groups, each of which consists of a 3 dimensional array of threads.*

The Dispatch Call [Code Listing 3, Line 39] assigns work to the GPU as a 3 dimensional array of thread groups, each of which consists of a 3 dimensional array of threads [Figure 12]. The total number of threads which DirectCompute will execute is calculated by multiplying the number of thread groups by the number of threads they contain. Each of these thread groups is subsequently computed using a SIMD engine (Yang, 2010, p.28). Prior to this, the Compute shader must be compiled into assembly code [Code Listing 3, Line 20]. Next, the input and output data which will be utilised by the GPU is transferred to buffers, which are subsequently transferred to the video memory of the GPU so that it can access the data [Code Listing 3, Lines 22 - 36]. Once the GPU has completed the processing the “Map” command transfers the output data from video memory back to the main memory of the computer [Code Listing 3, Lines 42 - 44].

```
1:      ID3D11Device*              pDevice = NULL;
2:      ID3D11DeviceContext*       pContext = NULL;
3:      ID3D11ComputeShader*       pCS = NULL;
4:      ID3D11Buffer*              pBuf = NULL;
5:      ID3D11Buffer*              pBufResult = NULL;
6:      ID3D11ShaderResourceView*  pBufSRV = NULL;
7:      ID3D11UnorderedAccessView* pBufResultUAV = NULL;
8:      int* pData = NULL;
9:
10:     struct BufType
11:     {
12:         int i;
13:     };
14:
15:     int __cdecl main()
16:     {
17:             LoadData(pData);
18:             CreateGraphicsDeviceAndContent(pDevice, pContext);
19:             // Compile the shader
20:             CreateComputeShader("BasicCompute11.hlsl", "CSMain", pDevice, &pCS);
21:
22:             // Input Buffer
23:             CreateStructuredBuffer(pDevice, sizeof(BufType), NUM_ELEMENTS,
        &pData[0], &pBuf);
24:             // Output Buffer
25:             CreateStructuredBuffer(pDevice, sizeof(BufType), NUM_ELEMENTS, NULL,
        &pBufResult
26:
27:             // Create ResourceViews so that GPU can access the data
28:             CreateBufferSRV(pDevice, pBuf, &pBufSRV);
29:             CreateBufferUAV(pDevice, pBufResult, &pBufResultUAV);
30:
31:             // Initialise the shader
32:             pContext->CSSetShader(pCS, NULL, 0);
33:             // Send Input Data to GPU
34:             pContext->CSSetShaderResources(0, 1, &pBufSRV);
35:             // Assign an output location for GPU's computations
36:             pContext->CSSetUnorderedAccessViews(0, 1, &pBufResultUAV, NULL);
37:
38:             // Instruct GPU to start processing
39:             pContext->Dispatch(NUM_ELEMENTS, 1, 1);
40:
41:             // Read back the result from GPU, blocks this CPU thread until GPU
        has finished
42:             ID3D11Buffer* debugbuf = CreateAndCopyToDebugBuf(pDevice, pContext,
        pBufResult);
43:             D3D11_MAPPED_SUBRESOURCE MappedResource;
44:             pContext->Map(debugbuf, 0, D3D11_MAP_READ, 0, &MappedResource);
45:     }
```

Code Listing 3 : DirectCompute Dispatch Function

In order for a GPU to execute a Compute Shader and process data, the CPU must perform a number of operations. Firstly, the objects which allow interaction with the driver for the GPU must be created [Lines 1 - 2, 7]. Next buffers are created to store the input and output data [Lines 22 - 25] and these buffers are converted to a GPU assessable format [Lines 27 - 29]. Subsequently the compiled shader, the input data and output location are transferred to the GPU [Lines 31 – 36]. The "Dispatch()" [Line 39] call instructs the GPU to allocate the specified number of thread groups and begin processing the

data asynchronously. The "Map()" command [Lines 41 - 44] transfers the processed data back to the CPU, furthermore it forces the CPU thread that executed the "Dispatch()" call to wait until the GPU has finished processing, effectively blocking the thread.

2.3. Image Processing

The algorithm applies a number of image processing techniques to the X-ray image in order to prepare it for use by the mesh generator. Most image processing algorithms are built around a combination of averaging, edge detection, neighbourhood and texture analysis (Jähne, 2005, p.11).

2.3.1. Medical Imaging

2.3.2.1. Image Processing Techniques

Median filters are typically utilised in medical imaging as a pre-processing step for image segmentation algorithms because of their ability to reduce noise whilst preserving edges, improving the accuracy of the segmentation. Due to the considerable computational cost of applying these filters, techniques have been developed to facilitate GPGPU processing (Voila et al., 2003), in order to reduce execution times.

Kindelan and Lezo (1984) have recommended using the Hueckel Basis Function to detect Arteries in medical images. This method suffers from a number of problems including difficulty processing areas which contain arteries which are occluded or intersecting arteries. Furthermore it is an interactive method, requiring user input to suggest an initial location for the algorithm to process the image.

Image registration is the technique used to correct inaccuracies contained within X-ray images by comparing multiple X-ray exposes of the same area. Strzodka et al. (2004) propose a method of Image registration which utilises GPGPU processing to improve the performance of the algorithm. Whilst it may prove useful to incorporate similar functionality into the proposed algorithm, it is not a simple prospect as multiple exposures are typically taken at different orientations in podiatric surgery. The result of this restriction is the complexity of the registration process is substantially increased.

A comprehensive discussion detailing a variety of image processing and analysis techniques implemented by a system for the segmentation and landmarking of 3D CT scans has been suggested (Banik et al., 2008). The system is broadly similar to the one we propose, in particular using histograms, thresholding and active shape model detection to select the areas of interest contained within the CT scan. The key difference is that system is designed to process 3D image data captured

by CT scans. Active Shape Model detection is a method commonly used by algorithms to obtain areas of interest in medical images, such as nodules on X-ray images of a lung (Wei et al., 2002).

Using an artificial neural network to classify regions of interest (Yin and Tian, 2008) has been developed; the system allows those regions extracted from an X-ray using edge chain code to be categorized. An alternative system of automated classification in medical images has been proposed (Shamir et al., 2008). The system assesses the severity of Osteoarthritis within a knee joint. It uses a simpler system than an artificial neural network, accomplishing the classification by comparing the knee joint depicted within an X-ray image to a number of preselected images containing knee joints suffering from Osteoarthritis to different extents.

The superimposing of bones on top of organs in X-ray images can be a problem for image segmentation techniques, however a method has been developed (Park et al., 2004) to limit the effect this may have.

The use of an "edge aware" bilateral grid (Chen et al., 2007) has been suggested as a method to improve the contrast of images, including medical images. The process increases contrast of the images whilst respecting the edges contained in the image. A side effect of this process is that any noise present is amplified along with the contrast. As the test images already have sufficient contrast for the algorithm to function correctly, it is not necessary to implement similar functionality at this time.

A system to validate the accuracy of 3D anatomical polygon meshes against CT or MRI images has been proposed (Cardoso, 2010). A simulated CT or MRI is generated from the polygon mesh; this allows the generated image to be compared with an image captured by a medical imaging device. Similar functionality could be useful later in the project, in determining if the mesh created by the procedural mesh generator is accurate.

2.3.2.2. Parallel Processing in Medical Imaging

GPGPU processing has gained considerable interest for computing various algorithms in medical imaging because of its ability to significantly reduce execution times compared to a CPU. This has found to be the case for image registration (Ansorge, 2008) and image segmentation (Roberts et al., 2010), with the GPU offering substantial performance improvements over CPU implementations.

Schellmann et al. (2008) performed a comparative study into the cost-effectiveness of various parallel processing architectures when used to perform medical image registration. The results of the study found that GPGPU offer the most cost effective platform, however the difficulty of development for the various architectures was not considered in the recommendations.

GPGPU processing has been shown to not only expedite the computation of medical imaging algorithms, but also offer the most cost-effective method of achieving improved performance (Schellmann et al., 2008) compared to shared memory and distributed CPU processing.

Multi-threading and various optimisation techniques have been trialled (Evans et al., 2010) to identify the optimal method for reducing the time required to produce a diagnostically useful image from the original image, when using a CPU to perform the computation. The research found that multi-threading provides the most substantial speedup surpassing all other optimisations by a minimum of a factor of five. The test was conducted using 16 threads which results in a 10x speedup. By rearranging Amdahl's Law to calculate P it can be determined that the parallel portion is [Equation 9] 96% of the total algorithm, resulting in a maximum potential speedup of 25x. This is actually misleading because half of the threads are executed using logical processors via HyperThreading. This offers much reduced computational power compared to 16 physical processing cores, so the number of processors would in fact be less. Amdahl's Law can be altered to account for HyperThreading when calculating P [Equation 10], however the performance penalty incurred by using HyperThreading is difficult to determine as it is specific to the particular algorithm and the employed processor to compute it.

$$P - PN = \frac{N}{S} - N$$

$$P - 16P = \frac{16}{10} - 16$$

$$-15P = -14.4$$

$$P = \frac{14.4}{15}$$

$$P = 96\%$$

Equation 9 : Calculating Parallel Portion

By re-arranging the formula proposed by Amdahl's Law it can be shown that the algorithm being optimised has 96% of its code which can be processed in parallel.

$$S = \frac{1}{P + \left(\frac{1-P}{N}\right)H}$$

Equation 10 : Amdahl's Law for HyperThreading

Amdahl's Law requires some alterations to reflect the performance of logical processors made available with HyperThreading (Akhter and Roberts, 2006, pp.18-19). *A logical processing core offers only partial performance of a physical core. A speedup* S *is achieved using* N *processors (both*

physical and logical) with a parallelisable portion P and a performance pentalty per thread of H to reflect the reduced performance of logical processors.

2.3.2. Applicable Image Processing Techniques

2.3.2.1. Median Filtering

The median filter was selected to form part of the image processing algorithm because of its ability to remove salt and pepper noise whilst retaining those edges which are present within the image (Chan et al., 2005). Median filters are also suggested to be a good pre-filter to apply before performing edge detection (Bovik et al., 1987). Applying a median filter to an image is a computationally expensive prospect, especially for large-radius masks. To simplify the process of removing both noise and the small unwanted elements typically present within an X-ray image, a large-radius filter should be used (Jähne, 2005, p.317).

Median filtering algorithms tend to fall into two distinct categories: Histogram-Based and Sorting-Based.

The basic histogram approach (Huang et al., 1979) [Code Listing 4] works by creating a histogram of all the pixels contained within the mask area. Once all these pixels have been added to the histogram, the algorithm iterates through the histogram bins until half the total number of pixels contained within the mask area have been counted. The histogram bin number at which this occurs is the median value for that particular mask area. The histogram approach is less efficient in terms of memory payload for filters with small radii than a sorting based method, until (16×16), where it achieves parity.

```
1:      //Iterate through all pixels within the mask
2:      for(i = 0; i < maskArea; i++)
3:      {
4:              //dataValue must be in 0-255 range for 256 bin histogram
5:              dataValue = data[maskPixelLocation];
6:              histogram[dataValue]++;
7:      }
8:      count = 0;
9:      for(i = 0; i < histogramSize; i++)
10:     {
11:             //Add the number of pixels at histogram location i
12:             count += histogram[i];
13:             //The median value is the value which the count equals half of the
        mask area. Mask area is an integer so rounding will mean we don't need to
        test if it is equal to.
14:             if(count > (maskArea / 2)
15:             {
16:                     //Return the median value
17:                     return i;
18:             }
19:     }
```

Code Listing 4 : Histogram Based Median Filter Algorithm

A histogram approach to median filtering initially determines which histogram bin the current data value corresponds to and then increments the count of the number of pixels in that particular bin [Lines 2 – 7]. When all the pixels within the mask area have been added to the histogram, the histogram bins are iterated through and the number of pixels contained within the bin is added to the count of the total number of pixels processed so far [Lines 9 and 12]. When this count value is greater than half of the mask area then the number of the particular bin being processed is the median value [Lines 14 – 18].

The sorting based [Code Listing 5] approach again reads all the pixels within the mask in the same manner but adds the values to a list instead of a histogram. This list is then subsequently sorted into ascending[17] value order, for example using bubble sort. The median value is located at the midpoint of the list and as such it is a trivial process to calculate the median value once the list has been sorted.

[17] The list can also be sorted into descending order.

```
1:     //Iterate through all pixels within the mask
2:     for(i = 0; i < maskArea; i++)
3:     {
4:            list.add(data[maskPixelLocation]);
5:     }
6:
7:     //Sort Items in the list using a fast sorting algorithm
8:     list.Sort();
9:
10:    int mid = list.Size() / 2;
11:    return list[mid];
```

Code Listing 5 : Sorting Based Median Filter Algorithm

The sorting based approach to median filtering starts by adding the data values of the pixels within the mask to a list or other storage container such as an array [Lines 2 – 5]. These values are subsequently sorted into numerical order using an appropriate sorting algorithm [Line 7]. Calculating the median value is then a trivial process, as it is the value at the midpoint of the list [Lines 9 and 10].

Numerous methods have been devised to reduce the time required to process and apply a median filter to an image. Of particular interest for this application are those techniques which can potentially be parallelised or those which perform efficiently using large-radius masks.

Two fast median filtering algorithms which use a successive binning method have been proposed (Tibshirani, 2008). The first, Binmedian, calculates the exact median value. The second, Binapprox finds the approximate median value and is therefore faster but introduces the possibility that the median value calculated may not be the exact value.

The Constant Time median filtering algorithm (Perreault and Hebert, 2007) is an efficient method for applying large-radius median filters to images, exhibiting order O(1) runtime complexity compared to the more typical O(n) of other median filtering algorithms. The algorithm reduces runtime complexity and therefore the time required to apply a median filter by reusing the majority of a previously calculated histogram to calculate median values of neighbouring pixels. The algorithm is less than ideal for parallel processing, particularly "massively" parallel implementations such as GPGPU. This is because it reduces runtime complexity by reusing the previously calculated histogram which occurs far less frequently when median values are calculated in parallel. This effect is cumulative, becoming more pronounced as the number of threads used to process the filter increases. It does however indicate that drastically improved performance can be achieved by minimising the number of redundant memory accesses required for large-radius filters.

There have been various attempts at producing a GPU accelerated median filter (Mitchell et al., 2003; McGuire, 2008; NVIDIA, 2010d). Whilst these have managed to yield significant performance improvements in comparison to their CPU counterpart implementations, they have been limited to

small-radius (3×3 and 5×5) masks due to limitations of current graphics processors. All of these GPGPU methods are typified by their adoption of a sorting based approach to median filtering, which is better suited to GPU architectures. There appears to currently be no GPU median filtering algorithms which utilise a histogram based approach. This is probably due in part to the restrictions placed on algorithm design by the hardware limitations of the GPU and the fact that histograms have a significantly larger memory payload than a sorting based approach, this being more pronounced for smaller radii.

The fast small-radius (McGuire, 2008) method is essentially a sorting based median filter based on Paeth's (1990) CPU algorithm which has been adapted to better suit current GPU architectures. This has been achieved by the use of a branchless bi-directional bubble sort using hardware accelerated min and max operations. A similar branchless median filter (Kachelriess, 2009) has been proposed for CPU utilising SSE instructions. The median filter is processed using a single pass. The fast, small-radius implementation allows the data to maintain parallelism for each pixel but comes at a cost of requiring redundant memory accesses, unlike the constant time algorithm. The Separable Median Filter Approximation (Mitchell et al., 2003) technique for median filtering uses a two pass solution, sorting first horizontally then vertically. It uses branches to sort these values; therefore it will incur a performance penalty for this. Additionally the median value calculated by this method is only an approximation, so the value may be close to the median value but not the actual value.

2.3.2.2. Histogram Calculation

Histograms are one of the basic tools for image analysis and are used by the optimum thresholding algorithm [Section 2.3.2.3]. Histograms are difficult to parallelise due to the number of potential conflicts that may occur by different threads writing to the same bin at the same time (this is reduced with larger numbers of bins). A popular method for parallel histogram calculation is by reduction, whereby each thread calculates its own sub-histogram for the area it is assigned to. When all the sub-histograms have been generated, the sub-histograms are then combined together into a single histogram.

A scatter based method of GPU histogram generation has been proposed (Scheuermann and Hensley, 2007) and makes use of various Shader Model 3 vertex shader functions, including Vertex Texture fetch. It allows a large number of bins[18] to be used, but it is restricted to processing textures

[18] Up to 1024 bins.

with dimensions which are a power of two[19]. This would require the X-ray images to be resized (Kazhdan and Hoppe, 2008, p.7) as their native dimensions are not a power of two.

A parallel reduction method of GPU histogram calculation using OpenCL (Podlozhnyuk, 2009) is an alternative; however the number of histogram bins is restricted by the size of the local cache memory available. Both of these approaches also suffer from performance bottlenecks when using a small number of bins, which may be exacerbated by distribution of the input data. In the case of the partially processed X-rays the distribution is confined to a small range of values resulting in an increased number of collisions. Two CUDA specific parallel reduction (Shams and Kennedy, 2007) methods similar to the previously mentioned OpenCL technique have been implemented and evaluated. The first and fastest method is broadly similar to the OpenCL technique but the second utilises global memory instead of the local cache memory. This has two benefits over using cache memory: the distribution of data is not a factor in performance and it allows significantly more bins to be used per histogram. However a significant performance penalty is incurred because cache memory is considerably faster than global memory.

2.3.2.3. Thresholding

The optimum thresholding (Myler and Weeks, 1993, p.175) technique analyses an image's histogram and locates the value at the valley between the two predominant peaks (one peak is the background and the other is the object) in the histogram. This value is subsequently used as the threshold value allowing a threshold operation to remove the image's background. The technique allows a threshold value to be determined automatically without requiring user input.

A method for GPU thresholding (Tatarchuk, 2008) an image as part of a GPU-Based Active Contours algorithm has been developed. This technique converts a colour image into grayscale and then subsequently uses a manually specified threshold value or one computed using the histogram. It is not explicitly stated which method is used or indeed if the GPU is used to calculate this threshold value if it is an automated calculation.

2.3.2.4. Sobel Edge Detection

Sobel edge detection filters are used to highlight edges contained within an image and are typically employed in conjunction with feature extraction algorithms to improve the accuracy of the segmentation. They are however affected by noise and therefore are not ideal for use with images which contain a high level of noise (Petrou and Bosogianni, 1999, p.306). Applying noise reduction techniques such as median filtering or a 3D bilateral filter (Langs and Biedermann, 2007) to the

[19] Graphics cards typically require textures to have dimensions which correspond to a power of two value.

image would prevent this from becoming an issue whilst preserving edges. An improved version of the Sobel filter (Wang, 2006) has been developed which offers enhanced performance when performing edge detection on images with a constrained range of luminance values. The basic Sobel filter should prove sufficient for this project as the X-ray images have a comparatively large range of luminance values.

2.3.2.5. Feature Extraction

Like most image processing techniques there are a number of methods available for detecting and extracting features from images, ranging from fairly simple contour tracking to complex algorithms that analyse not only contours but texture. Active Contours ("Snakes") (Kass et al., 1988) are a method of feature extraction which attempts to extract a shape by altering the location of a set of points so that they enclose the target feature. "Snakes" are the predominant method of feature extraction in medical imagery (McInerney and Terzopolous, 1996). Active Shape Models (ASMs) (Cootes et al., 1995) are an improvement of the "Snakes" feature extraction technique, being capable of locating objects whose appearance may vary. ASMs locate objects by being trained using a number of images with marked landmark points covering the possible shape variations. Active Appearance Models (AAMs) are gaining popularity because whilst ASMs are faster to implement than AAMs, an AMM requires fewer landmark points and typically converges to a better result, especially in terms of textures (Cootes et al., 1999). AAMs differ from ASMs mainly due to the fact that they also analyse the texture of areas surrounding the landmarks.

A GPGPU implementation of the "Snakes" feature extraction algorithm has been developed (Tatarchuk, 2008), offering real time performance. An alternative method of feature extraction is histogram-based searching (Sizintsev et al., 2008), whereby the histogram of a template image is compared to the histogram of a subsection of the image to determine the location of the template within the image. This method is not particularly useful for our algorithm as it will be difficult to generate templates which will function with the large amount of variety expected in the X-ray images the algorithm will process.

2.4. Conclusions

This literature review has shown that utilising parallel processing in general and GPGPU in particular can yield significantly improved performance for the computation of an algorithm compared to a sequential implementation. A considerable amount of research into GPGPU based image processing has been conducted, exploring a wide variety of image processing techniques. However the lack of a GPGPU median filtering algorithm that is capable of applying large-radius masks is a noticeable

omission. This can be ascribed to the difficulties of implementing such an algorithm on the restrictive architecture of a GPU and their limited use in image processing.

The performance benefits of utilising GPGPU are well documented, however they are still disputed particularly when the performance benefits are considered with the increased difficulty of development. These results must be approached with some scepticism for a number of reasons:

- A great deal of the research does not compare the performance results with the most efficient CPU algorithm.
- Those algorithms which benefit most from GPGPU processing are typically suited to SIMD processing.
- Testing is typically conducted using the most powerful GPU available at the time.

3. Methodology

3.1. Overview

The methodology adopted for this research is centred on conducting experiments which generate quantitative results. This will allow a quantitative analysis of the relative performance of the various algorithm implementations to be conducted.

3.2. Implementation

The algorithm will be implemented in C++ and HLSL. It will use compiler settings specified in Table 11 to apply the same optimisations to all versions of the algorithm. The GPGPU version will use pre-compiled shaders to minimise the overhead of compiling at runtime, as this is how GPGPU programs are typically implemented. Wherever possible, buffers for transferring data to and from the GPU will be re-used to minimise the overhead incurred by transferring data.

It may not be practical to perform GPGPU processing for all aspects of the algorithm, therefore those portions of the algorithm which cannot be processed efficiently by a GPU will utilise the MIMD implementation. The reasoning behind this decision is that GPGPU algorithms tend to be implemented to reduce the execution time of an algorithm and the MIMD should offer improved performance over the SISD implementation for those sections which cannot be effectively processed by a GPU.

3.3. Data Gathering

A number of experiments will be conducted on the various implementations of the algorithm. These will be mainly quantitative tests, measuring the time required to execute the algorithm or a particular component. The performance timings will be gathered using high performance timers, however a GPU is an asynchronous processing device so care has to be taken to ensure that the timers are setup correctly [Section 2.2.4.1].

3.3.1. Algorithm Experiments

The following tests will be performed on the algorithm and the output images it generates:

1. **Overall Algorithm (Quantitative):** This test will compare the relative execution times for the various implementations of the algorithm. The execution times will include the amount of time required to convert the data to an appropriate format.

2. **Individual Components (Quantitative):** A comparison of the execution times for individual components of the algorithm will be conducted. This will help to determine if particular components of the algorithm are particularly suited to parallel processing.
3. **Data Formats (Quantitative):** This test will determine what is the most appropriate data format for a particular processor architecture.
4. **Median Filters (Quantitative):** A comparative test of the performance between the various implementations of GPU median filters. Performance results of the CPU median filters with the same size masks will be included to allow better comparisons to be made. This test will be conducted with both a small-radius (3×3) mask and a large-radius (19×19) mask.
5. **Optimum Number of Threads (Quantitative):** An experiment to establish the optimum numbers of threads per thread group for the development GPU.
6. **Average Image Error (Quantitative):** A quantitative test to determine if there are any differences between images produced by each implementation of the algorithm. The test will record the number and average size of the errors.
7. **Image Comparison (Qualitative):** A visual inspection of the output images to establish how significant the impact of any errors or differences are on the images produced.
8. **Implementation Difficulty (Qualitative):** An assessment of the difficulty of implementing the algorithm on the various parallel architectures. The test is difficulty to conduct scientifically therefore the conclusions on this will strictly be the author's personal opinion.

To ensure the tests are accurate and the results are replicable the following steps will be taken:

- The test will be performed 50 times and the average time of these will be used in any calculations, this is to reduce the impact any anomalous results may have.
- The computer will be updated to ensure it has the latest video card drivers, DirectX and Operating System updates.
- All non-essential software, background services and power management will be disabled so that the performance of the CPU and GPU is not impaired.
- The performance timings for the overall and individual components will be collected at the same time, this will eliminate any differences in execution time that may occur if these experiments were conducted separately.
- High performance timers (Dawson, 2010) will be used to measure the execution times of the various algorithm implementations.

3.4. Results Analysis

The results of the various experiments will be analysed using the SISD implementation as the control group. This will allow the speedup of the multi-core CPU (MIMD) and GPU (SIMD) implementations of algorithm to be calculated using the formulas detailed in the literature review [Section 2.2.4].

When analysing the overall performance of the algorithm, the additional time required to convert data into the optimum format must be considered. This will be clearly indicated in the analysis and added to the overall execution time of the GPU algorithm. However when comparing individual components of the algorithm it would be unfair to do this as the primary concern is how fast the GPU implementation is at a particular operation because other applications of these components may not require the data to be converted, unlike our implementation.

There is a possibility that inconsistencies between the rounding of floating point values by CPUs and GPUs may introduce errors or produce different images. A quantitative assessment of these differences will be conducted by comparing the images produced with the control image generated by the SISD implementation. Whilst the various implementations may produce different images, these inconsistencies may not unduly affect the final output. Therefore a qualitative test in the form of a visual inspection will be conducted to ensure that the images produced by the different implementations are similar enough so as to not affect the results.

4. Implementation

4.1. Overview

The proposed algorithm is designed to provide an automated method for isolating and subsequently separating areas of interest such as bone from other unwanted objects contained within a digital X-ray image. These manipulated images will be used to generate a 3D mesh representing the foot depicted in the X-rays. The algorithm will be developed in three variants, one for each of the following processor architectures: SISD (Single Core CPU), MIMD (Multi-core CPU) and SIMD (GPU). The SISD implementation will use standard C++ code and will be used to provide a baseline value when calculating the amount of speedup produced by the parallel implementations. The MIMD variant will share the majority of its C++ code with the SISD implementation and use OpenMP to provide the thread management and other parallel processing constructs. The SIMD version will use DirectCompute to leverage the parallel processing capabilities of the GPU. The algorithm will be developed to be capable of processing data in both unsigned integer and floating point data formats.

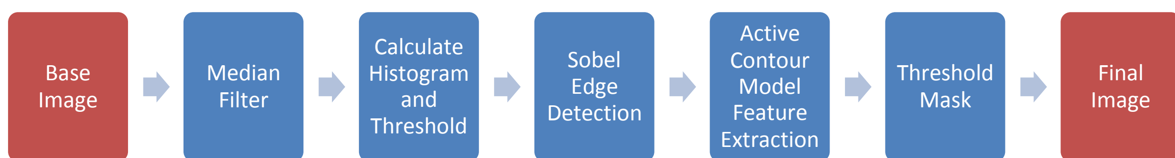

Figure 13 : Proposed Method for Processing an X-ray Image

This diagram demonstrates the process that will be applied to the original X-ray image in order to produce the required image.

4.2. General Algorithm Description

The algorithm prepares X-ray images by extracting image data from a DICOM file[20] and subsequently converting it to an appropriate data format for processing. The image data is processed in five discrete stages [Figure 13]; the selected image processing techniques are applied sequentially in the following order to the entire area of the image: Median Filtering, Histogram Calculation and Image Thresholding, Sobel Edge Detection, Active Contour Model (ACM) Feature Extraction, Threshold Mask.

[20] Digital X-rays also contain additional data including information about the method of capture, the X-ray device and patient details.

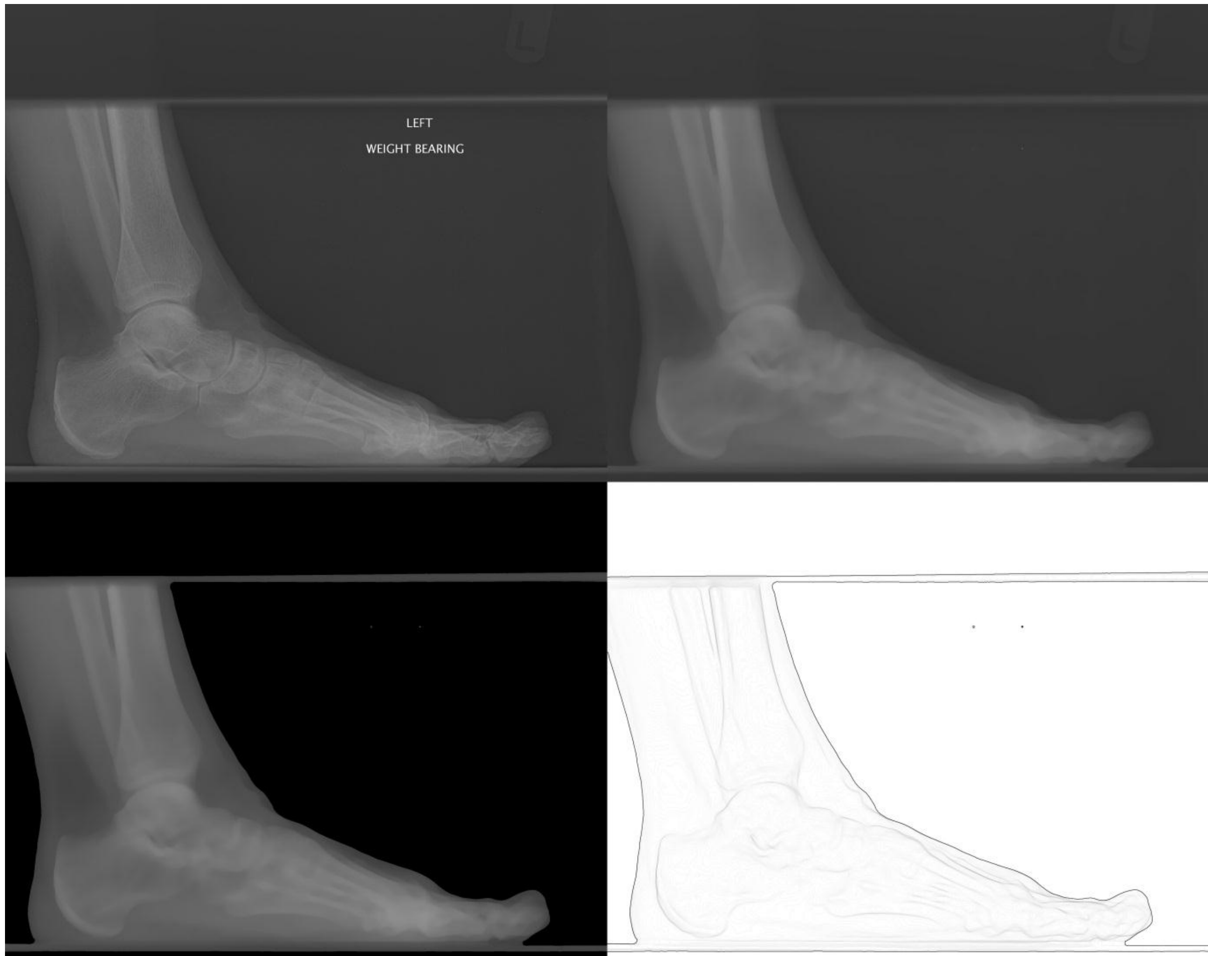

Figure 14 : Anticipated Algorithm Output

The image above depicts the anticipated output of the algorithm at the various stages of the algorithm. Clockwise from the top left corner the images are: Initial X-ray input, Large-Radius Median Filter, Threshold and Sobel Edge Detection.

4.2.1. Median Filter

The first stage of the algorithm consists of applying a large-radius (19×19) median filter to reduce the background salt-and-pepper noise present in the image. A median filter is particularly suited to this task compared to alternative noise reduction techniques such as arithmetic mean filtering because it preserves edges which are important later in the algorithm, when locating the bones contained within the image. Additionally a large-radius mask allows the algorithm to remove the relatively small unwanted objects, such as text [Figure 4, Figure 6] which may be present in the X-ray image. This removes the need to apply an additional feature extraction pass to the image before the median filter operation. If these elements were not removed from the image, the histogram and therefore the threshold value could be skewed, resulting in the threshold operation either retaining some of the unwanted elements or conversely removing too much of the desired information. The ability of the median filter to preserve edges depends on how large the radius of the median filter is,

relative to the size of the object containing those edges. The bones in the X-rays are comparatively large compared to the radius size and therefore their edges should be only slightly affected by the median filter.

4.2.2. Histogram and Image Thresholding

The thresholding process will help to remove the background(s) contained within the X-ray image whilst retaining the areas of interest. The threshold value is calculated by analysing the image's histogram, allowing the thresholding operation to be fully automated. The thresholding operation examines all the pixels within the image comparing them to the threshold value. All those which do not meet the threshold criteria are replaced with the minimum brightness value.

After the median filter operation, the algorithm subsequently calculates a 256-bin histogram of the entire image. This histogram is analysed to obtain a threshold value for use in the threshold operation. A 256-bin histogram was selected because it represented the best compromise between the memory payload of the histogram[21] and the production of an accurate threshold value.

The analysis of the histogram is accomplished calculating the gradient between each histogram bin and using this information to ascertain the locations of the peaks and troughs within the histogram. The threshold value is chosen by finding the leftmost peak which contains over 10%[22] of the image's total pixels and locating the value at the trough on its left-hand side. It selects the trough rather than the peak as this gets rid of additional noise at a cost of removing a minimal amount of detail around the bone edges. This allows the automated threshold value to function correctly when operating on images both with [Figure 6] and without [Figure 4] secondary backgrounds. Images with a secondary background produce two distinct peaks [Figure 16] whereas images without produce a single distinct peak [Figure 15]. For both types of image the tallest peak represents where the majority of the region of interest resides.

X-ray images are encoded in one of two formats, MONOCROME1 where black is used to represent bone or MONOCROME2 where white is used. The test X-ray images are encoded in MONOCROME1[23] format. As a result the algorithm searches right to left to locate the threshold value. For a MONOCROME2 image the data values would be inverted, requiring either the threshold value

[21] The number of bins affects how much memory a histogram occupies, with larger numbers of bins requiring large amounts of memory.

[22] This value will vary depending on how much of the total image area is occupied by areas of interest.

[23] Note that whilst the X-rays are processed in MONOCHROME1 format, in this document the X-ray images are depicted as MONOCROME2 to maintain consistency with how the majority of medical applications handle MONOCROME1 format X-ray images.

calculation to operate left to right or the pixel values to be inverted when they are obtained from the DICOM file.

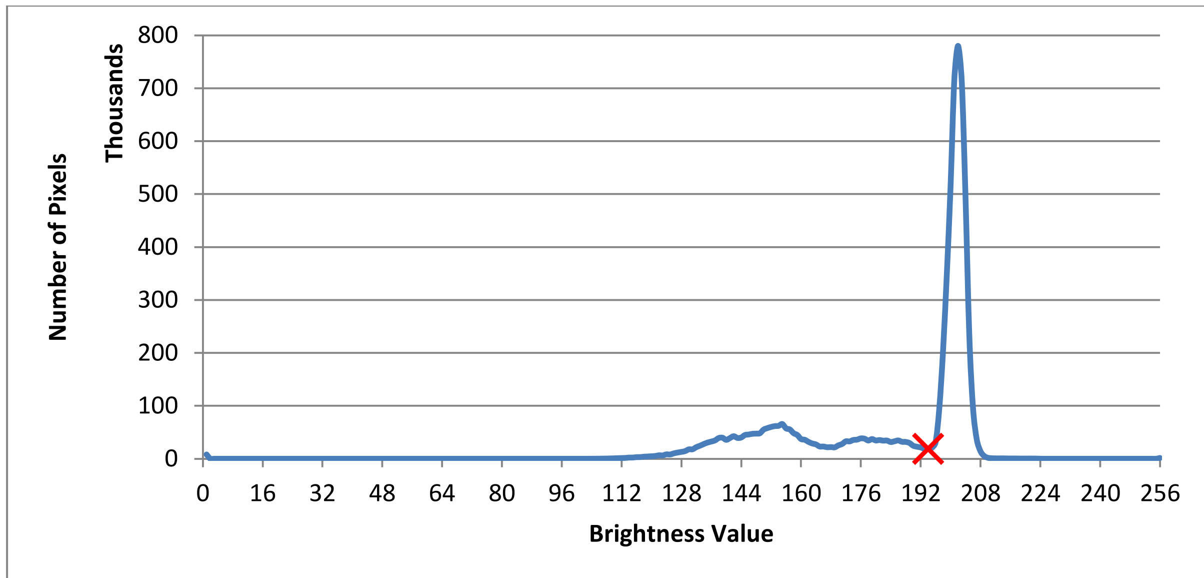

Figure 15 : Typical X-Ray Image Histogram

The above histogram is generated from the image in Figure 3. The soft tissue and bone produce a distinct peak. The threshold value is approximately located at the ***Red*** *X marked on the graph.*

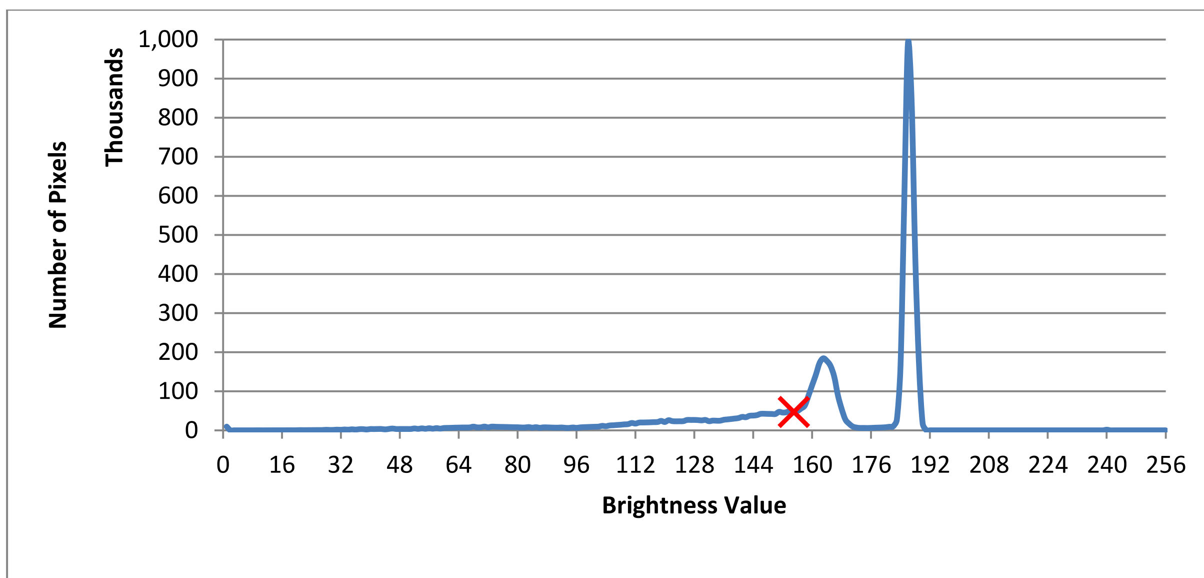

Figure 16 : X-ray Image Histogram with Secondary Background

The above histogram is generated from the image in Figure 5. The soft tissue and bone produced the largest peak and a second smaller distinct peak is produced by the secondary background. The threshold value is approximately located at the ***Red*** *X marked on the graph.*

4.2.3. Sobel Edge Detection

Sobel Edge detection is performed on the image resulting from the previous step [Section 4.2.2]. This will highlight the edges of the bones making them more defined and improving the accuracy of feature extraction process [Section 4.2.4].

4.2.4. Active Contour Model Feature Extraction

Active Contour Model (ACM) feature extraction will be used to ascertain the location of the bones contained within the image. All data outside the contours produced by ACM will be removed from the image as this is likely to be the remaining soft tissue which was not removed by the threshold operation. Overlapping bones could potentially be a considerable problem when segmenting the image. However as multiple X-rays images taken at a number of different angles are processed to generate a mesh, it should be possible to identify the bones by comparing images as they will not overlap in all images.

4.2.5. Threshold Mask

The final image is created by performing a threshold operation on the original unaltered X–ray image using a mask produced using the previous steps. The areas of interest selected contained within the contours generated using the feature extraction technique will be written to the mask image as white pixels and the remainder will be black pixels. Each pixel of the original X-ray is compared in a simple Boolean threshold operation, and those pixels whose corresponding pixel in the mask image is not white will be set to black. Those pixels contained in the areas of interest retain their original brightness value. This is to prevent the median filter from removing the fine details stored within the bones when applied to the image.

4.3. SISD Implementation

The SISD version was implemented in C++, sharing the majority of its source code with the OpenMP implementation [Section 4.4] but with the OpenMP statements removed. This forces the code to be executed sequentially rather than in parallel by the CPU.

4.4. MIMD Implementation

The MIMD implementation shares the majority of its code with its SISD counterpart. OpenMP instructions are used to execute code in parallel. Simultaneous MultiThreading (SMT) or HyperThreading (HT) Technologies are not supported any further than the default support provided by OpenMP constructs.

```
1:      // Wait.
2:      #pragma omp barrier
3:
4:      #pragma omp master
```

Code Listing 6 : OpenMP Synchronisation Constructs

The OpenMP instruction "`#pragma omp barrier`*" [Line 2] is a compiler directive used for synchronisation. It instructs the program to wait until all threads have completed processing the preceding parallel section. Once this has occurred the threads are allowed to continue. The "*`#pragma omp master`*" [Line 4] forces the subsequent code to only execute on a single thread.*

4.4.1. Median Filtering

Due to the requirement of the algorithm to employ a large-radius median filter, the most obvious solution is to adopt a histogram approach to median filtering [Code Listing 4]. This technique offers a reduced memory payload for the chosen radius compared to sorting based methods. Furthermore the code complexity of the median calculation is lower as it does not require an efficient sorting algorithm such as quicksort.

The techniques for reducing the execution time required to process a median filter utilised by the Constant Time Median Filter (Perreault and Hebert, 2007) could be incorporated. However this could reduce the scalability of the algorithm and possibly require a significant number of synchronisation constructs, thereby reducing the benefits derived from parallelisation. Therefore the current implementation is essentially only a parallelised version of the basic histogram method [Code Listing 4], but this does still allow those techniques employed by the Constant Time Median Filter to be incorporated if necessary.

The histogram approach is very scalable as each pixel's calculation of a median value is a self-contained process, requiring no explicit synchronisation. To process the median filter in parallel requires OpenMP compiler directive "`#pragma omp for`" [Code Listing 7, Line 8]. This instruction distributes the computation required for each pixel evenly to all available processing cores. Each thread requires its own location in memory to store a histogram of the area that it is currently computing, therefore the "`private`" keyword [Code Listing 7, Line 4] is employed to facilitate this.

```
1:      int index;
2:      int* histograms;
3:
4:      #pragma omp parallel private(histograms)
5:      {
6:              histograms = new int[256];
7:
8:              #pragma omp for
9:              for(int i = (radius * width); i < numPixels - (radius * width); i++)
10:             {
11:                     //RESET HISTOGRAM!
12:                     for(int j = 0; j < 256; j++)
13:                     {
14:                             histograms[j] = 0;
15:                     }
16:
17:                     int index;
18:                     for(int x = 0; x < radius; x++)
19:                     {
20:                             for(int y = 0; y < radius; y++)
21:                             {
22:                                     index = i + (((y - (radius/2))* width) + (x -
                (radius/2)));
23:                                     histograms[(int)(data[index] * 255)]++;
24:                             }
25:                     }
26:
27:                     int count;
28:                     count = 0;
29:                     for(int j = 0; j < 256; j++)
30:                     {
31:                             count += histograms[j];
32:                             if(count >= ((radius * radius) / 2))
33:                             {
34:                                     temp[i] = ((float)j / 255.0f);
35:                                     break;
36:                             }
37:                     }
38:             }
39:             delete[] histograms;
40:     }
```

Code Listing 7 : OpenMP Median Filter

This implementation is essentially the basic histogram median filter [Code Listing 4] adapted to suit an OpenMP parallel processing model; the median value for each pixel is calculated independently with the pixels being evenly distributed between the processing cores. The main alteration is the declaration of the histogram so that each thread has a separate histogram [Lines 4 and 6] to utilise, removing the need for synchronisation constructs.

4.4.2. Histogram and Thresholding

The calculation of the image's histogram and the optimum threshold value present a more complex problem to parallelise; specifically it requires a great deal of synchronisation to ensure that the correct values are calculated. Parallel reduction appears to be the most common method of utilising parallel processors to computing a histogram. Whilst OpenMP features an intrinsic instruction to

perform parallel reduction [Code Listing 8, Line 3], it cannot be employed when processing arrays and consequently requires the parallel reduction used to calculate the histogram calculation to be specified manually [Code Listing 9].

```
1:      #pragma omp parallel
2:      {
3:              #pragma omp for reduction(+ : nSum)
4:              for (i = 0; i <= 1000; i++)
5:              {
6:                      nSum += i;
7:              }
8:      }
```

Code Listing 8 : OpenMP Reduction Operation

The "`reduction( + : nSum)`*" [Line 3] instruction allows a number of processors to increment the value "*`nSum`*" in parallel with minimal synchronisation. Each thread increments its own copy of "*`nSum`*", and when all threads are finished these are combined by a single thread.*

Each thread calculates a histogram for a portion of the image [Code Listing 9, Lines 11 - 16]. Once this operation has been complete the thread is instructed to wait until all other threads have completed their histogram [Code Listing 9, Line 19]. A single thread combines these sub-histograms into a single histogram for the entire image [Code Listing 9, Line 22 – 29]. This requires that the threads write their particular sub-histograms to a shared location in memory. In order for threads to share a particular location in memory it must be marked with the "`shared`" keyword [Code Listing 9, Line 9].

```
1:      int amount = omp_get_max_threads() * 256; // Allocate enough memory for a
        histogram per thread.
2:      int* hist = new int[amount];
3:
4:      for(int i = 0; i < amount; i++)
5:      {
6:              hist[i] = 0;
7:      }
8:
9:      #pragma omp parallel shared(hist)
10:     {
11:             #pragma omp for
12:             for(int i = 0; i < numPixels; i++)
13:             {
14:                     int value = data[i] / 256; // Convert into 0-255 range.
15:                     hist[value + (omp_get_thread_num() * 256)]++;
16:             }
17:
18:             // Wait.
19:             #pragma omp barrier
20:
21:             // Execute only on a single thread and write the merged histogram to
        the beginning of the shared memory location.
22:             #pragma omp master
23:             for(int j = 0; j < 256; j++)
24:             {
25:                     for(int k = 0; k < omp_get_max_threads(); k++)
26:                     {
27:                             histogram[j] += hist[j + (k * 256)];
28:                     }
29:             }
30:     }
```

Code Listing 9 : OpenMP Histogram Calculation

The histogram of the entire image is unable to use the OpenMP "reduction" instruction because it cannot operate on array data structures. The code above effectively replicates the functionality provided by the "reduction" instruction, but for array data structures.

An alternative would be to use atomic operations [Code Listing 10, Line 14], which would allow all threads to write to the same histogram. However if two threads attempt to write to the same memory location at the same time, one thread must yield and remain idle until the other thread has completed its operation. The likelihood of this occurring is directly related to the distribution of the brightness values in the image, which in the case of the partially processed X-rays is quite low and will therefore impair performance.

```
1:      int* hist = new int[256];
2:
3:      for(int i = 0; i < amount; i++)
4:      {
5:              hist[i] = 0;
6:      }
7:
8:      #pragma omp parallel
9:      {
10:             #pragma omp for
11:             for(int i = 0; i < numPixels; i++)
12:             {
13:                     int value = data[i] / 256; // Convert into 0-255 range.
14:                     #pragma omp atomic
15:                     hist[value]++
16:             }
17:     }
```

Code Listing 10 : OpenMP Histogram Calculation via Atomic Operations

A histogram of the entire image can be calculated by a number of threads using atomic operations to ensure accuracy. The performance of this method of histogram calculation is directly related to the distribution of brightness values in the image and typically exhibits poor performance.

The gradients between histogram bins are calculated independently resulting in this operation being a simple process to parallelise [Code Listing 11, Line 2 – 9]. The next portion of the threshold operation ascertains the location of the peaks and toughs within the image's histogram [Code Listing 11, Line 13 – 43]. This is performed sequentially because a significant amount of the arithmetic relies on the previously calculated value. If the current location is determined to be either a peak or a trough, it along with related useful information is added to a list. This list is then analysed to ascertain which histogram bin contains the trough of the leftmost peak which contains at least 10% of the image's total number of pixels. The actual threshold operation is processed in parallel, as the operation performed on each pixel is completely independent, requiring no explicit synchronisation.

```
1:     //Reversed Iteration Because Image is inversed!
2:     #pragma omp parallel
3:     {
4:            #pragma omp for
5:            for(int i = 255; i >= 0; i--) //skip 0
6:            {
7:                   gradient[i] = ((histogram[i-1]) - histogram[i]) / ((i-1) - i);
8:            }
9:     }
10:    gradient[0] = 0;
11:    area = 0;
12:
13:    for(int i = 254; i > 0; i--) //SKIP 0 also SKIP 254
14:    {
15:           area += histogram[i];
16:
17:           if(gradient[i+1] > 0)
18:           {
19:                  if(gradient[i] <= 0)
20:                  {
21:                         Feature f;
22:                         f.location = i;
23:                         f.percentage = (float)area / (float)numPixels;
24:                         area = 0;
25:                         features.push_back(f);
26:                  }
27:           }
28:    }
29:
30:    for(int i = features.size() - 1; i > 0; i--)
31:    {
32:           if(features[i].percentage >= 0.1f)
33:           {
34:                  largestID = features[i].location;
35:           }
36:    }
37:
38:    //ID is in 0-255 Range
39:    threshold = largestID * (4096 / 256);
40:
41:    //IMAGE IS INVERTED SO >= THRSHOLD INSTEAD OF <=
42:    #pragma omp parallel
43:    {
44:           #pragma omp for
45:           for(int i = 0; i < numPixels; i++)
46:           {
47:                  if(data[i] >= threshold)
48:                  {
49:                         data[i] = 0;
50:                  }
51:           }
52:    }
53:    return data;
```

Code Listing 11 : OpenMP Threshold Operation

This source code calculates the appropriate threshold value by analysing the histogram [Lines 2 – 39]. Once this value is determined it then proceeds with the threshold operation using this value [Lines 42 – 52]. Not all of this process can be effectively processed in parallel [Lines 10 – 39]; therefore they are processed serially using a single processing core. The threshold value is located by analysing the

gradients between histogram bins [Lines 13 – 28] and determining how many of the image's total pixels are contained within the particular peak. The peaks are then examined in reverse order to find the leftmost peak which contains over 10% of the images total pixels [Lines 30 – 36].

4.4.3. Sobel

The Sobel filter is processed in parallel [Code Listing 7, Line 3] with each pixel being totally independent of its neighbours. No explicit synchronisation constructs are required because as previously mentioned the operation performed on each pixel is totally autonomous from all other pixels.

```
1:      #pragma omp parallel
2:      {
3:              #pragma omp for
4:              for(int i = width; i < numPixels - width; i++)
5:              {
6:                      int vert = ((data[i - width - 1] * matrix[0]) + (data[i -
        width] * matrix[1]) + (data[i - width + 1] * matrix[2]) + (data[i - 1] *
        matrix[3]) + (data[i] * matrix[4]) + (data[i + 1] * matrix[5]) + (data[i +
        width - 1] * matrix[6]) + (data[i + width] * matrix[7]) + (data[i + width +
        1] * matrix[8]));
7:
8:                      int horiz = ((data[i - width - 1] * matrix[9]) + (data[i -
        width] * matrix[10]) + (data[i - width + 1] * matrix[11]) + (data[i - 1] *
        matrix[12]) + (data[i] * matrix[13]) + (data[i + 1] * matrix[14]) + (data[i
        + width - 1] * matrix[15]) + (data[i + width] * matrix[16]) + (data[i +
        width + 1] * matrix[17]));
9:
10:                     int value = (int)sqrt((float)((vert*vert) + (horiz * horiz)));
11:
12:                     output[i] = (unsigned short) value;
13:             }
14:     }
```

Code Listing 12 : OpenMP Sobel Filter

Each pixel can be calculated in parallel with the Sobel filter [Lines 1 – 4]. The filter contains a comparatively large amount of arithmetic, multiplying each pixel within the mask area by a corresponding matrix element [Lines 6 and 8]. This is performed twice, once for the vertical edges and again for the horizontal edges. These two values are squared and then combined, after which the square root of this value is calculated [Line 10].

4.4.4. Active Contour Model Feature Extraction

Due to the substantial amount time required to develop the other components of the image processing algorithm, particularly the various approaches to GPU median filtering, it was decided to concentrate on improving the median filtering algorithm rather than implement an Active Contour Model algorithm and to develop this functionality at a later date.

4.4.5. Threshold Mask

This operation is fundamentally the same as the threshold operation [Section 4.4.2], except that the threshold value is altered per pixel according to values stored in a mask image.

4.5. SIMD Implementation

The DirectCompute implementation differs considerably from the CPU algorithms. This is due to a number of reasons, primarily because parallelisation is implicitly implied within the HLSL code. Synchronisation is still explicitly stated and additionally it can only be performed between the threads within the same thread group. Another issue is that data has to be transferred into a GPU accessible data structure and possibly converted to an appropriate data format before it can be processed. The transferring of data between the GPU and CPU is a necessity but incurs overhead so therefore should be kept to a minimum.

4.5.1. Median Filtering

Two median filtering algorithms are proposed, the first is a development of the fast, small-radius median filter algorithm (McGuire, 2008) which exploits the cache memory available in DirectCompute and the second is a histogram method again utilising cache memory.

The large-radius median filter required by the algorithm is particularly problematic for a GPGPU implementation because current GPU accelerated methods are limited to performing median filters with small-radius masks. Performing a small mask multiple times on the image is suggested as a potential workaround (McGuire, 2008), however for this particular problem it will not produce the desired result, as text and other unwanted objects will remain because they are larger than the radius of a small mask. A sorting based method [Code Listing 5] is impractical for use on a GPU as the memory payload increases with the radius size and using a large number of threads in parallel exacerbates the situation. The histogram method [Code Listing 4] has a fixed memory payload regardless of radius size[24] making it possible to implement using either the GPU's global or cache memory. The number of accesses to global memory required by median filters, particularly those with large-radius masks is an acute problem for GPU processing. GPGPU algorithms favour a high compute to memory access ratio (Thibieroz and Cebenoyan, 2009, p.3) to help hide the considerable latency incurred by accessing global memory on a GPU.

Due to the restrictions on how cache memory can be accessed in CS4.0 [Section 4.6.2] the GPU accelerated median filters will employ CS5.0 unless specifically stated otherwise. As the X-ray images

[24] This remains true whilst the number of pixels a single histogram bin may possibly contain is less than the maximum value of a 32bit unsigned integer.

tested have dimensions which are not a power of two values, it may not be possible to allocate the optimum number of threads per SIMD engine on all current GPU architectures. It is possible to pad the image to make it a power of two, however doing so would unfairly penalise Nvidia GPUs as they process data in scalar format rather than vectorized like AMD GPUs and in its current for DirectCompute does not have the functionality to detect which GPU is performing the processing.

4.5.1.1. Fast, Small-Radius Algorithm

This median filtering algorithm is essentially the fast, small-radius median filter algorithm proposed by McGuire (2008) implemented in compute shader form for purely comparative purposes. Initially it reads the mask area to a temporary variable. These values are then sorted using a min-max exchange [Code Listing 13, Line 35 – 39] avoiding the performance penalties incurred when performing a sorting algorithm which utilises branch operations (McGuire, 2008, p.168).

4.5.1.2. Caching Algorithm

The caching algorithm employs the fast, small-radius algorithm as the basis of the processing, but crucially it pre-loads the relevant portion of the image data into the available cache memory. The principal aim is to minimise the number of accesses to global memory required by the median filter, similar to the caching method suggested by Zink (2010). The median filter masks of neighbouring pixels overlap considerably [Figure 17], resulting in the fast, small-radius algorithm containing a large number of redundant accesses to global memory. As each thread group processes a strip of neighbouring pixels from the image to perform the median filter operation, a significant portion of these memory accesses can be transferred to cache memory by pre-loading the relevant portion of the image. The caching algorithm does not reduce the number of redundant memory accesses but it does transfer them to cache memory which offers improved performance compared to global memory (Green, 2009).

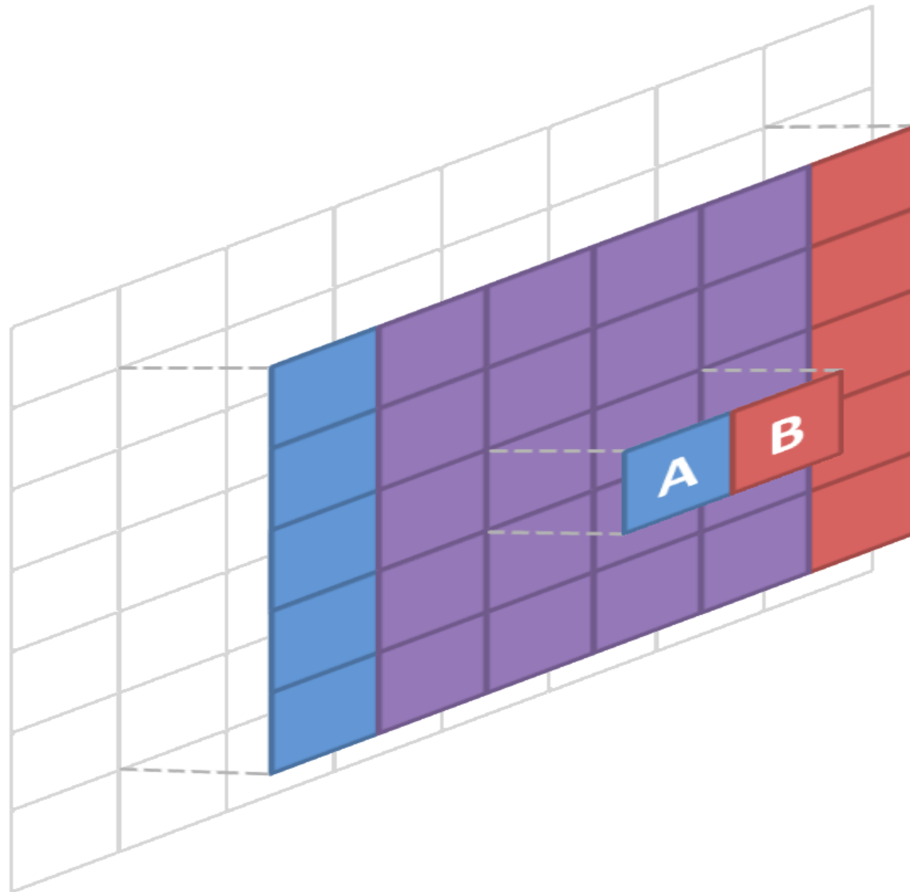

Figure 17 : Mask Overlaps between Neighbouring Pixels

This diagram depicts the amount that overlap (Purple) in a (5×5) median filter mask for two neighbouring pixels. The back layer (White Boxes) represents the entire X-ray image, the mid layer represents those pixels which need to be accessed to calculate the median values. The front layer is the two pixels for which the median values are being calculated. Pixel A's mask (Blue and Purple) and Pixel B's (Red and Purple) Mask contain 20 shared pixels within the overlapping area of the masks.

The caching algorithm functions by having each thread within a thread group transfer a single column[25] of pixels from global memory into cache memory. To achieve this, the algorithm requires one thread for each pixel that is contained within the strip to be processed and an additional (radius - 1) threads. An alternative method would be to use one thread for each pixel in the strip and to distribute the pixels contained within relevant portion of the image amongst these threads, removing the need for the additional (radius - 1) threads. This may however lead to the threads having unbalanced workloads, which impairs performance on SIMD architectures. A synchronisation construct [Code Listing 13, Line 18] is used to guarantee that the entirety of the relevant portion of the image is transferred into cache memory before the threads proceed with the median calculation.

[25] The algorithm would function in the same way if the strip were allocated vertically instead of horizontally, except that each thread would load a row of pixels instead of a column.

The calculation of median value remains identical to the fast, small-radius filter [Code Listing 13, Line 35 – 39] except that the additional threads used to help load the image do not perform the calculation and remain idle [Code Listing 13, Line 23].

For example, based on this algorithm [Figure 18], if a thread group operates on a strip of four pixels using a (3×3) median filter mask then the fast, small-radius algorithm would require four threads (one per pixel) but the caching algorithm would require an additional two (radius – 1) threads for a total of six.

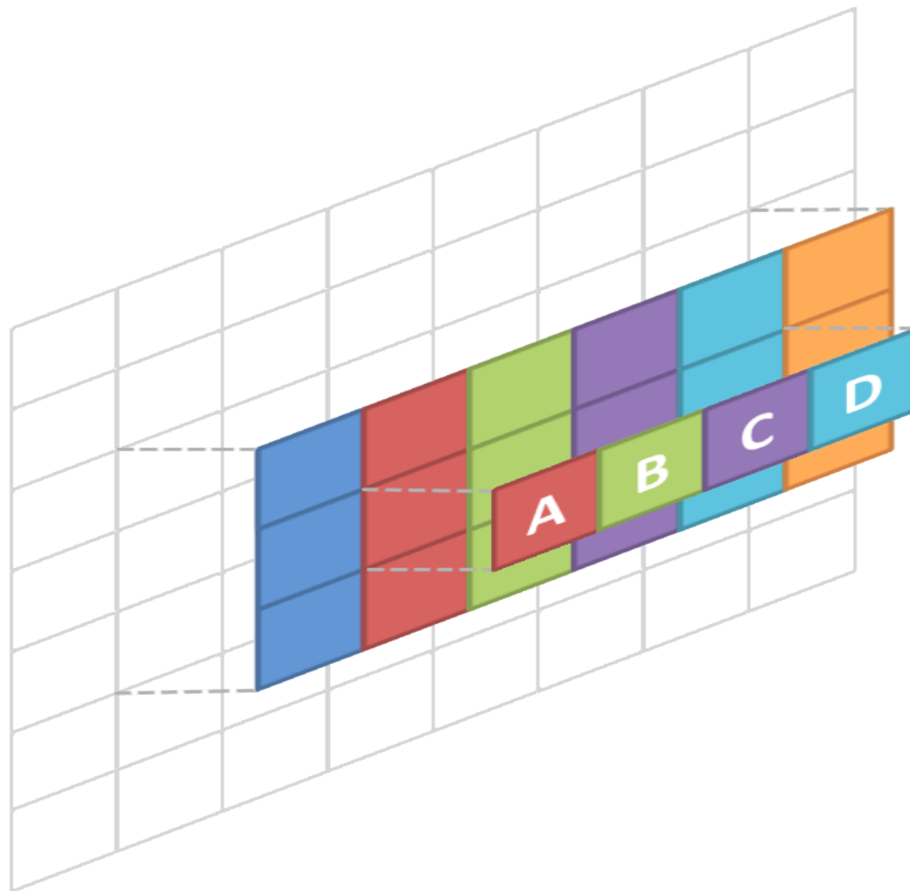

Figure 18 : Caching 3x3 Median Filter Diagram

*This diagram demonstrates the number of threads the caching algorithm requires to calculate the median values for a strip of four neighbouring using a (3×3) mask. The back layer (White Boxes) represents the entire X-ray image, the mid layer represents those pixels which need to be accessed to calculate the median values. The pixels are colour coded to identify which particular thread they are loaded into cache memory by. The **blue** pixels are loaded by thread 1, the **red** pixels by thread 2, the **green** pixels by thread 3, etc. The front layer is the four pixels for which the median values are being calculated, thread 1 and 6 (**Blue** and **Orange**) do not perform any median calculations once they have loaded data into the cache. The remaining threads (two to five) calculate the median values for their particular pixel (thread 2 = A, thread 3 = B, etc.) using all the relevant image data loaded into cache*

*memory. For example, thread 2 when calculating the median value of pixel A will use the data values loaded into cache memory by threads 1 to 3 (**Blue**, **Red**, **Green**) to calculate the median value.*

```
1:      groupshared uint cache[42*1*3];
2:
3:      [numthreads(42,1,1)]
4:      void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
5:      {
6:              uint v[9];
7:
8:              int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
        yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
9:
10:             int offset;
11:             //Each thread caches 3 values, one above, one below and the current
        location
12:             for(int p = 0; p < 3; p++)
13:             {
14:                     cache[(gt.x *3) + p] = Buffer0[i - width + (p * width)].i;
15:             }
16:
17:             //Wait till cache is filled
18:             GroupMemoryBarrierWithGroupSync();
19:
20:             int count = 0;
21:
22:             //Skip the outer threads
23:             if(gt.x != 0 && gt.x != xNumThreads)
24:             {
25:                     for(int x = 0; x < 3; x++)
26:                     {
27:                             for(int y = 0; y < 3; y++)
28:                             {
29:                                     offset = ((gt.x +(x-1))*3) + y;
30:                                     v[count] = cache[offset];
31:                                     count++;
32:                             }
33:                     }
34:                     uint temp;
35:                     mnmx6(v[0], v[1], v[2], v[3], v[4], v[5]);//Exchange 7
36:                     mnmx5(v[1], v[2], v[3], v[4], v[6]); //Exchange 6
37:                     mnmx4(v[2], v[3], v[4], v[7]); //Exchange 4
38:                     mnmx3(v[3], v[4], v[8]); //Exchange 3
39:                     BufferOut[i].i = v[4];
40:             }
41:     }
```

Code Listing 13 : Compute Shader 5.0 Caching Median Filter

This is the implementation of the Caching algorithm using the Compute Shader 5.0. The cache memory is allocated [Line 1] corresponding to the number of threads [Line 3] and the mask radius. Each thread loads a column of pixels equal to the size of the radius into cache memory [Lines 12 – 15]. Once completed the thread waits until all other threads have finished performing the same action. The threads specifically created to load data into the cache do not calculate a median value [Line 23]. The remaining threads use the min-max exchange sort to calculate the median [Lines 35 –

39]. The mask area for a particular pixel is accessed from the cache memory and loaded into temporary variables [Lines 25 -33].

4.5.1.3. Histogram Algorithm

The cache memory available in DirectCompute allows a histogram method of median filtering to be implemented. Essentially the histogram method proposed is the basic histogram median filter [Code Listing 4] with minor amendments to make it suitable for GPU processing. These include the allocation of cache memory to allow each thread to store a histogram and alteration of the median calculation logic to enable it to function with the restrictions on loops in HLSL. Each SIMD engine has 32KB of addressable cache memory and with each histogram requiring 1KB of cache memory this effectively limits the maximum number of threads to 32 per SIMD engine. The HLSL compiler is capable of unrolling the loop for median calculation to improve performance, but this operation is restricted to loops which do not use flow control to escape early [Code Listing 7, Line 35] (MSDN, 2010c). Loop unrolling improves performance because dynamic flow control can take longer to execute if there are a small number of instruction for each branch (Sander, 2005) as is the case here [Code Listing 14, Lines 4 - 7].

This algorithm cannot be realized in CS4.0 because threads are restricted to writing to a single location in cache memory, making it impractical to implement. This method requires a significant number of branching operations, however the number of branches is constant, unaffected by the radius size, unlike a sorting based filter.

```
1:      for(int i = 0; i < 256; i++)
2:      {
3:              count += cache[(threadID * 256) + i];
4:              if(count <= midPoint)
5:              {
6:                      value = i;
7:              }
8:      }
```

Code Listing 14 : HLSL Median Calculation

So that the for loop can be unrolled by the HLSL compiler the median calculation is altered slightly from the basic histogram median filter [Code Listing 4, Lines 14 - 18]. This allows the correct median value to be found without having to use an intrinsic flow control instruction to escape from the loop once it is found.

4.5.2. Histogram and Thresholding

As indicated to in the OpenMP implementation, the calculation of the histogram and optimum threshold value is difficult parallelise. This situation is exacerbated in GPGPU as it is a massively

parallel processor, so constructing the operation in a manner which allows it to be efficiently processed on a GPU is extremely difficult. As such the processing of this section of the algorithm will utilise the MIMD code and perform the computation on the CPU. This negates much of the overhead which would be incurred transferring data, in order to process the serial components on the CPU. The actual thresholding operation once the threshold value has been calculated is suitable for parallel processing and consequently will utilise the GPU. The shader itself is effectively a simple branch operation. The sheer number of processors executing the code in parallel should limit the performance impact of using a branch instruction. A vectorized implementation would allow the use of the hardware accelerated "`min`" intrinsic instruction to threshold the image improving performance but requiring a more complex threading model.

```
1:      [numthreads(40,1,1)]
2:      void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
3:      {
4:
5:              int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
        yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
6:
7:              uint value = Buffer0[i].i;
8:              if(value >= threshold)
9:              {
10:                     value = 0;
11:             }
12:
13:             BufferOut[i].i = value;
14:     }
```

Code Listing 15 : GPU Threshold Filter

In the DirectCompute version of the algorithm, only the actual threshold operation can be processed effectively by the GPU. The histogram and threshold value are calculated by the CPU and passed to the GPU via the Constant Registers.

4.5.3. Sobel

The final stage in the current implementation of the algorithm is to apply a Sobel filter to the image. The Sobel filter is ideally suited to a GPGPU, as each pixel can be processed totally independently and the operation has a high arithmetic intensity compared to the other image processing techniques implemented as a part of the algorithm. The proposed caching algorithm [Section 4.5.1.2] could be applied to further reduce the execution time of the algorithm.

4.5.4. Active Contour Model Feature Extraction

As previously mentioned [Section 4.4.4], it was decided to not implement Active Contour Model Detection until the remainder of the algorithm had been satisfactorily developed. However the GPU

accelerated active contours algorithm proposed by Tatarchuk (2008) could be used as the basis of the GPGPU implementation.

4.5.5. Threshold Mask

This operation is essentially the same as the threshold operation [Section 4.5.2], except that the result of the threshold operation is determined by performing a lookup which compares the pixel to the corresponding pixel in the mask image.

4.6. Development Issues

The development of the SIMD algorithm encountered a number of problems which required addressing.

4.6.1. Limited Data Types Available on GPU

The X-ray image's native data format is 16bit unsigned short integers (ushort) which whilst supported on the CPU are not available for use on the GPU. Instead DirectCompute supports the following scalar data types:

- 32bit Unsigned Integer (uint)
- 32bit Signed Integer (int)
- 32bit Floating Point (float)
- 64bit Double Precision Floating Point (double)
- 16bit Half Precision Floating Point (half)

Due to this restriction, the unsigned integer data type was chosen as it requires only a cast operation to convert the data from the unsigned short integer data type to unsigned integer, allowing the GPU processing to conform the objective stating that accuracy must not be compromised to expedite processing [Section 1.4.2-7]. This is a major problem for the histogram median filter as there is limited cache memory available. This lack of a 16bit integer data type effectively limits either the bins each histogram can contain or the maximum number of threads which can be executed in parallel per thread group as the unsigned integer data type has twice the memory payload of an unsigned short integer[26].

4.6.2. Reduced Functionality in CS4.0

The previously suggested method for improving the fast, small-radius median filter by caching data [Section 4.5.1.2] is restricted to Shader Model 5 hardware because CS4.0 offers far more limited functionality. The most significant problem is that CS4.0 devices can only write to locations in cache

[26] 16bits for an unsigned short integer compared to 32bits for an unsigned integer.

memory that corresponds to the particular thread identity number (Thibieroz, 2009, p.25). This issue can however be negated for (3×3) median filters using CS4.0 hardware by utilising a three element vector as the storage data type [Code Listing 16, Line 1 & 6] instead of the scalar data type [Code Listing 13, Line 1 & 6] as used in the CS5.0 implementation. The result is that each thread is required to write to a single location in cache memory instead of three. This allows the algorithm to conform to the restrictions of the Compute Shader 4 specification and therefore run on Shader Model 4 hardware. The remaining alterations to the caching algorithm allow it to sort the data in vector format. The inability of CS4.0 devices to write to random locations in cache memory also prevents the histogram method of median filtering [Section 4.5.1.3] being implemented due to the number of memory locations required per thread. CS4.0 has a number of additional limitations including:

- No Atomic Operations.
- Reduced Cache Memory Size.[27]
- Slower Execution.[28]

4.6.3. Other Issues

Due to the number of development issues, the considerable variety in the GPU architectures available and the lack of Nvidia GPUs to test with, it has not been possible to perform extensive testing, essentially limiting testing to the AMD HD 5000 series GPU architecture. Therefore all recommendations and assumptions about GPGPU are based solely on the GPGPU performance of the AMD HD 5000 series GPU architecture.

[27] 16KB per thread group instead of 32KB.

[28] A DirectX 11 GPU executing the same program, compiled to CS4.0 instead of CS5.0 exhibits reduced performance.

```
1:      groupshared uint3 cache[42];
2:
3:      [numthreads(42,1,1)]
4:      void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
5:      {
6:              uint3 v[3];
7:
8:              int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
        yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
9:
10:             int offset;
11:             //Each thread caches 3 values, one above, one below and the current
        location
12:             cache[gt.x].x = Buffer0[i - width].i;
13:             cache[gt.x].y = Buffer0[i].i;
14:             cache[gt.x].z = Buffer0[i + width].i;
15:
16:             //Wait till cache is filled
17:             GroupMemoryBarrierWithGroupSync();
18:
19:             int count = 0;
20:
21:             //Skip the outer threads
22:             if(gt.x != 0 && gt.x != xNumThreads)
23:             {
24:                     for(int x = 0; x < 3; x++)
25:                     {
26:                             offset = (gt.x + (x - 1));
27:                             v[count].x = cache[offset].x;
28:                             count++;
29:                             v[count].y = cache[offset].y;
30:                             count++;
31:                             v[count].z = cache[offset].z;
32:                             count++;
33:                     }
34:             }
35:             float temp;
36:             mnmx6(v[0].x, v[0].y, v[0].z, v[1].x, v[1].y, v[1].z); //Exchange 7
37:             mnmx5(v[0].y, v[0].z, v[1].x, v[1].y, v[2].x); //Exchange 6
38:             mnmx4(v[0].z, v[1].x, v[1].y, v[2].y); //Exchange 4
39:             mnmx3(v[1].x, v[1].y, v[2].z); //Exchange 3
40:             BufferOut[i].i = v[1].y;
41:     }
```

Code Listing 16 : Compute Shader 4.0 Caching Median Filter

The differences between the Compute Shader 4.0 version of the cached median filter and the Compute Shader 5.0 implementation [Code Listing 13] are highlighted in ***red****. The uint3 is a vector data type capable of storing three scalar values.*

4.6.4. Hardware Specific Optimisations

Achieving maximum performance of the GPU implementation proved to be a significant challenge throughout the development of the algorithm, because hardware specific optimisations were required to leverage maximum performance from the GPU. Some manufacturers utilise vector processors in their GPU architectures [Section 2.2.3.2] and as such require explicit vectorization of arithmetic to yield maximum performance unlike GPUs with scalar processors. Research indicates

that vectorization can produce performance improvements between 1.45x-6.7x (Jang et al., 2009, p.1).

Determining the optimum number of threads is a major problem as it varies depending on the particular GPU hardware executing the program. An additional issue with selecting the optimum number of threads is that values must be selected before the shader is compiled as they are manually specified in HLSL code [Code Listing 16, Line 3] using the "`numthreads()`" instruction.

5. Results and Analysis

5.1. Overall Performance

This section examines the speedup in performance achieved by the MIMD and SIMD parallel processor architectures when computing the algorithm for pre-processing x-ray images compared to a sequential SISD architecture. All three versions of the algorithm were tested using the X-ray data both in its native format and converted to floating point format.

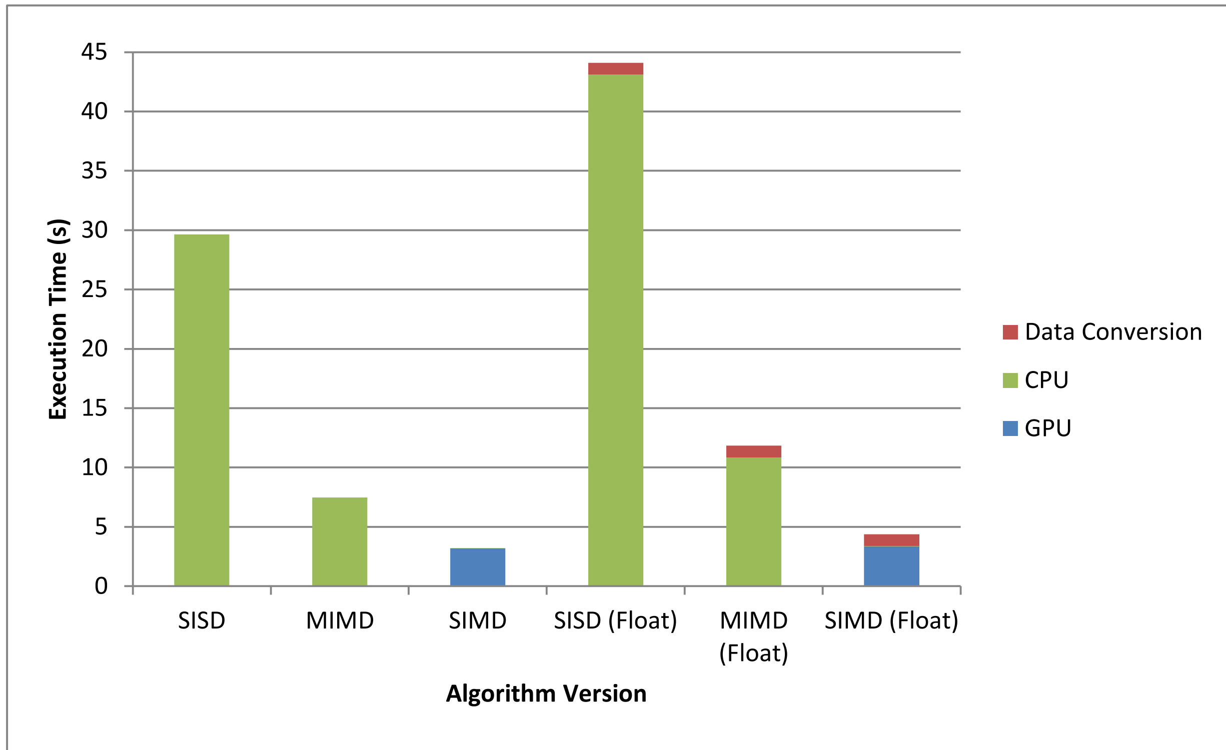

Figure 19 : Algorithm Performance Results

This graph illustrates the relative performance of the three processor architectures used to compute the algorithm. The SIMD algorithm requires the CPU to execute a small portion of the algorithm as the GPU is unable to efficiently process some components, however this is not clearly visible on the graph. The graph also demonstrates the size of the performance penalty incurred by converting the data to floating point format.

Table 3 : Overall Algorithm Speedup

Data Type	Speedup		
	SISD	MIMD	SIMD
Integer	1.00x	3.96x	9.22x
Float	1.00x	3.72x	10.08x

The results of this experiment clearly indicate that both forms of parallel processing offer a significant performance improvement over a serial processor. The efficiency of the MIMD implementation can be calculated [Equation 8] as 96%[29] and therefore it is possible to obtain a maximum potential speedup of 25x[30]. As previously discussed [Section 2.2.4.1], the efficiency of the SIMD algorithm cannot be calculated because only the execution times of the three implementations are directly comparable.

Additionally, the graph [Figure 19] demonstrates that processing the data in floating point format takes significantly longer, increasing the execution time by approximately 30% for the CPU implementations and 5% for the GPU. This increase is without considering the additional overhead incurred converting the data from its native data format to floating point. Whilst the performance penalty incurred converting data is relatively small, it has comparatively greater impact on the overall execution time of the GPU. These results would appear to support the hypothesis that whilst floating point data may be the optimum format for processing data on a GPU, it is only relevant for those algorithms which are arithmetically intensive rather than those which are memory access intensive.

The SIMD algorithm cannot be entirely processed by the GPU and requires the CPU to perform some of the processing. This constrains the maximum potential speedup as those sections which require processing on the CPU incur additional performance penalties due to the overhead of data being transferred to and from the GPU each time this occurs.

It should be acknowledged that both the SISD and MIMD algorithms could be optimised to reduce their execution times using the SIMD capabilities of the CPU via SSE instructions. Research into using SSE and AVX instructions for image processing (Larsson and Palmer, 2009) indicates that potentially a two to three fold reduction in execution times can be achieved. This would significantly reduce the disparity between the execution times attained by the SIMD and the MIMD implementations. However, this would necessitate additional code and increase code complexity as processing with SSE or AVX instructions requires data to be aligned to 16 and 32 bytes respectively.

5.2. Individual Component Performance

This section discusses the results of the testing on the individual components of the algorithm.

[29] This is based on the average performance of the MIMD implementation across both data types.
[30] Assuming sufficient processing cores are available to achieve this.

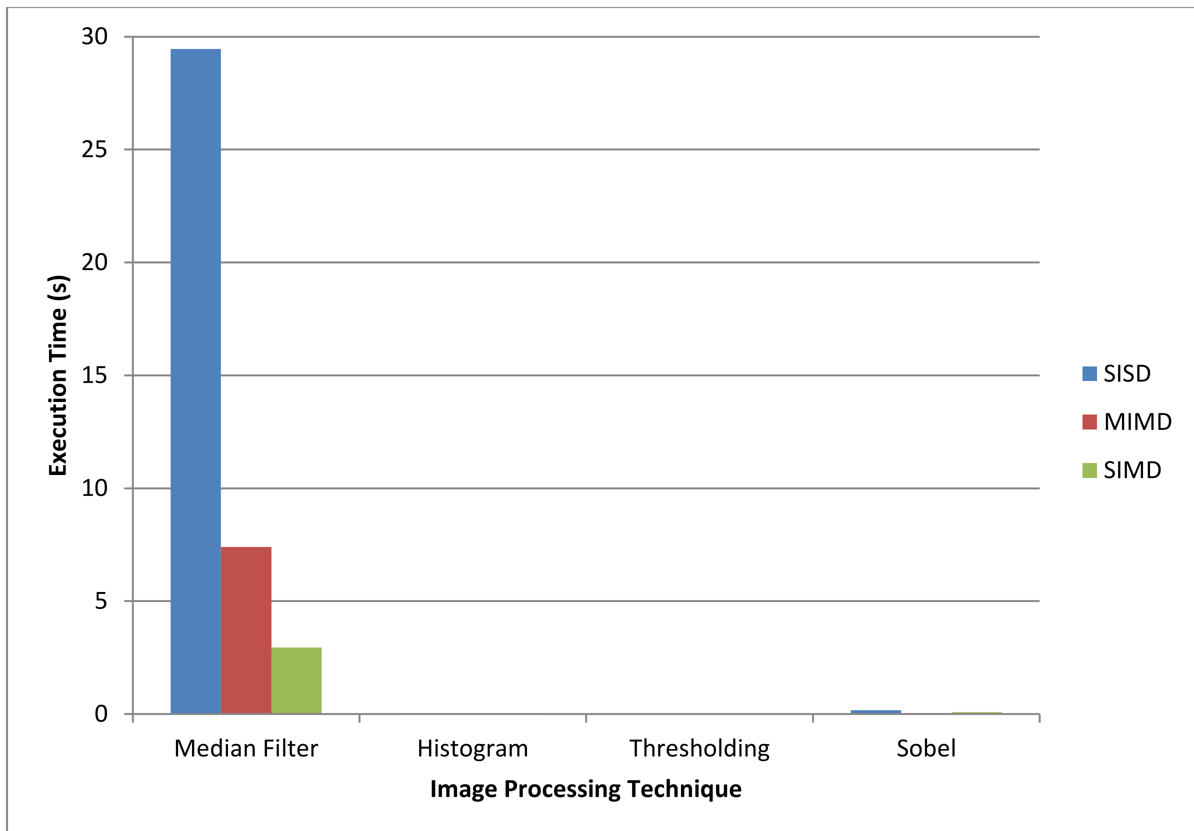

Figure 20 : Individual Component Performance Results (Overall)

This graph breaks down the total time required to process the algorithm into its individual components. The graph clearly illustrates that the median filter is by far the most time consuming component to process for all architectures.

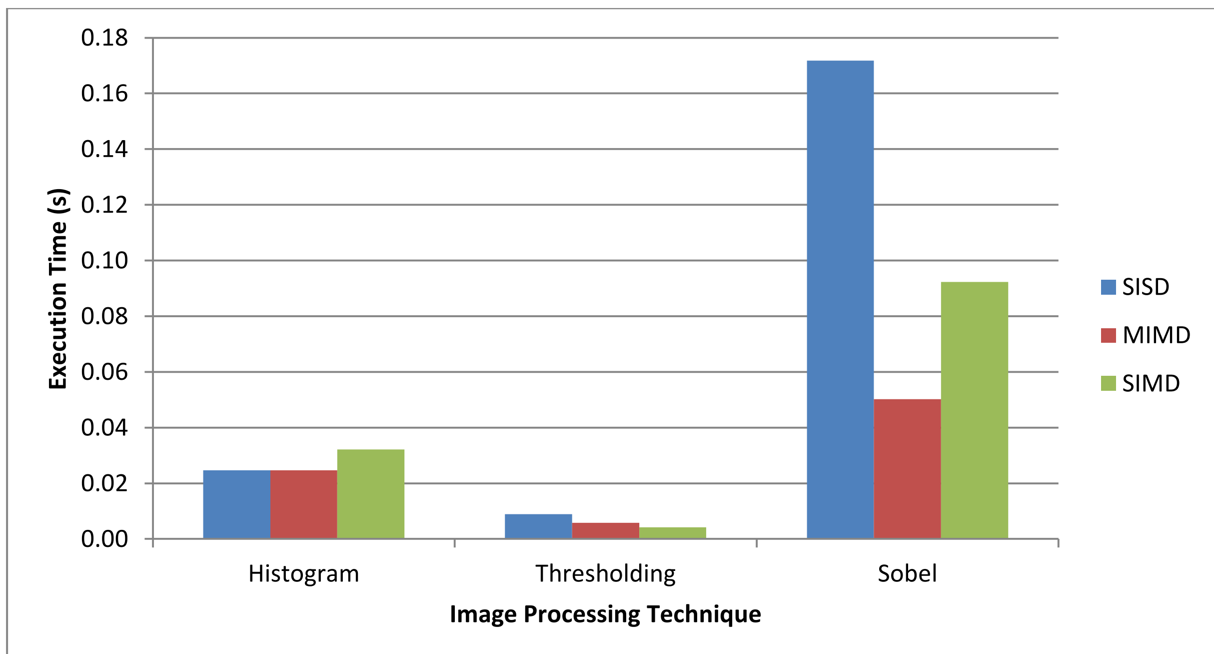

Figure 21 : Individual Component Performance Results (In Detail)

This graph excludes the Median filter so the Histogram, Thresholding and Sobel components can be examined in detail. The results indicate that the SIMD algorithm does not always offer considerably improved performance as is the case with the Median Filter.

Table 4 : Individual Component Speedup

Image Processing Technique	Speedup		
	SISD	MIMD	SIMD
Median	1.00x	3.98x	10.00x
Histogram	1.00x	1.00x	0.77x
Thresholding	1.00x	1.54x	2.13x
Sobel	1.00x	3.42x	1.86x

The overall performance graph [Figure 20] illustrates how the time required to compute the median filter dwarfs the combined execution time of the algorithm's other components. This is because the median filter is significantly more computationally intensive than the other techniques combined. This would be significantly less pronounced if a smaller radius mask was used, although it would not achieve the desired effect. The results indicate that GPGPU can achieve significant performance gains over other forms of processing. It also demonstrates that the parallelisation of the other components has produced negligible impact on the overall execution time and as such concentrating on improving the performance of the median filter would have been more beneficial. The results [Table 14] for the histogram component indicate that transferring data to and from the GPU is a comparatively costly process. This can be surmised because the GPU algorithm actually calculates the histogram component using the CPU and should therefore take the same amount of time to execute. However the GPU must transfer the data to the CPU and back again once the histogram is calculated, which must be the source of the reduced performance. Therefore it is recommended that wherever possible data transfers are kept to a minimum.

The thresholding and Sobel components imply that GPGPU may not always offer a significant performance improvement over a MIMD implementation. In fact the results indicate that the Sobel filter is slower to execute when processed on GPU than a multi-core CPU. As previously suggested the use of SSE or AVX instruction to enhance the CPU's performance would exacerbate the situation.

5.3. Data Format Performance

This section compares the overall execution time of the algorithm using different data types.

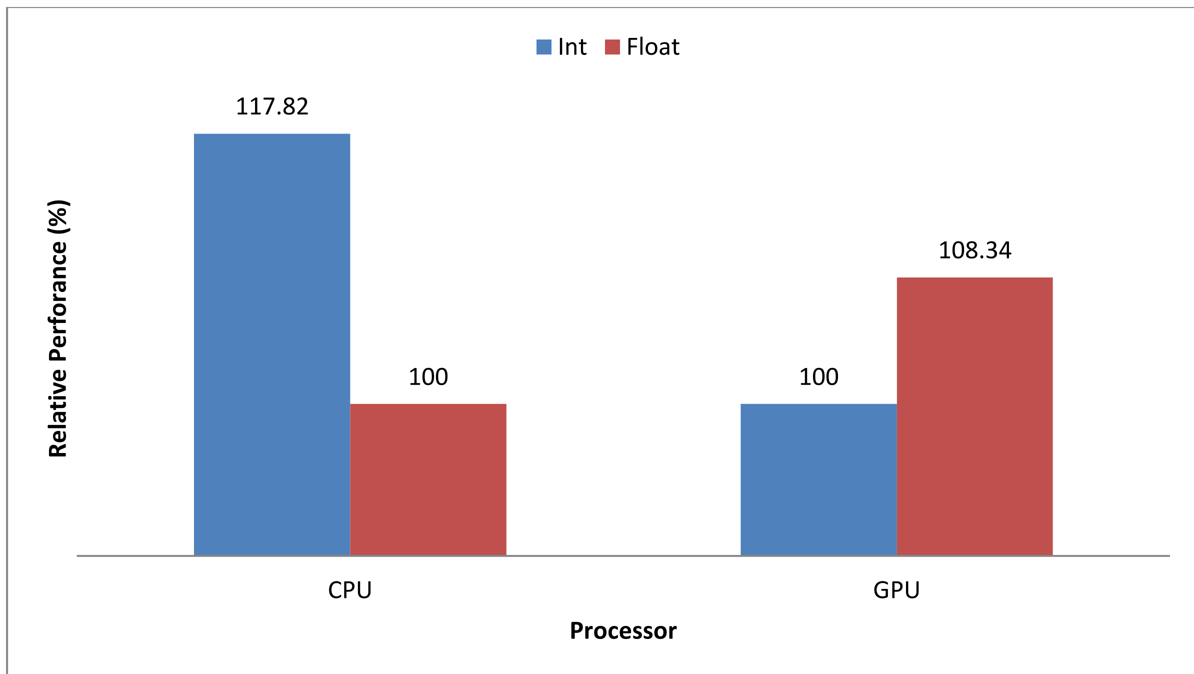

Figure 22 : Data Format Performance

The chart demonstrates the relative performance of different data types when processed using a particular processor architecture.

Based on the execution timings for a small-radius (3×3) mask using the fast, small-radius median filter, we can establish that the CPU suffers from approximately an 18% penalty from processing data in floating point format rather than integer. Conversely the opposite is true for GPU, albeit with a much smaller penalty of 8%. The magnitude of the performance differences recorded for this test does not reflect the results from testing the entirety of the algorithm [Section 5.1]. This can most likely be attributed to the fact that the median filter may not in retrospect have been the most appropriate choice to test the performance difference of the two data formats. This is because it has relatively few areas where floating point arithmetic occurs, the majority being integer mathematics regardless of the data format. Performing this test with the Sobel filter may have been a more appropriate choice in retrospect as it contains far more arithmetic calculations and would have better reflected the actual performance difference. However this is not without its own drawbacks, as the Sobel filter's performance is still relatively dependent on memory accesses.

5.4. Median Filtering Performance

This section details the results of the testing conducted on the various median filtering algorithms. The median filters were tested with a small (3×3) and a large (19×19) radius mask to allow a more comprehensive evaluation of the various algorithms. Specifically the test was used to determine if the caching algorithm gave any performance increase over the fast, small-radius and if the GPU histogram method offered enhanced performance over a parallel CPU implementation.

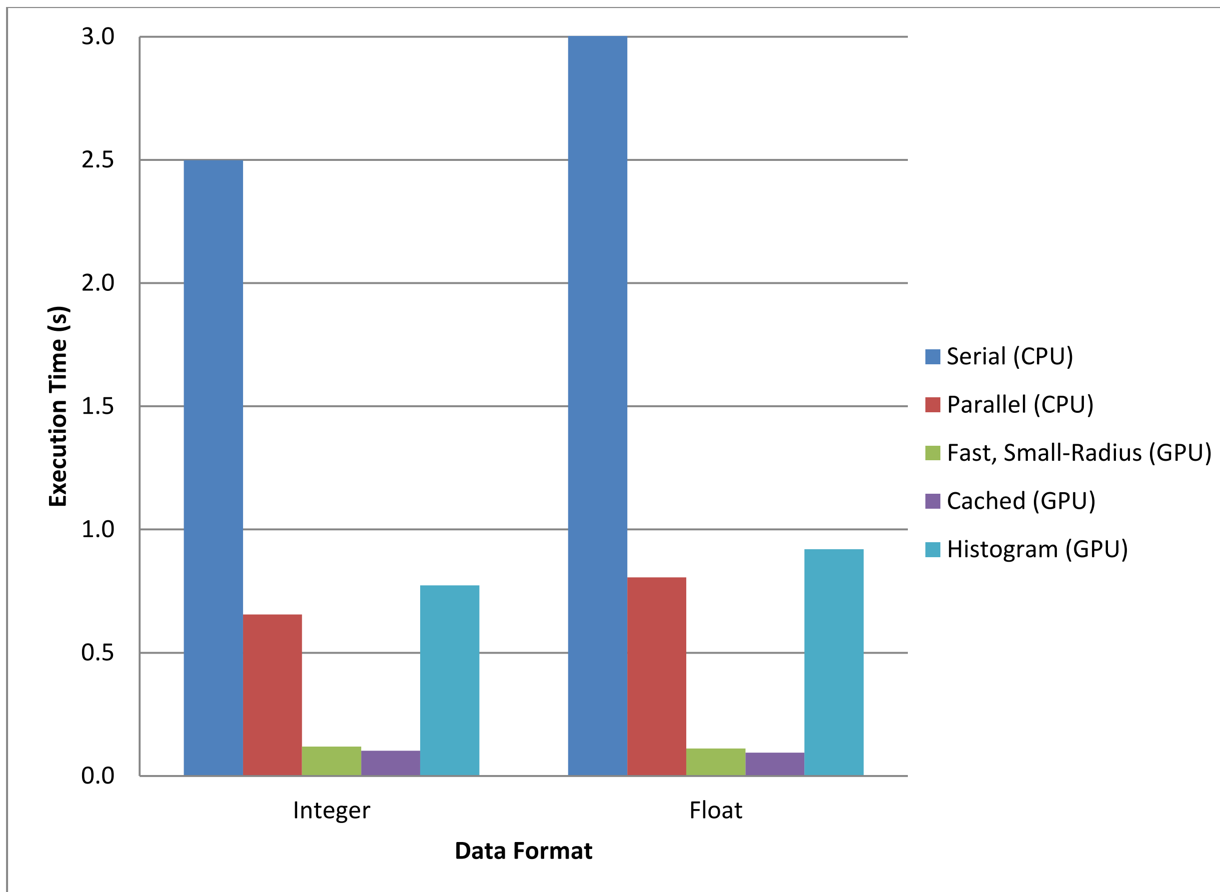

Figure 23 : Median Filter Performance Results

This chart depicts the performance timings of the various median filtering algorithms applying a (3×3) *mask to the X-ray image.*

The results of this experiment indicate that the GPU median filters (when it is possible to utilise them, i.e. for small radius masks or when sufficient cache memory is available, adopting a histogram based approach) in most cases offer superior performance to either of the CPU implementations. However for small-radius filters the GPU histogram method is in fact slower than the parallel CPU implementation. However this trend does not continue with large-radius masks, where the GPU histogram algorithm offers significantly improved performance compared to the performance of all other implementations [Figure 24].

The data supports the following conclusions:

- For small-radius masks the fast, small-radius and caching GPU algorithms considerably reduce execution times in comparison to the CPU implementations and the proposed GPU Histogram method [Table 5]. They offer at least a fivefold increase in performance compared to all other implementations.

- The GPU histogram method offers increasingly improved performance as the filter's mask radius increases. With a (19×19) radius mask the GPU histogram implementation reduces of the execution time by a factor of ten. This result replicates the findings of a comparative study into the potential speedup processing a median filter using a GPU rather than a CPU (Castaño-Díeza et al., 2008, p.12).
- That by transferring the redundant memory accesses required by the fast, small-radius filter from global memory to cache memory offers approximately a 15% reduction in the processing time required to process a (3×3) median filter.

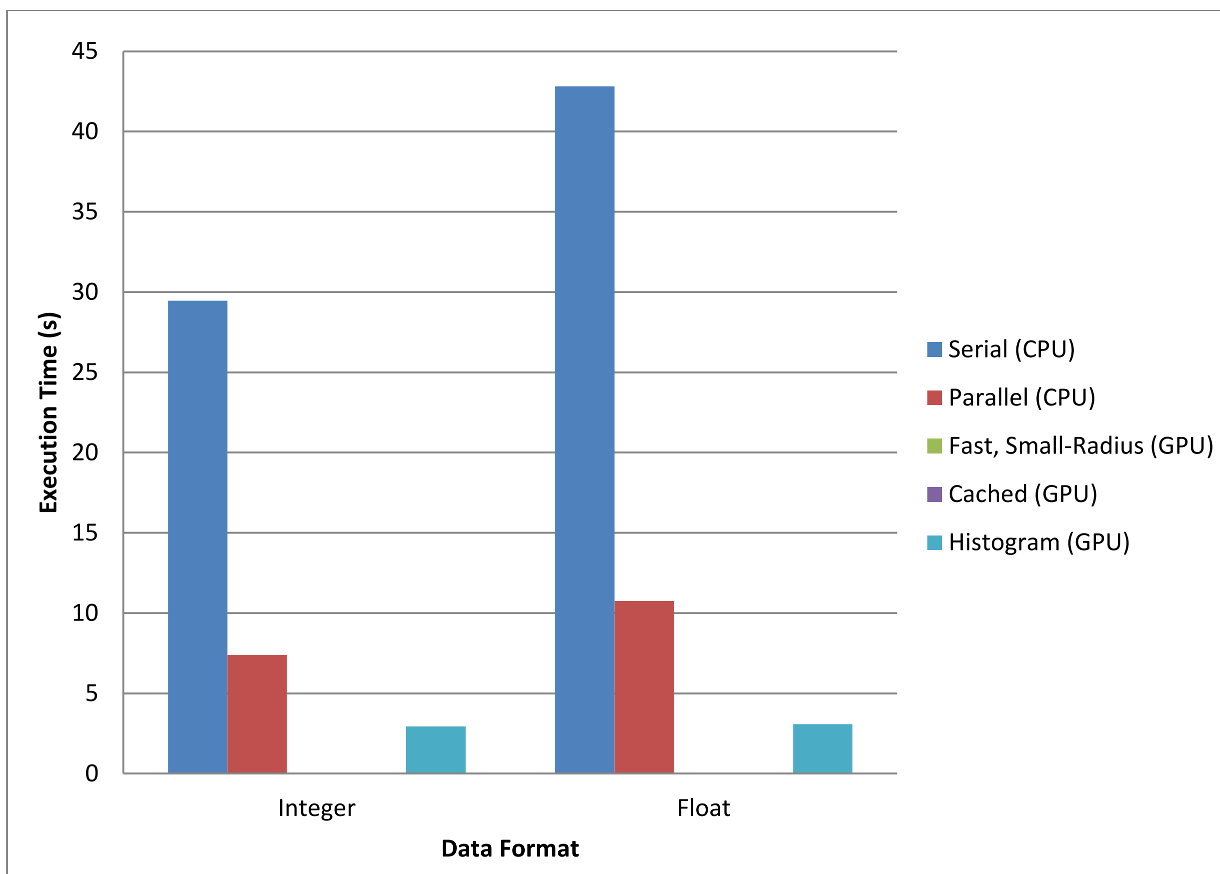

Figure 24 : Median Filter Performance Results (Large-radius)

This chart depicts the performance timings of the various median filtering algorithms applying a (19×19) *mask to the X-ray image. The Fast, Small-Radius and Cached GPU implementations are unable to apply such a large mask.*

A detailed analysis of the caching algorithm has revealed that the number of redundant global memory accesses has been reduced by 66%. This has not however produced a performance increase as substantial as anticipated. This could be attributed to the algorithm using unaligned memory accesses to retrieve data from the cache memory, causing them to be serialised and therefore impairing performance (Bordawekar et al., 2010a).

The performance improvement achieved by the caching algorithm does however illustrate that accessing global memory on the GPU incurs performance penalties and should be avoided if possible. The caching algorithm could in fact be applied to numerous other areas, for example the GPU Sobel mask and the GPU histogram median filters, both of which have redundant global memory accesses.

Table 5 : Median Filter Speedup

Radius	Speedup				
	Serial (CPU)	Parallel (CPU)	Fast, Small-Radius (GPU)	Caching (GPU)	Histogram (GPU)
3 x 3	1.00x	3.82x	21.10x	24.52x	3.23x
19 x 19	1.00x	3.98x	N/A	N/A	10.00x

5.5. Optimum Number of Threads

This test is designed to determine the optimum number of threads for the test hardware configuration. As previously identified [Section 4.6.5], it is difficult to accurately determine the optimum number of threads per thread group to use for specific GPU hardware. The optimum value depends on not only the algorithm's efficiency but also the number of SIMD engines the hardware contains and the number of processing cores they consist of. Performance timings were taken with the Sobel filter and the fast, small-radius median filter, primarily because these facilitated large thread groups which allowed more comprehensive testing to be conducted. The results of this experiment were considered when selecting the number of threads to utilise for the various components of the algorithm.

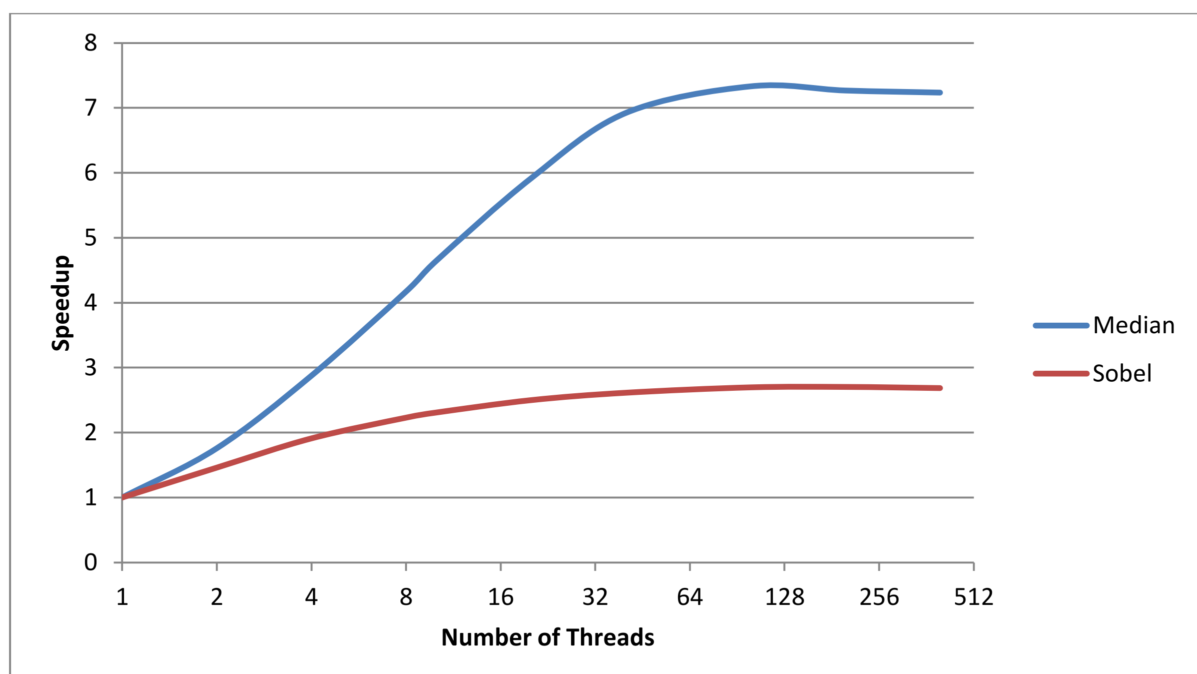

Figure 25 : Optimum Number of Threads

This graph illustrates the speedup obtained on the test GPU by increasing the number of threads each SIMD engine must execute.

The graph [Figure 25] exhibits a similar trend to that predicted by Amdahl's Law [Figure 11], with diminishing returns for larger thread groups. On closer inspection however, when the number of threads exceeds 100 the performance of the median filter is actually reduced. This can be attributed to poor balancing of work between thread groups when using large numbers of threads, as the total amount of computation required remains constant with only the number of threads being varied. Therefore when using a large number of threads some SIMD engines may be idle as they are being starved of work. This is because the processing is not distributed evenly to all SIMD engines. The number of units of work each thread group has to process can be calculated with the following equation [Equation 11]. If this number is not a whole number then it is likely starvation will occur in some SIMD engines.

$$W = \frac{A}{(T \times S)}$$

Equation 11 : Work per SIMD Engine

The formula calculates the amount of work W, allocated per SIMD Engine, when a GPU consists of S SIMD Engines executing T threads in parallel for the total amount of work A.

Using this equation it can be determined that for the X-ray images allocating 100 or more threads per SIMD will result in some starvation occurring. The Sobel filter does not appear to be affected by the starvation, which can probably be ascribed to the fact that the Sobel filter's execution time is an order of magnitude smaller than median filter and therefore not as affected by starvation as is the case with the median filter.

A further issue is that GPUs tend to have a number of processors per SIMD engine that corresponds to a power of two value[31]. Ideally the number of threads should be allocated accordingly, however this is not practical for this particular problem as the X-ray test images have dimensions that are not a power of two[32] and therefore cannot be distributed evenly when allocated in powers of two.

A number of conclusions can be drawn from this experiment including the fact that increasing the number of threads increases performance, following a similar trend to that proposed by Amdahl. However, unlike Amdahl's Law, performance can actually be inhibited if the number of threads

[31] This refers to the number of processors per SIMD engine. The number of SIMD engines available on a GPU is typically an even number but not necessarily one which is a power of two.

[32] The test X-ray image's dimensions are 2920 by 2320 pixels.

allocated leads to the starvation of some SIMD engines. The optimum number of threads is proportional to the computational intensity of the program being executed, with those programs which have a high computational intensity benefiting more from additional threads being allocated. The exact number of threads is difficult to predict due to the complexities of the interactions between the graphics processing hardware and the algorithm (Jang et al., 2009, p.5).

5.6. Average Image Error

This is a quantitative comparison of the number and size of the errors in the images produced by the various algorithms.

Table 6 : Number of Errors

Test	Number of Errors	Average Error	Maximum Error[33]
SISD (Int) – MIMD (Int)	0	0	0
SISD (Int) – SIMD (Int)	1,765,303	1	4095
SISD (Int) – SISD (Float)	1,899,198	156	4095
SISD (Float) – MIMD (Float)	0	0	0
SISD (Float) – SIMD (Float)	2,146,427	10	4096

The results in Table 6 suggest that both CPU implementations of the algorithm (SISD and MIMD) produce the same result and hence the algorithm does not display any instability (Mattson and Strandberg, 2008), as a consequence of the parallelisation of floating point calculations. The GPU implementation of the algorithm would appear produce images which contain a significant number of differences compared to those images produced by the various CPU implementations, however the average error between the implementations is less than 0.25% and so could be considered to be acceptable.

The number and size of errors between the floating point and integer results indicate substantial differences between the image produced using floating point data and those generated from the original integer data. The size of errors was unexpected, as in theory the images produced using the algorithm with both data types (int and float) should be almost identical, excluding those inconstancies caused by rounding. The cause of these can be identified by comparing the images produced [Figure 33].

These results should not however be considered in isolation, as they can only indicate that there may potentially be a problem, not the cause. Therefore these results need to be evaluated alongside the images produced so that the problem can be identified and subsequently rectified.

[33] The data value of a pixel in the X-ray can be anywhere in the range of 0 to 4095.

5.7. Image Comparison

The final test is a qualitative test achieved by visually inspecting the images produced by the algorithm. The differences between the images produced by the various implementations of the algorithm have been identified and highlighted by overlaying the images in Adobe Photoshop and performing a difference operation[34] (Adobe, 2010). This removes all of the image data apart from those areas which do not match.

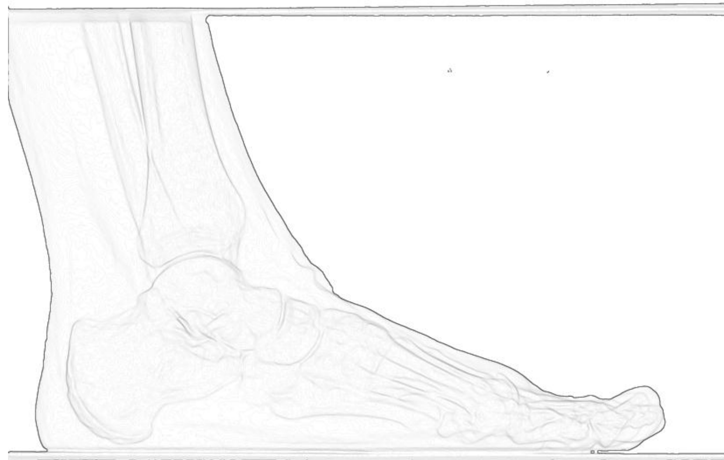

Figure 26 : Algorithm Final Output - Integer (Lateral)

This image is the output of the algorithm after the original lateral X-ray image has been processed using the algorithm. A considerable portion of the unwanted elements have been removed, leaving just the bone and soft tissue.

[34] The difference operation examines the colour information in each channel and subtracts either the blend colour from the base colour or vice versa, depending on which has the greatest brightness value.

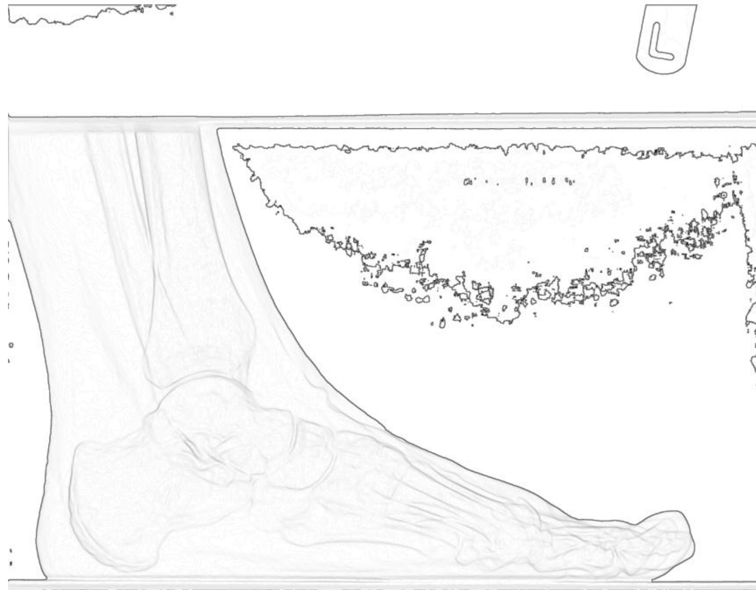

Figure 27 : Algorithm Final Output - Float (Lateral)

This image is the output of the algorithm after converting the lateral X-ray image data to floating point format. The algorithm has retained a significant proportion of the background and the orientation marker remains.

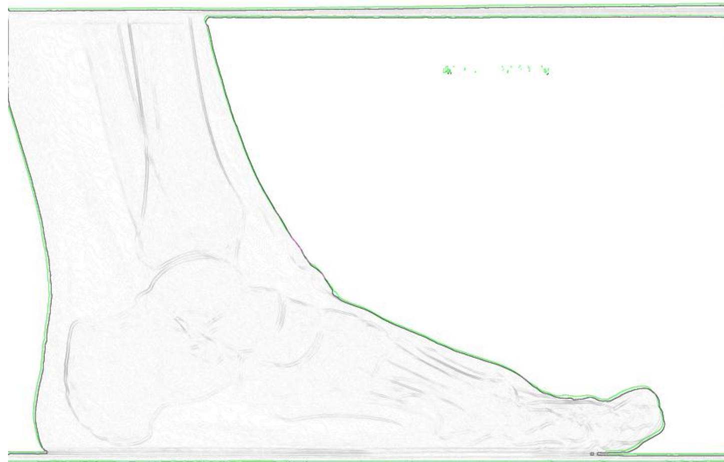

Figure 28 : Image Differences Integer CPU – Integer GPU (Lateral)

This image illustrates the differences between the output of the CPU and GPU algorithms. Whilst it highlights a considerable number of differences, the majority of these can be ascribed to the images produced being slightly misaligned.

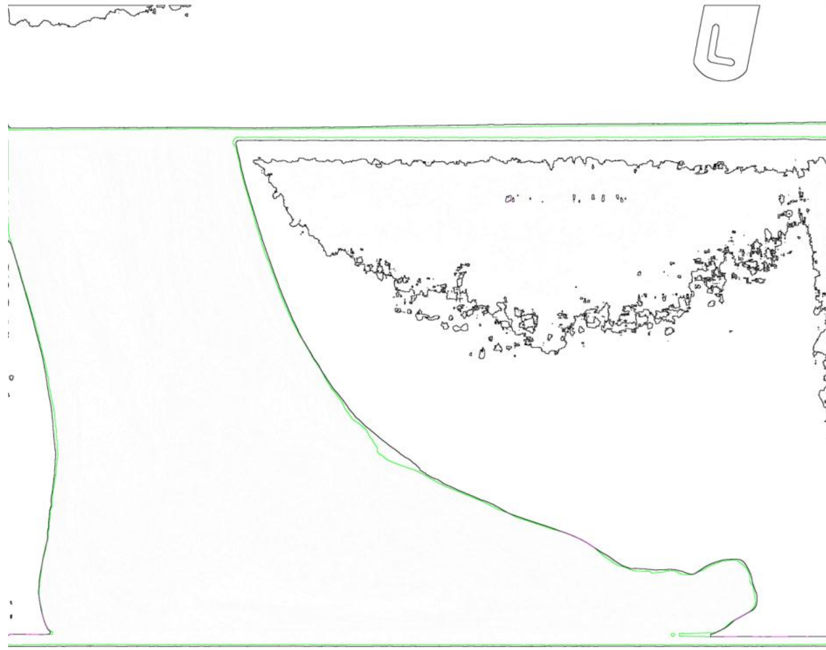

Figure 29 : Image Differences Integer – Floating Point (Lateral)

This image emphasises the differences between the images generated by the CPU algorithm when using different data types. The floating point output retains a significantly larger proportion of the unwanted elements including the orientation marker.

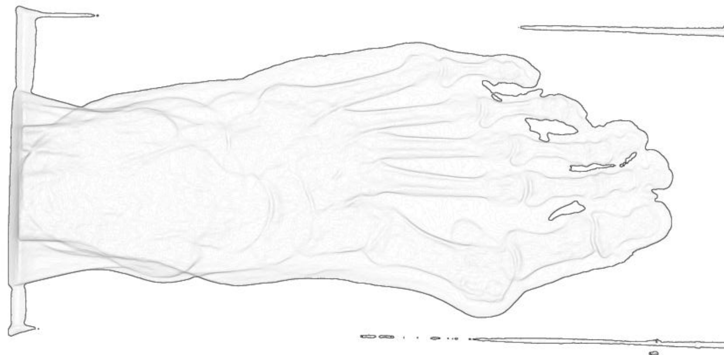

Figure 30 : Algorithm Final Output - Integer (Planar)

This image is the output of the algorithm after the original planar X-ray image has had the algorithm applied. A considerable portion of the unwanted elements including the secondary background have been removed, leaving just the areas of interest.

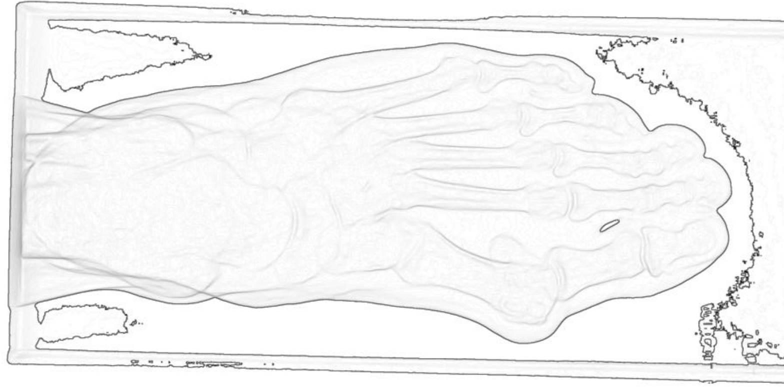

Figure 31 : Algorithm Final Output - Float (Planar)

This image is the output of the algorithm after converting the planar X-ray image data to floating point format. The algorithm has retained a significant proportion of the secondary background and other unwanted elements.

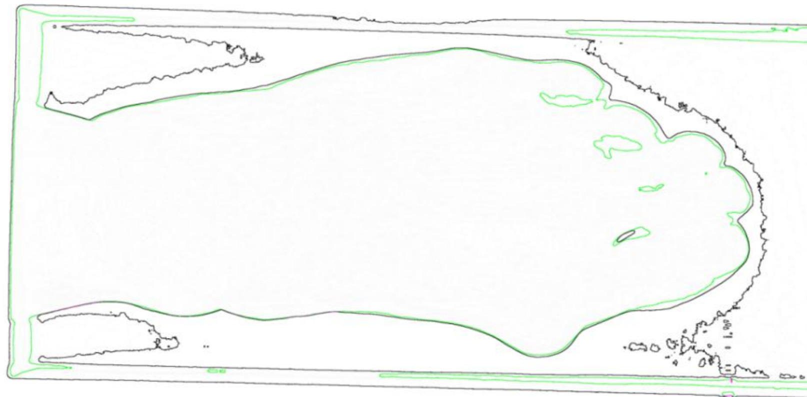

Figure 32 : Image Differences Integer – Floating Point (Planar)

This image highlights the differences between the images produced by the CPU algorithm when operating on different data types. The floating point output retains far more of the unwanted elements.

Figure 33 : Image Differences Integer CPU – Integer GPU (Planar)

This image illustrates the differences between the output of the CPU and GPU algorithms. Whilst a considerable number of differences are highlighted, the majority of these can be attributed to the images produced being slightly out of alignment with each.

A visual inspection of the images produced by the various implementations of the algorithm indicates that the when operating on the same data type, the images generated are broadly similar with only minor discrepancies. Conversely when comparing the images produced by using different data types it is clear that the images have substantial differences. This is particularly evident when ascertaining how many of the unwanted elements have been removed by the algorithm, especially the lateral X-ray image [Figure 27] which retains the orientation marker and a considerable portion of background. These findings correspond to those of the quantitative analysis [Section 5.6]. The results of the Photoshop difference operation further substantiate this claim, with the outputs of the various algorithm implementations containing only relatively minor differences when operating on the same data type [Figure 28 and Figure 33].

The algorithm appears to be capable of removing the majority of the secondary background [Figure 6] contained within the planar X-ray image without unduly affecting the image produced using the lateral X-ray which contains no such background. Therefore it can be assumed that the method proposed for calculating the threshold value [Section 4.4.2] functions correctly.

The Sobel Edge detector appears to perform satisfactorily when highlighting the bone edges and does not need replacing with a more complex and robust method of feature extraction. The bones in the output images are defined well enough that the Active Contour Model method of Feature Extraction should be able to correctly identify and extract the bone portions of the image from the soft tissue. A couple of areas (particularly the toes) are less well defined than other areas, which can be attributed to the non-uniform brightness of bone identified earlier [Section 6.1.1.1]. This could be rectified incorporating functionality which will allow the algorithm to adapt to non-uniform brightness. However, whilst it is difficult to ascertain from the images [Figure 26, Figure 27, Figure 29 and Figure 30] there appears to be enough contrast in those regions which are less well defined for ACM to still perform feature extraction.

Based on the output images [Figure 26 and Figure 27], it can be assumed that the orientation does not adversely affect the images generated by the algorithm. It would also give the impression that it is not necessary to implement any additional functionality to account for orientation in the algorithm. This is quite likely to occur on the condition that a similar proportion of an image is occupied with bone and soft tissue, otherwise the peak detection will need to be adjusted

accordingly. However this cannot be stated categorically, with only limited testing on two orientations being conducted. Ideally extensive testing would be performed using X-rays taken at additional orientations.

The images produced by the SIMD algorithm reveal that the GPU retains a slightly larger amount of the unwanted data compared with the output of the CPU implementations [Figure 28 and Figure 33]. These minor inconsistencies can be attributed to the fact the GPU median filter produces a marginally different image to the CPU median filter. This primarily manifests itself in the output image being marginally out of alignment with the original image. Additionally it should be considered that the GPU and CPU operate with differing levels of precision for floating point operations, specifically in the case of the square root operation in the Sobel filter [Code Listing 12, Line 10].

The most significant problem identified by visual image inspection is the substantial discrepancies [Figure 29] between the images produced using the floating point data and those generated with the native integer data. This indicates that the floating point algorithms contain some errors which are affecting the images produced. A floating point version of the algorithm was developed to determine if utilising data in floating point format would yield enhanced performance over integer data on the GPU. Based on the results [Section 5.1 and Section 5.3] of the data format comparison it would appear to produce only a relatively modest improvement, particularly when considering the additional overhead incurred converting data into floating point format. These results suggest that development of the floating point version of the algorithm should be discontinued. However it cannot be discounted that some of the performance disparity between data types may be due to the floating point algorithms retaining considerably more unwanted data and therefore requiring additional processing, hence the longer execution times.

6. Conclusions

6.1. Overview

The project has two primary objectives, firstly that the algorithm should be capable of isolating bone contained within an X-ray image and secondly that it should be accelerated using parallel processing technologies. Both of these goals have been realised to varying extents. Referring to the individual aims [Section 1.4] established at the instigation of this project it can be concluded that the majority of these have been achieved, predominantly in regards to the algorithm's implementation. The project cannot however be considered a complete success due to the omission of the Active Contour Model feature extraction functionality, which is a major component of the algorithm tasked with isolating the bone portions of the image from soft tissue.

6.1.1. Individual Project Aims

6.1.1.1. Algorithm Objectives

1. **Separate Soft Tissue and Bone**: The algorithm is capable of identifying and isolating those areas of the image which contain either soft tissue or bone. However it currently cannot separate them from each other. This functionality would have been provided by the Active Contour Model feature extraction technique which was not developed.
2. **Remove Unwanted Features**: The use of a large-radius median filter has proven successful at removing the unwanted elements [Figure 4 and Figure 6] contained within the X-ray image in addition to diminishing the amount of noise present.
3. **Noise Reduction:** The median filter appears to have successfully reduced the noise present in the image. The final stage in the image processing algorithm prevents the median filter from affecting the fine details in the bone portions of the image.
4. **Planar and Lateral X-Rays**: The algorithm would appear to operate correctly for X-rays taken at both planar and lateral orientations. In spite of this, there is no specific functionality to counteract the effects that intermediate angles may have on an X-ray image and therefore as a result may not be processed correctly.
5. **Non-Uniform Brightness**: There is no specific functionality to counter non-uniform brightness but the images produced by the algorithm do not appear to have been negatively affected.

6.1.1.2. Implementation Objectives

1. **DICOM X-Rays**: The algorithm is capable of reading the X-ray image data directly from a DICOM file. Ideally, it should utilise any relevant information contained within the DICOM file's header to improve the images produced.

2. **Fully Automated**: In its current form the algorithm is totally automated, requiring only the directory where the DICOM file is located to process the X-ray image.
3. **Consumer Hardware**: The SISD and MIMD algorithms can be executed on any consumer PC employing Microsoft Windows XP or later. Hardware restrictions limit the SIMD algorithm to relatively recent systems featuring a DirectX 11 complaint GPU.
4. **Single Instruction Single Data Implementation**: A SISD version of the algorithm was implemented and used to provide baseline performance timings.
5. **Multiple Instruction Multiple Data Algorithm**: The MIMD algorithm can be considered efficient and scalable, with 96% efficiency and a maximum potential speedup of 25x.
6. **Single Instruction Multiple Data Algorithm**: A GPU accelerated version of the algorithm was implemented, however it was not practical to employ the GPU for all the processing required. The SIMD algorithms offers significantly reduced execution times compared to both CPU implementations.
7. **Native Resolution**: All versions of the algorithm operate on the image at its native resolution and the data formats used for processing ensure that accuracy is not lost during conversion.

6.2. Parallel Processing

The implementation of all three versions of the algorithm has allowed a comprehensive analysis of their relative performance to be conducted, which has shown that the SIMD and MIMD parallel algorithms offer enhanced performance over the SISD implementation. Whilst the SIMD algorithm offers improved performance over both of its CPU counterparts, it should be noted that a number of methods exist that would considerably reduce the execution time of the CPU algorithms and consequently diminish the performance advantage that GPGPU implementation enjoys.

6.2.1. Multi-core CPU

The MIMD implementation of the algorithm which uses a multi-core CPU can be considered efficient, with an average speedup of 3.84x equating to 96% of the algorithm being executed in parallel and a maximum potential speedup of 25x[35]. This would indicate that the MIMD algorithm is scalable well in excess of current and planned consumer CPUs. It may be advisable to sacrifice some of this scalability in order to enhance performance, incorporating some of the techniques implemented in the Constant Time Median Filter (Perreault and Hebert, 2007). This would reduce the execution time and reduce the disparity between the processing times achieved by the CPU and those attained by the GPU. However this should not result in an increase in the performance gap between the MIMD and SISD implementation because these improvements would be applied to

[35] Assuming sufficient processing cores are available to achieve this.

both. When considering reducing scalability it must be acknowledged that consumers CPUs have an order of magnitude less processing cores than available in GPUs and so would be less likely to be affected.

In the author's opinion OpenMP yields excellent returns when processing data parallel problems, considering that CPUs have significantly fewer processing cores. The OpenMP API is comparatively straightforward to implement and has a mature development environment, with numerous software packages featuring exhaustive debugging and profiling capabilities for parallel processing.

6.2.2. GPGPU

The GPGPU version of the algorithm offers increased performance over both the parallel and serial CPU implementations. In particular, the median filter as the most computationally intensive process used by the algorithm benefited considerably, with a tenfold increase in performance over the serial CPU implementations. This is consistent with the suggested amount of speedup typically achieved by DirectCompute compared to CPU implementations (Boyd, 2010, p.37). Two strategies were proposed to compute a median filter using GPGPU and have proven to be effective, offering much reduced execution times in comparison to the CPU implementations. There are nevertheless a number of restrictions in their use. The GPU histogram method is only suitable for large-radius filters. Furthermore due to its high memory requirements it may not be possible to make optimal use of all available processing cores in a SIMD engine. Whilst the caching algorithm provides improved performance over the fast, small-radius algorithm it cannot be used for large-radius filters due to hardware limitations. Both of the techniques are further constrained by the fact that they require access to the cache memory in a manner that is only achievable with Shader Model 5 devices, reducing where they can be utilised.

The caching algorithm can potentially be adapted to other areas of the algorithm such as the Sobel filter. The improved performance from caching data substantiates the claim that global memory access on the GPU has a higher latency than the cache memory (NVIDIA, 2010c, p.30).

The results would also appear to indicate that the optimum number of threads to allocate is relative not only to the number of processors provided by the specific GPU architecture used for processing, but also the computational intensity of the calculations being performed. Furthermore the results suggest that measures should be taken to ensure that SIMD engines are not starved of data and therefore idle, especially when performing computationally intensive operations. For this problem it can be more significant than it first appears, GPU architectures are typically designed with SIMD engines being allocated processors in groups corresponding to power of two values. Consequently

images with dimensions that are power of two values allow optimal allocation of work, however the test X-ray images have dimensions which are not a power of two value.

The algorithm can be considered scalable based on the recommendation made by Fung (2010, p.6) about maximising scalability in DirectCompute programs. The recommendation is that each dispatch call should issue processing so that it is spread across thousands of thread groups, which in turn allows a GPGPU program to scale across multiple future generations of GPU. However the performance of GPGPU algorithms can be limited by the CPU as it is required to transfer data to and from the GPU and process those areas which cannot be effectively parallelised.

GPGPU processing does however have a number of disadvantages:

- It is an incredibly inefficient method for processing non-parallelisable problems.
- Data may need to be converted to an appropriate format for GPGPU processing, thus incurring additional overheads.
- Attaining optimal performance requires various optimisations to be implemented. This is a particular problem for AMD GPUs as it requires the explicit vectorization of arithmetic operations.

Based on the results and subsequent analysis the following recommendations can be made:

- GPGPU is an excellent method for processing data parallel problems, offering substantial performance improvements.
- For large-radius median filter masks the GPU histogram method should be considered as it offers superior performance to its CPU counterparts.[36]
- The strategy of using the cache memory to reduce memory access times resulting from redundant global memory access appears sound and should be transferred to other applicable areas of GPGPU processing.
- Determining the optimum number of threads to allocate per SIMD engine is complex but may have considerable impact on performance.
- The use of floating point data to enhance the performance of GPU processing is only worthwhile if the overhead incurred converting the data is less than the time gained by processing using floating point rather than integer format. Additionally the performance benefit only occurs for arithmetic instructions, not for memory accesses. Therefore it is only advisable for arithmetically intensive operations.

[36] Provided the target system has an appropriate GPU.

In the author's opinion, whilst DirectCompute is capable of producing substantial speedups compared to CPU based implementations, it requires specialist knowledge to obtain optimum performance. The development environment can be considered immature due its lack of sophisticated software packages for debugging and profiling DirectCompute shaders, compared to those available for parallel processing on the CPU.

6.3. Algorithm

The algorithm in its current form is not capable of performing all the tasks required to isolate the bone portions of the X-ray image. Specifically it cannot separate soft tissue and bone. It is however capable of removing unwanted elements from digital X-ray images in addition to highlighting the edges of soft tissue and bone. Incorporating Active Contour Model feature extraction into the algorithm would rectify this issue. It could also prove useful to investigate the feasibility of using feature extraction in the initial processing pass in place of a large-radius median filter to determine if this would either produce a more accurate image or result in reduced execution times. The application of the algorithm to an X-ray is a fully automated procedure, only requiring the directory containing the DICOM file to be specified to process the image.

The images produced by the CPU and GPU versions of the algorithm do contain some minor differences, but these can be considered irrelevant for our purposes. However there are considerable differences between the images produced using different data types which implies that there are a number of errors in the floating point algorithm which need to be addressed.

6.4. Summary

In summary we can conclude that although the algorithm has some significant functionality missing, it has proved capable of isolating the soft tissue and bone portions of the X-ray image whilst reducing noise and removing the unwanted elements from the image. From the performance timings collected during the experimentation we can also state that GPGPU processing in particular and parallel processing in general can considerably reduce the execution times of algorithms that operate on data parallel problems.

7. Future Work

There are a number of avenues that could potentially be explored as part of further development of the algorithm. The decision to not implement the feature extraction component [Section 4.4.4] requires that any future development of the algorithm should initially be concentrated on rectifying this. Furthermore there are a number of areas in which improvements to the algorithm could be made, especially in regards to improving its functionality.

7.1. Active Contour Model Feature Extraction

The bone detection will be performed using Active Contour Models to process the image. It is hoped that the GPGPU algorithm suggested by Tatarchuk (2008) can be adapted to a DirectCompute implementation and if possible modified to make use of cache memory now available on the GPU. The suggested algorithm samples a region of pixels for each control point; therefore the caching algorithm used for median filtering [Section 4.5.1.2] could potentially be used.

7.2. GPU Median Filter

There are a number of approaches which could be used to improve the performance of the GPU histogram median filter [Section 4.5.1.3]. On the development GPU, approximately 10KB of cache memory remains unused, which is due to the X-ray image's dimensions restricting the number of threads per thread group that can be allocated without starvation of some SIMD engines occurring.

An additional sub-histogram which corresponds to the main histogram could be created to reduce the number of loop iterations used to calculate the median value. For example if an additional four bin histogram was created it would reduce the amount of bins that need searching from 256 possible locations to 64 [Code Listing 18, Line 27].

An alternative technique would be to use the remaining cache memory to store as much of the shared image data in the cache memory as possible, similar to the caching algorithm. However depending on the radius size of the median filter mask, it may not be practical to load all the required image data into cache memory. Therefore steps must be taken to allow the shader to determine if a particular pixel is stored in cache memory or global memory. Obviously, the performance impact of using cache memory is dependent how much of the image data can be loaded into cache memory.

The performance of the GPU cached median filter can also be improved [Section 4.5.1.2]. Currently each group of pixels is sampled as a row. If this was altered to a block then additional redundant memory accesses could be prevented.

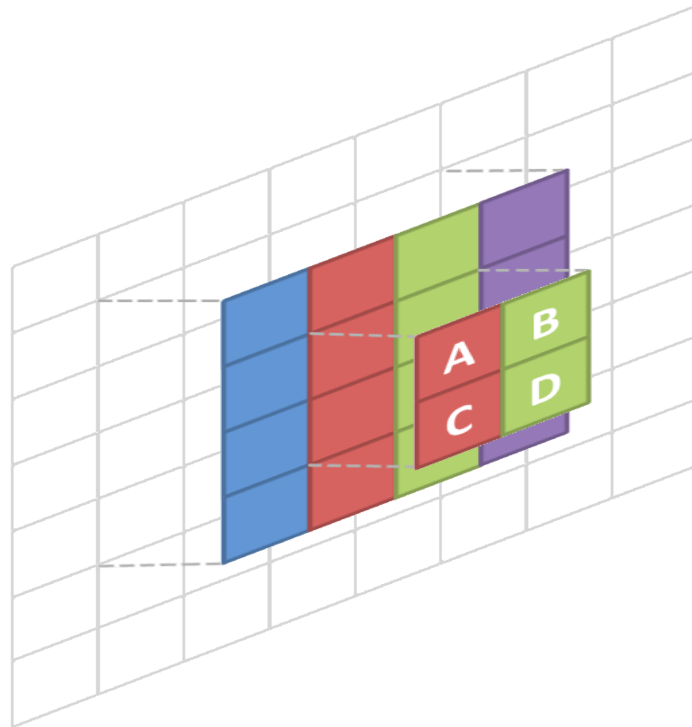

Figure 34 : Improved Caching Median Filter

Allocating thread groups in a block rather than a strip [Figure 18] increases the number of redundant memory access that are transferred from global memory to cache. In this example the small fast algorithm requires 36 accesses to global memory, the original caching algorithm 18 and the improved caching 16. The effect is relative to the block size, increasing for large block sizes.

7.3. X-Ray Header Information

DICOM based X-ray images contain a wealth of additional information about the patient and how the image was captured. This information is stored within the DICOM file's header and could potentially be used to improve the generated mesh. The most significant of the stored information for the algorithm's purposes is the Photometric Interpretation flag. Typically for an X-ray it is either set to MONOCHROME1 or MONOCHROME2. MONOCHROME1 format means that black[37] represents dense materials such as bone whereas MONOCHROME2 is the opposite, with white signifying dense materials. For the algorithm this is a considerable problem as in its current form the threshold function will not operate correctly with MONOCRHOME2 images, removing bone information instead of the image background. To fix this is a trivial process, requiring the relevant tag within the DICOM header to be found and its value ascertained and the appropriate threshold function to be selected.

[37] Low bit values

7.4. Integration

The X-ray processing algorithm and the mesh generation code need to be transferred from their current standalone implementations into a unified implementation. This would allow them to be fully integrated into VirtuOrtho which will produce a more streamlined process for mesh generation than is currently implemented, benefiting the end user.

7.5. Generating Skin Mesh

Although the simulation is focused primarily on simulating the cutting of bone, it may in fact be desirable to generate an additional mesh representing the soft tissue. This would serve to further increase the realism of the virtual operation within VirtuOrtho. The mesh generation process could be adapted so that rather than discarding any soft tissue information found within the X-ray image, it could instead be used to generate an outer mesh depicting the layers of skin and soft tissue surrounding the bone. This process would be broadly similar to how the bone mesh is currently being generated. The cutting algorithm would however require alterations to reflect the increased elasticity of skin and soft tissue compared to bone, which is currently being modelled.

7.6. Parallel Mesh Generation Pipeline.

Currently the mesh generation pipeline implemented for VirtuOrtho is a serial process, with the X-ray image processing algorithm being the only part of the process to take advantage of parallel processors. Clearly this is a series performance bottleneck and should ideally be rectified by re-designing the code so that is capable of taking advantage of parallel processors. The knowledge about parallel algorithm design gained during the development of the image processing algorithm should be used to improve the performance of the mesh generator.

DirectCompute would appear to be a suitable candidate for providing parallel processing capabilities for the mesh generator. This is due to the process being arithmetically intensive with comparatively minimal memory bandwidth requirements, which compliment a GPU's parallel processing architecture. DirectCompute is also capable of operating on data in the same format as used by the DirectX graphics API, thus reducing the amount of data format conversions required.

7.7. Benchmarking

Achieving optimal performance has been a significant issue throughout the development of the DirectCompute implementation of the image processing algorithm. This has been further compounded by the fact that minor alterations to the number of threads, thread groups and other settings can significantly impair performance depending on the architecture of the GPU the algorithm is being processed on. GPU architectures additionally can potentially be substantially

different, not only between different manufacturers but also between different generations from the same manufacturer!

Currently the DirectCompute API does not provide facilities to apply the optimum settings automatically during the compilation of the compute shader. The simplest solution to this would be to include GPU specific settings, however determining the appropriate values manually would be a time consuming and laborious process. A potential solution (Fung, 2010, p.5) would be to adopt a similar process to that used by Windows Vista/7 and some PC games whereby a benchmark is run during the software's installation to determine the settings which achieve optimal performance.

7.8. Alternative GPGPU Implementations

In order to perform a complete analysis of the GPGPU algorithm's effectiveness it should be implemented in either CUDA or OpenCL in addition to its current DirectCompute form. Doing this would also allow a critical review of the different GPGPU APIs to be conducted and their relative benefits and drawbacks to be established. A traditional pixel shader approach can be ruled out because of the algorithm's requirement for a large-radius median filter, which without the cache memory made available by DirectCompute and the other GPGPU APIs is impractical to implement.

8. References

Adobe (2010) **Photoshop Elements Help** [Internet], Available from: <http://help.adobe.com> [Accessed 11 Nov 2010].

Akhter, S. and Roberts, J. (2006) **Multi-Core Programming.** Intel Press.

Amdahl, G. (1967) Capabilities, Validity of the Single Processor Approach to Achieving Large-Scale Computing. In: **AFIPS Conference Proceedings**., p.483–485.

Ansorge, R. (2008) **AIRWC : Accelerated Image Registration With CUDA.** Cambridge, University of Cambridge.

Banik, S., Rangayyan, R. M. and Boag, G. S. (2008) Landmarking and Segmentation of 3DCT Images. **SYNTHESIS LECTURES ON BIOMEDICAL ENGINEERING #30**.

Bilodeau, B. (2009) Your Game Needs Direct3D 11, So Get Started Now! In: **Game Developer Conference**. San Francisco.

Bit-tech.net (2009) **ATI Radeon HD5870 Architecture Analysis** [Internet], Available from: <http://www.bit-tech.net/hardware/graphics/2009/09/30/ati-radeon-hd-5870-architecture-analysis/1> [Accessed 11 Aug 2010].

Bit-tech.net (2010) **GeForce GTX 580** [Internet], Available from: <http://www.bit-tech.net/hardware/graphics/2010/11/09/nvidia-geforce-gtx-580-review/2> [Accessed 15 Nov 2010].

Bleiweiss, A. (2008) GPU Accelerated Pathfinding. In: **Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware**. Aire-la-Ville, Eurographics Association, pp.65-74.

Bordawekar, R., Bondhugula, U. and Rao, R. (2010a) Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application! In: **Proceedings of the 19th international conference on Parallel architectures and compilation techniques**. Vienna, ACM, pp.537-538.

Bordawekar, R., Bondhugula, U. and Rao, R. (2010b) **Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU.** IBM.

Bovik, A. C., Huang, T. S. and Munson, D. C. (1987) The Effect of Median Filtering on Edge Estimation and Detection. **IEEE Transactions on Pattern Analysis and Machine Intelligenc**, 9 (2), pp.181-194.

Boyd, C. (2010) **DirectCompute Lecture Series 101: Introduction to DirectCompute**. [Internet], Available from: <http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-101-Introduction-to-DirectCompute> [Accessed 10 Oct 2010].

Boyd, C. (2008) DirectX 11 Compute Shader. In: **SIGGRAPH**.

Brookwood, N. (2010) **AMD Fusion™ Family of APUs: Enabling a Superior, Immersive PC Experience.** Insight64.

Cardoso, A. D.S. (2010) **Generation of planar radiographs from 3D anatomical models using the GPU.** University of Porto.

Castaño-Díeza, D., Mosera, D., Schoeneggera, A., Pruggnallera, S. and Frangakis, A. S. (2008) Performance evaluation of image processing algorithms on the GPU. **Journal of Structural Biology**, 164 (1), pp.153-160.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J. and Menon, R. (2000) **Parallel Programming in OpenMP.** Morgan Kaufmann.

Chan, R. H., Ho, C.W. and Nikolova, M. (2005) Salt-and-Pepper Noise Removal by Median-type Noise Detectors and Edge-preserving Regularization. **IEEE Transactions on Image Processing**, 14 , pp.1479-1485.

Chen, J., Paris, S. and Durand, F. (2007) Real-Time Edge-Aware Image Processing with the Bilateral Grid. **ACM Trans. Graph**, 26 (3).

Christadler, I. (2010) **Performance and Productivity of new Programming Languages "PRACE OpenDialog with European Tier-0 Users".** Hamburg, PRACE.

Chu, M. M. (2010) **GPU Computing: Past, Present and Futurewith ATI Stream Technology.**

Cootes, T. F., Edwards, G. J. and Taylor, C. (1999) Comparing Active Shape Models with Active Appearance Models. In: T. Pridmore, D. E. ed. **British Machine Vision Conference**., pp.173-182.

Cootes, T. F., Taylor, C. J., Cooper, D. H. and Graham, J. (1995) Active Shape Models-Their Training and Application. **Computer Vision and Image Understanding**, 61 (1), pp.38-59.

Danielsson, P. E. (1984) Vices and Virtues of image parallel machines. In: Levialdi, S. ed. **Digital Image Analysis**.

Davidson, A. (2006) Using a Graphics Processor Unit (GPU) for Feature Extraction from Turbulent Flow datasets. In: **The National Conference On Undergraduate Research**. Asheville.

Davis, J. E., Ozsoy, A., Patel, S. and Taufer, M. (2009) **Towards Large-Scale Molecular Dynamics Simulations on Graphics Processors.** Newark, University of Delaware.

Dawson, B. (2010) **Coding For Multiple Cores on Xbox 360 and Microsoft Windows.** Advanced Technology Group.

Evans, K., Sych, T. and Dunsavage, K. (2010) **Improving Medical Imaging Performance on the Intel® Xeon® Processer 5500 series.** Intel Corporation.

FAKULTI KEJURUTERAAN ELEKTRIK. (n.d.) **X-Ray Image Processing.** UNIVERSITI TEKNOLOGI MALAYSIA.

Flynn, M. J. (1966) Very High-Speed Computing Systems. **Proceeding of IEEE**, 54 (12), pp.1901-1909.

Foster, I. (1995) **Designing and Building Parallel Programs.** Addison-Wesley Publishing Company.

Fried, M. (2010) **GPGPU Architecture Comparision of AMD and Nvidia GPUs.** Microway.

Fung, J. (2010) **DirectCompute Lecture Series 210: DirectCompute GPU Optimizations and Performance**. [Internet], Available from: <http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-210-GPU-Optimizations-and-Performance> [Accessed 10 Oct 2010].

Gelsinger, P. P. (2008) **Intel Architecture Press Briefing**. [Internet], Available from: <http://download.intel.com/pressroom/archive/reference/Gelsinger_briefing_0308.pdf> [Accessed 15 Sep 2010].

Grama, A. and Kumar, V. (2008) Scalability of Parallel Programs. In: Sanguthevar Rajasekaran, J. R. ed. **Handbook of Parallel Computing: Models, Algorithms and Applications**. Chapman & Hall, pp.43-45.

Green, S. (2009) DirectX 10/11Visual Effects. In: **GDC**.

Gustafson, J. L. (1988) Reevaluating Amdahl's Law. **Communications of the ACM**, 31, pp.532-533.

Howes, L. (2010) **DirectCompute Lecture Series 230: GPU Accelerated Physics**. [Internet], Available from: <http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-230-GPU-Accelerated-Physics> [Accessed 10 Oct 2010].

Huang, T., Yang, G. and Tang, G. (1979) A fast two-dimensional median filtering algorithm. **IEEE Transactions on Acoustics, Speech and Signal Processing**, 27 (1), pp.13-18.

Isensee, P. (2006) Utilizing Multicore Processors with OpenMP. In: Dickheiser, M. ed. **Game Programming Gems 6**. Charles River Media.

Jähne, B. (2005) **Digital Image Processing: Concepts, Algorithms and Scientific Applications.** Springer.

JaJa, J., Varshney, A. and Shi, Q. (2008) Parallel Algorithms for Volumetric Surface Construction. In: Rajasekaran, S. and Reif, J. eds. **Handbook of Parallel Computing: Models, Algorithms and Applications**. Chapman & Hall.

Jang, B., Do, S., Pien, H. and Kaeli, D. (2009) Architecture-Aware Optimization Targeting Multithreaded Stream Computing. In: **Second Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2009)**. Washington DC, ACM.

Jeong, W.K., Fletcher, P. T., Tao, R. and Whitaker, R. T. (2007) Interactive Visualization of Volumetric White Matter Connectivity in DT-MRI Using a Parallel-Hardware Hamilton-Jacobi Solver. In: **IEEE Conference on Visualization**.

Kachelriess, M. (2009) Branchless vectorized median filtering. **Nuclear Science Symposium Conference Record (NSS/MIC)**, pp.4099 - 4105.

Kass, M., Witkin, A. P. and Terzopoulos, D. (1988) Snakes: Active contour models. **International Journal of Computer Vision**, 4 (1), pp.321-331.

Kazhdan, M. and Hoppe, H. (2008) Streaming Multigrid for Gradient-Domain Operations on Large Images. **ACM Transactions on Graphics**, 27 (3).

Khronos Group (2010) **OpenCL** [Internet], Available from: <http://www.khronos.org/opencl/> [Accessed 05 Jul 2010].

Kindelan, M. and Lezo, J. S.D. (1984) Atery Dectection and Tracking in Coronary Angiography. In: Levialdi, S. ed. **Digital Image Analysis**., pp.283-294.

Langs, A. and Biedermann, M. (2007) Filtering Video Volumes Using the Graphics Hardware. In: Pedersen, B. E.A.K. ed. **SCIA 2007**. Springer-Verlag, p.878–887.

Larsson, P. and Palmer, E. (2009) **Image Processing Acceleration Techniques using Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions.**

Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R. and Dubey, P. (2010) Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In: **Proceedings of the 37th annual international symposium on Computer architecture**. New York, ACM, pp.451-460.

Lindberg, P. (2009) **Basic OpenMP Threading Overhead.** Intel Corporation.

Lönroth, M. U. (2009) **Advanced Real-time Post-Processing using GPGPU techniques.** DICE.

Magro, W., Petersen, P. and Shah, S. (2002) Hyper-Threading Technology. **Intel Technology Journal**, 4 (1).

Mattson, T. and Strandberg, K. (2008) **Parallelization and Floating Point Numbers.** Intel Corporation.

McGuire, M. (2008) A Fast, Small-Radius GPU Median Filter. In: Engel, W. ed. **ShaderX6**. Charles River Media, pp.165-173.

McInerney, T. and Terzopolous, D. (1996) Deformable Models in Medical Image Analysis, a Survey. **Medical Image Analysis**, 1 (2), pp.91-108.

Mitchell, J. L., Ansari, M. Y. and Hart, E. (2003) Advanced Image Processing with DirectX 9 Pixel Shaders. In: Engel, W. ed. **ShaderX2: Shader Programming Tips and Tricks with DirectX 9**. Wordware, pp.439-464.

Moore, G. E. (1965) Cramming more components onto integrated circuits. **Electronics**, 38 (8).

MSDN (2010a) **Common-Shader Core (DirectX HLSL)** [Internet], Available from: <http://msdn.microsoft.com/en-us/library/bb509580(v=VS.85).aspx> [Accessed 10 Oct 2010].

MSDN (2010b) **Compute Shader Overview** [Internet], Available from: <http://msdn.microsoft.com/en-us/library/ff476331(v=VS.85).aspx> [Accessed 15 Jul 2010].

MSDN (2010c) **for Statement (DirectX HLSL)** [Internet], Available from: <http://msdn.microsoft.com/en-us/library/bb509602(v=VS.85).aspx> [Accessed 09 13 2010].

MSDN (2010d) **numthreads** [Internet], Available from: <http://msdn.microsoft.com/en-us/library/ff471442(VS.85).aspx> [Accessed 03 May 2010].

Myler, H. R. and Weeks, A. R. (1993) **The Pocket Handbook of Image Processing Algorithms in C.** Prentice Hall.

NVIDIA (2010a) **CUDA** [Internet], Available from: <http://www.nvidia.co.uk/object/what_is_cuda_new_uk.html> [Accessed 03 Jun 2010].

NVIDIA (2010b) **CUDA Supporting GPUs** [Internet], Available from: <http://www.nvidia.co.uk/object/cuda_gpus_uk.html> [Accessed May 2010].

NVIDIA. (2010c) **OpenCL Programming Guide for the CUDA Architecture**. [Internet], Available from: <http://developer.nvidia.com/object/opencl.html> [Accessed 06 Dec 2010].

NVIDIA (2010d) **OpenCL SDK** [Internet], Available from: <http://developer.download.nvidia.com/compute/opencl/sdk/website/samples.html> [Accessed 08 Dec 2010].

NVIDIA. (2009) **Optimization : NVIDIA OpenCL Best Practices Guide**. [Internet], Available from: <http://developer.nvidia.com/object/opencl.html> [Accessed 09 Jun 2010].

NVIDIA. (1999) **Transform and Lighting**. [Internet], Available from: <http://www.nvidia.com/object/transform_lighting.html> [Accessed 04 Jul 2010].

Paeth, A. W. (1990) Median finding on a 3x3 Grid. In: Glassner, A. ed. **Graphic Gems**. Boston, Academic Press, pp.171-175.

Park, M., Jin, J. S. and Wilsoni, L. S. (2004) Detection of Abnormal Texture in Chest X-rays with Reduction of Ribs. In: Piccardi, M., Hintz, T., He, X., Huang, M. L., Feng, D. F. and Jin, J. eds. **Information Processing VIP2003**. Sydney, Australian Computer Society.

Perreault, S. and Hebert, P. (2007) Median Filtering in Constant Time. **IEEE Transactions on Image Processing**, 16 (9), pp. 2389-2394.

Petrou, M. and Bosogianni, P. (1999) **Image Processing Fundamentals.** WILEY.

Pfister, H. (n.d.) **General Purpose Computing using Graphics Hardware.** Harvard University.

Podlozhnyuk, V. (2009) **Histogram calculation in OpenCL.** nVidia.

Preis, T., Virnau, P., Paul, W. and Schneider, J. J. (2009) Accelerated fluctuation analysis by graphic cards and complex pattern formation in financial markets. **New Journal of Physics**, 11 (9).

Roberts, M., Sousa, M. C. and Mitchell, J. R. (2010) A Work-Efficient GPU Algorithm for Level Set Segmentation. In: **Annual Conference on Computer Graphics - SIGGRAPH**. Los Angeles, ACM.

Roosta, S. H. (1999) **Parallel Processing and Parallel Algorithms – Theory and Computation.** Springer.

Saccone, R. (2007) Improve Scalability With New Thread Pool APIs. **MSDN Magazine**, 08 Oct.

Sander, P. (2005) DirectX9 High Level Shading Language. In: **SIGGRAPH**.

Sandy, M. (2010) **DirectCompute Lecture Series 110: Memory Patterns**. [Internet], Available from: <http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-110-Memory-Patterns> [Accessed 10 Oct 2010].

Schellmann, M., Vörding, J., Gorlatch, S. and Meiländer, D. (2008) Cost-Effective Medical Image Reconstruction: From Clusters to Graphics Processing Units. In: **Conference On Computing Frontiers**. Ischia, ACM, pp.283-292.

Scheuermann, T. and Hensley, J. (2007) Efficient Histogram Generation Using Scattering on GPUs. In: **I3D 2007**. Association for Computing Machinery, Inc.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T. and Abrash, M. (2008) **Larrabee: A Many-Core x86 Architecture for Visual Computing.** Intel Corporation.

SensAble Technologies (2010) **PHANTOM Omni® Haptic Device** [Internet], Available from: <http://www.sensable.com/haptic-phantom-omni.htm> [Accessed 19 May 2010].

Shamir, L., Ling, S., Scott, W., Bos, A., Orlov, N., Macura, T., Eckley, D. and Goldberg, I. (2008) Knee X-ray image analysis method for automated detection of Osteoarthritis. **IEEE Transactions on Biomedical Engineering**.

Shams, R. and Kennedy, R. A. (2007) Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices. In: **International Conference on Signal Processing and Telecommunication Systems**. Canberra.

Sizintsev, M., Derpanis, K. G. and Hogue, A. (2008) Histogram-based search: A comparative study. In: **Computer Vision and Pattern Recognition**.

Strzodka, R., Droske, M. and Rumpf, M. (2004) Image Registration by a Regularized Gradient Flow. A Streaming Implementation in DX9 Graphics Hardware. **Computing 73**, 73, p.373–389.

Tatarchuk, N. (2008) GPU-Based Active Contours for Real-Time Object Tracking. In: Engel, W. ed. **ShaderX6: Advanced Rendering Techniques**. Charles River Media, pp.145-160.

Thibieroz, N. (2009) Shader Model 5.0 and Compute Shader. In: **Game Developer Conference**. San Francisco.

Thibieroz, N. and Cebenoyan, C. (2009) DirectCompute Performance on DX11 Hardware. In: **GDC09**.

Tibshirani, R. J. (2008) **Fast Computation of the Median by Successive Binning.** Stanford, Stanford University.

Valve (2010) **Hardware Survey** [Internet], Available from: <http://store.steampowered.com/hwsurvey> [Accessed 08 Nov 2010].

Voila, I., Kanistar, A. and Groller, M. E. (2003) Hardware-Based Nonlinear Filtering and Segmenation using High Level Shading Languages. In: **14th IEEE Visualization Conference**. Seattle, pp.309-316.

Walbourn, C. (2005) **Game Timing and Multicore Processors.** Advanced Technology Group.

Wang, C. W. (2006) Real Time Sobel Square Edge Detector for Night Vision Analysis. In: Campilho, A. and Kamel, M. eds. **ICIAR 2006**. Springer-Verlag Berlin Heidelberg, p.404 – 413.

Wei, J., Hagihara, Y., Shimizu, A. and Kobatake, H. (2002) **Optimal image feature set for detecting lung nodules on chest.** Tokyo, Tokyo University of Agriculture and Technology.

Yang, J. (2010) **Basics of DirectCompute Application Development**. [Internet], Available from: <http://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-120-Basics-of-DirectCompute-Application-Development> [Accessed 10 Oct 2010].

Yin, Y. and Tian, G. Y. (2008) Feature Extraction and Optimisation for X-ray Weld Image Classification. In: **17th World Conference on Nondestructive Testing**.

Zink, J. (2010) Ambient Occlusion Using DirectX Compute Shader. In: Lake, A. ed. **Game Programming Gems 8**. Course Technology PTR, pp.50-73.

9. Appendices

9.1. Appendix A: Hardware Statistics

This appendix contains information regarding the hardware configurations of PC gamers that use the popular Steam software. The data is obtained automatically as part of the Steam Hardware Survey (Valve, 2010). Whilst this data does not relate directly to the target audience of the VirtuOrtho, it does give a general indication of the current trends that consumer hardware will follow. This can be attributed to the fact that PC gamers tend to be early adopters of computer technology and PC gamers are currently the main driving force behind GPU development.

Table 7 : Steam Hardware Survey, Number of CPU cores

Number of CPU cores	Steam Users (%)
1	11.22
2	53.81
3	1.20
4	33.13
6	0.57
8	0.06

The survey of CPUs shows that single core CPUs are in the minority with dual and quad core processors currently being the most popular derivatives of multi-core processors.

Table 8 : Steam Hardware Survey, Advanced CPU Feature Support

Advanced CPU Features	Steam Users (%)
SSE2	99.13
SSE3	97.29
SSE4.1	39.81
HyperThreading	16.05
SSE4.2	14.65
SSE4a	12.46

The survey of advanced CPU features[38] demonstrates that SSE2 and SSE3 instructions are almost universally supported in current CPUs, whereas HyperThreading and later versions of SSE instructions have far more limited support.

[38] i.e. Features which are not part of the x86 or x64 instruction sets.

Table 9 : Steam Hardware Survey, DirectX 11 Graphics Cards

DirectX 11 Graphics Card	Steam Users (%)
AMD Radeon HD 5450	0.54
AMD Radeon HD 5500 Series	0.30
AMD Radeon HD 5600 Series	0.70
AMD Radeon HD 5700 Series	7.30
AMD Radeon HD 5800 Series	7.60
AMD Radeon HD 5900 Series	0.64
NVIDIA GeForce GTX 460	1.74
NVIDIA GeForce GTX 465	0.28
NVIDIA GeForce GTX 470	1.24
NVIDIA GeForce GTX 480	1.08
Total DirectX 11 GPUs	**21.42%**

The survey of Steam users with DirectX 11 capable graphics cards indicates that they have been widely adopted, with over 20% of PCs featuring a DirectX 11 graphics card. They appear to have achieved good market penetration between their initial retail launch (September 2009) and the time the survey was conducted (November 2010).

9.2. Appendix B: Test Hardware

This appendix details the configuration of both test hardware and the C++ compiler of Visual Studio 2010.

Table 10 : Test Hardware Configuration

Hardware	Model	Specifications	Notes
CPU	Intel Core i5 750	Quad Core @ 2.66GHz	No HyperThreading
RAM	DDR3	4GB @ 1333MHz	
GPU	AMD HD 5770	800 Stream Processors, GDDR5 1024MB	Catalyst 10.8 Drivers
OS	Windows 7	x64 Version	

Note: The Core i5 750 is a derivative of the Core i7 processor. It however lacks the HyperThreaded cores of an i7 processor; therefore it has four physical processing cores and no logical cores. It also has a Dual Channel memory controller instead of the Triple Channel controller found in Core i7 CPUs.

Table 11 : Visual Studio 2010 C++ Compiler Settings

Compiler Setting	Value
Target Platform	x64
Optimisation	Maximize Speed (/O2)
Whole Program Optimisation	Yes (/GL)
Enhanced Instruction Set	Streaming SIMD Extensions 2 (/arch:SSE2) (/arch:SSE2)
Floating Point Model	Precise (/fp:precise)
Open MP Support	Yes (/openmp)

9.3. Appendix C: YEF Proposal

This appendix contains the initial project proposal for VirtuOrtho.

Table 12 : YEF Funding Application Form

Project Title Serious Games for Medical Training A novel approach to pre-med virtual environments / operations
Layman's Summary When planning surgical procedures medical professionals desire as much information as possible. In the lead application – podiatric surgery – current practice is to take X-ray images of the foot from the top and side and then use these to plan the surgical procedure. Current research aims to produce a computer software system (serious game) capable of taking these single plane X-ray images of a patient's foot and construct a virtual 3D representation of it. A functional application of this system will then be developed, suitable of everyday use by medical professionals, allowing different orthopaedic surgical techniques to be practiced and experimental procedures to be assessed. In addition to pre-operative planning, the application could be used to: Instruct students with human physiology and anatomy using realistic 3D representations. Familiarise trainee and pre-reg medical professionals in generic surgical procedures Aid the explanation of complex surgical techniques to patients
Purpose of the project and benefits The purpose of the software is to develop a 3D representation and manipulation system using an existing mass-produced, inexpensive hardware platform. This will facilitate its broad uptake in teaching (of both students and pre-registration healthcare professionals), and pre-operative planning and simulation. The need for improved systems for teaching and learning is well established, especially for the development of knowledge and familiarity of human anatomy. Such knowledge has traditionally been gained from planar pictures, models and ultimately dissection. Potentially the proposed system will allow the 3D representation, manipulation and "digital dissection" of human using an

inexpensive hardware platform, which makes it suitable for large scale use within the academic and vocational training of all healthcare professionals.

Surgical pre-planning requires imaging of the patient anatomy; this subsequently viewed in two dimensions which quite naturally limits accurate visualisation. This limitation, combined with the increasing move to minimally invasive laparoscopic surgery requires new approaches to visualisation and pre-operative planning. The proposed development will allow the 3D representation, planning and "digital practice" of surgery.

The product will initially be focused on Podiatry surgical trainees. With the ever growing need for better, cheaper and faster medical and surgical training and assistance systems, software that delivers these criteria will have real value.

Intellectual Property (IP)

The software will generate IP for the training software.

IP will be generated from the techniques implemented from the 2D X-Ray to 3D model / mesh algorithms.

Describe the novel aspects of your proposed project/venture

The use of 3D representations in the teaching and development of a range of students (from undergraduate to post-reg medical professionals) is novel. There are currently available simulation and representation products; however these are all single plane representations so the anatomical architecture is poorly represented.

This project will deliver a low cost system suitable for applications ranging from "mass teaching" through to high-end surgical procedure planning.

The novelty and advantages of using a currently available platform for the delivery of the application cannot be overestimated.

We are particularly keen to link this project to the regional academic, industry and NHS strength in Healthcare technologies.

Target markets

As this project is a collaboration between ourselves and the School of Podiatry, the initial

development will be aimed at the podiatric surgery market. We realise this represents only a small fraction of the potential market, however it does represent a significant initial market. There are currently 17 Podiatric degree programmes throughout the UK, which represents a market of over 750 units. In the following phases of development the largest opportunity lies within “mass teaching” of undergraduate medical, dental and allied healthcare professionals the target market is c. 25,000 units.

Sales values are of course difficult to predict, however a unit value of £2,500 for the mass teaching model is estimated (based on a 50% discount vs. Anatomical models). Using this estimate of unit price, the value of the market is over £60M within the UK alone.

Our exploitation route would be via collaborative development and licensing with appropriate partners; we have confirmed interest from educational software providers for the “mass teaching” model.

9.4. Appendix D: Performance Results Data

This appendix contains the averaged results for the various experiments [Section 3.3.1] conducted as part of this project. All timings were gathered using the high performance timer and the values obtained rounded to four decimal places. All speedup values are calculated to two decimal places.

9.4.1. Overall Execution Time

The overall execution time experiment measures how long it takes a particular version of the algorithm to process an X-ray image. The total time for each algorithm is broken down into how long the CPU, GPU and data conversion take.

Table 13 : Overall Execution Time

Algorithm	Execution Time (s)			
	GPU	CPU	Data Conversion	Total
SISD (Int)	N/A	29.6541	N/A	29.6541
MIMD (Int)	N/A	7.4839	N/A	7.4839
SIMD (Int)	3.1818	0.0363	N/A	3.2181
SISD (Float)	N/A	43.0914	0.9989	44.0909
MIMD (Float)	N/A	10.8444	0.9989	11.8433
SIMD (Float)	3.3498	0.0250	0.9989	4.3737

Note: N/A signifies that the use of a particular processor or data conversion was unnecessary for the specific implementation of algorithm.

9.4.2. Individual Component Execution Time

The individual component execution time experiment measures how long it each component of the algorithm takes to execute on a particular parallel processor architecture.

Table 14 : Individual Component Execution Time

Image Processing Technique	SISD	MIMD	SIMD
Median Filter	29.448	7.3948	2.9442
Histogram	0.0246	0.0246	0.0321
Thresholding	0.0089	0.0058	0.0041
Sobel	0.1717	0.0502	0.0923

Note: All tests of individual components were performed using the data in its native format (Integer)

9.4.3. Data Format Performance

The data format experiment gauges the performance penalty incurred by certain data types on different processor architectures.

Table 15 : Data Format Performance

	Serial (CPU)	Parallel (CPU)	Fast, Small-Radius (GPU)	Caching (GPU)
Integer Execution Time (s)	2.4986	0.6549	0.1184	0.1019
Floating Point Execution Time (s)	3.0081	0.8055	0.1107	0.0939
Speedup (%)	16.937	18.701	-6.9220	-8.4758

9.4.4. Median Filter Performance

The median filter experiment assesses the performance of the two proposed GPU median filters against a histogram implementation for serial and parallel CPUs, in addition to the fast, small-radius GPU algorithm. The algorithms were tested with a small-radius (3×3) mask and a large-radius (19×19) mask.

Table 16 : Median Filter (3 x 3) Performance

Data Type	Execution Time (s)				
	Serial (CPU)	Parallel (CPU)	Fast, Small-Radius (GPU)	Caching (GPU)	Histogram (GPU)
Integer	2.4986	0.6549	0.1184	0.1019	0.7731
Float	3.0081	0.8055	0.1107	0.0939	0.9201

Table 17 : Median Filter (19 x 19) Performance

Data Type	Execution Time (s)				
	Serial (CPU)	Parallel (CPU)	Fast, Small-Radius (GPU)	Caching (GPU)	Histogram (GPU)
Integer	29.4488	7.3948	N/A	N/A	2.9442
Float	42.8207	10.7648	N/A	N/A	3.0836

Note: N/A denotes that the Fast, Small-radius and Caching algorithms are unable to calculate median filter with large masks.

9.4.5. Optimum Number of Threads

The optimum number of threads experiment analyses the performance the fast, small-radius median filter and a Sobel filter when process on the development GPU using various numbers of threads per thread group.

Table 18 : Optimum Number of Threads (Median)

Number of Threads	Execution Time (s)	Speedup
1	0.7235	1.00x
2	0.4115	1.75x
4	0.2516	2.87x
8	0.1734	4.17x
10	0.1556	4.64x
20	0.1223	5.91x
40	0.1045	6.92x
100	0.0986	7.33x
200	0.0995	7.26x
400	0.1000	7.23x

Table 19 : Optimum Number of Threads (Sobel)

Number of Threads	Execution Time (s)	Speedup
1	0.2403	1.00x
2	0.1646	1.46x
4	0.1258	1.91x
8	0.1078	2.22x
10	0.1041	2.30x
20	0.0961	2.49x
40	0.0919	2.61x
100	0.0891	2.69x
200	0.0889	2.70x
400	0.0895	2.68x

9.5. Appendix E: Source Code

This appendix contains the source code for the GPU implementations of various components of the algorithm.

9.5.1. DirectCompute Fast, Small-Radius Median Filter

```
1:    #define s2(a, b) temp = a; a = min(a,b); b = max(temp, b);
2:    #define mn3(a, b, c) s2(a, b); s2(a, c);
3:    #define mx3(a, b, c) s2(b, c); s2(a, c);
4:
5:    #define mnmx3(a, b, c) mx3(a, b, c); s2(a, b);
6:    #define mnmx4(a, b, c, d) s2(a, b); s2(c, d); s2(a, c); s2(b, d);
7:    #define mnmx5(a, b, c, d, e) s2(a, b); s2(c, d); mn3(a, c, e); mx3(b, d, e);
8:    #define mnmx6(a, b, c, d, e, f) s2(a, d); s2(b, e); s2(c, f); mn3(a, b, c);
      mx3(d, e, f);
9:
10:   [numthreads(40,1,1)]
11:   void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
12:   {
13:          uint v[9];
14:
15:          int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
      yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
16:
17:          int count = 0;
18:          int offset;
19:          for(int x = 0; x < 3; x++)
20:          {
21:                 for(int y = 0; y < 3; y++)
22:                 {
23:                        offset = i + ((x - 1) * width) + (y - 1);
24:                        v[count] = Buffer0[offset].i;
25:                        count++;
26:                 }
27:          }
28:          uint temp;
29:          mnmx6(v[0], v[1], v[2], v[3], v[4], v[5]); // 7 exchanges
30:          mnmx5(v[1], v[2], v[3], v[4], v[6]); // 6 exchanges
31:          mnmx4(v[2], v[3], v[4], v[7]); // 4 exchanges
32:          mnmx3(v[3], v[4], v[8]); // 3 exchanges
33:          BufferOut[i].i = v[4];
34:   }
```

Code Listing 17 : GPU Fast, Small-Radius Median Filter

This is a direct port of the Fast, Small-Radius Median Filter (McGuire, 2008) *from a Pixel Shader to a Compute Shader implementation.*

9.5.2. GPU Histogram Median Filter

```
1:      groupshared uint cache[256 * 20 * 1];
2:
3:      [numthreads(20,1,1)]
4:      void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
5:      {
6:              int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
        yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
7:
8:              for(int p = 0; p < 256; p++)
9:              {
10:                     cache[(gt.x * 256) + p] = 0;
11:             }
12:
13:             uint count = 0;
14:             int offset;
15:             for(int x = 0; x < 19; x++)
16:             {
17:                     for(int y = 0; y < 19; y++)
18:                     {
19:                             offset = i + ((x - 1) * width) + (y - 1);
20:                             count = (uint)(gt.x * 256) + (Buffer0[offset].i /
        16.0);
21:                             cache[count]++; //PUT INTO HISTOGRAM
22:                     }
23:             }
24:
25:             count = 0;
26:             uint value = 0;
27:             for(int z = 0; z < 256; z++)
28:             {
29:                     if(count <= ((radius * radius) / 2))
30:                     {
31:                             value = z;
33                      }
34              }
35              BufferOut[i].i = value * 16;
36:     }
```

Code Listing 18 : GPU Histogram Median Filter

The GPU Histogram Median Filter is effectively the algorithm proposed by Huang et al. (1979) *[Code Listing 4] implemented in a manner which is suitable for processing using a GPU via the DirectCompute API. The main difference between the two is the actual median calculation, which lacks an early escape from the "*`for`*" loop [Lines 27 - 35].*

9.5.3. GPU Sobel Filter

```
1:     [numthreads(40,1,1)]
2:     void CSMain( uint3 g : SV_GroupID, uint3 gt : SV_GroupThreadID)
3:     {
4:             int i = ((xNumThreads * yNumThreads) * g.x) + ((xNumThreads *
       yNumThreads) * g.y * (width/xNumThreads)) + ((gt.x * yNumThreads) + gt.y);
5:             uint value;
6:             value = Buffer0[i].i;
7:
8:             if((uint)i >= width && (uint)i < numPixels - width)
9:             {
10:                    int id = (gt.x * yNumThreads) + gt.y;
11:
12:                    int3 p1, p2 ,p3;
13:
14:                    p1 = int3(Buffer0[i - width - 1].i, 0, Buffer0[i - width +
       1].i * -1);
15:                    p2 = int3(Buffer0[i - 1].i * 2, 0, Buffer0[i + 1].i * -2);
16:                    p3 = int3(Buffer0[i + width - 1].i, 0, Buffer0[i + width +
       1].i * -1);
17:
18:                    p1 = p1 + p2 + p3;
19:
20:                    int vert = p1.x + p1.y + p1.z;
21:
22:                    p1 = int3(Buffer0[i - width - 1].i, Buffer0[i - width].i * 2,
       Buffer0[i - width + 1].i);
23:                    p2 = int3(Buffer0[i - 1].i * -1, 0, Buffer0[i + 1].i);
24:                    p3 = int3(Buffer0[i + width - 1].i * -1, Buffer0[i + width].i
       * -2, Buffer0[i + width + 1].i * -1);
25:
26:                    p1 = p1 + p2 + p3;
27:
28:                    int horiz = p1.x + p1.y + p1.z;
29:
30:                    float sqrtVal = (vert*vert) + (horiz * horiz);
31:                    int temp= (int)sqrt(sqrtVal);
32:
33:                    value = clamp(temp,0,4095);
34:            }
35:            BufferOut[i].i = value;
36:    }
```

Code Listing 19 : GPU Sobel Filter

The GPU Sobel Filter is essentially the same as CPU implementation [Code Listing 12], with minor alterations to conform to the DirectCompute specification [Lines 1 – 6, 35].

9.6. Appendix F: Compute Shader Functionality

The appendix details the differences between the two main versions of the Compute Shader, version 4.0 and 5.0. The data is based on the specification of the Compute Shader (MSDN, 2010b) listed in the DirectCompute documentation.

Table 20 : Compute Shader Functionaility

Functionality	Compute Shader 5.0	Compute Shader 4.0
Atomic Instructions	Yes	No
Double Precision Floating Point Values	Yes	No
Cache Memory	32KB	16KB
Limited Cache Region	No	Yes, threads are limited to 256 byte region of cache memory for writing.
Threads must access cache memory via SV_GroupIndex	No	Yes, when writing to cache memory.
Threads access any location in cache memory	Yes	No, threads can only write to their specific 256 byte region in cache memory.
Number of unordered access view resources that can be bound to a single Compute Shader	>1	1
Max. Number of Threads per Thread Group	1024	768
Max. Number of Threads in Z Dimension	64	1
Max. Number of Thread Groups per Dispatch call	65,535	65,535
Dispatch Indirect	Yes	No