The University of Maine
DigitalCommons@UMaine

Electronic Theses and Dissertations | Fogler Library

Summer 8-21-2015

Accelerating Scientific Computing Models Using GPU Processing

Raymond F. Flagg III
University of Maine - Main, raymond.flagg@maine.edu

Follow this and additional works at: http://digitalcommons.library.umaine.edu/etd

Part of the Hardware Systems Commons, and the Other Computer Engineering Commons

Recommended Citation
Flagg, Raymond F. III, "Accelerating Scientific Computing Models Using GPU Processing" (2015). *Electronic Theses and Dissertations*. 2307.
http://digitalcommons.library.umaine.edu/etd/2307

This Open-Access Thesis is brought to you for free and open access by DigitalCommons@UMaine. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of DigitalCommons@UMaine.

ACCELERATING SCIENTIFIC COMPUTING MODELS USING GPU PROCESSING

By

Raymond Forrest Flagg III

B.S. University of Maine, 2012

A THESIS

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

(in Computer Engineering)

The Graduate School

The University of Maine

August 2015

Advisory Committee:

Bruce Segee, Henry R. and Grace V. Butler Professor of Electrical and Computer Engineering, Advisor

Yifeng Zhu, Dr. Waldo "Mac" Libbey '44 Professor of Electrical and Computer Engineering

Peter Koons, Professor of Geodynamics; Professor, Climate Change Institute

THESIS ACCEPTANCE STATEMENT

On behalf of the Graduate Committee for Raymond Forrest Flagg III, I affirm that this manuscript is the final and accepted thesis. Signatures of all committee members are on file with the Graduate School at the University of Maine, 42 Stodder Hall, Orono, Maine.

Bruce Segee, Date

Henry R. and Grace V. Butler Professor of Electrical and Computer Engineering

© 2015 Raymond Forrest Flagg III

All Rights Reserved

LIBRARY RIGHTS STATEMENT

In presenting this thesis in partial fulfillment of the requirements for an advanced degree at The University of Maine, I agree that the Library shall make it freely available for inspection. I further agree that permission for “fair use” copying of this thesis for scholarly purposes may be granted by the Librarian. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Signature:

Date:

ACCELERATING SCIENTIFIC COMPUTING MODELS USING GPU PROCESSING

By Raymond Forrest Flagg III

Thesis Advisor: Dr. Bruce Segee

An Abstract of the Thesis Presented
in Partial Fulfillment of the Requirements for the
Degree of Master of Science
(in Computer Engineering)
August 2015

GPGPUs offer significant computational power for programmers to leverage. This computational power is especially useful when utilized for accelerating scientific models. This thesis analyzes the utilization of GPGPU programming to accelerate scientific computing models.

First the construction of hardware for visualization and computation of scientific models is discussed. Several factors in the construction of the machines focus on the performance impacts related to scientific modeling.

Image processing is an embarrassingly parallel problem well suited for GPGPU acceleration. An image processing library was developed to show the processes of recognizing embarrassingly parallel problems and serves as an excellent example of converting from a serial CPU implementation to a GPU accelerated implementation. Genetic algorithms are biologically inspired heuristic search algorithms based on natural selection. The Tetris genetic algorithm with A* pathfinding discusses memory bound limitations that can prevent direct algorithm conversions from the CPU to the GPU. An

analysis of an existing landscape evolution model, CHILD, for GPU acceleration explores that even when a model shows promise for GPU acceleration, the underlying data structures can have a significant impact upon that ability to move to a GPU implementation. CHILD also offers an example of creating tighter MATLAB integration between existing models.

Lastly, a parallel spatial sorting algorithm is discussed as a possible replacement for current spatial sorting algorithms implemented in models such as smoothed particle hydrodynamics.

TABLE OF CONTENTS

LIST OF TABLES xiii

LIST OF FIGURES xv

Chapter

1 INTRODUCTION 1

2 VISUALIZATION WALL 4

2.1 Driving Factors 4

2.2 New Hardware 5

2.2.1 Xorg.conf 5

2.2.1.1 PCIe Address 5

2.2.1.2 Device Section 6

2.2.1.3 Screen Section 7

2.2.1.4 ServerLayout Section 9

2.2.2 Issues 11

2.3 New Computer 11

2.3.1 First Try 11

2.3.1.1 OS Installation 11

2.3.1.2 Issues 12

2.3.2 Second Try 12

2.3.3 Third Try 13

2.4 Xrandr 13

2.5 Xorg 14
2.5.1 Default Configuration 14
2.5.2 Device Section 15
2.5.3 ZaphodHeads Option 16
2.5.4 Screen Section 17
2.6 Computer Failure 18
2.7 Summary 18
3 BUILDING GPGPU MACHINES 20
3.1 NVIDIA GPUs 20
3.2 Compute Capability 21
3.2.1 Architecture Specifications 24
3.2.2 GPU Specifications 25
3.3 Top of the Line GPUs 26
3.4 Precision vs. Cost 27
3.5 Supporting Hardware 27
3.5.1 Processor 27
3.5.1.1 Processor Manufacturer 28
3.5.1.2 Processor Selection 28
3.5.1.2.1 5930K vs. 5960X 29
3.5.1.2.2 Xeon vs. Core i7 30
3.5.2 Motherboard 31

3.5.3 Storage....31
3.5.3.1 RAID....31
3.5.3.1.1 RAID 0....32
3.5.3.1.2 RAID 1....32
3.5.3.1.3 RAID 5....33
3.5.3.2 SSD vs. Hard Drive....33
3.5.4 RAM....33
3.5.5 Power Supply....34
3.5.5.1 Power Supply Rating....34
3.5.5.2 Modular....35
3.5.6 Cooling....36
3.5.6.1 Air Cooling....37
3.5.6.2 Liquid Cooling....37
3.5.7 Case....38
3.6 GPGPU Machine Configurations....38
3.7 Additional Considerations....39
3.8 Summary....40
4 CUDA PROGRAMMING....41
4.1 Terminology....41
4.1.1 Host and Device....42
4.1.2 Kernel....42
4.1.3 __global__ Keyword....42
4.1.4 __device__ Keyword....42

4.1.5 __host__ Keyword 43

4.1.6 Threads, Blocks, and Grids 43

4.1.7 Warps 43

4.1.8 Global and Shared Memory 44

4.2 Code 44

4.2.1 Internal Kernel Variables 44

4.2.2 Kernel Configuration 45

4.2.3 Dimensionality 45

4.2.4 Streams 46

4.3 Occupancy 47

4.3.1 Occupancy Calculator Code 48

4.3.2 Alternative Versions 49

4.4 Memory Management 49

4.4.1 Explicit Memory Management 49

4.4.2 Unified Memory 51

4.5 Data Structures 52

4.5.1 Memory Coalescence 52

4.5.2 Good Data Structures for GPUs 53

4.5.3 Data Structures to Avoid 53

4.5.4 Array of Structures versus Structure of Arrays 53

4.5.5 Thrust Library 54

4.6 Error Handling 54
4.7 Multiple GPUs 55
4.8 CUDA SDK and Toolkit 56
4.9 NVCC 56
4.9.1 PTX 56
4.9.2 Unsupported NVCC Compile Options 57
4.10 Summary 57
5 IMAGE PROCESSING GPU ACCELERATION 58
5.1 ImageMagick 58
5.1.1 MagickWand API 59
5.1.2 GPU Compilation 61
5.2 GPU Filters 62
5.3 Timing 62
5.3.1 Timer Functions 62
5.3.2 Timers 63
5.4 Image Conversion 63
5.5 Results 64
5.6 Summary 69
6 TETRIS GENETIC ALGORITHM AND PATHFINDING 70
6.1 Genetic Algorithms 70
6.1.1 Genome 71
6.1.2 Initial Population 71
6.1.3 Evaluation 71

6.1.4 Breeding 72

6.1.4.1 Roulette Wheel Selection 72

6.1.4.2 Roulette Wheel Example 73

6.1.4.3 Crossover 74

6.1.5 Mutation 74

6.1.6 Elitism 75

6.2 Tetris Genome 75

6.2.1 Chromosomal Terminology 76

6.2.2 Chromosome Selection 78

6.2.3 Additional Considerations 78

6.3 Tetris Piece Placement with A* Pathfinding 79

6.3.1 Costs 79

6.3.1.1 G Cost 79

6.3.1.2 H Cost 79

6.3.2 A* Basics 80

6.3.3 Tetris Specific Implementation 80

6.3.3.1 Regions 80

6.3.3.2 Costs 81

6.4 C# CPU Results 81

6.5 Additional Considerations 82

6.6 GPU Implementation 83

6.6.1 C# to C++ 84

6.6.2 Issues 84

6.6.3 Potential Solutions 85
6.6.3.1 Global Memory Solution 85
6.6.3.2 Thrust Library Solution 85
6.6.3.3 Warp Pathfinding Solution 86
6.7 Summary...86
7 CHILD 87
7.1 MATLAB Integration..87
7.1.1 MEX...88
7.1.2 MEX Modifications 88
7.1.3 MATLAB to MEX Interface 89
7.1.4 MEX Compilation...90
7.2 GPU Implementation..91
7.2.1 Underlying Data Structures 92
7.2.2 Memory Transfers...93
7.3 Summary...93
8 PARALLEL SPATIAL SORTING ALGORITHM 95
8.1 Related Work 95
8.2 Algorithm Explanation 96
8.3 Algorithm Steps 97
8.4 Caveats..99
8.4.1 Caveat 1 99
8.4.2 Caveat 2 99

8.4.3 Caveat 3 ... 100

8.4.4 Caveat 4 ... 100

8.5 Example ... 100

8.6 Scalability ... 104

8.7 Dimensionality ... 105

8.8 Potential Applications ... 105

8.8.1 SPH ... 106

8.8.2 Agent Based Modeling ... 106

8.9 Summary ... 106

9 FUTURE WORK ... 108

9.1 Visualization Wall ... 108

9.2 Testis Genetic Algorithm and Pathfinding ... 108

9.3 Image Processing ... 109

9.4 CHILD ... 109

9.5 Parallel Spatial Sorting Algorithm ... 109

10 CONCLUSION ... 111

REFERENCES ... 112

APPENDIX A. NVIDIA QUADRO NVS 420 XORG.CONF ... 115

APPENDIX B. AUTO CONFIGURED XORG.CONF ... 119

APPENDIX C. FIRST RADEON HD 5870 XORG.CONF ... 123

APPENDIX D. SECOND RADEON HD 5870 XORG.CONF ... 127

APPENDIX E. GPGPU MACHINE CONFIGURATION SPECIFICATIONS ... 131

APPENDIX F. IMAGE PROCESSING TESTS ... 133

APPENDIX G. IMAGE PROCESSING SOURCE 137

APPENDIX H. TETRIS GENETIC ALGORITHM SOURCE.................................138

APPENDIX I. TETRIS GENETIC ALGORITHM FULL RESULTS 139

APPENDIX. J MATLABCOMPILE.SH...140

APPENDIX K. CHILD MATLAB PATCH..141

BIOGRAPHY OF THE AUTHOR 155

LIST OF TABLES

Table 3.1: Compute Capability Comparison Table Part 1 [2] 22

Table 3.2: Compute Capability Comparison Table Part 2 [2] 23

Table 3.3: Comparison of i7-5960X [12] and i7-5930K [11]..30

Table 3.4: Comparison of E7-4809 v2 [14], E7-8893 v2 [13], and i7-5930K [11] 30

Table 3.5: 80 PLUS Certification Ratings [17]...35

Table 3.6: GPGPU Machine Configurations 39

Table 4.1: Automatically Defined dim3 Variables..44

Table 4.2: Triple Chevron Parameter Definitions [2]...45

Table 4.3: cudaMemcpyKind Values 51

Table 4.4: Multiple GPU CUDA Functions..55

Table 5.1: MagickWand API Functions for Reading and Writing Images 60

Table 5.2: MagickWand API Objects for Reading and Writing Images 60

Table 5.3: MonkTimer Functions...63

Table 5.4: Generated Image Resolutions..64

Table 5.5: GPU Timer Values for Five GPUs 65

Table 5.6: Setup Timer Values for Five GPUs 66

Table 6.1: Example Roulette Wheel Selection Ranges 73

Table 6.2: Tetris Chromosome Descriptions 75

Table 6.3: Example Calculated Chromosome Values 77

Table 6.4: Best C# CPU AI Individuals Produced..82

Table 7.1: CHILD Source Files Requiring Modification for MEX Compilation 89

Table 7.2: mexFunction() Parameters..90

Table E.1: First Machine Configuration..131

Table E.2: Second Machine Configuration..132

Table F.1: Image Processing Open Timer Values..133

Table F.2: Image Processing Setup Timer Values..134

Table F.3: Image Processing GPU Timer Values..135

Table F.4: Image Processing Save Timer Values..136

LIST OF FIGURES

Figure 1.1: CPU and GPU Theoretical Floating Point Performance [2] 2

Figure 2.1: Example Quadro NVS 420 Xorg Device Section 7

Figure 2.2: Example Quadro NVS 420 Xorg Screen Section 8

Figure 2.3: Differing Xorg Screen Section Options for Single Monitor 9

Figure 2.4: Quadro NVS 420 Xorg ServerLayout Section 10

Figure 2.5: IRL Virtualization Wall Monitor and Screen Layout 10

Figure 2.6: ATI Radeon HD 5870 Port Layout 13

Figure 2.7: Command to Produce Xorg Default Configuration 14

Figure 2.8: Example Radeon HD 5870 Xorg Device Section 16

Figure 2.9: xrandr Command to List Video Port Names 17

Figure 2.10: Radeon HD 5870 Xorg Device Section With ZaphodHeads Option 17

Figure 2.11: Sample Radeon HD 5870 Xorg Screen Section 18

Figure 3.1: NVIDIA GTX 980 Specifications [5] 25

Figure 3.2: RAID Levels 0, 1, and 5 32

Figure 3.3: Modular and Non-Modular Power Supplies [18][19] 36

Figure 3.4: Liquid Cooling vs. Air Cooling 37

Figure 3.5: Case Modification 40

Figure 4.1: Architecture Comparison from CUDA C Programming Guide [2] 41

Figure 4.2: Triple Chevron Parameters and Syntax 45

Figure 4.3: dim3 struct Variables 46

Figure 4.4: Creation of Two CUDA Streams [2] 46

Figure 4.5: Using Streams for Asynchronous Memory Transfers 47

Figure 4.6: Calculating MyKernel Occupancy Programmatically[2] 48

Figure 4.7: Typical CUDA Program Flow from CUDA Wikipedia Article [21] 50

Figure 4.8: cudaMalloc and cudaFree Function Prototypes 50

Figure 4.9: cudaMemcpy Function Prototype...51

Figure 4.10: cudaMallocManaged Function Prototype 52

Figure 4.11: Error Checking Using checkCudaErrors Macro..54

Figure 4.12: Compiling for Multiple Compute Capabilities with nvcc [22] 57

Figure 5.1: Command to Generate MagickWand C++ Compiler Flags [24] 61

Figure 5.2: Arch Linux MagickWand C++ Compiler Flags...61

Figure 5.3: Example Image Convert Command..63

Figure 5.4: GPU Timer Comparison for GTX 580, GTX 680, GTX 980 67

Figure 5.5: Setup Timer Comparison for GTX 580, GTX 680, GTX 980......................68

Figure 6.1: Example Roulette Wheel Selection Pie Chart 73

Figure 6.2: Example Breeding with Random Crossover 74

Figure 6.3: Example Tetris Board with Highlighted Scoring Criteria 77

Figure 6.4: Normal Tetris Pieces...83

Figure 6.5: Additional Tetris Pieces 83

Figure 7.1: Wrapping exit() Functions for MEX Compilation 88

Figure 7.2: Command to Compile CHILD from CHILD Main Directory 90

Figure 7.3: Command to Compile CHILD matlabInterface Object Files........................91

Figure 7.4: Linked List Insertion and Removal 92

Figure 8.1: Grid of Blocks with Coordinates 96

Figure 8.2: Example Block to Global Memory Object Array Mapping 97

Figure 8.3: Example Grid of Blocks Layout and Shared Memory Timestep 0 101

Figure 8.4: Example Global Memory with Active Objects 101

Figure 8.5: Example Grid of Blocks Layout and Shared Memory Timestep 1 102

Figure 8.6: Example Grid of Blocks Layout and Shared Memory Timestep 2 103

Figure 8.7: Example Shared Memory Just Prior to Timestep 3 103

Figure 8.8: Example Global Memory Array During Timestep 3 104

Figure 8.9: Grid of GPU Grids 105

CHAPTER 1

INTRODUCTION

Dedicated Graphics Processing Units (GPUs), often referred to as graphics cards, have a long history, becoming common in video game consoles and early computers with their use being driven by the demand for better graphics for gaming. This has fueled companies such as NVIDIA and AMD to create ever more powerful GPUs. GPUs are highly parallel processors and General Purpose GPU (GPGPU) computing went mainstream in 2007 with the release of NVIDA's CUDA toolkit [1].

GPGPUs are graphics processing units used for general purpose processing as opposed to strictly rendering graphics. GPGPUs are one of the most popular types of accelerators, processors designed to speed up computation by leveraging high parallelism. The massively parallel architecture of GPUs allows for much higher theoretical processing power compared to that of CPUs. Figure 1.1 shows a comparison of the theoretical processing power between modern CPUs and GPUs.

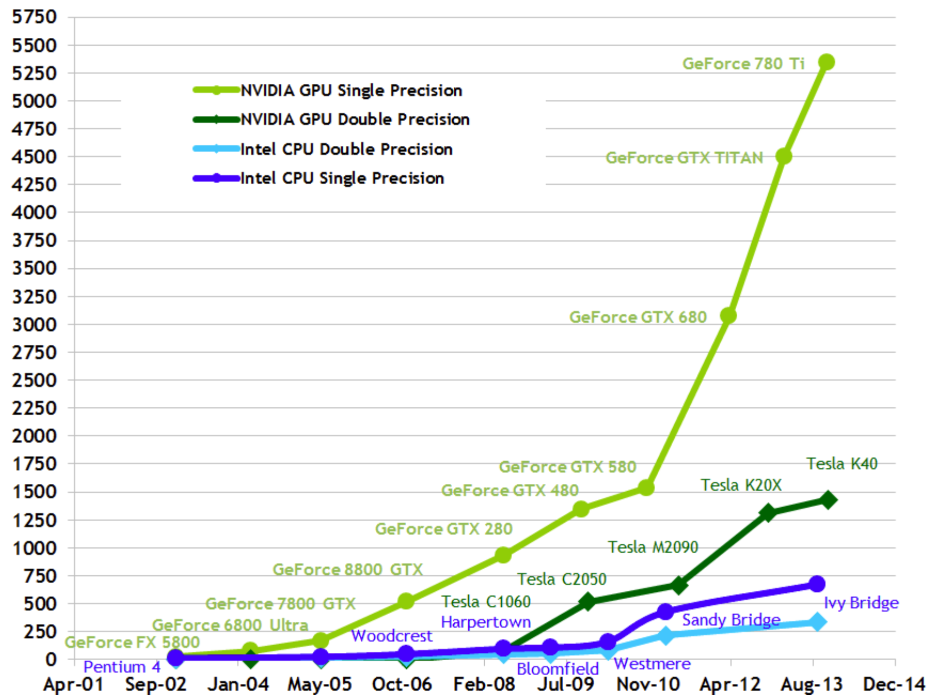

Figure 1.1: CPU and GPU Theoretical Floating Point Performance [2]

As GPU development has continued, many scientific models, especially those originally designed for High Performance Computing (HPC) have begun including support for the use of GPUs as accelerators to decrease the runtime, often by an order of magnitude [3]. The increased GPU accelerator support is reflected in supercomputer designs, with 88 of the top 500 supercomputers including some form of accelerator/co-processor technology to achieve faster computation [4]. Many programs that are computationally intensive may be good candidates for GPU acceleration. Not all applications are suitable for GPU acceleration as there are unique limits imposed by GPU architecture not found in normal HPC architectures.

The low entry cost of purchasing high end consumer GPUs that can be used for GPGPU programming has made it possible to run massively parallel scientific models on the desktop as opposed to requiring a supercomputer. The GTX 980 has over 4.5 TFlops

of floating point processing power [5]. A desktop machine containing two GTX 980s would have enough processing power to be on the TOP 500 Supercomputer list in 2008.

The focus of this thesis is the analysis of utilizing the highly parallel nature of GPUs to accelerate scientific models using consumer hardware.

The first step in researching the application of GPUs in scientific computing is having strong hardware. In order to accommodate model visualization for large scientific models, the use of multiple screens driven by a single computer to create large displays (visualization walls) is an excellent starting point. The creation and maintenance of visualization walls is discussed in Chapter 2, followed by the design and building of GPGPU specific machines for scientific research in Chapter 3.

The architectural differences between CPUs and GPUs offer unique challenges in developing models that effectively utilize the large computational power offered by GPUs. Chapter 4 offers an introduction to GPU programming using CUDA.

Chapters 5 through 7 discuss converting pre-existing models from CPU implementations to GPU implementations. Chapter 5 focuses on an embarrassingly parallel problem while Chapters 6 and 7 covers some of the pitfalls associated with implementing GPU accelerated versions of models that seem, at first glance, like good candidates for GPU acceleration. Chapter 7 also discusses creating tighter integration between existing models using MATLAB as a common interface. Chapter 8 looks at designing a GPU sorting algorithm to overcome some of the limitations found in earlier chapters.

CHAPTER 2

VISUALIZATION WALL

As scientific models continue to grow in complexity and the amount of data they are able to process continues to increase, visualizing that data becomes more difficult. Viewing models on a large screen or projector has the drawback of losing low level resolution as the number of pixels doesn't increase, only the size of the pixels. The increasing prevalence of 4K screens will help mitigate this effect to some extent, but 4K screens are still much more expensive than 1080p and lower resolution screens. Visualization walls are much more affordable and can offer much higher resolution while still offering a large display area. Visualization walls combine the output from multiple high resolution screens to form a single, much higher resolution, display. The combined display of a visualization wall allows the big picture view while keeping smaller scale resolution so that fine detail isn't lost. As the prices of 4K screens drop, visualization walls using 4K screens will continue to offer extremely high resolution displays.

This chapter outlines the creation of a visualization wall from standard consumer hardware with the only custom modifications applied to the xorg.conf configuration file. Several iterations of hardware are detailed including an analysis of the creation of the xorg.conf configuration file for each iteration.

2.1 Driving Factors

The original visualization wall in the Instrumentation Research Laboratory (IRL) located at the University of Maine in Barrows Hall was driven by an older machine that had experienced memory issues related to the motherboard rather than to the DIMMs. These issues required that either the motherboard be replaced or a different computer be

found. Given the age of the machine and the financial constraints of the setup, a different computer was required.

2.2 New Hardware

The first step was to find a computer that would work. The old machine had an AMD processor and it was determined that another computer with an AMD processor would be the easiest replacement. A suitable replacement was located, the hard drive migrated, and xorg.conf reconfigured to adjust for the change in graphics card PCIe bus changes. The same three NVIDIA Quadro NVS 420's (Q420) were used in this configuration and no updates were made to the OS, a version of Arch Linux running a 3.x kernel with unknown custom configurations. The cards were connected such that a card was located at PCIe address 03:0.0, one at address 07:0.0, and one at address 0c:0.0. The top and bottom cards had four monitors connected to each with the middle card having a single monitor connected.

2.2.1 Xorg.conf

The previous version of the xorg.conf file (configuration file) was modified to work with the new configuration. It wasn't a one to one modification and the original configuration was not saved during this stage, which prevents any in depth analysis of the changes. Changes were made to the ServerLayout, Device, and Screen sections and the entire modified configuration file can be seen in Appendix A. It was determined through trial and error that five Screens were needed.

2.2.1.1 PCIe Address

The way that the Q420 handles the four connected monitors is such that ports one and two can be connected as TwinViews and ports three and four can also be connected

as TwinViews. As there are nine monitors and three Q420's, two cards were completely filled and the third had only a single monitor connected as described in Section 2.2. Each card was defined in a Device section with the BusID specifying the PCIe address with the periods replaced by colons such that 03:0.0 becomes 3:0:0. Each card is addressable at two specific addresses, the one at which it is specifically defined, and the next sequential address. In the case of the card located at 03:0.0, the card, and two of the monitors attached to it, are addressable at 04:0.0. For addresses above 09:0.0, such as 0c:0.0, the numerical value is used in the configuration file. In the example, 0c:0.0, PCI:12:0:0 is used.

2.2.1.2 Device Section

Five devices were defined in the xorg.conf file. Each device has a specific BusID. The BusID's used were PCI:3:0:0, PCI:4:0:0, PCI:7:0:0, PCI:8:0:0, and PCI:12:0:0. The definition of the Device section is shown in Figure 2.1 where PCI:X:0:0 is replaced by the correct address and DeviceX is replaced by sequentially increasing values starting at Device0.

```
Section "Device"
    Identifier      "DeviceX"
    Driver          "nvidia"
    VendorName      "NVIDIA Corporation"
    BoardName       "Quadro NVS 420"
    BusID           "PCI:X:0:0"
EndSection
```

Figure 2.1: Example Quadro NVS 420 Xorg Device Section

2.2.1.3 Screen Section

For each device a screen was defined, resulting in five screens starting from Screen0 and increasing sequentially up to Screen4. Many of the options are left over from the original configuration file. The only options modified were the Identifier and the Device. An example Screen section used is shown in Figure 2.2 where ScreenX and DeviceX are changed to the corresponding screen and device values.

```
Section "Screen"
    Identifier      "ScreenX"
    Device          "DeviceX"
    Monitor         "Monitor0"
    DefaultDepth    24
    Option          "ConnectedMonitor" "DFP,DFP"
    Option          "UseDisplayDevice" "DFP-0,DFP-1"
    Option          "CustomEDID" "DFP-0:/etc/X11/edid.bin;DFP-
    1:/etc/X11/edid.bin"
    Option          "TwinView" "1"
    Option          "TwinViewXineramaInfoOrder" "DFP-0"
    Option          "metamodes" "DFP-0: nvidia-auto-select +0+0,
    DFP-1: nvidia-auto-select +0+1024"
    SubSection      "Display"
            Depth   24
    EndSubSection
EndSection
```

Figure 2.2: Example Quadro NVS 420 Xorg Screen Section

The options for TwinView and TwinViewXineramaInfoOrder allow the screen to encompass two monitors. In Screen4, several options are slightly different and are shown in Figure 2.3.

```
Option          "TwinView" "0"
Option          "TwinViewXineramaInfoOrder" "DFP-0"
Option          "metamodes" "DFP-0: nvidia-auto-select +0+0"
```

Figure 2.3: Differing Xorg Screen Section Options for Single Monitor

These options remove the use of TwinView and specify only a single monitor as this screen has only a single monitor attached.

2.2.1.4 ServerLayout Section

The physical layout of the nine monitors for the visualization wall is a three by three grid. The ServerLayout section defines the positions of the screens as well as the input devices and the Xinerama option. The screen positions are absolute, but relative positions could also have been used. The SeverLayout section in use is shown in Figure 2.4.

```
Section "ServerLayout"
    Identifier      "Layout0"
    Screen          0  "Screen0" 0 0
    Screen          1  "Screen1" 1280 0
    Screen          2  "Screen2" 0 2048
    Screen          3  "Screen3" 2560 0
    Screen          4  "Screen4" 2560 2048
    InputDevice     "Keyboard0" "CoreKeyboard"
    InputDevice     "Mouse0" "CorePointer"
    Option          "Xinerama" "1"
EndSection
```

Figure 2.4: Quadro NVS 420 Xorg ServerLayout Section

Each monitor is set to the maximum resolution, 1280x1024. Figure 2.5 shows the layout of the physical monitors with each monitor labeled and each screen color coded.

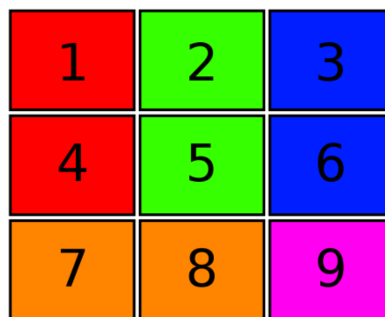

Figure 2.5: IRL Virtualization Wall Monitor and Screen Layout

Screen0 is shown in red, Screen1 in green, Screen2 in blue, Screen3 in orange, and Screen4 in purple. The working configuration file using the Quadro NVS 420 graphics cards can be found in Appendix A.

2.2.2 Issues

This setup worked, but it was impossible to update the system for several reasons. The system had not been updated for quite some time and several changes had been made to the file system used by Arch Linux in the interval. There were also the unknown custom configurations that might be overwritten by an update. The file system changes made to Arch Linux are difficult to apply all at once but an attempt was made to update the system. The process predictably broke almost everything and it became necessary to reinstall the operating system.

2.3 New Computer

Due to budget constraints, a suitable computer could not be purchased and so existing hardware was utilized. Several computers not currently in use were tested as viable replacements.

2.3.1 First Try

The intent for the first replacement computer was to simply use the old NVIDIA Quadro NVS 420's that powered the previous version of the wall along with a new install of Arch Linux. It is difficult to install an operating system on the wall when all the monitors are connected due to ergonomics, the size of the screen, and the limited range a wired keyboard and mouse have. As such, an old video card was placed in the computer in place of the Q420's for installation purposes.

2.3.1.1 OS Installation

The installation followed the Arch Linux Beginner's Guide [6] and was reasonably straight forward. The hard drive for the computer has a capacity of 320GB. The partitioning scheme chosen was 50GB for the root file system partition, 16GB for the

swap partition, and the remaining space for the home partition. Once the OS had been installed, including a graphical environment, in this case KDE, the Q420's were installed and the computer was moved to the wall.

2.3.1.2 Issues

After some research, it turns out that the current NVIDIA drivers no longer support the Q420 and the legacy drivers are not only difficult to come by, but not fully supported by newer versions of Xorg. Since the purpose of this whole operation was to get the wall back up and running in such as state that updating the OS was possible, it was decided that the Q420's were not the best choice of video cards for the job.

2.3.2 Second Try

A visualization wall that had previously been located at the Foster Center for Student Innovation was removed during the 2013-2014 school year and the machine driving that wall had been sitting unused since it's removal. The Innovation Center wall was a 16 monitor setup using three ATI Radeon HD 5870's (R5870). The R5870's used are hex head graphics card with six mini-displayport outputs. There were no known issues with the machine before it was removed from the Innovation Center and so the hard drive from the wall in the IRL was reconfigured for this hardware. As it turns out, this computer also had an issue with its RAM. Unfortunately, the CPU heatsink obstructs at least one of the RAM DIMMs and would require removal in order to test the RAM. This is a long process and would require purchasing thermal compound to reapply the heatsink and as other machines were available, this machine became undesirable for the wall.

2.3.3 Third Try

At this point, the original replacement computer was configured to use two of the R5870's and to drive the wall. This configuration worked and the following sections use this computer. The current configuration has an AMD Phenom 8650 Triple-Core Processor running at 1.15GHz with 4GB DDR2 RAM. The same procedure for installing the OS was followed as outlined in Section 2.3.1.1 with the exception that the installation was done with a single R5870 installed. With a working Arch Linux installation in place, the second R5870 was installed and the computer was moved for final installation.

The R5870 has six mini-displayport heads that are enumerated starting at DisplayPort-0. Figure 2.6 shows the first port as seen from the rear of the card when installed.

Figure 2.6: ATI Radeon HD 5870 Port Layout

2.4 Xrandr

With the installation of an up-to-date OS and kernel, xrandr was an available option for configuration. When only a single graphics card was installed, xrandr worked very well at setting up screens. However, when a second card was added, xrandr did not recognize it automatically. When the command to link the second card to the screens was initialized, some of the screens would become available, but not all of them and other

commands to position them did not work correctly or were ignored. It became evident that as of August of 2014, xrandr was still not suitable for multiple video cards and multiple monitors.

2.5 Xorg

At this point, Xorg seemed to be the only option left. The majority of posted information regarding multihead configuration files focuses on NVIDIA graphics cards and uses TwinView as outlined in Section 2.2.1.3. The TwinView option is only available for NVIDIA graphics cards and cannot be used with AMD cards.

2.5.1 Default Configuration

The first step to success was generating a default configuration file by running command shown in Figure 2.7. This command must be run as the privileged user root.

```
Xorg -configure
```

Figure 2.7: Command to Produce Xorg Default Configuration

It is possible to run the command shown in Figure 2.7 using sudo, though the documentation suggests that using sudo may not work. The configuration command creates a default configuration file located at /root/xorg.conf.new. The default configuration file can be seen in Appendix B. The configuration file didn't create a fully configured screen layout, but did correctly identify two graphics cards and create Device sections for each card with all options listed and commented out.

2.5.2 Device Section

From the newly created configuration file and many Google searches, it was determined that the Device section has a Screen option that can be specified. According to the xorg.conf man page [7],

> *Screen number*
>
> *This option is mandatory for cards where a single PCI entity can drive more than one display (i.e., multiple CRTCs sharing a single graphics accelerator and video memory). One Device section is required for each head, and this parameter determines which head each of the Device sections applies to. The legal values of number range from 0 to one less than the total number of heads per entity. Most drivers require that the primary screen (0) be present.*

This revelation led to creating separate Device sections for each port used on each card. An example Device section is shown in Figure 2.8.

```
Section "Device"
        Identifier      "CardX"
        Driver          "radeon"
        BusID           "PCI:X:0:0"
        Screen          Y
EndSection
```

Figure 2.8: Example Radeon HD 5870 Xorg Device Section

The Identifier CardX is replaced by a sequentially increasing number starting at Card0. Screen Y is replaced with a number between zero and one less than the number of heads on the card and refers to which head is being referenced. For the R5870's, the maximum screen number is five. For a single card, the numerical values for CardX and Screen Y are the same. When a second card is introduced, the first card will still have the same values, but while the CardX values for the second card will continue to increase, the Screen Y values will start over at zero.

2.5.3 ZaphodHeads Option

The example Device section in Figure 2.8 is still missing one option, ZaphodHeads. The ZaphodHeads option is only available for AMD/ATI cards. This option allows a specific device to be bound to a specific output. Unfortunately, the man page for xorg.conf doesn't contain any information regarding this option, though the ArchWiki ATI page has useful information [8]. The output value used can be referenced in two ways. If xrandr is enabled, the name to use can be found by running the command shown in Figure 2.9.

```
xrandr -q
```

Figure 2.9: xrandr Command to List Video Port Names

Each head has an associated name. Most current systems have xrandr enabled, but in the event it has not been enabled, the xorg log file found at /var/log/Xorg.0.log can be used and should give the correct name for the output. In the case of the R5870, each port's name is of the form DisplayPort-X, where X is replaced by a numerically increasing value starting at zero on the first card. A complete Device section including the ZaphodHeads option is shown in Figure 2.10.

```
Section "Device"
        Identifier      "CardX"
        Driver          "radeon"
        BusID           "PCI:1:0:0"
        Screen          X
        Option          "ZaphodHeads" "DisplayPort-X”
EndSection
```

Figure 2.10: Radeon HD 5870 Xorg Device Section With ZaphodHeads Option

2.5.4 Screen Section

Each individual monitor has a Screen section that binds a device to a screen. There should be one screen per device and so this setup has nine devices and nine screens. A sample Screen section is shown in Figure 2.11.

```
Section "Screen"
        Identifier      "ScreenX"
        Device          "CardX"
        Monitor         "Default-Monitor"
EndSection
```

Figure 2.11: Sample Radeon HD 5870 Xorg Screen Section

ScreenX and CardX are replaced by previously defined values as described in Section 2.5.2. The full configuration file using the R5870s can be found in Appendix C.

2.6 Computer Failure

The computer used and outlined in Section 2.3.3 finally died and another replacement computer was required. Fortunately, the computer outlined in Section 2.3.2, that had previously been located at the Foster Center for Student Innovation, was still available. The memory issues it had been experiencing were related to a poorly seated DIMM and were thus easily fixed. The original assumption that the heat sink required removal in order to reseat the DIMM turned out to be fallacious. This computer was used as a replacement for the previous computer and the only necessary changes to the Xorg configuration file were to reflect the different PCIe bus addresses as discussed in Section 2.2.1.1. The full Xorg configuration file can be found in Appendix D.

2.7 Summary

The visualization wall modifications outlined in this chapter are an excellent starting point for building visualization walls that can be easily maintained. The visualization wall currently residing in the IRL has been continuously kept up to date and

is running the most current Linux kernel available through Arch Linux. The only modification to the system is the custom Xorg configuration file shown in Appendix D.

CHAPTER 3

BUILDING GPGPU MACHINES

When considering how to build a GPGPU specific machine using consumer hardware, there are a number of differences from building a machine that focuses on CPU processing. While one would normally start with processor selection and build the machine around the desired processor, the first step in building a GPGPU machine is to determine what type of GPU best fits one's purposes. There are two major GPU manufacturers to choose from in the consumer market, NVIDIA and AMD. While NVIDIA and AMD hold the majority market for consumer discrete GPUs, but the market isn't equally divided. NVIDIA currently has a much higher market share compared to AMD. In the fourth quarter of 2014, NVIDIA had 76% of the discrete GPU market leaving AMD with the remaining 24% [9]. It is also important to consider that AMD GPUs can only run programs written in OpenCL, while NVIDIA GPUs allow for programs to be run written in both OpenCL and CUDA. Given this, NVIDIA is the best choice as it offers the largest number of options for programming.

This chapter outlines the selection criteria for building GPGPU machines using consumer hardware and concludes with two GPGPU machine configurations created from the outlined selection criteria using the latest available NVIDIA consumer GPUs.

3.1 NVIDIA GPUs

When talking about NVIDIA GPU selection in the consumer space, compute capability is the main differentiating factor. NVIDIA compute capability defines the physical CUDA features of a GPU. This includes the number of CUDA cores, amount of shared memory, bus speeds, etc.

3.2 Compute Capability

It was once easy to tell which card was “king” when looking at usefulness from a CUDA perspective as higher compute capability directly correlated to better CUDA specifications. The highest level single GPU cards for any given generation were the best because they had more CUDA cores and all new cards supported the highest compute capability. This is no longer the case. The divergence of compute capability has made it much less clear as to what compute capability and what card is best in a given situation. Some of the 3.x level compute capabilities are better suited for shared memory tasks than the 5.x level compute capabilities. Tables 3.1 and 3.2 show the comparison between compute capabilities for 2.x and higher.

<table>
<tr><th></th><th colspan="6">Compute Capability</th></tr>
<tr><th>Technical Specifications</th><th>2.x</th><th>3.0</th><th>3.2, 3.5</th><th>3.7</th><th>5.0</th><th>5.2</th></tr>
<tr><td>Maximum dimensionality of grid of thread blocks</td><td colspan="6">3</td></tr>
<tr><td>Maximum x-dimension of a grid of thread blocks</td><td>65535</td><td colspan="5">2^32-1</td></tr>
<tr><td>Maximum y- or z-dimension of a grid of thread blocks</td><td colspan="6">65535</td></tr>
<tr><td>Maximum dimensionality of thread block</td><td colspan="6">3</td></tr>
<tr><td>Maximum x- or y-dimension of a block</td><td colspan="6">1024</td></tr>
<tr><td>Maximum z-dimension of a block</td><td colspan="6">64</td></tr>
<tr><td>Maximum number of threads per block</td><td colspan="6">1024</td></tr>
<tr><td>Warp size</td><td colspan="6">32</td></tr>
<tr><td>Maximum number of resident blocks per multiprocessor</td><td>8</td><td colspan="3">16</td><td colspan="2">32</td></tr>
<tr><td>Maximum number of resident warps per multiprocessor</td><td>48</td><td colspan="5">64</td></tr>
<tr><td>Maximum number of resident threads per multiprocessor</td><td>1536</td><td colspan="5">2048</td></tr>
<tr><td>Number of 32-bit registers per multiprocessor</td><td>32 K</td><td colspan="2">64 K</td><td>128 K</td><td colspan="2">64K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread block</td><td>32 K</td><td colspan="5">64 K</td></tr>
<tr><td>Maximum number of 32-bit registers per thread</td><td colspan="2">63</td><td colspan="4">255</td></tr>
<tr><td>Maximum amount of shared memory per multiprocessor</td><td colspan="3">48 KB</td><td>112 KB</td><td>64KB</td><td>96 KB</td></tr>
<tr><td>Maximum amount of shared memory per thread block</td><td colspan="6">48 KB</td></tr>
<tr><td>Number of shared memory banks</td><td colspan="6">32</td></tr>
<tr><td>Amount of local memory per thread</td><td colspan="6">512 KB</td></tr>
<tr><td>Constant memory size</td><td colspan="6">64 KB</td></tr>
<tr><td>Cache working set per multiprocessor for constant memory</td><td colspan="4">8 KB</td><td colspan="2">10 KB</td></tr>
<tr><td>Cache working set per multiprocessor for texture memory</td><td>12 KB</td><td colspan="5">Between 12 KB and 48 KB</td></tr>
<tr><td>Maximum width for a 1D texture reference bound to a CUDA array</td><td colspan="6">65536</td></tr>
<tr><td>Maximum width for a 1D texture reference bound to linear memory</td><td colspan="6">2^27</td></tr>
</table>

Table 3.1: Compute Capability Comparison Table Part 1 [2]

	Compute Capability					
Technical Specifications	**2.x**	**3.0**	**3.2, 3.5**	**3.7**	**5.0**	**5.2**
Maximum width and number of layers for a 1D layered texture reference	16384 x 2048					
Maximum width and height for a 2D texture reference bound to a CUDA array	65536 x 65535					
Maximum width and height for a 2D texture reference bound to linear memory	65000 x 65000					
Maximum width and height for a 2D texture reference bound to a CUDA array supporting texture gather	16384 x 16384					
Maximum width, height, and number of layers for a 2D layered texture reference	16384 x 16384 x 2048					
Maximum width, height, and depth for a 3D texture reference bound to a CUDA array	2048 x 2048 x 2048	4096 x 4096 x 4096				
Maximum width (and height) for a cubemap texture reference	16384					
Maximum width (and height) and number of layers for a cubemap layered texture reference	16384 x 2048					
Maximum number of textures that can be bound to a kernel	128	256				
Maximum width for a 1D surface reference bound to a CUDA array	65536					
Maximum width and number of layers for a 1D layered surface reference	65536 x 2048					
Maximum width and height for a 2D surface reference bound to a CUDA array	65536 x 32768					
Maximum width, height, and number of layers for a 2D layered surface reference	65536 x 32768 x 2048					
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array	65536 x 32768 x 2048					
Maximum width (and height) for a cubemap surface reference bound to a CUDA array	32768					
Maximum width (and height) and number of layers for a cubemap layered surface reference	32768 x 2046					
Maximum number of surfaces that can be bound to a kernel	8	16				
Maximum number of instructions per kernel	512 million					

Table 3.2: Compute Capability Comparison Table Part 2 [2]

3.2.1 Architecture Specifications

The architecture of the GPU, Tesla (not to be confused with the Tesla line of cards), Fermi, Kepler, or Maxwell, shows the first part of the compute capability, 1, 2, 3, or 5 respectively. Within each architecture there are various levels of compute capability. The most relevant compute capabilities when selecting a consumer grade GPGPU are currently 3.5 and 5.2. Tables 3.1 and 3.2 above show a comprehensive list of differences between compute capabilities. Upon inspection it becomes obvious that there are a large number of reference points for comparison. However, only a few of these have any practical relevance to GPGPU programming. By limiting the examination to only compute capabilities 3.5 and 5.2, many reference points become extemporaneous as these compute capabilities have many similarities. The two main points of interest are the maximum number of resident blocks per multiprocessor and the maximum amount of shared memory per multiprocessor. The maximum number of resident blocks per multiprocessor is an indication of the hard limit for how many simultaneous blocks a single multiprocessor can handle. Having a higher maximum number of resident blocks allows for more efficient parallelism at the multiprocessor level. Correlated to this is the maximum amount of shared memory per multiprocessor. These are two of the limiting factors related to how many blocks will actually be run on a multiprocessor. Compute capability 3.5 allows for 16 blocks per multiprocessor with 48K of shared memory while compute capability 5.2 allows for 32 blocks per multiprocessor with 96K of shared memory. This is the same ratio of blocks to shared memory, but if fewer blocks are run per multiprocessor, then more shared memory is available for each block.

3.2.2 GPU Specifications

While looking at compute capability can help narrow down what line of cards will work best, the actual specifications of the card are used to select the "best" card. NVIDIA provides a large list of specifications for their cards. The specifications for the GTX 980 are shown below in Figure 3.1.

GTX 980 Engine Specs:	
CUDA Cores	2048
Base Clock (MHz)	1126
Boost Clock (MHz)	1216
Texture Fill Rate (GigaTexels/sec)	144
GTX 980 Memory Specs:	
Memory Clock	7.0 Gbps
Standard Memory Config	4 GB
Memory Interface	GDDR5
Memory Interface Width	256-bit
Memory Bandwidth (GB/sec)	224
GTX 980 Technology Support:	
NVIDIA SLI® Ready	Yes (4-way)
NVIDIA G-Sync™-Ready	Yes
NVIDIA GameStream™-Ready	Yes
GeForce ShadowPlay™	Yes
NVIDIA GPU Boost™	2.0
Dynamic Super Resolution	Yes
MFAA	Yes
NVIDIA GameWorks™	Yes
Microsoft DirectX	12 API
OpenGL	4.4
CUDA	Yes
Bus Support	PCI Express 3.0
OS Certification	Windows 8 & 8.1, Windows 7, Windows Vista, Linux, FreeBSD x86
Display Support:	
Maximum Digital Resolution*	5120x3200
Maximum VGA Resolution	2048x1536
Standard Display Connectors	Dual Link DVI-I, HDMI 2.0, 3x DisplayPort 1.2
Multi Monitor	4 displays
HDCP	Yes
Audio Input for HDMI	Internal
GTX 980 Graphics Card Dimensions:	
Height	4.376 inches
Length	10.5 inches
Width	Dual-width
Thermal and Power Specs:	
Maximum GPU Tempurature (in C)	98 C
Graphics Card Power (W)	165 W
Minimum System Power Requirement (W)	500 W
Supplementary Power Connectors	2x 6-pins

Figure 3.1: NVIDIA GTX 980 Specifications [5]

The extensive list of specifications provided can be overwhelming to parse, but for GPGPU use, many of the specifications can be safely ignored. The specifications of interest lie in the Engine and Memory Specs sections. The number of CUDA cores lists the total number of CUDA cores between all multiprocessors and so correlates to the number of threads that can be run in parallel. When comparing GPUs of the same architecture, the number of CUDA cores can be directly compared with higher values being better.

3.3 Top of the Line GPUs

At the time of building the current GPGPU machines, the top of the line consumer GPU used for CUDA was the GTX 980. The GTX 990 could be considered “better” than the GTX 980, though for GPGPU programming, this isn’t the case. The GTX 990 is a dual processor GPU meaning that there are essentially two GTX 980s on the same physical card. The reason this isn't considered the top of the line is that the bus speed is shared between the two GPUs and thus halves the data transfer rate to and from the separate GPUs on the card. The better option in this case is to use two separate GTX 980s so that the bus speed isn't affected. This isn’t a consideration when picking a GPU for gaming, but the largest bottleneck in GPGPU programming is often transferring data to and from the GPU. It therefore makes sense that the best way to minimize that bottleneck is to make sure not to limit the bandwidth for data transfers by trying to transfer data on the same PCIe bus. This holds true for linked PCIe slots on the motherboard. It is much better to have completely separate slots for each card than to have linked slots.

3.4 Precision vs. Cost

There is a second consideration when looking at NVIDIA GPUs for GPGPU machines. The GeForce family of GPUs is designed for single precision floating point calculations. The physical architecture of the GPUs limits double precision calculations significantly compared to single precision calculations. For applications where double precision is necessary, the Tesla line from NVIDIA offers greatly improved double precision performance, though at a much higher cost. Tesla GPUs are cost prohibitive for most research projects, but are often found in supercomputers as accelerators.

3.5 Supporting Hardware

After selecting the appropriate GPU, the rest of the supporting hardware can be chosen. The choice of supporting hardware follows a very similar path as when selecting hardware for a regular computer build. However, unlike when building a regular computer, the motherboard and processor are given equal weighting for a GPGPU machine. This is largely due to the fact that the PCIe slots need to be fully independent and, for the newest GPUs, support PCIe 3.0 x16.

3.5.1 Processor

Processor selection has less importance for a GPGPU machine than for a regular computer because the emphasis of a GPGPU machine is on the GPU rather than on the CPU. This doesn't mean, however, that CPU selection isn't important. For comparison sake, a strong CPU is needed. Having a powerful CPU as a baseline gives what can be considered a worst case speedup compared to the GPU. In most circumstances, the actual speedup when comparing a serial CPU implementation vs. a multi-threaded CPU implementation vs. a GPU implementation will heavily depend upon the CPU and GPU.

For existing systems, it's much more plausible to add a new GPU than to change the CPU. Therefore, testing a strong GPU vs. a strong CPU shows the minimum speedup that can be expected for a program. In most circumstances, the speedup will actually be higher because the difference between the CPU and the GPU will be greater with a less powerful CPU.

It is also important to remember that for running any of the serial sections of a program, the CPU will still be utilized and that not all programs that will be run will support GPU acceleration or be good candidates for GPU acceleration. For these reasons, CPU selection still has an impact, though less than for a regular computer.

3.5.1.1 Processor Manufacturer

There are two major CPU manufacturers, Intel and AMD. Intel has a much larger market share compared to AMD, 72% vs. 28% [10], and thus there are often more motherboard choices for Intel processors than for AMD. Intel processors are also often able to get more processing power out of fewer cores than AMD, but are usually more expensive.

3.5.1.2 Processor Selection

Having decided upon an Intel processor, the next step is to determine what type of processor fits both the budget and processing requirements. The top of the line consumer Intel processor line is the 4th generation Core-i7 Haswell-E family. The Haswell family is divided between the regular Core-i7s with up to quad-core processors and the E editions with either six or eight cores. The three E edition processors are top of the line and the real choice is between the Core i7-5930K and the Core i7-5960X.

3.5.1.2.1 5930K vs. 5960X

There are two major differences between the Core i7-5930K and the Core i7-5960X. The 5930K has six physical cores running at 3.5 GHz while the 5960X has eight physical cores running at 3.0 GHz. Both processors use the LGA2011-3 socket, and so are interchangeable. Choosing between these two processors is heavily dependent upon use case of the machine and the budget. The 5960X has two more physical cores and so multithreaded programs that aren't as computationally intensive, but that benefit from good parallelism, will be a better fit for this processor. The 5930K only has six physical cores, but runs 0.5 GHz faster, meaning that actual computation will be faster, but won't have as high a level of parallelism. It is also important to note that the 5960X is significantly more expensive, nearly twice the cost, compared to the 5930K. Since GPGPU machines are built to utilize the GPU for high parallelism, the 5930K is a better compliment to the GPU than the 5960X because of the faster processor speed. If the two speeds were closer, say only a difference of 0.2 GHz, then the choice would be less clear, though the price difference would still be a significant factor. In that case, the extra cores could outweigh such a miniscule speed difference if the budget allowed for the extra expense. Table 3.3 shows a comparison of the most important factors between the 5960X and the 5930K.

	Core i7-5960X	Core i7-5930K
Recommended Customer Price	$1059.00	$594.00
Number of Cores	8	6
Number of Threads	16	12
Processor Base Frequency	3 GHz	3.5 GHz

Table 3.3: Comparison of i7-5960X [12] and i7-5930K [11]

3.5.1.2.2 Xeon vs. Core i7

Another option would be to use a server processor such as one of the Intel Xeon processors. These processors can be found with many more cores than a Core i7, but they often operate at a lower frequency and almost always cost considerably more than the Core i7 equivalent. The same reasoning used to determine the better Core i7 processor applies in this case as well. Table 3.4 shows a comparison between two Xeon processors, the E7-4809 v2 and the E7-8890 v2, and the i7-5930K.

	Xeon E7-4809 v2	Xeon E7-8893 v2	Core i7-5930K
Recommended Customer Price	$1223.00	$6841.00	$594.00
Number of Cores	6	6	6
Number of Threads	12	12	12
Processor Base Frequency	1.9 GHz	3.4 GHz	3.5 GHz

Table 3.4: Comparison of E7-4809 v2 [14], E7-8893 v2 [13], and i7-5930K [11]

All three processors have six physical cores. The main differences are the processor frequency and price. The E7-4809 is more than twice the cost of the i7-5930K and runs at a significantly lower frequency, clearly making the E7-4809 a poor choice. The E7-8893 is much more closely matched to the i7-5930K in terms of processor

frequency, but costs 10x more than the i7-5930K. This clearly shows that the speed tradeoffs and extra cost, make the Core i7-5930K not only a better deal but better suited for a GPGPU machine.

3.5.2 Motherboard

There are a number of considerations when selecting a motherboard, not least of which is checking CPU support and PCIe slot configuration. One of the main selection criteria for the motherboard is PCIe compatibility. Top of the line GPUs need to be in a PCIe 3.0 x16 slot in order to gain the best performance. Both the GTX 980 and the GTX Titan Black are compatible with PCIe 3.0 x16. When building a machine around either of these two cards, the motherboard should have the same number of PCIe 3.0 x16 slots as GPUs. The GPUs can be run in PCIe 2.0 slots or PCIe 3.0 x8, but performance will suffer due to the slower bus speed.

3.5.3 Storage

Picking the right storage configuration depends upon two major factors, dataset size and long term retention. If the machine needs to provide long term retention of research data, a RAID array is a good way to avoid data loss in the case of disk failure. RAID arrays can also be beneficial when dealing with large datasets as they can increase write and read speeds, depending upon the RAID type.

3.5.3.1 RAID

RAID or redundant array of independent disks is a data storage virtualization technology that utilizes multiple disks. There are a number of RAID levels, but the most

common are 0, 1, and 5. Figure 3.2 shows the layout for RAID levels 0, 1, and 5. Each of the levels is then discussed in the following sections.

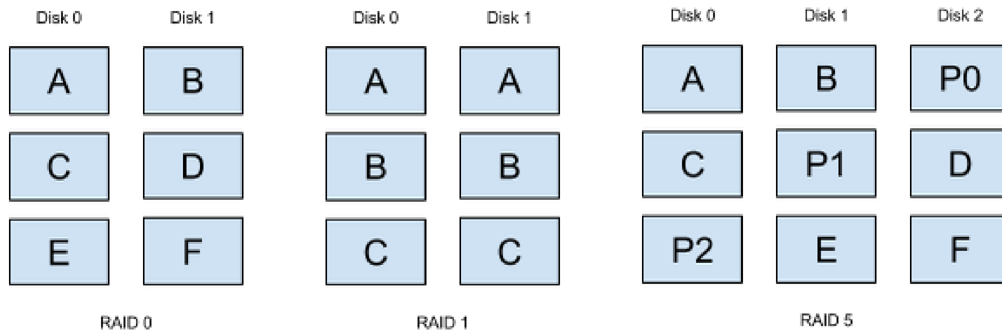

Figure 3.2: RAID Levels 0, 1, and 5

3.5.3.1.1 RAID 0

RAID 0 consists of striping data. Striping refers to distributing the data in roughly equal chunks across all the disks, allowing for faster read and write operations due to the ability to concurrently perform the operations. RAID 0 lacks mirroring or parity and thus results in utilizing the full capacity of all disks in the array. The tradeoff is a lack of redundancy or error correction. When using a RAID 0 array, the result of a disk failure is often an unrecoverable array and a total loss of data.

3.5.3.1.2 RAID 1

RAID 1 consists of mirroring. Mirroring refers to identical copies of data on separate disks. This requires at least two disks, but often more are used. RAID 1 offers high sustained read throughput as all disks can be read simultaneously. Write throughput suffers, however, as all disks must be updated.

3.5.3.1.3 RAID 5

RAID 5 consists of block-level striping with distributed parity. Block-level striping, as the name implies, performs striping at the block level, rather than at the bit-level, byte-level, or some other stride. Distributed parity utilizes parity across all disks, rather than having a dedicated disk for parity. The benefit of using distributed parity is that a RAID 5 array can lose a single disk and rebuild the lost information from the remaining disks. Using parity begins to limit the total capacity of usable disk space, but the tradeoff is disk failure tolerance. A RAID 5 array requires a minimum of three disks.

3.5.3.2 SSD vs. Hard Drive

Solid State Drives (SSDs) can be a consideration when selecting hard drives. SSDs have much higher sequential read and write speeds compared to typical hard drives. Previous work by Jason Monk showed no significant use of included SSDs for the associated research [15]. These machines were built to continue similar research and so SSDs were not used.

3.5.4 RAM

One important feature is the amount RAM supported by the motherboard. It's a good idea to get a motherboard that supports a large amount of RAM to give sufficient memory levels and to allow for future expansion should the need arise. When using large datasets, having a large amount of RAM can improve performance because of the faster access speeds of RAM vs a disk drive. As the price of RAM has dropped, the maximum amount of RAM a motherboard will support has continued to increase. Many high end motherboards now support 64GB of RAM or more. At a certain point, when building a GPGPU machine, the benefit of more RAM starts to drop off as most datasets aren't large

enough to utilize all of the available RAM and the GPUs simply don't have the same order of magnitude of RAM available.

3.5.5 Power Supply

Power supply selection depends mostly on the power requirements for the CPU and GPUs. Most of the rest of components have fairly low power consumption in comparison. A number of online power supply calculators exist. Most of calculators take into account the CPU(s), GPU(s), number of disk drives, motherboard, and some of the other peripherals. Newegg has a power supply calculator that can be used for a general idea of necessary wattage [16].

It's important to remember that the calculator gives only a general idea of power requirements and that the values returned should be taken as a minimum in most cases. It can be advantageous to make certain to have a large enough power supply for any future expansion as new GPUs become available, the system is expanded, and power requirements change.

3.5.5.1 Power Supply Rating

The rating system used for power supplies is the 80 PLUS system, a voluntary certification program that certifies efficiency greater than 80% at loads of 20%, 50%, and 100% with a true power factor of greater than or equal to 0.9. The levels of 80 PLUS certification are shown below in Table 3.5.

80 PLUS Certification	115V Internal Non-Redundant			
% of Rated Load	10%	20%	50%	100%
80 PLUS	---	80%	80%	80% / PFC .90
80 PLUS Bronze	---	82%	85% / PFC .90	82%
80 PLUS Silver	---	85%	88% / PFC .90	85%
80 PLUS Gold	---	87%	90% / PFC .90	87%
80 PLUS Platinum	---	90%	92% / PFC .95	89%
80 PLUS Titanium	90%	92% / PFC .95	94%	90%

Table 3.5: 80 PLUS Certification Ratings [17]

The higher levels of 80 PLUS certification are more desirable due to greater efficiency under load. When selecting a power supply, a rating of 80 PLUS Gold is often sufficent.

3.5.5.2 Modular

Modular power supplies are highly desirable as they allow for more efficient wire management by eliminating unnecessary cables. A fully modular power supply is best, but partially modular power supplies are more common and less expensive. Figure 3.3 shows the difference in cabling between a regular power supply and a fully modular power supply.

Figure 3.3: Modular and Non-Modular Power Supplies [18][19]

The main difference between fully modular and partially modular power supplies is that the 24-pin CPU power connector is removable for a fully modular power supply while it’s built in on the partially modular models.

3.5.6 Cooling

Cooling a computer can be done in one of two ways, air cooling or liquid cooling. Figure 3.4 shows two cooling systems installed in two different machines with the air cooling system on the right and liquid cooling system on the left.

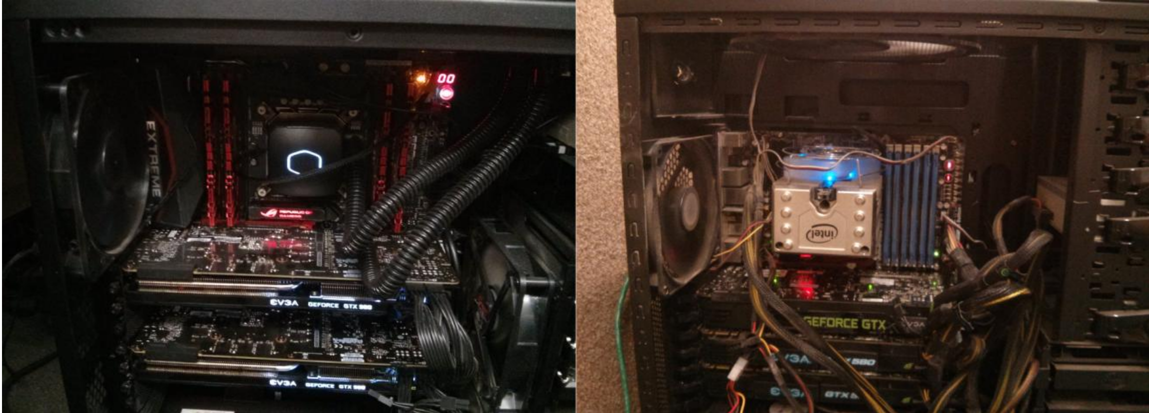

Figure 3.4: Liquid Cooling vs. Air Cooling

3.5.6.1 Air Cooling

The most common cooling technique is to use air and fans. The CPU has a heat sink and a fan, each GPU has a heat sink and fan built in, the power supply has a fan, and there are usually between two and six other fans in the case to help move air around. Usually some of the fans are configured to pull air in while others are configured to push air out. This creates a good flow that helps to keep all the components cool. Most processors come with a heat sink and fan, but oftentimes the extreme editions of the Core-i7 family are sold without the heat sink and fan.

3.5.6.2 Liquid Cooling

The alternative to air cooling is liquid cooling. There are several types of liquid cooling systems and the components that the system cools depends on how extensive the cooling system is. It's common to use liquid cooling for the CPU and air cooling for the GPU. There are systems that replace the air cooling on the GPU, but they aren't common and usually require a custom cooling system. Liquid cooling systems for the CPU are fairly common and maintenance free systems are easily purchased. These maintenance free system are completely sealed and don't require extra reservoirs or changing of the

cooling liquid. This makes them ideal drop in replacements for the CPU heat sink and fan. They also have the added benefit of being nearly silent. These systems have a small closed heat sink attached to the CPU with two hoses running to the radiator with a pump to keep the cooling liquid circulating. The radiator draws the heat from the circulating liquid and can often keep the CPU cooler than an air cooling system. Fans are mounted directly on the radiator to either push cool air into the radiator or draw the hot air away.

3.5.7 Case

Choosing a good liquid cooling system ties directly into choosing the case for the machine. It's important to make sure that there is enough room to mount the radiator and the fans either inside the case or on the back or top of the case.

The choice of case for the machine is fairly important, but is more about aesthetics than functionality. The main concerns when choosing a case are finding one that's compatible with the motherboard, large enough for the cooling system, has enough bays for the required number of drives, and has good routing for the wiring. The case should be the final component selected.

3.6 GPGPU Machine Configurations

Following the procedures and reasoning outlined in the preceding sections, two GPGPU machines were constructed. The difference between the two machines is that one has dual GTX 980 GPUs, machine A, while the other has dual GTX Titan Black GPUs, machine B. The selection of the two different GPUs was entirely dependent on the intended research for each machine. Machine A will primarily be used for research that heavily utilizes shared memory, for which the GTX 980 is better suited. Machine B will primarily be used for research with lower shared memory requirements and larger

datasets, for which the GTX Titan Black is better suited. Table 3.6 shows the configurations for the two GPGPU machines.

	Machine A	Machine B
CPU:	Intel Core i7-5930K	Intel Core i7-5930K
GPU:	2 x NVIDIA GTX 980	2 x NVIDIA GTX Titan Black
Memory:	32 GB DDR4 2133	32 GB DDR4 2133
Motherboard:	ASUS Rampage V Extreme	ASUS Rampage V Extreme
Storage:	4 x 2TB 7200 RPM HDD	5 x 2TB 7200 RPM HDD
Case:	Rosewill Throne-Window - Black	Rosewill Throne-Window - Black
Cooling:	Cooler Master Nepton 280L	Cooler Master Nepton 280L
Power Supply:	Corsair 1500W Fully Modular	Corsair 1500W Fully Modular

Table 3.6: GPGPU Machine Configurations

Appendix E contains the full configuration specifications for both machines.

3.7 Additional Considerations

Even with careful selection criteria, there was a compatibility issue between the Rosewill Throne case and the Nepton 280L liquid cooling system. The ASUS Rampage V Extreme motherboard didn't leave enough room at the top of the case for the fans to be mounted internally below the radiator. The top of the case is designed with plastic vents above the mesh of the case. It was necessary to modify the plastic vents to allow the fans to be mounted on top of the case. Figure 3.5 shows the top of the case after modification.

Figure 3.5: Case Modification

3.8 Summary

The GPGPU machine configuration criteria outlined in this chapter served as the basis for the construction of the two GPGPU machines outlined. The two machines were used for several of the tests contained in the remainder of this thesis. NVIDIA GPU's were selected for each machine to allow for the use of both CUDA and OpenCL progamming. The machines are identical with the exception of the GPUs. One machine contains two GTX 980s while the second machine contains two GTX Titan Blacks. Each machine has room and sufficient power for additional GPUs.

CHAPTER 4

CUDA PROGRAMMING

This chapter serves as an introduction to Compute Unified Device Architecture (CUDA) programming starting with basic terminology and syntax. Kernel writing and configuration are explored, followed by more advanced concepts including occupancy, memory management and data structures. The chapter concludes by outlining the compilation process and multiple GPU programming.

Effectively programming NVIDIA GPUs requires an understanding of the difference between a typical CPU architecture and a GPU architecture. GPUs are highly parallel systems with many ALUs controlled by less sophisticated control structures. Figure 4.1 shows the difference between a typical CPU with a small number of ALUs and a sophisticated control structure and large cache and a typical GPU with many ALUs per control structure and much smaller cache.

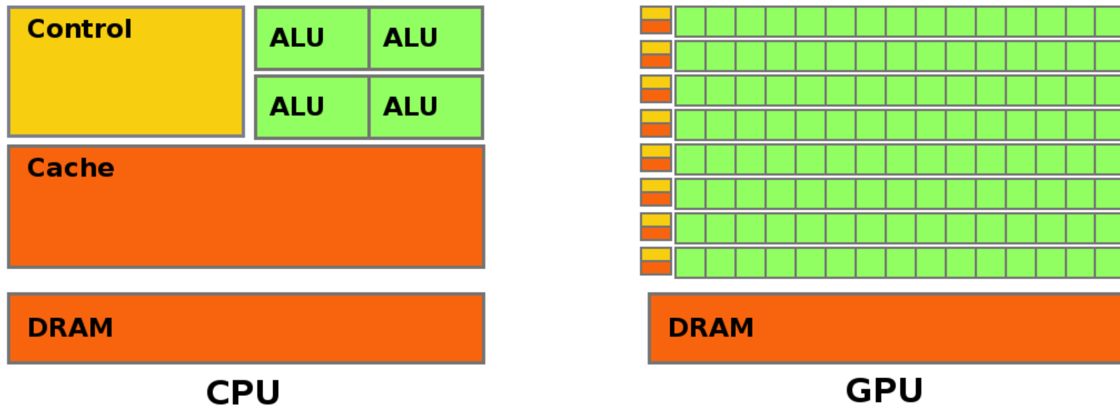

Figure 4.1: Architecture Comparison from CUDA C Programming Guide [2]

4.1 Terminology

In order to become familiar with GPGPU programming in CUDA, a number of terms must be defined.

4.1.1 Host and Device

In GPGPU programming, two important terms to understand are Host and Device. The host refers to the CPU running the the C/C++ CUDA program. The device refers to the GPU being utilized. It's possible to have multiple devices per host, allowing the programmer to take advantage of all available GPUs. The host and each device maintain separate memory spaces. This separation used to require explicit copy requests to move data between memory spaces. Since the introduction of Unified Memory, this is no longer the case. Memory management is discussed in more detail in Section 4.4.

4.1.2 Kernel

A Kernel refers to functions that run on a device rather than on the host. Kernels are launched from the host to run on a device with a specified number of CUDA threads that run in parallel. The number of threads is specified at runtime.

4.1.3 __global__ Keyword

The __global__ keyword is used to declare kernels that can be launched from the host and run on a device. The __global__ keyword is one of the most common keywords used in CUDA programming. Functions marked as __global__ are configured to specify the dimensionality of parallelism. This is discussed in more detail in Sections 4.2.2 and 4.2.3.

4.1.4 __device__ Keyword

The __device__ keyword is used to declare a function that can be called by CUDA threads. Functions marked as __device__ can be called in __global__ functions as well as by other __device__ functions. They can't be called directly from the host.

4.1.5 __host__ Keyword

The __host__ keyword is an optional keyword used to declare a traditional function that is called by the host and run on the host. All functions not specified otherwise are host functions. The majority of the time, the __host__ keyword isn't explicitly used, except to avoid duplicating functions that will be called on both the host and device. In these cases, both the __device__ and __host__ keywords are used to declare the function.

4.1.6 Threads, Blocks, and Grids

A kernel is run as a series of thread. The threads are organized into blocks that collectively make up the kernel grid. All threads within a block are guaranteed to run at the same time and can share memory, using shared memory, amongst themselves. There are restrictions on the number of threads in a block that are defined by the compute capability of the device. Section 3.2 contains more information about compute capability.

4.1.7 Warps

CUDA threads are grouped into warps. Warps have a Single Instruction Multiple Data (SIMD) architecture that results in a single instruction being run by all threads in the warp, all on separate data. The largest impact of this SIMD architecture is seen with divergent code in conditional statements such as if statements. If not all threads in a warp have the same path, such as when some threads execute a conditional and some do not, threads not executing the current instruction sit idle until the paths reconverge. For all compute capabilities above 2.0, the warp size is 32 threads.

4.1.8 Global and Shared Memory

The memory spaces for hosts and devices are completely separate. Devices have two major types of memory available, global memory and shared memory. Global memory is the largest amount of memory, GDDR5 on the newest models, and can be accessed by all threads running on a device. While global memory has the largest capacity, it also has the highest access latency. Shared memory, on the other hand, is tied to a specific block of threads and can only be accessed by the threads within that block. The memory access times for shared memory are much faster, but the capacity is highly diminished. The GTX 980 has 4GB of global memory while only 96KB of shared is available to a single multiprocessor, which must be split between up to 32 blocks [5][2].

4.2 Code

With the necessary terminology defined, the process of writing CUDA code can begin.

4.2.1 Internal Kernel Variables

When writing a kernel, a number of automatically defined dim3 variables are provided. These variables provide information related to thread location. Table 4.1 provides information about the automatically defined variables.

Variable	Description
dim3 gridDim	Dimensions of the grid.
dim3 blockDim	Dimensions of each block.
dim3 blockIdx	Current block index within the grid.
dim3 threadIdx	Current thread index with the block.

Table 4.1: Automatically Defined dim3 Variables

4.2.2 Kernel Configuration

Once a kernel has been written, it is configured at runtime to define the dimensionality of how it's executed on the GPU. The syntax used for launching a kernel is the triple chevron, <<<...>>>. The triple chevron syntax takes up to four parameters as shown in Figure 4.2 and explained in Table 4.2.

```
kernel<<<numBlocks, threadsPerBlock, sharedMemPerBlock, stream>>>(A, B,
    C, ...);
```

Figure 4.2: Triple Chevron Parameters and Syntax

Argument Name	**Description**
numBlocks	Number of blocks in the grid.
threadsPerBlock	Number of threads in each block.
sharedMemPerBlock	Amount of shared memory within each block. (optional, defaults to 0)
stream	Associated stream. (optional, defaults to 0, see Section 4.2.4)

Table 4.2: Triple Chevron Parameter Definitions [2]

The regular function parameters A, B, C, … are passed to each thread. Primitive types such int, float, char, etc are copied to device memory upon kernel launch. Pointer types such as int*, float*, char*, etc are also copied, but the memory that they point to is not. It is important to remember that when passing pointers in a kernel, the memory they map to must be device memory rather than host memory. Section 4.4 has a more detailed explanation of explicit memory management.

4.2.3 Dimensionality

CUDA has a number of useful features for multidimensional programming in up to three dimensions. Blocks of threads can have up to three dimensions, as can grids of

blocks. In order to define multiple dimensions, the dim3 structure is used. Figure 4.3 shows the dim3 structure available in CUDA.

```
struct dim3 {
    unsigned int x, y, z;
}
```

Figure 4.3: dim3 struct Variables

When using the triple chevron syntax as described in Section 4.2.2, the parameters for the number of blocks and the number of threads can be specified with the dim3 structure for up to three dimensions. When integers are specified in place of the dim3 structure, the value is used for the dimension in the X direction and the Y and Z dimensions default to one.

4.2.4 Streams

In CUDA, a stream is a sequence of operations performed sequentially. Streams are not limited to kernel execution, but can also include memory transfers. Streams are created using the cudaStreamCreate method. Figure 4.4 shows an example of creating two streams.

```
cudaStream_t streams[2];
for (int i = 0; i < 2; ++i) {
    cudaStreamCreate(&streams[i]);
}
```

Figure 4.4: Creation of Two CUDA Streams [2]

To use an initialized stream, the optional stream triple chevron parameter is specified. Streams are useful for issuing asynchronous memory transfers as any action

issued to the stream will wait for previously issued actions to complete before executing. Figure 4.5 shows an example of issuing a host to device memory transfer, followed by a kernel launch, followed by a device to host memory transfer using the example streams shown in Figure 4.4.

```
cudaMemcpyAsync(device_ptr, host_ptr, size, cudaMemcpyHostToDevice,
        stream[1]);
kernel1<<32, 1024, 0, stream[1]>>>(device_ptr);
cudaMemcpyAsync(host_ptr, device_ptr, size, cudaMemcpyDeviceToHost,
        stream[1]);
```

Figure 4.5: Using Streams for Asynchronous Memory Transfers

4.3 Occupancy

Global memory accesses in CUDA have the most latency and can have a large impact on performance for programs heavy utilizing global memory. CUDA attempts to hide this latency by warp switching if enough active warps exist. Warp switching occurs when a warp contains more idle threads than active ones. If a more active warp exists, the current warp is switched out and the more active warp begins to run. The concept of occupancy relates to how many active warps exist compared to the maximum number of warps. Equation 4.1 shows how occupancy can calculated.

$$Occupancy = \frac{ActiveWarps}{MaximumActiveWarps} \tag{4.1}$$

Full occupancy occurs when the number of active warps is equal to the maximum possible number of active warps. The number of active warps depends on a number of factors, including register usage, shared memory usage, and block size [20].

4.3.1 Occupancy Calculator Code

The CUDA toolkit has an occupancy calculator API that can be used to predict kernel occupancy based upon block size and shared memory usage. Figure 4.6 shows example code to calculate the occupancy of the kernel MyKernel.

```
// Device code
__global__ void MyKernel(int *d, int *a, int *b) {
        int idx = threadIdx.x + blockIdx.x * blockDim.x;
        d[idx] = a[idx] * b[idx];
}

// Host code
int main() {
        int numBlocks; // Occupancy in terms of active blocks
        int blockSize = 32; // These variables are used to convert occupancy to warps
        int device; cudaDeviceProp prop;
        int activeWarps;
        int maxWarps;
        cudaGetDevice(&device);
        cudaGetDeviceProperties(&prop, device);

        cudaOccupancyMaxActiveBlocksPerMultiprocessor( &numBlocks, MyKernel, blockSize, 0);
        activeWarps = numBlocks * blockSize / prop.warpSize;
        maxWarps = prop.maxThreadsPerMultiProcessor / prop.warpSize;
        std::cout << "Occupancy: " << (double)activeWarps / maxWarps * 100 << "%" << std::endl;
        return 0;
}
```

Figure 4.6: Calculating MyKernel Occupancy Programmatically[2]

4.3.2 Alternative Versions

The CUDA toolkit also provides a standalone version of the occupancy calculator located at <CUDA_Toolkit_Path>/include/cuda_occupancy.h. In addition to the occupancy calculator API and standalone version, a spreadsheet is also provided that can be used to calculate occupancy based on a number of factors including block size and register and shared memory usage [2].

4.4 Memory Management

Prior to CUDA 6.0, the only way to handle memory management was through explicit memory allocation and transfers between the independent memory spaces for host and device memory. CUDA 6.0 introduced the concept of unified memory as an alternative approach to memory management. Since memory transfers to and from the GPU often take much more time than the actual computation, memory management is an extremely important topic.

4.4.1 Explicit Memory Management

When using explicit memory management, several steps must be performed in order to use data on a device. First, device memory is allocated. Second, the desired data is copied to the device. Third, the necessary operations are performed on the device memory. Lastly, the results of the operations are copied back to the host. Figure 4.7 shows a typical CUDA program flow.

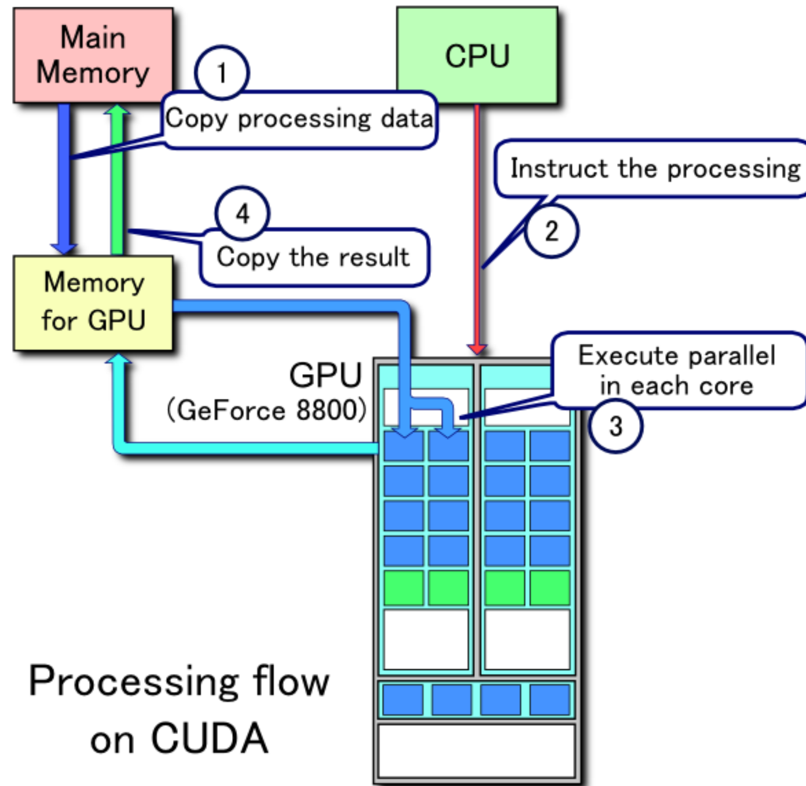

Figure 4.7: Typical CUDA Program Flow from CUDA Wikipedia Article [21]

Explicit memory management using the CUDA Runtime API is primarily based on the two functions cudaMalloc and cudaFree. These functions respectively allocate and free device memory. Figure 4.8 shows both function prototypes.

```
cudaError_t cudaMalloc(void** device_ptr, size_t size);
cudaError_t cudaFree(void* device_ptr);
```

Figure 4.8: cudaMalloc and cudaFree Function Prototypes

The return value type, cudaError_t, describes the function completion. CUDA error handling will be discussed in Section 4.6. Both cudaMalloc and cudaFree behave similarly to their conventional C language counterparts, malloc and free. The biggest difference between the CUDA and C allocation functions is that cudaMalloc takes a double pointer to device memory rather than single pointer. The double pointer points to

the address of the allocated memory in the case of successful allocation rather than returning a pointer to the allocated memory.

Copying memory between host memory space and device memory space is primarily handled by cudaMemcpy. The cudaMemcpy prototype is shown in Figure 4.9.

```
cudaError_t cudaMemcpy(void* dest, void* src, size_t count, enum
        cudaMemcpyKind kind);
```

Figure 4.9: cudaMemcpy Function Prototype

As with cudaMalloc and cudaFree, cudaMemcpy is very similar to its C language counterpart used on Unix systems, memcpy. The major difference is the addition of the kind parameter that specifies the copy domains. Table 4.3 shows the possible values of the kind parameter.

cudaMemcpyKind Value	**Memory Space Copy Direction**
cudaMemcpyHostToHost	Copies from Host to Host
cudaMemcpyHostToDevice	Copies from Host to Device
cudaMemcpyDeviceToHost	Copies from Device to Host
cudaMemcpyDeviceToDevice	Copies from Device to Device
cudaMemcpyDefault	Used for Unified Virtual Memory Space

Table 4.3: cudaMemcpyKind Values

Asynchronous memory copies are also possible using cudaMemcpyAsync. The additional cudaStream_t parameter is used with cudaMemcpyAsync. Streams are discussed in more detail in Section 4.2.4.

4.4.2 Unified Memory

CUDA 6.0 introduced the concept of managed unified memory. Unified memory allows the same pointer to reference memory on both the host and device without

explicitly copying memory using cudaMemcpy. The main advantage unified memory provides is the simplification of the program as host and device specific pointers and explicit memory copies are not required [2]. It's important to note that memory copies are still taking place, so no speedup should be expected when using unified memory versus explicit memory management. Unified memory is allocated using the function cudaMallocManaged, which takes the same arguments as cudaMalloc, outlined in Section 4.4.1. Figure 4.10 shows the prototype for cudaMallocManaged.

```
cudaError_t cudaMallocManaged(void** data_ptr, size_t size);
```

Figure 4.10: cudaMallocManaged Function Prototype

4.5 Data Structures

Associated with memory management is the use of appropriate data structures. For sequential CPU based applications the choice of data structures, as it relates to memory management, matters considerably less than it does for GPU applications. In sequential applications, often data can be stored in many different types of data structures without causing significant issues. This isn't the case for GPU applications as data will be accessed in parallel and memory coalescence (discussed next) becomes a primary concern.

4.5.1 Memory Coalescence

Global memory can only be accessed by 32, 64, or 128-byte memory transactions [2]. The effect of this memory access limitation is that extra memory is often included in memory transactions when accessing global memory. In order to take advantage of this access pattern, having data stored in sequential locations that will be accessed by all threads in a warp becomes important. Utilizing this access pattern to limit the number of

global memory accesses is known as memory coalescence. Effectively planning data structures and algorithms to maximize memory coalescence is an important factor in GPU programming.

4.5.2 Good Data Structures for GPUs

Data structure selection can play an important role in GPU programming. Arrays, and data structures such as vectors that are built upon arrays, are the preferred data structures for GPU programming due to fact that the elements of an array are held in sequential memory locations. This sequential nature offers the best option for memory coalescence.

4.5.3 Data Structures to Avoid

Data structure such as linked lists are poor choices for GPU programming due to the fact that adjacent elements are not stored in adjacent memory locations but referenced by pointers from the preceding elements. This causes linked lists to have poor memory coalescence and is the reason they should be avoided in GPU programming when possible.

4.5.4 Array of Structures versus Structure of Arrays

In a CPU based application, it's very common to have arrays of structures that hold information about individual objects. A good example of this would be a pixel in an image. A pixel structure would typically have values for red, blue, green, and alpha. For sequential access to pixels, it makes more sense to keep all the data related to a single pixel in a structure and create an array of structures to hold the data for an entire image. The SIMD architecture of the GPU means that instead of accessing each pixel one at a time and performing all of the operations on the associated data, multiple pixels values,

such as red values, are accessed at a single time and the operations performed on those values are performed in parallel. To achieve better memory coalescence, it is preferable to store the pixel data as a structure of arrays for each type of data. This results in separate arrays for the red, blue, green, and alpha values.

4.5.5 Thrust Library

When programming in C++, the Standard Template Library (STL) contains many useful data structures, such as vectors, that simplify memory management by encapsulating the necessary memory allocations for actions such as resizing. Unfortunately, the STL isn't available for use on the GPU. The CUDA toolkit does, however, contain the Thrust library that makes a number of the same containers available on the GPU.

4.6 Error Handling

Many CUDA API calls return a cudaError_t error type containing information regarding the call completion. cudaError_t is an enum type with many possible values. The large number of possible return values complicates error handling. Fortunately, the CUDA toolkit provides a good error checking macro for C++ called checkCudaErrors. The checkCudaErrors macro can be wrapped around CUDA calls to correctly handle and output error information. The checkCudaErrors macro is located in the <nvidia_sdk_samples>/common/inc/helper_cuda.h header file. Figure 4.11 shows an example of wrapping cudaMemcpy using checkCudaErrors.

```
checkCudaErrors(cudaMemcpy(device_ptr, host_ptr, size, cudaMemcpyHostToDevice));
```

Figure 4.11: Error Checking Using checkCudaErrors Macro

4.7 Multiple GPUs

One way to increase the amount of GPU processing is to utilize multiple GPUs. CUDA has a set of API calls to obtain information about available GPUs and to select between the available devices. Table 4.4 shows several useful functions for multiple GPU programming.

Function Prototype	**Description**
cudaError_t cudaGetDeviceCount(int *count);	Fills count with the number of available devices.
cudaError_t cudaGetDevice(int *device);	Fills device with a reference to the currently selected device.
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device);	Fills prop with the properties of the specified device.
cudaError_t cudaSetDevice(int device);	Sets the current device to the specified device.

Table 4.4: Multiple GPU CUDA Functions

The cudaDeviceProp struct contains a number of useful device properties including device name, total global memory, shared memory per block, registers per block, and warp size. This is by no means an exhaustive list of the properties contained in the cudaDeviceProp struct.

The use of cudaSetDevice sets all future kernel launches and memory allocations to use the last selected device. Each CPU thread can set its own device, allowing for multithreaded CPU applications where each thread leverages a single GPU. This can make using multiple GPUs much more manageable. Data can be directly transferred between GPUs using cudaMemcpy with the kind parameter set to cudaMemcpyDeviceToDevice.

4.8 CUDA SDK and Toolkit

The NVIDIA drivers provide the necessary libraries in order to run CUDA applications, however, the CUDA toolkit is required in order to compile CUDA source code. NVIDIA supports compilation for Windows, OS X, and a number of the most popular Linux distributions such as Fedora, Redhat, and Ubuntu. Other distributions, such as Arch Linux, provide user constructed toolkit installers. Arch Linux, for example, provides the cuda package from the community repository that contains both the CUDA SDK as well as the CUDA toolkit. Due to the open source nature of the Linux operating system and the large community that has grown around it, this thesis focuses on CUDA in Linux as opposed to Windows or OS X.

4.9 NVCC

The NVIDIA CUDA Compiler (NVCC) is used to compile device specific code, including host code containing kernel launches. NVCC compiles object files or executables for host code by forwarding the applicable code to a compatible underlying compiler, usually GCC or G++ [2].

4.9.1 PTX

Device specific code and host code containing kernel launches is compiled into Parallel Thread Execution (PTX) code. PTX code is usable by the device used at runtime. During compilation, the minimum compute capability of the devices that will be used should be specified. This allows the PTX code to use the best instruction set for a given device. The -arch flag is used to specify the compute capability and it is possible to compile for multiple compute capabilities simultaneously. Figure 4.12 shows an example of compiling for compute capabilities 2.0, 2.1, and 3.0.

```
nvcc x.cu \
--generate-code arch=compute_20,code=sm_20 \
--generate-code arch=compute_20,code=sm_21 \
--generate-code arch=compute_30,code=sm_30
```

Figure 4.12: Compiling for Multiple Compute Capabilities with nvcc [22]

4.9.2 Unsupported NVCC Compile Options

It is possible to compile host code that doesn't contain any device code directly with GCC or G++ and then link those files using NVCC. This can be useful when NVCC doesn't support necessary compile options. Alternatively, the -Xcompiler option can be used to pass compile options directly to the underlying compiler. Each option that must be passed to the underlying compiler must be preceded by the -Xcompiler option.

4.10 Summary

This chapter presented the basics of CUDA programming used for writing highly parallel programs to run on NVIDIA GPUs. The highly parallel nature of the GPU architecture was discussed showing how threads are grouped into warps of 32 threads that are guaranteed to run concurrently and organized into blocks that aren't guaranteed to run concurrently or in any particular order. Basic code structure including kernel writing, memory management, and error checking were explored, as were more advanced topics such as occupancy and multiple GPU integration. The chapter concluded by discussing program compilation.

CHAPTER 5

IMAGE PROCESSING GPU ACCELERATION

Image processing is inherently parallel as all actions in image processing are applied to either every pixel or groups of pixels and the outcome rarely has large internal dependency. Many of the simplest image processing techniques are embarrassingly parallel, such as inversion and certain smoothing algorithms. Embarrassingly parallel refers to the fact that splitting the operations for parallel execution requires little to no work. A key feature of embarrassingly parallel problems is that there are almost no interdependency issues when parallelizing the problem. The embarrassingly parallel nature of image processing makes it an excellent candidate for GPU acceleration and a perfect example of moving a highly parallel algorithm from the CPU to the GPU.

This chapter explores a simple image processing library using the ImageMagick MagickWand C API to read and write image files. Two filters, inversion and smoothing, are converted for GPU implementation and the inversion filter tested on several generations of NVIDIA GPU architectures.

5.1 ImageMagick

The image processing functions created utilize the ImageMagick C language API, MagickWand, in order to read in and write out the image files. These functions could be removed and replaced by custom image reading and writing libraries, but that would be an excellent example of reinventing the wheel unnecessarily. The ImageMagick API is capable of reading and or writing a very extensive list of image formats, over 200 to date [23]. Replacing the ability to work with all of these image formats would be no small

task, though supporting the most common image formats such as PNG and JPEG would be feasible, though outside the scope of the intention of this work.

5.1.1 MagickWand API

A small number of functions from the MagickWand API are necessary for image reading and writing. The process for reading and writing images follow very similar steps. When using the MagickWand API the environment must first be initialized. After initialization, a new MagickWand is created into which the image to be manipulated is read. Once the image is stored in a MagickWand object, the pixels are transferred to a custom image object and the associated MagickWand API objects are destroyed. After all operations on the image have been performed, the modified image can be written out using a similar process. First the environment is initialized and a blank MagickWand object is created to hold the modified image. Next, the pixels are transferred to the new MagickWand object and the image is written to a new image file. Finally, all MagickWand API objects are destroyed. Table 5.1 shows some helpful MagickWand API calls used in reading and writing images while Table 5.2 shows the MagickWand API objects useful for the same operations.

API Call	Description
MagickWandGenesis()	Initializes the MagickWand environment.
NewMagickWand()	Creates a new MagickWand object.
MagickReadImage(MagickWand* wand, const char* filename)	Reads in an image from a file to store in a MagickWand object.
MagickGetImageHeight(MagickWand* wand)	Returns the pixel height of a MagickWand object.
MagickGetImageWdith(MagickWand* wand)	Returns the pixel width of a MagickWand object.
NewPixelIterator(MagickWand* wand)	Used for iterating over pixels in a MagickWand object.
DestoryPixelIterator(PixelIterator* iterator)	Use for PixelIterator destruction.
DestroyMagickWand(MagickWand* wand)	Used for MagickWand destruction.
MagickWandTerminus()	Cleans up the MagickWand environment.
PixelSetRed/Green/Blue(PixelWand*, double value)	Sets the red/green/blue pixel value.
MagickWriteImage(MagickWand* wand)	Writes a MagickWand object image to an image file.

Table 5.1: MagickWand API Functions for Reading and Writing Images

Object Type	Description
MagickWand	Object that holds the image for modification.
PixelIterator	Object for iterating over the pixels in a MagickWand object.
PixelWand	Object to reference the actual pixels in a MagickWand object.
MagickBooleanType	Boolean return value for error checking.

Table 5.2: MagickWand API Objects for Reading and Writing Images

5.1.2 GPU Compilation

As explained in Section 4.9, to compile for the GPU rather than for the CPU, NVCC is used. In order to compile the ImageMagick libraries, certain flags must be passed to the compiler. The documentation for ImageMagick states that the necessary flags can be found using the command shown in Figure 5.1.

```
pkg-config --cflags --libs MagickWand
```

Figure 5.1: Command to Generate MagickWand C++ Compiler Flags [24]

Running the command shown in Figure 5.1 on an Arch Linux system produces the flags shown in Figure 5.2.

```
-fopenmp -DMAGICKCORE_HDRI_ENABLE=1 -
DMAGICKCORE_QUANTUM_DEPTH=16 -fopenmp -
DMAGICKCORE_HDRI_ENABLE=1 -
DMAGICKCORE_QUANTUM_DEPTH=16 -I/usr/include/ImageMagick-6 -
lMagickWand-6.Q16HDRI -lMagickCore-6.Q16HDRI
```

Figure 5.2: Arch Linux MagickWand C++ Compiler Flags

The nvcc compiler doesn’t recognize the -fopenmp flag, which must be passed directly to the gcc/g++ compiler. This is accomplished by adding the –Xcompiler option, discussed in Section 4.9.2, directly before the -fopenmp option. It is important to notice that the -fopenmp option is included twice from the output of the command from Figure 5.1 and thus the -Xcompiler option must be placed in front of each occurrence. This means that the easiest way to include the options shown in Figure 5.2 is to manually include them rather than using the backtick method of insertion suggested in the MagickWand documentation.

5.2 GPU Filters

A number of filters were designed for the CPU version of the image processing library. These filters offer processing tasks such as inversion, smoothing, binary conversion, dilation, erosion, and histogram equalization. Only the inversion and smoothing filters were adapted for use on the GPU as they are both excellent examples of embarrassingly parallel algorithms. The inversion filter inverts the pixel values for red, blue, and green by subtracting the current value from the maximum possible pixel value. The smoothing filter uses a square mask of a user supplied size, with only odd sizes between three and twenty-one available, to implement a uniform filter. The uniform filter uses the same value for all mask locations and averages over the entire mask area.

5.3 Timing

In order to compare the differences between GPU runtimes, timers were used for each section of relevant code as well as for the total time the process took. The subsections timed were image opening, setup, GPU runtime, image saving, and total runtime.

5.3.1 Timer Functions

The timer class used is a custom class that utilizes clock_gettime functions to act as a simple stopwatch. Table 5.3 shows the available functions and describes their use.

MonkTimer(const char* name);	Creates a new timer with the specified name.
start();	Starts the timer.
stop();	Stops the timer.
Display();	Prints the associated timer information.

Table 5.3: MonkTimer Functions

5.3.2 Timers

The opening timer records the time it takes to actually open the image using the MagickWand API and to move that data into custom arrays for processing. The setup timer looks at the time to set up the filter, including moving the image to and from the GPU, and the time required to apply the filter on the GPU. The GPU timer only records the time to apply the filter on the GPU. The saving timer records the time it takes to write the image to file, including moving the data back into the format used by the MagickWand API.

5.4 Image Conversion

The original image selected is an ultra-high resolution image from the Hubble Space Telescope with a resolution of 15852x12392 [25]. Smaller resolutions of the image were generated using the command shown in Figure 5.3, where X is the percentage of the original image resolution for the new image.

```
convert <inputImage> -resize X% <outputImage>
```

Figure 5.3: Example Image Convert Command

The reduction in image resolution began at ninety-five percent of the original image resolution and went down to five percent of the original resolution, with versions

created every at every five percent difference. Table 5.4 shows the image resolutions for each image produced.

Image Name	% Size	Resolution
images/hs-2006-10-a-full_jpg.jpg	100	15852x12392
images/hs-2006-10-a-full_95.jpg	95	15059x11772
images/hs-2006-10-a-full_90.jpg	90	14267x11153
images/hs-2006-10-a-full_85.jpg	85	13474x10533
images/hs-2006-10-a-full_80.jpg	80	12682x9914
images/hs-2006-10-a-full_75.jpg	75	11889x9294
images/hs-2006-10-a-full_70.jpg	70	11096x8674
images/hs-2006-10-a-full_65.jpg	65	10304x8055
images/hs-2006-10-a-full_60.jpg	60	9511x7435
images/hs-2006-10-a-full_55.jpg	55	8719x6816
images/hs-2006-10-a-full_50.jpg	50	7926x6196
images/hs-2006-10-a-full_45.jpg	45	7133x5576
images/hs-2006-10-a-full_40.jpg	40	6341x4957
images/hs-2006-10-a-full_35.jpg	35	5548x4337
images/hs-2006-10-a-full_30.jpg	30	4756x3718
images/hs-2006-10-a-full_25.jpg	25	3963x3098
images/hs-2006-10-a-full_20.jpg	20	3170x2478
images/hs-2006-10-a-full_15.jpg	15	2378x1859
images/hs-2006-10-a-full_10.jpg	10	1585x1239
images/hs-2006-10-a-full_5.jpg	5	793x620

Table 5.4: Generated Image Resolutions

5.5 Results

Table 5.5 shows the GPU timer values for applying inversion filters on the various image resolutions shown in Table 5.4.

GPU Timer					
Image Resolution	**GTX 580 #1**	**GTX 580 #2**	**GTX 680**	**GTX 980 #1**	**GTX 980 #2**
793x620	0.00447	0.00417	0.00477	0.00213	0.00217
1585x1239	0.01016	0.00993	0.01258	0.00451	0.00461
2378x1859	0.02074	0.02053	0.02937	0.00727	0.00774
3170x2478	0.03593	0.03552	0.05006	0.02070	0.02533
3963x3098	0.05635	0.05620	0.07558	0.04451	0.05386
4756x3718	0.08581	0.08551	0.11173	0.06848	0.08177
5548x4337	0.11442	0.11411	0.15051	0.08755	0.10681
6341x4957	0.14890	0.14859	0.20137	0.13074	0.15970
7133x5576	0.18138	0.18109	0.24994	0.17177	0.20912
7926x6196	0.22115	0.22092	0.30870	0.21266	0.25953
8719x6816	N/A	N/A	0.37558	0.25551	0.31061
9511x7435	N/A	N/A	0.44697	0.30280	0.36920
10304x8055	N/A	N/A	N/A	0.36121	0.44257
11096x8674	N/A	N/A	N/A	0.43466	0.52400
11889x9294	N/A	N/A	N/A	0.58338	0.62538
12682x9914	N/A	N/A	N/A	0.79818	0.81874
13474x10533	N/A	N/A	N/A	0.97888	N/A
14267x11153	N/A	N/A	N/A	N/A	N/A
15059x11772	N/A	N/A	N/A	N/A	N/A
15852x12392	N/A	N/A	N/A	N/A	N/A

Table 5.5: GPU Timer Values for Five GPUs

Table 5.6 shows the setup timer values for applying inversion filters on the various image resolutions shown in Table 5.4.

Setup Timer					
Image Resolution	**GTX 580 #1**	**GTX 580 #2**	**GTX 680**	**GTX 980 #1**	**GTX 980 #2**
793x620	0.06793	0.06646	0.06983	0.17005	0.15652
1585x1239	0.14771	0.14149	0.15169	0.20072	0.20169
2378x1859	0.28203	0.26742	0.29257	0.28463	0.27282
3170x2478	0.47098	0.44347	0.48668	0.40936	0.40518
3963x3098	0.71133	0.67344	0.73529	0.55955	0.54639
4756x3718	1.00294	0.93329	1.04171	0.75449	0.75243
5548x4337	1.35162	1.27654	1.38539	0.92489	1.00996
6341x4957	1.74896	1.61217	1.82514	1.28526	1.24104
7133x5576	2.19037	2.03059	2.27789	1.47909	1.53798
7926x6196	2.68658	2.53414	2.75602	1.79652	1.84085
8719x6816	N/A	N/A	3.38904	2.13404	2.23138
9511x7435	N/A	N/A	3.97476	2.52831	2.60918
10304x8055	N/A	N/A	N/A	2.91457	2.99937
11096x8674	N/A	N/A	N/A	3.40801	3.51061
11889x9294	N/A	N/A	N/A	3.96724	4.02279
12682x9914	N/A	N/A	N/A	4.65806	4.67136
13474x10533	N/A	N/A	N/A	5.27606	N/A
14267x11153	N/A	N/A	N/A	N/A	N/A
15059x11772	N/A	N/A	N/A	N/A	N/A
15852x12392	N/A	N/A	N/A	N/A	N/A

Table 5.6: Setup Timer Values for Five GPUs

Many of the images were too large to fit into device memory uncompressed and the library doesn't currently support splitting the image for processing. No timing data is available for those images. The values for each timer can be seen in Appendix F.

Figure 5.4 shows the graphical results of the values listed in Table 5.5.

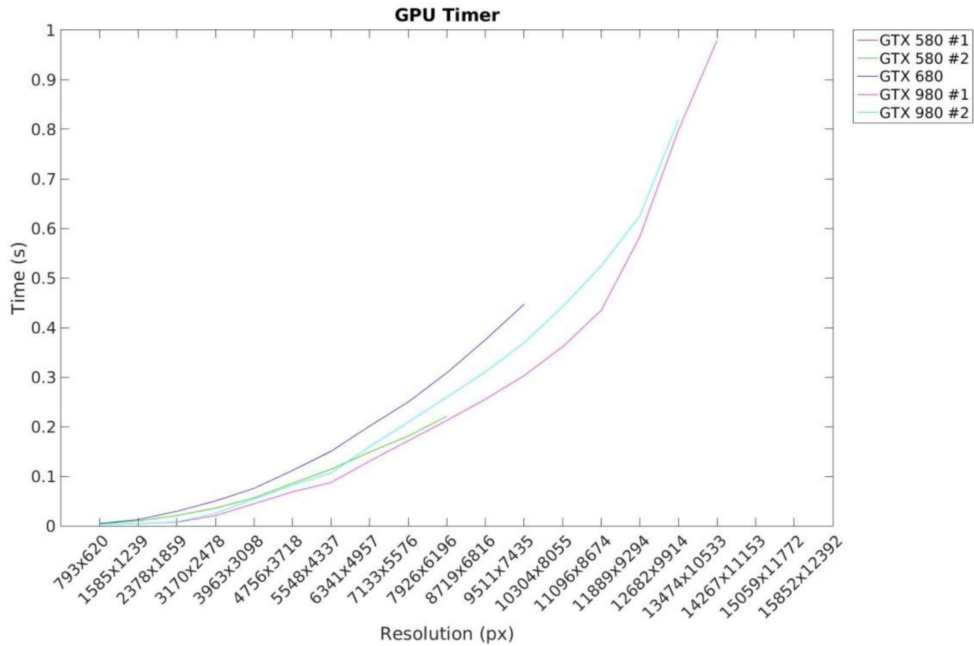

Figure 5.4: GPU Timer Comparison for GTX 580, GTX 680, GTX 980

The newer GTX 980s were able to perform the inversions on larger images due to the increased amount of device memory. The discrepancy between the two GTX 980s comes from the fact that one card was driving the display for the computer while the tests were being conducted. This is also the reason that one card was able to handle an additional image resolution.

Figure 5.5 shows the graphical results of the values listed in Table 5.6.

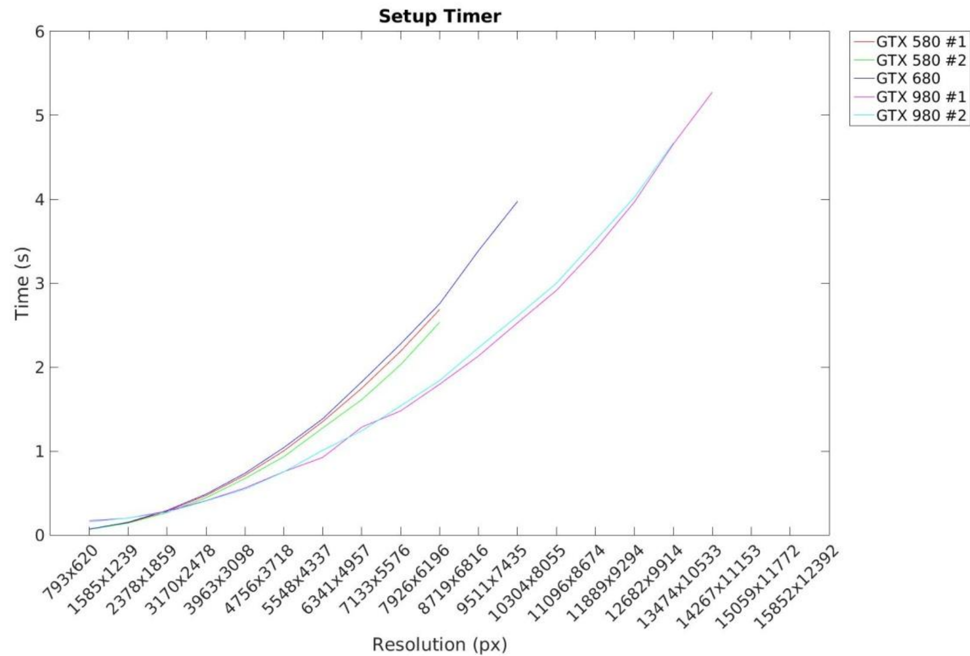

Figure 5.5: Setup Timer Comparison for GTX 580, GTX 680, GTX 980

The machines tested are the original GPGPU machine built by Jason Monk [15] and the new GPGPU machine with specifications outlined in Section 3.6. The comparison isn't necessarily apples to apples because of the differing computational capacities of the supporting hardware. However, useful information can still be gleaned by comparing the GPU runtime. The GPU runtime is a direct measure of how well the given GPU processes the image and doesn't rely on the supporting hardware. The setup time can also be useful as it relates directly to the transfer times to and from the GPU, though it remains important to remember that the differences in supporting hardware have an effect on these times and so they offer only general information. Note that the setup time is considerably longer than the computation time. In CUDA programming, this is often the case and reinforces the importance of proper memory management in order gain appreciable speedup.

As expected, the GTX 980s performed much better than the earlier architecture cards when comparing the GPU computation time.

5.6 Summary

This chapter covered the conversion of two simple image processing filters for use on NVIDIA GPUs using the ImageMagick MagickWand C API to read and write the image files. The inversion filter was explored on several generations of NVIDIA GPU architectures and the computational time was compared to show the evolution of CUDA processing power for an embarrassingly parallel problem.

CHAPTER 6

TETRIS GENETIC ALGORITHM AND PATHFINDING

As part of a project for a video game design class, a Tetris AI genetic algorithm utilizing A* pathfinding was designed using the Unity3d game engine. The AI was written in the C# programming language and was considered for GPU acceleration.

This chapter discusses genetic algorithms, including basic genome selection, individual evaluation and breeding, as well as mutation and elitism. The specific Tetris genome is then explored followed by A* pathfinding. The chapter concludes with an examination of the difficulties involved in moving to a GPU implementation as well as potential workarounds for the associated issues.

6.1 Genetic Algorithms

Genetic algorithms are biologically inspired heuristic search algorithms based on natural selection used to search large search spaces that would otherwise be difficult to traverse. Genetic algorithms start with an initial population of individuals who are often randomly generated. Each individual has a "genome" that defines selection characteristics. Each individual in the population is evaluated using a fitness function and given a score. This score is used to breed the current generation to create the next generation. Individuals with "better" scores have a higher likelihood of being chosen to breed and thus of passing on their genome, much like natural selection. As generations progress, individuals who are more fit pass on their genomes whilst individuals who are less fit die out and overall the entire population becomes more fit.

6.1.1 Genome

In genetic algorithms, the genome of an individual is the collection of selection criteria against which the individual is measured. The genome is made up of chromosomes that are usually represented by floating point values. Each chromosome correlates to a specific aspect of the problem of interest. This means that chromosome selection is problem dependent and thus difficult to generalize. Section 6.2.2 shows the chromosome selection for an actual problem.

6.1.2 Initial Population

The genome for the initial population is often generated randomly. This is one of the benefits of using genetic algorithms because the first generation doesn't require any specific knowledge to have the genetic algorithm work. The values can be chosen in other ways to improve the initial population, but it isn't necessary. In fact, seeding the initial population can be detrimental in the long run as the practice can cause sections of the search space to be eliminated, potentially missing "better" values. The size of the population can have an effect on the quality of the final solution. Up to a certain point, larger populations are generally better because there is a more diverse genome and a larger space is searched in fewer generations.

6.1.3 Evaluation

To evaluate individuals, a fitness function is created. The fitness function is problem dependant and picking a good fitness function is an important part of using genetic algorithms. Sometimes the fitness function will be obvious and is thus trivial to find. In other circumstances, there may be a number of ways to measure fitness and none may be obviously better than the others. Keep in mind that the fitness function is

attempting to measure how well a given genome solves a problem and that it's possible to choose a fitness function that breeds for a different problem than the intended problem. This represents a potential hazard when using biologically inspired search algorithms and is especially prevalent in neural networks. The output of the fitness function is the score for that genome.

6.1.4 Breeding

After all of the individuals have been evaluated, breeding occurs. In genetic algorithms, it is important to make sure that the most fit individuals have the highest chance of breeding. The most common way to ensure this is to use the roulette wheel selection method to pick individuals for breeding. Roulette wheel selection is discussed in more detail in the next section. Individuals are usually considered gender neutral, allowing any individual to breed with any other individual. They are often also allowed to breed with themselves, though this simply results in the individual being carried into the next generation.

6.1.4.1 Roulette Wheel Selection

Roulette wheel selection is a weighted selection method. The scores for all of the individuals are summed to get a total. The assumption when using this method is that the scores are all positive. Each individual is assigned a group of numbers between zero and the total. The size of the group of numbers is equal to the individual's score. To select individuals for breeding, random numbers are chosen between zero and the total and the resulting individuals are bred. The best way to visualize this is with an example.

6.1.4.2 Roulette Wheel Example

In this example, there are five individuals whose properties are shown in the below in Table 6.1. A pie chart, Figure 6.1, is often used to visualize such data.

ID	Score	Range
1	3	0-2
2	5	3-7
3	7	8-14
4	1	15
5	8	16-23

Table 6.1: Example Roulette Wheel Selection Ranges

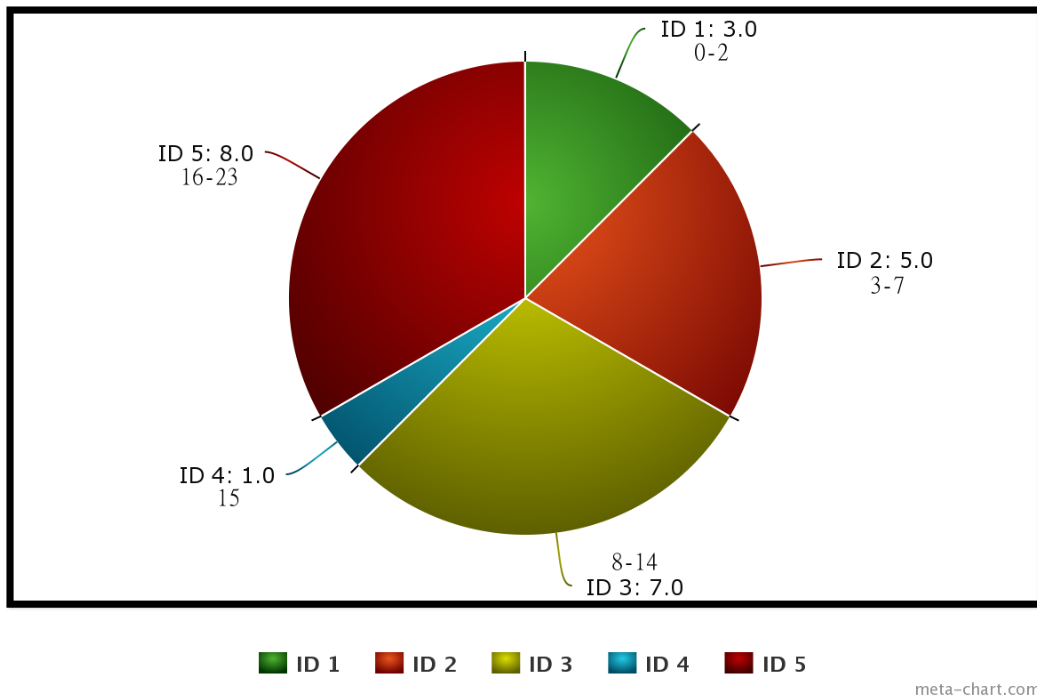

Figure 6.1: Example Roulette Wheel Selection Pie Chart

The total for this example is 24 and each individual has a range of values between 0 and 23. By choosing a random value from the set [0,24), any value chosen will fall into one of the ranges associated with a specific individual. The individuals with the largest

scores, individuals three and five in this case, have a higher chance of being chosen than those with lower scores. However, it's important to notice that while the likelihood of choosing a low scoring individual is low, it can still happen.

6.1.4.3 Crossover

After two parents have been selected, their genomes must be combined. This is achieved by selecting a crossover point in the genome and taking all of the chromosomes from one side of the crossover point from one parent and the rest from the other parent. The crossover point is often randomly chosen for each pairing. Sometimes two individuals are created from each pair of parents, rather than a single individual, by swapping the chromosomes up to the crossover point. Figure 6.2 shows an example of creating a single child from two parents using a crossover point after the second chromosome.

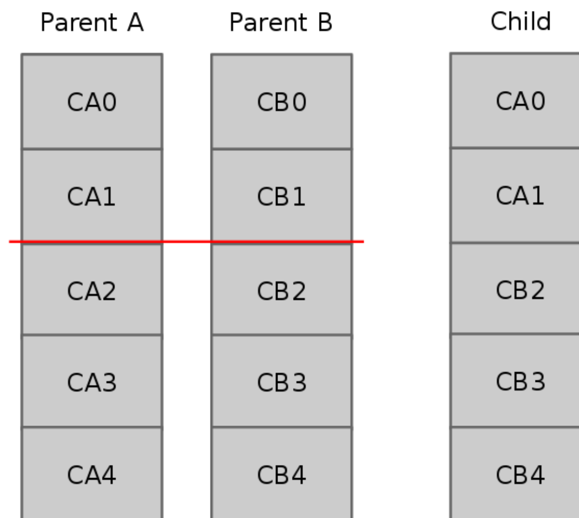

Figure 6.2: Example Breeding with Random Crossover

6.1.5 Mutation

In nature, chromosomes can mutate and add the potential for extra variation. The same principle applies in genetic algorithms. The mutation rate is used to occasionally

mutate chromosomes. This can often help spread the search area, moving beyond local maxima, and may help find better solutions. The mutation rate is often very small, especially for large populations.

6.1.6 Elitism

In genetic algorithms, elitism involves keeping some small portion of the most fit individuals alive into the next generation unchanged. This technique can be useful to ensure that good individuals aren't be lost by bad luck in the breeding process and crossover point selection. Elitism is usually limited to an extremely small selection of the population.

6.2 Tetris Genome

The genome used for the Tetris AI has six chromosomes, holes, blockades, row removal, max height, average height, max block height, and average block height. The description of each chromosome is expressed in Table 6.2.

Chromosomes	**Description**
Holes	number of holes created
Blockades	number of blockades created
Row Removal	number of rows removed
Max Height	maximum height of highest occupied space
Average Height	average height of all occupied spaces
Max Block Height	maximum height of current piece
Average Block Height	average height of current piece

Table 6.2: Tetris Chromosome Descriptions

In order to choose the best location for piece placement, each end location is given a score according to Equation 6.1.

score = hole chromosome * number of holes +

blockade chromosome * num of blockades +

row removal chromosome * number of rows removed + (6.1)

max height chromosome * max height +

average height chromosome * average height +

max block height chromosome * max block height +

average block height chromosome * average block height

6.2.1 Chromosomal Terminology

In order to discuss the selected chromosomes that make up the genome, two terms must be defined. The term hole refers to any empty space that is completely surrounded by blocks. A blockade refers to the number of occupied blocks above a hole. Figure 6.3 shows an example of a Tetris board to illustrate how each chromosome is calculated.

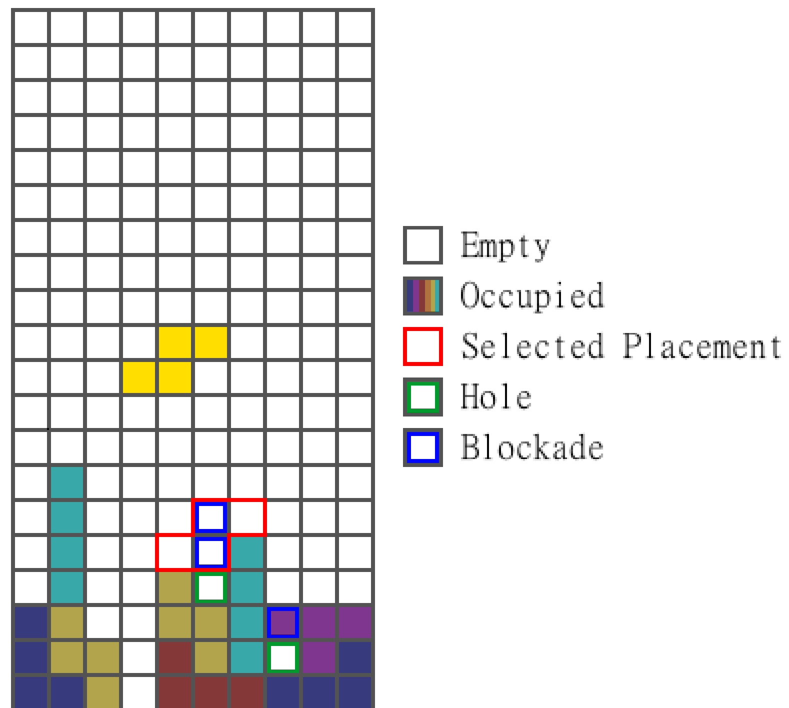

Figure 6.3: Example Tetris Board with Highlighted Scoring Criteria

Holes are outlined in green, blockades in blue, and the selected placement location in red. The calculated values for each of the chromosome are shown in Table 6.3.

Chromosomes	**Value**
Holes	2
Blockades	3
Row Removal	0
Max Height	7
Average Height	2.94
Max Block Height	6
Average Block Height	5.5

Table 6.3: Example Calculated Chromosome Values

Most the values in Table 6.3 can simply be read directly from Figure 6.2. The number of holes, blockades, rows removed, the max height, and the max block height can all be counted directly. The average block height and average height are merely calculated using the arithmetic mean.

6.2.2 Chromosome Selection

The first set of chromosomes chosen included holes, blockades, row removal, max height, and average height. This set of chromosomes failed to achieve good results. In an attempt to keep the height of placed pieces lower, the max block and average block height chromosomes were added. The addition of these chromosomes made a noticeable difference and allowed the resulting individuals to place many more pieces and remove many more lines before reaching the top of the board.

6.2.3 Additional Considerations

In order to ensure that the genetic algorithm was training for the desired outcome, all individuals were given the same sequence of pieces generated at the beginning of each new generation. Two different approaches to this were considered, testing all generations on the same sequence and testing each generation on several distinct sequences and averaging the scores. The first approach was discarded as it had a high potential of resulting in training for a single sequence and not an arbitrary one. The second approach was implemented and training was done for three sequences per generation. The end condition for training was set such that an individual that could place 500 pieces without breaking the board. This was achieved in a reasonably small number of generations, once in as few as the second generation simply by good luck on the initial population

6.3 Tetris Piece Placement with A* Pathfinding

A* Pathfinding is the process of selecting the shortest path from a starting location to an ending location, often without prior knowledge of the landscape. Pathfinding algorithms are a form of graph traversal. A* pathfinding breaks up the traversal area into small regions that are either walkable or not walkable. Examples of real world non-walkable regions include areas such as walls, steep slopes, and water.

6.3.1 Costs

Each region has three associated cost values, the G, H, and F costs. The G and H costs are discussed in next two sections. The F Cost is simply the combination of the H and G costs. The F cost is used to find the next region.

6.3.1.1 G Cost

The G cost defines how far a given region is from the starting region. Certain types of movement require different G costs. Generally, moving horizontally or vertically has a single G cost while moving diagonally has a greater G cost because the actual movement distance is greater.

6.3.1.2 H Cost

Each region also has an H cost, known as the heuristic cost. The heuristic cost is defined by a heuristic that estimates how far the region is from the end region. This is only an estimation because to know the exact distance, the path would have to be known, which is the point of A* pathfinding. Any number of obstacles can be in the way of the path estimated by the heuristic making the path longer than a straight path. The main point of the H cost is that it cannot overestimate the distance. A common simple heuristic used is the Manhattan method. The Manhattan method looks at the necessary

movement vertically and the necessary movement horizontally, sums them, and calculates the cost based on those values.

6.3.2 A* Basics

With the traversal area divided up into regions, the starting region is added to an open list as are all regions immediately adjacent to the starting region. Each of the adjacent regions has the starting region as its parent. From the open list, the region with the smallest F cost is found and is used as the next region. That region is removed from the open list and moved to the closed list. All of its adjacent regions are added to the open list. If one of the adjacent regions already exists in the open list and has a lower G cost coming from the current region than from the previous region, its costs are updated and its parent is changed from the previous region to the current region. This process continues until the end region exists in the closed list. From there, the path is found by simply following the end region backwards from its parent until the starting region is reached.

6.3.3 Tetris Specific Implementation

Using the basic A* pathfinding information outlined in Section 6.3.2, a Tetris specific A* pathfinding algorithm was designed. There are some differences from the basic algorithm, including what regions are applicable and how the costs were selected.

6.3.3.1 Regions

A region is defined as any space the current piece will fit into in a given rotation. Each rotation at a given location is its own region. However, these regions don't all have the same parent. The initial rotation, the one that is the same as the previous regions rotation, has the previous region as a parent. Each rotation in the new location has the

previous rotation as its parent and is only added to the open list if the previous rotation was also added. It is also important to note that movement is limited to either left, right, or down. Movement up is not an option. Diagonal movement isn't a direct option as diagonal movement is achieved through a combination of vertical and horizontal movement. Given the differences for movement, there are a maximum of twelve adjacent regions if all rotations are viable, four rotations for each of the three available movements, left, right, and down.

6.3.3.2 Costs

Costs were given round values, but are otherwise completely arbitrary. The three types of costs are horizontal, vertical, and rotational. 2:1 cost ratios were used for horizontal to rotational and vertical to rotational movement. Rotational movement has a lower cost in an attempt to encourage rotational movement before vertical or horizontal movement. The current costs are set to ten for each horizontal and vertical movement and five for rotational movement. These costs work well enough, though there may be better cost ratios. It's worth noting that it is the cost ratios and not the actual cost values that matter.

6.4 C# CPU Results

The C# algorithm was run a number of times with varying numbers of generations. Table 6.4 shows three of the best individuals over the longest run, 40 generations, starting with the best individual labeled as individual A, along with their associated chromosome values.

Individuals	A	B	C
Generation	35	17	40
Hole	-4.914	-3.303	-4.914
Rows Removed	1.4331	1.4331	1.252
Blockade	-1.484	-1.484	0.363
Max Height	-3.235	-3.235	-3.235
Average Height	0.893	0.893	0.3768
Max Block Height	-1.569	-0.5643	-1.569
Average Block Height	-1.042	-1.042	-1.042

Table 6.4: Best C# CPU AI Individuals Produced

Most of the chromosomes values are fairly close, if not exactly the same. The low variation in values suggests that the piece placement scoring function and the fitness function are both working. The fact that individuals in widely spaced generations share the same values indicates that the piece placement scoring function may not be well tuned beyond a certain point and a local maximum of the search space has been reached. It is possible that the search space is rough with a number of local maxima that produce reasonable results. All three of the listed individuals were able to place 500 pieces before breaking the field, but it should be possible to do better with a better piece placement scoring function.

6.5 Additional Considerations

It is important to note the differences between normal Tetris and the version tested. In a normal game of Tetris, the scoring mechanism is built into the removal of rows and could be directly mapped to the fitness function. Also, there is a speed element in normal Tetris that causes the pieces fall faster as the levels increase. The version

implemented doesn't take either of these into consideration as the actual game the AI was developed for is only based upon Tetris. The other main difference is that the piece set is slightly increased. Figure 6.4 shows the pieces in a standard game of Tetris.

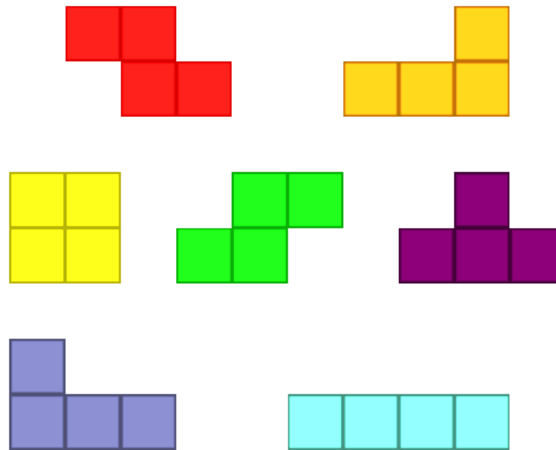

Figure 6.4: Normal Tetris Pieces

Additional pieces were added as part of the game mechanic and these additional pieces are shown in Figure 6.5.

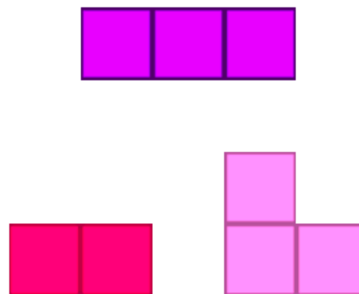

Figure 6.5: Additional Tetris Pieces

6.6 GPU Implementation

Genetic algorithms are often good candidates for GPU acceleration due to the fact that the majority of the computation is highly parallelizable. The same process is applied to each member of the population and intermediate generations are usually not saved, if only because they aren't needed except for record keeping. Given this highly parallel

nature, the Tetris AI genetic algorithm seemed like an excellent candidate for GPU acceleration. The original intent was to have each thread perform the necessary calculations and pathfinding for a single individual, thus allowing for very large populations.

6.6.1 C# to C++

With a working model written in C#, the next step was to convert the necessary portions of the program to run in C++. Only after a working C++ version was implemented could a GPU accelerated version could be explored. The conversion from C# to C++ is non-trivial, partially due to the fact that C++ requires explicit memory management while C# has a garbage collector that, expect in the case of unmanaged resources such as files, network connections, database connections, etc, handles memory management [26]. While explicit memory management isn't overly complicated, it can be extremely tedious and improper management often results in segmentation faults and memory leaks.

6.6.2 Issues

During the conversion from C# to C++, it started to become obvious that the assumption that this genetic algorithm was a good candidate for GPU acceleration might be false. The open and closed lists used in the A* pathfinding algorithm are vectors in C# that are dynamically resized as objects are added and removed. C++ has a Standard Template Library (STL) vector that works in a similar fashion, but cannot be used on the GPU. For small vectors, simply using a predefined sized array would be sufficient, but the open and closed lists can get quite large and local and shared memory constraints become a concern. The open lists ranges in size anywhere from the low teens to upwards

of two hundred objects. The closed list ranges in size anywhere from low teens upwards of four hundred objects. These large ranges make implementing dynamic lists on the GPU much more difficult.

Due to time constraints, these memory concerns ended development of the GPU accelerated version of the program until a workaround could be devised.

6.6.3 Potential Solutions

Three possible solutions to the memory concerns have been identified, but none have yet been implemented.

6.6.3.1 Global Memory Solution

The first solution is the simplest and uses global memory to store all open and closed lists for each individual. This solution should work for reasonably sized populations, though it has the potential to be considerably slower than the second solution due to utilizing global memory rather than shared memory.

6.6.3.2 Thrust Library Solution

The second solution involves using the Thrust library provided as part of the CUDA toolkit. The reason the Thrust library wasn't used in the first place is that it was believed that dynamic allocation on a device was not possible. The Thrust vector documentation implies this is not the case, though verifying this will require testing. If the Thrust vector can be dynamically modified, a shared memory Thrust vector, or even global memory Thrust vector, could provide a good option for open and closed lists in the A* pathfinding algorithm, though size may still become an issue as the lists have the potential to be larger than the available shared memory. Since A* pathfinding is type of graph problem, a third solution exists.

6.6.3.3 Warp Pathfinding Solution

The third solution would be to use a single warp of threads to do the pathfinding for each individual with the open and closed lists in either shared memory, if possible, or global memory. Bleiweiss et al. showed that A* pathfinding was capable of running on the GPU [27] and so this approach may well be the most effective as the pathfinding should be quickly solved by multiple threads. Keep in mind though that the population size may suffer for the same reasons outlined in the first two solutions.

6.7 Summary

This chapter discussed genetic algorithms in general and a Tetris AI genetic algorithm with A* pathfinding specifically. A GPU implementation of the Tetris AI genetic algorithm was explored as well as the issues associated with the adaptation. Potential workarounds for the GPU specific issues were also covered.

CHAPTER 7

CHILD

The Channel-Hillslope Integrated Landscape Development Model (CHILD) is a landscape evolution model developed by the Department of Civil and Environmental Engineering at the Massachusetts Institute of Technology [28]. CHILD is written in C++ and calculates topographical evolution over time. Many types of models can be simulated using CHILD's numerous options.

Previous work showed that CHILD was a good candidate for multithreading and indicated potential for GPU parallelization. Profiling from that work indicated the functions taking the largest percentages of time, tLNode::EroDep and tBedErodePwrLow::DetachCapacity [15].

This chapter explores the process of creating MATLAB executables to access custom C/C++ functions, followed by the creation of a MATLAB interface for CHILD. The chapter concludes by exploring potential GPU acceleration of CHILD and the underlying issues associate therewith.

7.1 MATLAB Integration

CHILD is written in C++ and doesn't currently have MATLAB integration. Within the research team, interest in tighter MATLAB integration has been discussed. The current method has been to use a system call within MATLAB to launch CHILD. An alternative to this is to compile CHILD as a MEX application so that it can be launched directly from within MATLAB.

7.1.1 MEX

MEX stands for MATLAB executable and allows custom C, C++, and FORTRAN functions to be called directly from within MATLAB. The output is a MEX-file, a shared library that MATLAB can run directly.

7.1.2 MEX Modifications

At least one modification must be made to any function that will be compiled into, or accessed by, a MEX-file. Any exit() call must be replaced with a call to mexErrMsgTxt(const char *c). This is done to avoid MATLAB exiting when an exit() call is reached. MATLAB doesn't create new processes when a MEX function is called and so any exit() call issues from a MEX function will exit MATLAB rather than the intended function. The mexErrMsgTxt(const char *c) passes the error message pointed to by the const char *c back to MATLAB and exits the MEX program as intended, returning control to MATLAB. Figure 7.1 shows an example of wrapping an exit() call for use within a MEX-file.

```
#ifdef MATLAB
    const char* c = "";
    mexErrMsgTxt(c);
#endif
    exit(EXIT_FAILURE);
```

Figure 7.1: Wrapping exit() Functions for MEX Compilation

A number of CHILD's source files required modification for wrapping exit() calls. Modifications following the approach outlined in Figure 7.1 were made to the files listed in Table 7.1.

Files Requiring MATLAB Specific Modifications
errors.cpp
TipperTriangulatorError.cp
tOption.cpp
tStratGrid.cpp
meander.cpp
tStreamNet.cpp

Table 7.1: CHILD Source Files Requiring Modification for MEX Compilation

Each of the files shown in Table 7.1 also require that the file mex.h be added as an #include to access the mexErrMsgTxt(const char *c) function call. In order to allow compilation of both regular CHILD and CHILD with a MATLAB interface, the #define MATLAB constant was included for all MATLAB specific calls, including the #include for mex.h. MEX compilation defines the MATLAB constant on the command line.

7.1.3 MATLAB to MEX Interface

When creating MEX-files, the mexFunction() is used as an interface between MATLAB and the MEX-file. The four mexFunction parameters are described in Table 7.2.

Argument	Description
int nlhs	Number of mxArray pointers in plhs
mxArray* plhs	Left hand side mxArray pointers passed in from MATLAB
int nrhs	Number of mxArray pointer in prhs
const mxArray* prhs	Right hand side mxArray pointers to return to MATLAB

Table 7.2: mexFunction() Parameters

Since CHILD doesn't need to return any values to MATLAB, both nrhs and prhs can be ignored. It is important to note that plhs should never be modified as this can cause undesirable side effects [29]. Using this function as a stepping stone, it would be possible to both pass data into and out of CHILD for greater MATLAB integration between models.

7.1.4 MEX Compilation

In order to compile a MEX-file, a MATLAB compiler is used. In Linux, this compiler is called mex. To compile CHILD, no extra mex specific options were necessary, though the compilation was a three step process. The first step of the process was to compile CHILD outside of MATLAB to generate the correct object files. CHILD compilation uses the make program to compile the source code and defaults to using the G++ compiler. Figure 7.2 shows the command used to compile CHILD on a Linux system from the main CHILD directory.

```
make -f Code/childir.mk
```

Figure 7.2: Command to Compile CHILD from CHILD Main Directory

Once the full CHILD source has been compiled, the necessary object files should have been created in their respective subfolder within the CMakeFiles/child-shared.dir directory. With the majority of the CHILD object files created, the modified files from

Table 7.1 must have object files generated with the MATLAB constant passed from the command line. From the matlabInterface/objects directory, this compilation is achieved as shown in Figure 7.3.

```
mex -c ../../errors/errors.cpp ../../tMesh/TipperTriangulatorError.cpp
../../tOption/tOption.cpp ../../tStratGrid/tStratGrid.cpp
../../tStreamMeander/meander.cpp ../../tStreamNet/tStreamNet.cpp
CFLAGS="-fPIC -DMATLAB"
```

Figure 7.3: Command to Compile CHILD matlabInterface Object Files

The final step is to link all the necessary object files together to create the MEX-file. The second and third steps are both achieved by running the matlabCompile.sh bash script from the matlabInterface directory. The full file can be found in Appendix J.

7.2 GPU Implementation

Knowing that CHILD was potentially suitable for GPU acceleration [15], a more detailed analysis of the code was undertaken to determine if the EroDep and or DetachCapacity functions could be adapted for the GPU. At first glance, both functions appear to be good candidates for acceleration due to little interdependency and the fact that each function is called for each individual node. This design is perfect for implementing on a GPU; however, further examination reveals significant issues.

Two major issues prevent CHILD in its current form from being a good candidate for GPU acceleration. The underlying data structures used throughout CHILD are problematic as are the large number of transfers to and from the GPU.

7.2.1 Underlying Data Structures

All nodes in CHILD are set up as a linked list, rather than in an array or vector. Linked lists are great for certain types of data management, specifically adding or removing elements. In a linked list, the link between each element is described by a pointer rather than having elements located at sequential memory locations. To add or remove an element, the pointer from the previous element is simply redirected. To insert an element, the previous element's pointer is directed to the new element and the new element's pointer is directed to the next element. To remove an element, the previous element's pointer is directed to what becomes the new next element. Figure 7.4 shows the insertion and removal operations in a linked list.

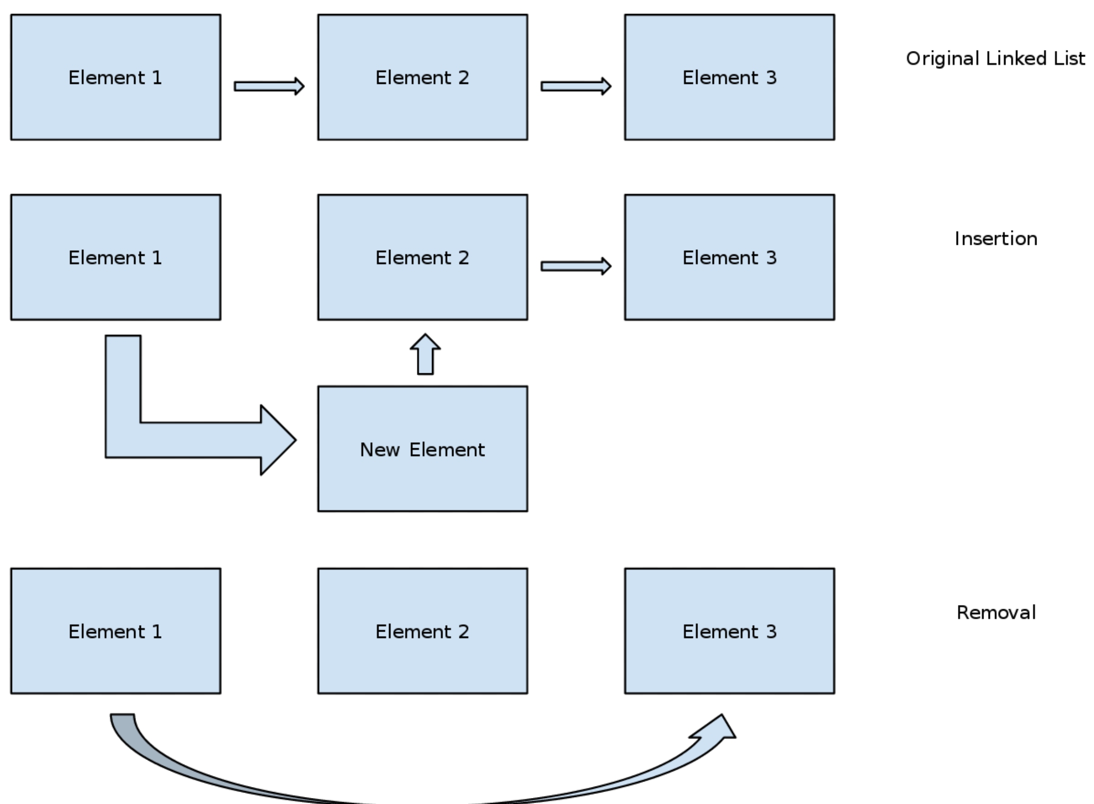

Figure 7.4: Linked List Insertion and Removal

When a pointer to the element in question already exists, the operation is O(1), though traversing the list to an element is O(n). As outlined in Section 4.5.2, the best data structure for GPGPU computation is an array. Linked lists are used heavily throughout the CHILD code base and in order to effectively move to the GPU, all instances where linked list data would be accessed on the GPU would have to be modified to use an array or vector. Rewriting the underlying data structure of CHILD is outside of the scope of what a single person could manage in any sort of reasonable timeframe since CHILD has some 100,000+ lines of source code.

7.2.2 Memory Transfers

The second major issue to consider is that for each timestep both EroDep and DetachCapacity are called on all nodes, but those calculations must be pushed to and from the GPU at each timestep. As is discussed and illustrated in Section 5.5 the data transfer times to and from the GPU are often much higher than the computation times. In order to make the best use of the GPU and to see any sort of meaningful speedup, minimal data transfers should be implemented compared to the amount of computation. This simply wouldn't be the case in CHILD unless more than EroDep and DetachCapacity functions could be moved to the GPU or both functions could be run on the GPU without moving data back to the CPU. Taking these limitations into account, CHILD is not a good candidate for GPU acceleration without extensive modification to the underlying data structures and perhaps not even then.

7.3 Summary

This chapter discussed generating MATLAB executables from C/C++ functions. The steps necessary to create a MATLAB interface for CHILD were also explored,

including a three step compilation process. The issues associated with potential GPU acceleration of CHILD were also discussed.

CHAPTER 8

PARALLEL SPATIAL SORTING ALGORITHM

The algorithm outlined in this chapter is intended as a replacement for a spatial sorting algorithm for models where objects move through three dimensional space. The original concept for this algorithm came from the intent to replace a radix sort in a smooth particle hydrodynamics (SPH) implementation [30]. The new implementation was intended to model landscape evolution. The original implementation used SPH to model avalanche flow, which is closely related to landscape evolution.

This chapter will explore a parallel spatial sorting algorithm and the steps required to implement the algorithm on NVIDIA GPUs. The potential for multiple GPU scalability will be discussed. The chapter will conclude with a discussion of potential applications of the algorithm.

8.1 Related Work

Satish et al. showed the design of a CUDA based parallel radix sort algorithm with high efficiency. This approach was able to achieve a 4x speedup compared to GPUSort [31].

Domínguez et al. evaluated cell-linked and Verlet lists to create neighbor lists for use in smoothed particle hydrodynamics. They proposed an approach utilizing dynamic Verlet list updating for neighbor searches [32].

Gonnet et al. described a parallel hierarchical cell decomposition approach to neighbor finding in smoothed particle hydrodynamics. This approach achieved a 40x speedup over the Gadget-2 simulation code [3].

8.2 Algorithm Explanation

Consider an array of objects in global memory that have positions in three dimensional space and move within the confines of a defined three dimensional rectangular prism. Further, consider the blocks in a grid on a GPU as representing rectangular prisms in three dimensional space. Each block is made of up a group of threads as discussed in Section 4.1.7. Figure 8.1 shows a visual representation of the grid of blocks.

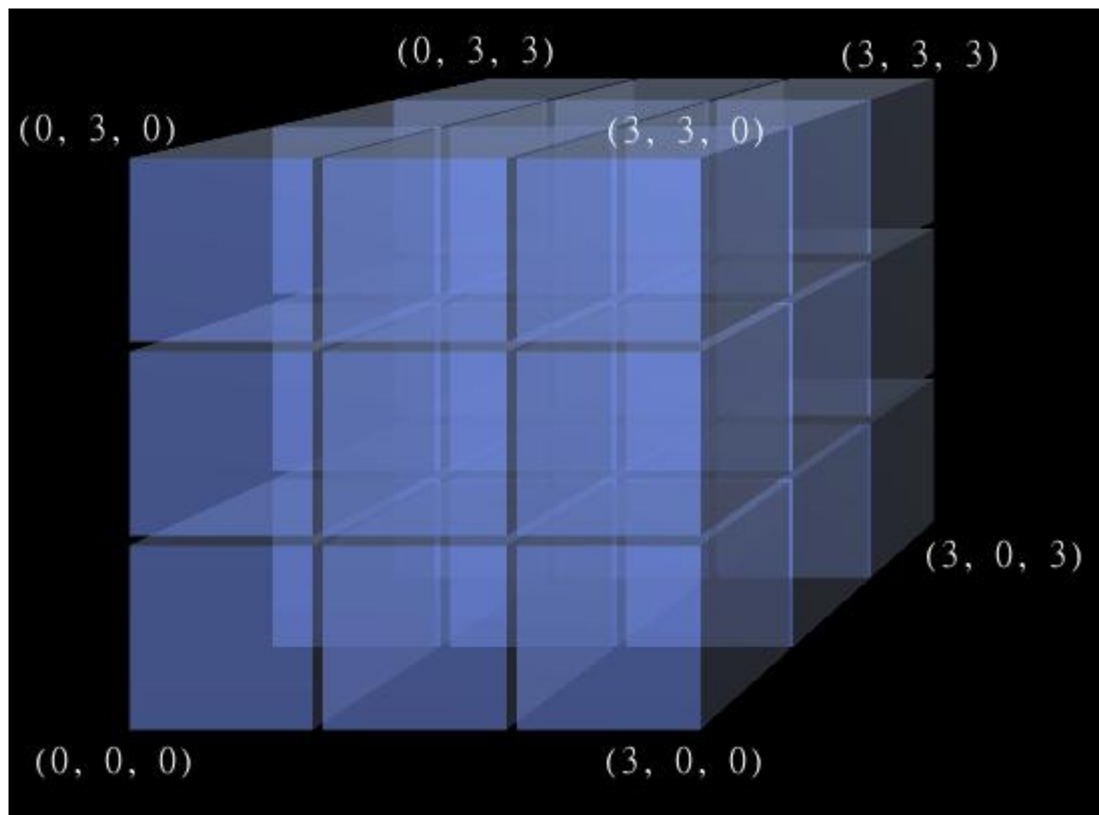

Figure 8.1: Grid of Blocks with Coordinates

The defined three dimensional space of the grid of blocks in Figure 8.1 is three meters long by three meter wide by three meters tall. Each block encompasses one cubic meter.

Each thread in a block represents a "hole" that can either be filled by an object making the thread active or is empty making the thread inactive. Objects move through three dimensional space and cross the boundaries from one block into another as time progresses. Each block may contain static information related the three dimensional space it represents. Blocks map directly to sections of the object array in global memory. Figure 8.2 shows how four blocks, each with six threads, map to an example global memory object array.

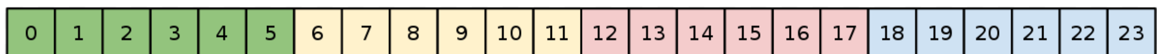

Figure 8.2: Example Block to Global Memory Object Array Mapping

8.3 Algorithm Steps

In order to perform the movement actions, the following steps should be followed. Note that step ten is optional for most timesteps, but required after the last timestep to return the objects to the host.

1. Initialize the global memory and fill it with active and inactive objects such that the objects are grouped by their three dimensional positions in the appropriate blocks. This may result in having a number of inactive objects in the array within each block.
2. Each block pulls the section of the global memory array it is associated with into a shared memory array.
3. Perform the calculations to determine how the objects move and, if applicable, interact with each other. The results of these calculations overwrite the values in the shared memory array.

4. Push the new values from the shared memory array back to the global memory array. At this point, each object's position has been updated, but objects have not yet moved from their original blocks.
5. Each block counts the number of inactive objects currently located within that block.
6. Each block creates a second shared memory array to hold the objects that will be moved neighboring blocks into their section of the global memory array.
7. Each block searches the global memory array for objects in adjacent blocks that will be moved into it, copies the objects to the shared memory array, and marks each object to be moved in the global memory array as inactive. There may be multiple blocks reading the same array values, but simultaneous reads from global memory do not cause race conditions the way that simultaneous writes would. It is important to make certain that there are enough inactive objects currently in the array to hold the objects to be moved. This is where the values from step five are utilized.
8. Each block combines the new objects in the second shared memory array together with the objects in the first shared memory array, using inactive objects to hold the new objects. As long as there are enough inactive objects to hold the new objects, no issues arise. At this point, the objects have been updated with new positions.
9. The first shared memory array, now holding all of the objects, is pushed back to the global memory array. All objects have now been placed in the appropriate

locations in the global memory array associated with the block their position is located within.

10. The global memory array is transferred back to the host for further processing or writing to disk. This step is optional and only performed after all timesteps have been taken or if a snapshot of the current state of the objects is desired.
11. If more timesteps must still be performed, repeat the cycle starting at Step 2.

8.4 Caveats

Some of steps in the algorithm have caveats associated with them that need to be explored. Possible solutions for each caveat are discussed when available.

8.4.1 Caveat 1

Mapping a block to three dimensional space is potentially non-trivial. In the simplest case, the grid is comprised of blocks of equal size and as long as the grid and the number of blocks are chosen appropriately, no issues arise. The major issue comes when some blocks might be different sizes. The layout of a grid with blocks of varying size represents a problem that requires the user to specify how each block maps to three dimensional space and requires extra memory, most likely in global memory, to hold the necessary block and neighbor mappings.

8.4.2 Caveat 2

The second shared memory array used at step six needs to be an additional array, separate from the array that holds the objects pulled from global memory. At this point, the amount of shared memory starts to come into play and a GPU with a high enough compute capability needs to be used.

8.4.3 Caveat 3

Searching adjacent blocks in step seven requires searching up to twenty-seven adjacent blocks when the movement in a single timestep of any object remains less than single block distance. If it's possible for an object to move farther than one full block, the number of blocks to search becomes much larger and may encompass the entire grid. The increased number of blocks to search increases the likelihood of running out of space in a block for more objects.

8.4.4 Caveat 4

Combining the new objects with the current objects in step eight is more complicated than it sounds. A potential approach to this would be use a single thread to move all new objects from the second shared memory array into the first shared memory array. This approach will take longer, but will make certain that no objects are overwritten.

8.5 Example

The steps outlined in Section 8.3 can be confusing to visualize without an example. A simple example should suffice for clarification. This example contains two blocks, each with eight threads. Four active objects start in each block at random locations. Figure 8.3 shows the grid layout of the blocks, with three dimensional spatial coordinates at each corner, associated color coded shared memory arrays for each block, and the active objects located within each block.

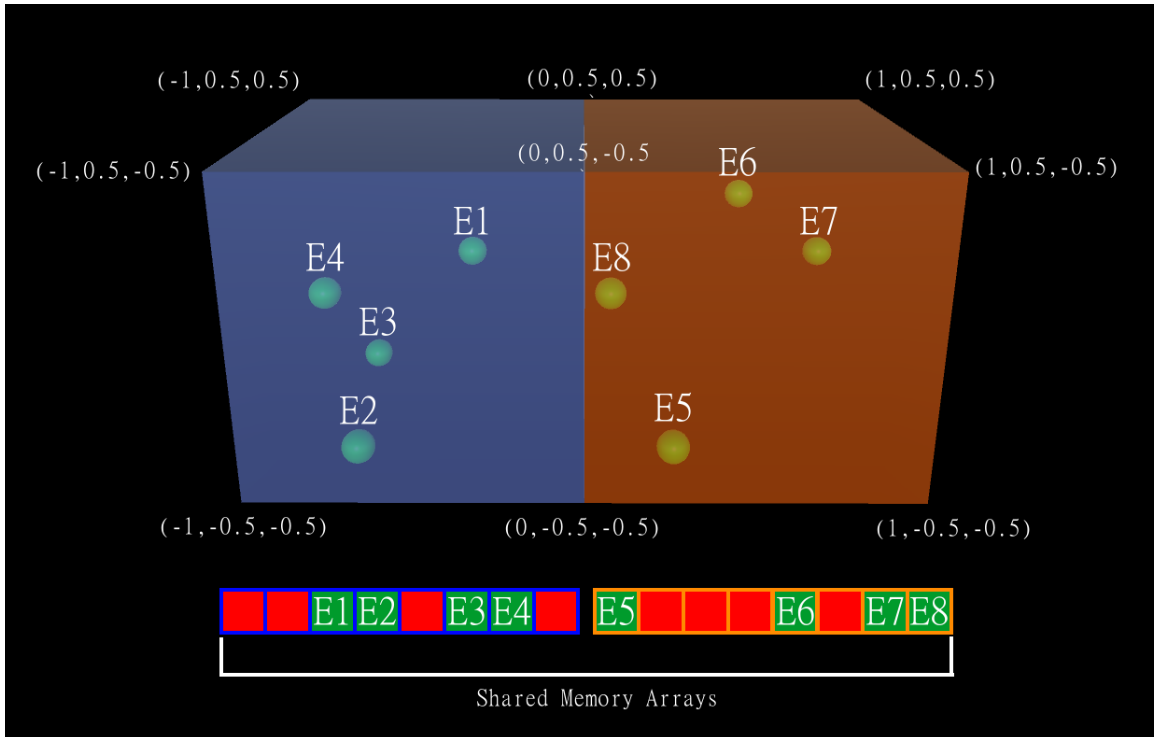

Figure 8.3: Example Grid of Blocks Layout and Shared Memory Timestep 0

Figure 8.4 shows the global memory array color coded to match the blocks in Figure 8.3. The active objects are highlighted in green. Notice that the global memory array is exactly same as the combination of the two shared memory arrays because each block pulls the corresponding sections of global memory into shared memory.

Figure 8.4: Example Global Memory with Active Objects

In the first timestep, the active objects move to new locations within the same block in three dimensional space. When the objects move and don't leave their current block, only their own position data must change. Therefore, when the shared memory arrays are pushed back to update the global memory array, it still looks exactly the same as that shown in Figure 8.4.

In the first timestep, no active objects move outside of their current block. Figure 8.5 shows the same grid layout of the blocks as Figure 8.3, with the updated active objects locations after the first timestep.

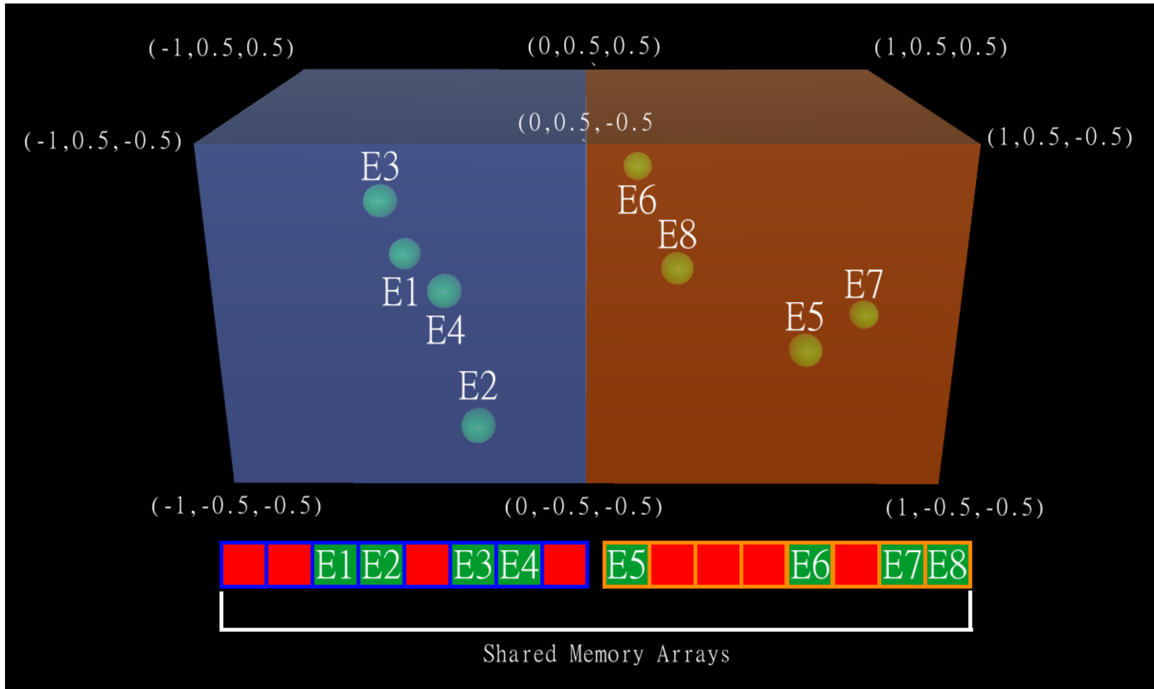

Figure 8.5: Example Grid of Blocks Layout and Shared Memory Timestep 1

In the second timestep, some of the active objects move from one block to the other. Object E2 moves from the blue block to the orange block and object E6 moves from the orange block to the blue block. Figure 8.6 shows the grid layout of blocks after the second timestep.

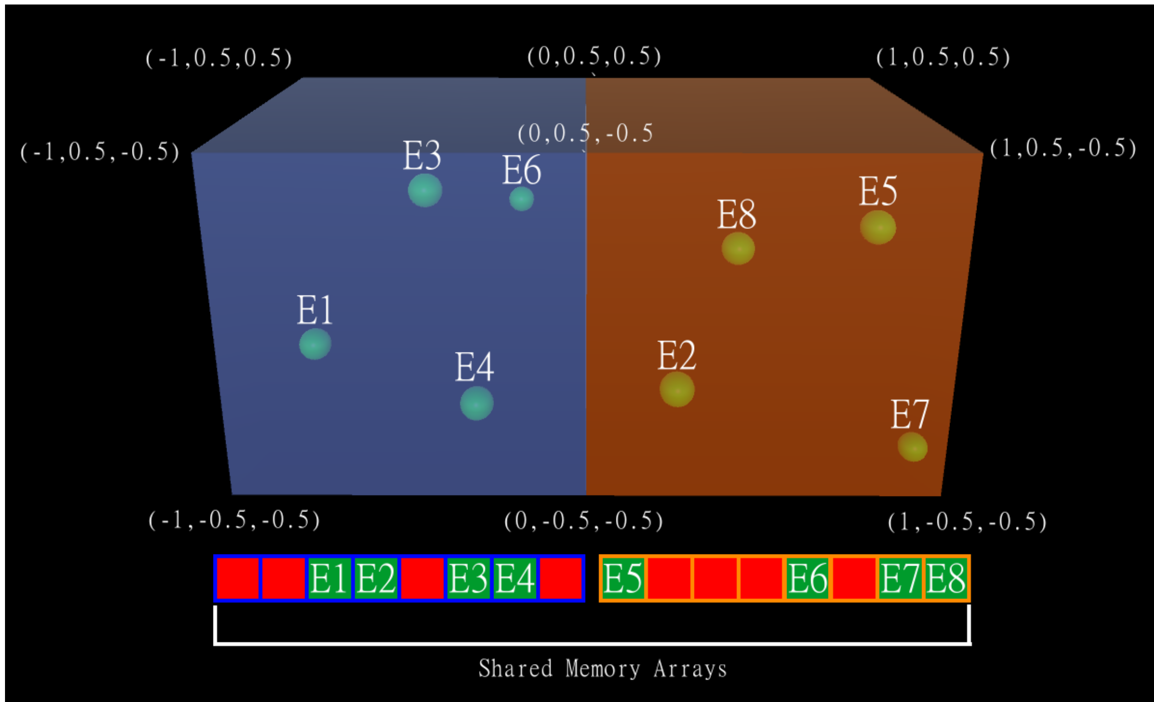

Figure 8.6: Example Grid of Blocks Layout and Shared Memory Timestep 2

At this point, the global memory array still looks exactly the same as shown in Figure 8.4.

Before the third timestep occurs, the shared memory arrays are updated to reflect object movement. This step takes place before every timestep, but until now hasn't had any effect. Each block searches the other block for any objects that have moved. In this case object E2 is moving into the orange block and object E6 is moving into the blue block. Figure 8.7 shows the first shared memory arrays after they have been updated.

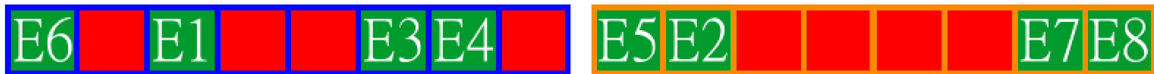

Figure 8.7: Example Shared Memory Just Prior to Timestep 3

The values shown in Figure 8.7 are used in the third timestep, before which the global memory array is updated as shown in Figure 8.8.

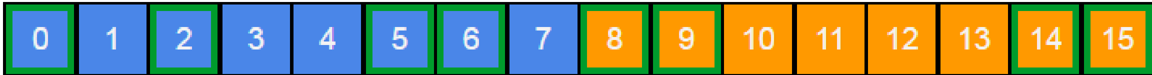

Figure 8.8: Example Global Memory Array During Timestep 3

This simple example shows how a small number of active object move between blocks and should suffice as a starting point to scale up the problem domain to include more blocks, threads, and active objects.

8.6 Scalability

The spatial sorting algorithm outlined shows promise for scaling to multiple GPUs. The inherent ability of the algorithm to split the spatial domain makes it an excellent candidate for multiple GPU parallelism. The theoretical maximum number of inter-GPU transfers occurs when the individual GPU grids are arranged such that the inner GPUs have a maximum of twenty-seven neighbors. Figure 8.9 shows an example GPU grid arrangement with twenty-eight GPUs with the center GPU highlighted. The surrounding GPU grids are the twenty-seven possible neighbors.

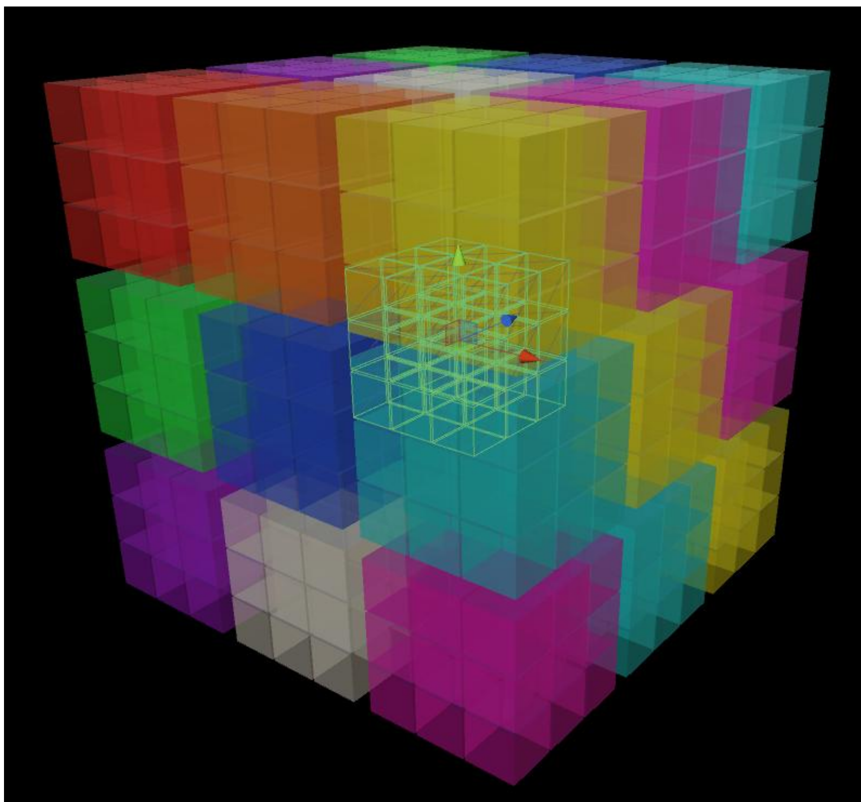

Figure 8.9: Grid of GPU Grids

8.7 Dimensionality

The intent for this algorithm is as a replacement for sorting algorithms in three dimensional space. The algorithm cannot handle models in higher dimensions, and the concept of objects moving through higher dimensions doesn't make sense for modeling the physical world at the macro scale. However, the algorithm should work just as well in lower dimensions, though finding reasons to model object movement in a single dimension might prove difficult.

8.8 Potential Applications

Two applicable applications for this spatial sorting algorithm include SPH and agent based modeling.

8.8.1 SPH

SPH or Smoothed Particle Hydrodynamics is a particle based mesh-free Lagrangian method for modeling fluid flows. SPH was originally developed in 1977 by Gingold and Monaghan for use with astrophysics and is often used to model fluid dynamics [33].

In SPH, each particle has several associated parameters, including, but not limited to, position, velocity, density, and smoothing length. The smoothing length defines the spherical radius around a particle in which all other particles therein contribute influence. One of the strengths of SPH is that the smoothing length need not be constant across all particles, though this makes finding neighboring particles more difficult. Approaches to sorting for locating neighbors include cell-linked or Verlet lists, or more efficient variants thereof [32], and radix sort [34]. The parallel spatial sorting algorithm outlined in this chapter provides a potential replacement for these and other neighbor sorting algorithms.

8.8.2 Agent Based Modeling

Agent Based Modeling (ABM) involves modeling a system using a collection of autonomous agents. The behavior of the agents is defined by a set of basic rules. The interaction between the agents and between the agents and the environment in the system can be used to study emergent phenomena. Since the agents in ABMs interact with each other related to their spatial locality, the parallel spatial sorting algorithm outlined in this chapter again provides a potential replacement for other neighbor finding algorithms.

8.9 Summary

This chapter outlined an algorithm for a parallel spatial sorting algorithm for NVIDIA GPUs. The basic algorithm is discussed, including the steps necessary to

implement the algorithm on a GPU. A simple example was explored, followed by multiple GPU scalability of the algorithm. Two potential applications into which the algorithm fits are also discussed.

CHAPTER 9

FUTURE WORK

The models and methods outlined in this thesis show the potential for GPU acceleration in large and small scale modeling. While these models provide a basis for GPU acceleration, a number of opportunities are left for exploration, several of which are discussed in this chapter.

9.1 Visualization Wall

The current setup for the IRL visualization wall is fully maintainable and running the latest kernel. However, it would be beneficial to replace the operating system with a more stable, long term Linux distribution if possible as actively maintaining Arch Linux is a time consuming process. An alternative to choosing a different Linux distribution would be to utilize the Arch Linux LTS kernel for added stability.

9.2 Testis Genetic Algorithm and Pathfinding

The opportunity still exists to implement a GPU accelerated Tetris AI genetic algorithm using A* pathfinding. One of the three suggested solutions to the memory management of the open and closed lists outlined in Section 6.6.3 should allow progress to continue. Other areas of consideration for future work include implementing a chromosome for lookahead and finding a better scoring function for piece placement. Currently the upcoming pieces, including the next piece, are completely ignored. Taking the next piece into consideration has the potential to improve placement considerably. Improving the piece placement scoring function, Equation 6.1, could allow for better genomes to be produced, perhaps in fewer generations.

9.3 Image Processing

The CPU version of the image processing library contains several more filters that have not yet been adapted for GPU acceleration. None of the filters utilize algorithms that present issues for GPU adaptation. The library is also an excellent candidate to explore unified memory more thoroughly due to the embarrassingly parallel nature of the filters. Another possible avenue for exploration includes the use of multiple GPUs. As discussed in Section 5.5, a number of the images produced were too large for the device memory and a multiple GPU implementation would allow for those larger images to be processed. In addition to implementing multiple GPU support, it would be beneficial to implement the ability to split the images for processing so that large images could be processed on a single GPU.

9.4 CHILD

While the underlying data structures used in CHILD make it a poor candidate for GPU acceleration, greater MATLAB integration remains promising. Continuing development of the MATLAB to MEX interface by adding the ability to pass MATLAB arrays directly into CHILD and return arrays to MATLAB could greatly improve MATLAB integration. Using similar tactics for other models would allow multiple models to interface through MATLAB.

9.5 Parallel Spatial Sorting Algorithm

The parallel spatial sorting algorithm outlined has yet to be implemented on a GPU, but looks to be a good candidate, especially for multiple GPU acceleration. The two modeling methods outlined in Sections 8.8.1 and 8.8.2 are promising possibilities for

testing the algorithm, though any model requiring spatial sorting could benefit from the use of the algorithm, especially on multiple GPUs.

While all reference to multiple GPU acceleration within this thesis have focused on GPUs within the same machine, the parallel spatial sorting algorithm has the best opportunity to leverage a large number of GPUs across multiple machines, such as seen within a supercomputer. The University of Maine supercomputer recently purchased several new NVIDIA Tesla GPUs that would be an excellent test bed for this algorithm.

CHAPTER 10

CONCLUSION

GPGPUs offer significant computational power for programmers to leverage. This computational power is especially useful when utilized for accelerating scientific models. The construction of hardware for visualization and computation of scientific models was discussed in this thesis, as was a basic understand of GPGPU programming to utilize said hardware.

This thesis discussed several models that were examined for GPU acceleration. Each model offered a new perspective on the benefits of, and issues associated with, GPU acceleration. The image processing library showed how to recognize embarrassingly parallel problems and served as an excellent example of converting from a serial CPU implementation to a GPU accelerated implementation. The Tetris genetic algorithm with A* pathfinding discussed memory bound limitations that can prevent direct algorithm conversions from the CPU to the GPU. Analyzing CHILD for GPU acceleration showed that even when a model shows promise for GPU acceleration, the underlying data structures, as well as the necessity to transfer data back to the CPU too often, can have a significant impact upon that ability to move to a GPU implementation. Integrating CHILD more closely into MATLAB by creating a MEX executable provided a path towards tighter integration between scientific models with MATLAB as a common access point. Lastly, the parallel spatial sorting algorithm discussed is a possible replacement for current spatial sorting algorithms implemented in models such as smoothed particle hydrodynamics and shows promise for scalability to multiple GPUs.

REFERENCES

[1] M. Harris, "History of GPGPU." http://gpgpu.org/oldsite/data/history.shtml, 2013. [Online; Accessed 15-July-2015].

[2] NVIDIA, CUDA C Programming Guide. NVIDIA, March 2015.

[3] Pedro Gonnet, Efficient and Scalable Algorithms for Smoothed Particle Hydrodynamics on Hybrid Shared/Distributed-Memory Architectures, CoRR, abs/1404.2303 (2014).

[4] TOP500 List, "CHINA'S TIANHE-2 SUPERCOMPUTER MAINTAINS TOP SPOT ON LIST OF WORLD'S TOP SUPERCOMPUTERS," http://www.top500.org/blog/lists/2015/06/press-release/, [Online; Accessed 26-July-2015].

[5] NVIDIA, "GeForce GTX 980," http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications, 2015. [Online; Accessed 16-July-2015].

[6] "Beginner's Guide," https://wiki.archlinux.org/index.php/beginners%27_guide, 2014. [Online; Accessed 13-August-2014].

[7] D. Dawes, "xorg.conf - Configuration File for Xorg," http://ftp.x.org/pub/X11R6.8.0/doc/xorg.conf.5.html, 2014. [Online; Accessed 13-August-2014].

[8] "ATI," https://wiki.archlinux.org/index.php/ATI#Dual_Head_setup, 2014. [Online; Accessed 13-August-2014].

[9] J. Peddie, "Add-in board market down in Q4, Nvidia increases market share lead," http://jonpeddie.com/press-releases/details/add-in-board-market-down-in-q4-nvidia-increases-market-share-lead, 2015. [Online; Accessed 16-July-2015].

[10] PassMark, "AMD vs Intel Market Share," https://www.cpubenchmark.net/market_share.html, 2015. [Online; Accessed 26-July-2015].

[11] Intel, "Intel® Core™ i7-5930K Processor (15M Cache, up to 3.70 GHz)," http://ark.intel.com/products/82931/Intel-Core-i7-5930K-Processor-15M-Cache-up-to-3_70-GHz, 2014. [Online; Accessed 16-July-2015].

[12] Intel, "Intel® Core™ i7-5960X Processor Extreme Edition (20M Cache, up to 3.50 GHz)," http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz, 2014. [Online; Accessed 16-July-2015].

[13] Intel, "Intel® Xeon® Processor E7-8893 v2 (37.5M Cache, 3.40 GHz)," http://ark.intel.com/products/75260/Intel-Xeon-Processor-E7-8893-v2-37_5M-Cache-3_40-GHz, 2014. [Online; Accessed 16-July-2015].

[14] Intel, "Intel® Xeon® Processor E7-4809 v2 (12M Cache, 1.90 GHz)," http://ark.intel.com/products/75245/Intel-Xeon-Processor-E7-4809-v2-12M-Cache-1_90-GHz, 2014. [Online; Accessed 16-July-2015].

[15] Jason Monk, Using GPU Processing to Solve Large-Scale Scientific Problems, Master's Thesis, University of Maine (May 2013).

[16] Newegg, "Power Supply Calculator," http://images10.newegg.com/BizIntell/tool/psucalc/index.html?name=Power-Supply-Wattage-Calculator. [Online; Accessed 26-July-2015].

[17] Ecova, "What is 80 PLUS certified?," http://www.plugloadsolutions.com/80PlusPowerSupplies.aspx#, 2015. [Online; Accessed 16-July-2015].

[18] Newegg, "CORSAIR AXi series AX860i," http://www.newegg.com/Product/Product.aspx?Item=N82E16817139041&cm_re=modular_power_supply-_-17-139-041-_-Product, 2015. [Online; Accessed 26-July-2015].

[19] Newegg, "EVGA 100-W1-500-KR," http://www.newegg.com/Product/Product.aspx?Item=N82E16817438016, 2015. [Online; Accessed 26-July-2015].

[20] D. J. Luitjens and D. S. Rennich, "CUDA Warps and Occupancy." Webinar, July 2011.

[21] Tosaka, "CUDA Processing Flow." http://en.wikipedia.org/wiki/File:CUDA_processing_flow_(En).PNG, 2008. [Online; Accessed 26-July-2015].

[22] NVIDIA, CUDA Compiler Driver NVCC. NVIDIA, March 2015.

[23] ImageMagick, "ImageMagick Formats," http://www.imagemagick.org/script/formats.php, 2015. [Online; Accessed 17-July-2015].

[24] ImageMagick, "ImageMagick Formats," http://www.imagemagick.org/script/magick-wand.php, 2013. [Online; Accessed 23-January-2013].

[25] NASA, "Picture Album: M101," http://hubblesite.org/gallery/album/entire/pr2006010a/warn/, 2006. [Online; Accessed 20-July-2015].

[26] Microsoft, "Cleaning Up Unmanaged Resources," https://msdn.microsoft.com/en-us/library/498928w2(v=vs.110).aspx, 2015. [Online; Accessed 20-July-2015].

[27] Avi Bleiweiss. 2008. GPU accelerated pathfinding. In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (GH '08). Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 65-74.

[28] G. E. Tucker, S. T. Lancaster, N. M. Gasparini, and R. L. Bras, The Channel-Hillslope Integrated Landscape Development Model (CHILD), ch. 12. Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, 2002.

[29] MathWorks, "mexFunction (C and Fortran)," https://www.mathworks.com/help/matlab/apiref/mexfunction.html, 2015. [Online, Accessed 28-September-2013].

[30] Krog, Ø.E.: GPU-based Real-Time Snow Avalanche Simulations. Master's thesis, NTNU (June 2010)

[31] N. Satish, M. Harris, M. Garland, Designing efficient sorting algorithms for manycore GPUs, in: Proceedings of IEEE International Parallel and Distributed Processing Symposium 2009, 2009.

[32] JM Domínguez, AJC Crespo, M Gómez-Gesteira, and JC Marongiu, Neighbour lists in smoothed particle hydrodynamics, International Journal for Numerical Methods in Fluids, 67 (2011), pp. 2026-2042.

[33] Robert A Gingold and Joseph J Monaghan, Smoothed particle hydrodynamics-theory and application to non-spherical stars, Monthly notices of the royal astronomical society, 181 (1977), pp. 375-389.

[34] D. Valdez-Balderas, JM Domínguez, BD Rogers, and AJC Crespo, Towards accelerating Smoothed Particle Hydrodynamics simulations for free-surface flows on multi-GPU clusters, ArXiv e-prints, 1210.1017 (2012).

APPENDIX A

NVIDIA QUADRO NVS 420 XORG.CONF

```
Section "ServerLayout"
	Identifier		"Layout0"
	Screen		0  "Screen0" 0 0
	Screen		1  "Screen1" 1280 0
	Screen		2  "Screen2" 0 2048
	Screen		3  "Screen3" 2560 0
	Screen		4  "Screen4" 2560 2048
	InputDevice		"Keyboard0" "CoreKeyboard"
	InputDevice		"Mouse0" "CorePointer"
	Option		"Xinerama" "1"
EndSection

Section "Files"
EndSection

Section "InputDevice"
	# generated from default
	Identifier	"Mouse0"
	Driver	"mouse"
	Option	"Protocol" "auto"
	Option	"Device" "/dev/psaux"
	Option	"Emulate3Buttons" "no"
	Option	"ZAxisMapping" "4 5"
EndSection

Section "InputDevice"
	# generated from default
	Identifier		"Keyboard0"
	Driver		"kbd"
EndSection

Section "Monitor"
	Identifier		"Monitor0"
	VendorName		"Unknown"
	ModelName		"DELL 1907FP"
	HorizSync		30.0 - 81.0
	VertRefresh		56.0 - 76.0
	Option		"DPMS"
EndSection

Section "Device"
	Identifier		"Device0"
```

```
        Driver                  "nvidia"
        VendorName              "NVIDIA Corporation"
        BoardName               "Quadro NVS 420"
        BusID                   "PCI:3:0:0"
EndSection

Section "Device"
        Identifier              "Device1"
        Driver                  "nvidia"
        VendorName              "NVIDIA Corporation"
        BoardName               "Quadro NVS 420"
        BusID                   "PCI:4:0:0"
EndSection

Section "Device"
        Identifier              "Device2"
        Driver                  "nvidia"
        VendorName              "NVIDIA Corporation"
        BoardName               "Quadro NVS 420"
        BusID                   "PCI:7:0:0"
EndSection

Section "Device"
        Identifier              "Device3"
        Driver                  "nvidia"
        VendorName              "NVIDIA Corporation"
        BoardName               "Quadro NVS 420"
        BusID                   "PCI:8:0:0"
EndSection

Section "Device"
        Identifier              "Device4"
        Driver                  "nvidia"
        VendorName              "NVIDIA Corporation"
        BoardName               "Quadro NVS 420"
        BusID                   "PCI:12:0:0"
EndSection

Section "Screen"
        Identifier              "Screen0"
        Device                  "Device0"
        Monitor                         "Monitor0"
        DefaultDepth  24
        Option                  "ConnectedMonitor" "DFP,DFP"
        Option                  "UseDisplayDevice" "DFP-0,DFP-1"
```

```
	Option			"CustomEDID" "DFP-0:/etc/X11/edid.bin;DFP-
1:/etc/X11/edid.bin"
	Option			"TwinView" "1"
	Option			"TwinViewXineramaInfoOrder" "DFP-0"
	Option			"metamodes" "DFP-0: nvidia-auto-select +0+0, DFP-1:
nvidia-auto-select +0+1024"
	SubSection		"Display"
		Depth		24
	EndSubSection
EndSection

Section "Screen"
	Identifier		"Screen1"
	Device			"Device1"
	Monitor				"Monitor0"
	DefaultDepth	24
	Option			"ConnectedMonitor" "DFP,DFP"
	Option			"UseDisplayDevice" "DFP-0,DFP-1"
	Option			"CustomEDID" "DFP-0:/etc/X11/edid.bin;DFP-
1:/etc/X11/edid.bin"
	Option			"TwinView" "1"
	Option			"TwinViewXineramaInfoOrder" "DFP-0"
	Option			"metamodes" "DFP-0: nvidia-auto-select +0+0, DFP-1:
nvidia-auto-select +0+1024"
	SubSection		"Display"
		Depth		24
	EndSubSection
EndSection

Section "Screen"
	Identifier		"Screen2"
	Device			"Device2"
	Monitor				"Monitor0"
	DefaultDepth	24
	Option			"ConnectedMonitor" "DFP,DFP"
	Option			"UseDisplayDevice" "DFP-0,DFP-1"
	Option			"CustomEDID" "DFP-0:/etc/X11/edid.bin;DFP-
1:/etc/X11/edid.bin"
	Option			"TwinView" "1"
	Option			"TwinViewXineramaInfoOrder" "DFP-0"
	Option			"metamodes" "DFP-0: nvidia-auto-select +0+0, DFP-1:
nvidia-auto-select +1280+0"
	SubSection		"Display"
		Depth		24
	EndSubSection
EndSection
```

```
Section "Screen"
	Identifier		"Screen3"
	Device		"Device3"
	Monitor			"Monitor0"
	DefaultDepth	24
	Option		"ConnectedMonitor" "DFP,DFP"
	Option		"UseDisplayDevice" "DFP-0,DFP-1"
	Option		"CustomEDID" "DFP-0:/etc/X11/edid.bin;DFP-
1:/etc/X11/edid.bin"
	Option		"TwinView" "1"
	Option		"TwinViewXineramaInfoOrder" "DFP-0"
	Option		"metamodes" "DFP-0: nvidia-auto-select +0+0, DFP-1:
nvidia-auto-select +0+1024"
	SubSection		"Display"
		Depth	24
	EndSubSection
EndSection

Section "Screen"
	Identifier		"Screen4"
	Device		"Device4"
	Monitor			"Monitor0"
	DefaultDepth	24
	Option		"ConnectedMonitor" "DFP,DFP"
	Option		"UseDisplayDevice" "DFP-0,DFP-1"
	Option		"CustomEDID" "DFP-0:/etc/X11/edid.bin;DFP-
1:/etc/X11/edid.bin"
	Option		"TwinView" "0"
	Option		"TwinViewXineramaInfoOrder" "DFP-0"
	Option		"metamodes" "DFP-0: nvidia-auto-select +0+0"
	SubSection		"Display"
		Depth	24
	EndSubSection
EndSection

Section "Extensions"
	Option	"Composite" "Disable"
EndSection
```

APPENDIX B

AUTO CONFIGURED XORG.CONF

```
Section "ServerLayout"
	Identifier		"X.org Configured"
	Screen			0  "Screen0" 0 0
	Screen			1  "Screen1" RightOf "Screen0"
	InputDevice		"Mouse0" "CorePointer"
	InputDevice		"Keyboard0" "CoreKeyboard"
EndSection

Section "Files"
	ModulePath		"/usr/lib/xorg/modules"
	FontPath		"/usr/share/fonts/misc/"
	FontPath		"/usr/share/fonts/TTF/"
	FontPath		"/usr/share/fonts/OTF/"
	FontPath		"/usr/share/fonts/Type1/"
	FontPath		"/usr/share/fonts/100dpi/"
	FontPath		"/usr/share/fonts/75dpi/"
EndSection

Section "Module"
	Load		"glx"
EndSection

Section "InputDevice"
	Identifier		"Keyboard0"
	Driver			"kbd"
EndSection

Section "InputDevice"
	Identifier		"Mouse0"
	Driver			"mouse"
	Option			"Protocol" "auto"
	Option			"Device" "/dev/input/mice"
	Option			"ZAxisMapping" "4 5 6 7"
EndSection

Section "Monitor"
	Identifier		"Monitor0"
	VendorName		"Monitor Vendor"
	ModelName		"Monitor Model"
EndSection

Section "Monitor"
```

```
        Identifier              "Monitor1"
        VendorName              "Monitor Vendor"
        ModelName               "Monitor Model"
EndSection

Section "Device"
                ### Available Driver options are:-
                ### Values: <i>: integer, <f>: float, <bool>: "True"/"False",
                ### <string>: "String", <freq>: "<f> Hz/kHz/MHz",
                ### <percent>: "<f>%"
                ### [arg]: arg optional
                #Option         "Accel"                         # [<bool>]
                #Option         "SWcursor"                      # [<bool>]
                #Option         "EnablePageFlip"                # [<bool>]
                #Option         "ColorTiling"                   # [<bool>]
                #Option         "ColorTiling2D"                 # [<bool>]
                #Option         "RenderAccel"                   # [<bool>]
                #Option         "SubPixelOrder"                 # [<str>]
                #Option         "AccelMethod"                   # <str>
                #Option         "EXAVSync"                      # [<bool>]
                #Option         "EXAPixmaps"                    # [<bool>]
                #Option         "ZaphodHeads"                   # <str>
                #Option         "EnablePageFlip"                # [<bool>]
                #Option         "SwapbuffersWait"               # [<bool>]
        Identifier                      "Card0"
        Driver                          "radeon"
        BusID                           "PCI:1:0:0"
EndSection

Section "Device"
                ### Available Driver options are:-
                ### Values: <i>: integer, <f>: float, <bool>: "True"/"False",
                ### <string>: "String", <freq>: "<f> Hz/kHz/MHz",
                ### <percent>: "<f>%"
                ### [arg]: arg optional
                #Option         "Accel"                         # [<bool>]
                #Option         "SWcursor"                      # [<bool>]
                #Option         "EnablePageFlip"                # [<bool>]
                #Option         "ColorTiling"                   # [<bool>]
                #Option         "ColorTiling2D"                 # [<bool>]
                #Option         "RenderAccel"                   # [<bool>]
                #Option         "SubPixelOrder"                 # [<str>]
                #Option         "AccelMethod"                   # <str>
                #Option         "EXAVSync"                      # [<bool>]
                #Option         "EXAPixmaps"                    # [<bool>]
                #Option         "ZaphodHeads"                   # <str>
```

```
		#Option		"EnablePageFlip"		# [<bool>]
		#Option		"SwapbuffersWait"		# [<bool>]
	Identifier			"Card1"
	Driver				"radeon"
	BusID				"PCI:2:0:0"
EndSection

Section "Screen"
	Identifier		"Screen0"
	Device			"Card0"
	Monitor				"Monitor0"
	SubSection		"Display"
		Viewport	0 0
		Depth		1
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		4
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		8
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		15
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		16
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		24
	EndSubSection
EndSection

Section "Screen"
	Identifier		"Screen1"
	Device			"Card1"
	Monitor				"Monitor1"
	SubSection		"Display"
		Viewport	0 0
		Depth		1
	EndSubSection
	SubSection "Display"
```

```
		Viewport	0 0
		Depth		4
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		8
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		15
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		16
	EndSubSection
	SubSection "Display"
		Viewport	0 0
		Depth		24
	EndSubSection
EndSection
```

APPENDIX C

FIRST RADEON HD 5870 XORG.CONF

```
Section "ServerLayout"
	Identifier		"X.org Configured"
	Screen		0  "Screen0" 0 0
	Screen		1  "Screen1" 1280 0
	Screen		2  "Screen2" 2560 0
	Screen		3  "Screen3" 0 1024
	Screen		4  "Screen4" 1280 1024
	Screen		5  "Screen5" 2560 1024
	Screen		6  "Screen6" 0 2048
	Screen		7  "Screen7" 1280 2048
	Screen		8  "Screen8" 2560 2048
	InputDevice		"Mouse0" "CorePointer"
	InputDevice		"Keyboard0" "CoreKeyboard"
	Option		"Xinerama"
EndSection

Section "Files"
	ModulePath		"/usr/lib/xorg/modules"
	FontPath		"/usr/share/fonts/misc/"
	FontPath		"/usr/share/fonts/TTF/"
	FontPath		"/usr/share/fonts/OTF/"
	FontPath		"/usr/share/fonts/Type1/"
	FontPath		"/usr/share/fonts/100dpi/"
	FontPath		"/usr/share/fonts/75dpi/"
EndSection

Section "Module"
	Load		"glx"
EndSection

Section "InputDevice"
	Identifier		"Keyboard0"
	Driver		"kbd"
EndSection

Section "InputDevice"
	Identifier		"Mouse0"
	Driver		"mouse"
	Option		"Protocol" "auto"
	Option		"Device" "/dev/input/mice"
	Option		"ZAxisMapping" "4 5 6 7"
EndSection
```

```
Section "Monitor"
	Identifier		"Default-Monitor"
EndSection

Section "Device"
	Identifier		"Card0"
	Driver			"radeon"
	BusID			"PCI:1:0:0"
	Screen			0
	Option			"ZaphodHeads" "DisplayPort-0"
EndSection

Section "Device"
	Identifier		"Card1"
	Driver			"radeon"
	BusID			"PCI:1:0:0"
	Screen			1
	Option			"ZaphodHeads" "DisplayPort-1"
EndSection

Section "Device"
	Identifier		"Card2"
	Driver			"radeon"
	BusID			"PCI:1:0:0"
	Screen			2
	Option			"ZaphodHeads" "DisplayPort-2"
EndSection

Section "Device"
	Identifier		"Card3"
	Driver			"radeon"
	BusID			"PCI:1:0:0"
	Screen			3
	Option			"ZaphodHeads" "DisplayPort-3"
EndSection

Section "Device"
	Identifier		"Card4"
	Driver			"radeon"
	BusID			"PCI:1:0:0"
	Screen			4
	Option			"ZaphodHeads" "DisplayPort-4"
EndSection

Section "Device"
```

```
	Identifier		"Card5"
	Driver			"radeon"
	BusID			"PCI:1:0:0"
	Screen			5
	Option			"ZaphodHeads" "DisplayPort-5"
EndSection

Section "Device"
	Identifier		"Card6"
	Driver			"radeon"
	BusID			"PCI:2:0:0"
	Screen			0
	Option			"ZaphodHeads" "DisplayPort-6"
EndSection

Section "Device"
	Identifier		"Card7"
	Driver			"radeon"
	BusID			"PCI:2:0:0"
	Screen			1
	Option			"ZaphodHeads" "DisplayPort-7"
EndSection

Section "Device"
	Identifier		"Card8"
	Driver			"radeon"
	BusID			"PCI:2:0:0"
	Screen			2
	Option			"ZaphodHeads" "DisplayPort-8"
EndSection

Section "Screen"
	Identifier		"Screen0"
	Device			"Card0"
	Monitor			"Default-Monitor"
EndSection

Section "Screen"
	Identifier		"Screen1"
	Device			"Card1"
	Monitor			"Default-Monitor"
EndSection

Section "Screen"
	Identifier		"Screen2"
	Device			"Card2"
```

```
        Monitor                 "Default-Monitor"
EndSection

Section "Screen"
        Identifier              "Screen3"
        Device                  "Card3"
        Monitor                 "Default-Monitor"
EndSection

Section "Screen"
        Identifier              "Screen4"
        Device                  "Card4"
        Monitor                 "Default-Monitor"
EndSection

Section "Screen"
        Identifier              "Screen5"
        Device                  "Card5"
        Monitor                 "Default-Monitor"
EndSection

Section "Screen"
        Identifier              "Screen6"
        Device                  "Card6"
        Monitor                 "Default-Monitor"
EndSection

Section "Screen"
        Identifier              "Screen7"
        Device                  "Card7"
        Monitor                 "Default-Monitor"
EndSection

Section "Screen"
        Identifier              "Screen8"
        Device                  "Card8"
        Monitor                 "Default-Monitor"
EndSection
```

APPENDIX D

SECOND RADEON HD 5870 XORG.CONF

```
Section "ServerLayout"
        Identifier    "X.org Configured"
        Screen     0  "Screen0" 0 0
        Screen     1  "Screen1" 1280 0
        Screen     2  "Screen2" 2560 0
        Screen     3  "Screen3" 0 1024
        Screen     4  "Screen4" 1280 1024
        Screen     5  "Screen5" 2560 1024
        Screen     6  "Screen6" 0 2048
        Screen     7  "Screen7" 1280 2048
        Screen     8  "Screen8" 2560 2048
        InputDevice    "Mouse0" "CorePointer"
        InputDevice    "Keyboard0" "CoreKeyboard"
        Option        "Xinerama"
EndSection

Section "Files"
        ModulePath   "/usr/lib/xorg/modules"
        FontPath     "/usr/share/fonts/misc/"
        FontPath     "/usr/share/fonts/TTF/"
        FontPath     "/usr/share/fonts/OTF/"
        FontPath     "/usr/share/fonts/Type1/"
        FontPath     "/usr/share/fonts/100dpi/"
        FontPath     "/usr/share/fonts/75dpi/"
EndSection

Section "Module"
        Load  "glx"
EndSection

Section "InputDevice"
        Identifier  "Keyboard0"
        Driver      "kbd"
EndSection

Section "InputDevice"
        Identifier  "Mouse0"
        Driver      "mouse"
        Option      "Protocol" "auto"
        Option      "Device" "/dev/input/mice"
        Option      "ZAxisMapping" "4 5 6 7"
EndSection
```

```
Section "Monitor"
        Identifier   "Default-Monitor"
EndSection

Section "Device"
        Identifier  "Card0"
        Driver      "radeon"
        BusID       "PCI:2:0:0"
        Screen      0
        Option      "ZaphodHeads" "DisplayPort-0"
EndSection

Section "Device"
        Identifier  "Card1"
        Driver      "radeon"
        BusID       "PCI:2:0:0"
        Screen      1
        Option      "ZaphodHeads" "DisplayPort-1"
EndSection

Section "Device"
        Identifier  "Card2"
        Driver      "radeon"
        BusID       "PCI:2:0:0"
        Screen      2
        Option      "ZaphodHeads" "DisplayPort-2"
EndSection

Section "Device"
        Identifier  "Card3"
        Driver      "radeon"
        BusID       "PCI:2:0:0"
        Screen      3
        Option      "ZaphodHeads" "DisplayPort-3"
EndSection

Section "Device"
        Identifier  "Card4"
        Driver      "radeon"
        BusID       "PCI:2:0:0"
        Screen      4
        Option      "ZaphodHeads" "DisplayPort-4"
EndSection

Section "Device"
```

```
	Identifier  "Card5"
	Driver      "radeon"
	BusID       "PCI:2:0:0"
	Screen      5
	Option      "ZaphodHeads" "DisplayPort-5"
EndSection

Section "Device"
	Identifier  "Card6"
	Driver      "radeon"
	BusID       "PCI:6:0:0"
	Screen      0
	Option      "ZaphodHeads" "DisplayPort-6"
EndSection

Section "Device"
	Identifier  "Card7"
	Driver      "radeon"
	BusID       "PCI:6:0:0"
	Screen      1
	Option      "ZaphodHeads" "DisplayPort-7"
EndSection

Section "Device"
	Identifier  "Card8"
	Driver      "radeon"
	BusID       "PCI:6:0:0"
	Screen      2
	Option      "ZaphodHeads" "DisplayPort-8"
EndSection

Section "Screen"
	Identifier "Screen0"
	Device     "Card0"
	Monitor    "Default-Monitor"
EndSection

Section "Screen"
	Identifier "Screen1"
	Device     "Card1"
	Monitor    "Default-Monitor"
EndSection

Section "Screen"
	Identifier "Screen2"
	Device     "Card2"
```

```
        #Monitor    "DisplayPort-2"
        Monitor    "Default-Monitor"
EndSection

Section "Screen"
        Identifier "Screen3"
        Device     "Card3"
        Monitor    "Default-Monitor"
EndSection

Section "Screen"
        Identifier "Screen4"
        Device     "Card4"
        Monitor    "Default-Monitor"
EndSection

Section "Screen"
        Identifier "Screen5"
        Device     "Card5"
        Monitor    "Default-Monitor"
EndSection

Section "Screen"
        Identifier "Screen6"
        Device     "Card6"
        Monitor    "Default-Monitor"
EndSection

Section "Screen"
        Identifier "Screen7"
        Device     "Card7"
        Monitor    "Default-Monitor"
EndSection

Section "Screen"
        Identifier "Screen8"
        Device     "Card8"
        Monitor    "Default-Monitor"
EndSection
```

APPENDIX E

GPGPU MACHINE CONFIGURATION SPECIFICATIONS

Part	Name	Specification	Price	Qty	Cost
Case	Rosewill Throne-Window Black	ATX Full Tower	$169.99	1	$169.99
Power Supply	Corsair AX1500i	1500W ATX 80 PLUS Titanium Full Modular	$461.81	1	$461.81
Motherboard	ASUS Rampage V Extreme	LGA 2011-v3 Intel X99	$498.99	1	$498.99
CPU	Intel Core i7-5930K	Haswell-E LGA 2011-v3 140W 6-core 3.5 GHz 15MB	$670.24	1	$670.24
GPU	EVGA Superclocked GTX 980	GTX 980 4GB GDDR5 PCIe 3.0 x16	$622.78	2	$1,245.56
RAM	G.Skill Ripjaws 4 Series 32GB	32GB (4 x 8GB) 288-Pin DDR4 2133	$479.99	1	$479.99
Optical Drive	Lite-On It	24x DVD-RW	$17.71	1	$17.71
Hard Drives	Seagate 2TB Barracuda	2TB 7200 RPM SATA 6.0 G/s 64MB	$104.99	4	$419.96
Total Cost					$3,964.25

Table E.1: First Machine Configuration

Part	**Name**	**Spec**	**Price**	**Qty**	**Cost**
Case	Rosewill Throne-Window Black	ATX Full Tower	$169.99	1	$169.99
Power Supply	Corsair AX1500i	1500W ATX 80 PLUS Titanium Full Modular	$461.81	1	$461.81
Motherboard	ASUS Rampage V Extreme	LGA 2011-v3 Intel X99	$498.99	1	$498.99
CPU	Intel Core i7-5930K	Haswell-E LGA 2011-v3 140W 6-core 3.5 GHz 15MB	$670.24	1	$670.24
GPU	EVGA Superclocked GTX Titan Black	GTX Titan Black 6GB GDDR5 PCIe 3.0 x16	$1,222.22	2	$2,444.44
RAM	G.Skill Ripjaws 4 Series 32GB	32GB (4 x 8GB) 288-Pin DDR4 2133	$479.99	1	$479.99
Optical Drive	Lite-On It	24x DVD-RW	$17.71	1	$17.71
Hard Drives	Seagate 2TB Barracuda	2TB 7200 RPM SATA 6.0 G/s 64MB	$104.99	5	$524.95
Total Cost					$5,268.12

Table E.2: Second Machine Configuration

APPENDIX F

IMAGE PROCESSING TESTS

Open Timer					
Image Resolution	**GTX 580 #1**	**GTX 580 #2**	**GTX 680**	**GTX 980 #1**	**GTX 980 #2**
793x620	0.06022	0.06053	0.06014	0.07141	0.05917
1585x1239	0.26126	0.26566	0.26526	0.23433	0.22368
2378x1859	0.61919	0.61967	0.61762	0.52405	0.49091
3170x2478	1.09994	1.12566	1.12504	0.94441	0.95717
3963x3098	1.78247	1.80044	1.80112	1.49596	1.50720
4756x3718	2.58364	2.62191	2.61840	2.21637	2.13052
5548x4337	3.56217	3.61459	3.56201	2.92795	2.91538
6341x4957	4.70008	4.45329	4.71567	3.80094	3.85230
7133x5576	5.94769	5.88825	6.04672	4.91367	4.89208
7926x6196	7.43774	7.29170	7.23369	6.02091	6.18280
8719x6816	N/A	N/A	8.94446	7.09851	7.29531
9511x7435	N/A	N/A	10.74893	8.77665	8.71247
10304x8055	N/A	N/A	N/A	9.99032	10.26523
11096x8674	N/A	N/A	N/A	11.31017	11.95100
11889x9294	N/A	N/A	N/A	13.46067	13.75117
12682x9914	N/A	N/A	N/A	15.66043	15.71900
13474x10533	N/A	N/A	N/A	17.68167	N/A
14267x11153	N/A	N/A	N/A	N/A	N/A
15059x11772	N/A	N/A	N/A	N/A	N/A
15852x12392	N/A	N/A	N/A	N/A	N/A

Table F.1: Image Processing Open Timer Values

Setup Timer					
Image Resolution	**GTX 580 #1**	**GTX 580 #2**	**GTX 680**	**GTX 980 #1**	**GTX 980 #2**
793x620	0.06793	0.06646	0.06983	0.17005	0.15652
1585x1239	0.14771	0.14149	0.15169	0.20072	0.20169
2378x1859	0.28203	0.26742	0.29257	0.28463	0.27282
3170x2478	0.47098	0.44347	0.48668	0.40936	0.40518
3963x3098	0.71133	0.67344	0.73529	0.55955	0.54639
4756x3718	1.00294	0.93329	1.04171	0.75449	0.75243
5548x4337	1.35162	1.27654	1.38539	0.92489	1.00996
6341x4957	1.74896	1.61217	1.82514	1.28526	1.24104
7133x5576	2.19037	2.03059	2.27789	1.47909	1.53798
7926x6196	2.68658	2.53414	2.75602	1.79652	1.84085
8719x6816	N/A	N/A	3.38904	2.13404	2.23138
9511x7435	N/A	N/A	3.97476	2.52831	2.60918
10304x8055	N/A	N/A	N/A	2.91457	2.99937
11096x8674	N/A	N/A	N/A	3.40801	3.51061
11889x9294	N/A	N/A	N/A	3.96724	4.02279
12682x9914	N/A	N/A	N/A	4.65806	4.67136
13474x10533	N/A	N/A	N/A	5.27606	N/A
14267x11153	N/A	N/A	N/A	N/A	N/A
15059x11772	N/A	N/A	N/A	N/A	N/A
15852x12392	N/A	N/A	N/A	N/A	N/A

Table F.2: Image Processing Setup Timer Values

GPU Timer					
Image Resolution	**GTX 580 #1**	**GTX 580 #2**	**GTX 680**	**GTX 980 #1**	**GTX 980 #2**
793x620	0.00447	0.00417	0.00477	0.00213	0.00217
1585x1239	0.01016	0.00993	0.01258	0.00451	0.00461
2378x1859	0.02074	0.02053	0.02937	0.00727	0.00774
3170x2478	0.03593	0.03552	0.05006	0.02070	0.02533
3963x3098	0.05635	0.05620	0.07558	0.04451	0.05386
4756x3718	0.08581	0.08551	0.11173	0.06848	0.08177
5548x4337	0.11442	0.11411	0.15051	0.08755	0.10681
6341x4957	0.14890	0.14859	0.20137	0.13074	0.15970
7133x5576	0.18138	0.18109	0.24994	0.17177	0.20912
7926x6196	0.22115	0.22092	0.30870	0.21266	0.25953
8719x6816	N/A	N/A	0.37558	0.25551	0.31061
9511x7435	N/A	N/A	0.44697	0.30280	0.36920
10304x8055	N/A	N/A	N/A	0.36121	0.44257
11096x8674	N/A	N/A	N/A	0.43466	0.52400
11889x9294	N/A	N/A	N/A	0.58338	0.62538
12682x9914	N/A	N/A	N/A	0.79818	0.81874
13474x10533	N/A	N/A	N/A	0.97888	N/A
14267x11153	N/A	N/A	N/A	N/A	N/A
15059x11772	N/A	N/A	N/A	N/A	N/A
15852x12392	N/A	N/A	N/A	N/A	N/A

Table F.3: Image Processing GPU Timer Values

Save Timer					
Image Resolution	**GTX 580 #1**	**GTX 580 #2**	**GTX 680**	**GTX 980 #1**	**GTX 980 #2**
793x620	0.25422	0.25558	0.25367	0.24702	0.45319
1585x1239	0.99420	0.99518	0.99659	0.96172	1.33285
2378x1859	2.17783	2.18110	2.18906	1.94356	2.68687
3170x2478	3.66344	3.68839	3.80651	3.36115	4.64885
3963x3098	5.50760	5.46167	5.53940	5.11637	7.07745
4756x3718	7.81318	7.86898	7.84524	7.14361	10.03811
5548x4337	10.60433	10.67557	10.54413	9.49074	13.50097
6341x4957	13.86053	13.80147	13.88243	12.48383	17.57727
7133x5576	17.32810	17.58893	17.33497	15.81620	22.13507
7926x6196	21.58243	21.50620	21.43030	19.39990	27.57047
8719x6816	N/A	N/A	26.14163	23.81357	33.41327
9511x7435	N/A	N/A	31.48467	28.34207	39.93007
10304x8055	N/A	N/A	N/A	33.23827	46.75543
11096x8674	N/A	N/A	N/A	38.56040	54.05287
11889x9294	N/A	N/A	N/A	44.20510	61.85707
12682x9914	N/A	N/A	N/A	50.70307	71.73130
13474x10533	N/A	N/A	N/A	57.62500	N/A
14267x11153	N/A	N/A	N/A	N/A	N/A
15059x11772	N/A	N/A	N/A	N/A	N/A
15852x12392	N/A	N/A	N/A	N/A	N/A

Table F.4: Image Processing Save Timer Values

APPENDIX G

IMAGE PROCESSING SOURCE

The full source code for the implemented image processing library is available on github at https://github.com/fostro/ECE533ImageProcessing. This library is licensed under GNU General Public License Version 2.

APPENDIX H

TETRIS GENETIC ALGORITHM SOURCE

The full source code and working Unity3d project for the Tetris AI genetic algorithm with A* pathfinding implementation is available for download for personal use via Google Drive at the following address https://drive.google.com/file/d/0B49gTqY3DhdmNkVBUG1QV19wOVE/view?usp=sharing. The Tetris AI Unity implementation is Copyright 2014 Dream-Crusher Labs, LLC.

APPENDIX I

TETRIS GENETIC ALGORITHM FULL RESULTS

The full results of running the Tetris genetic algorithm within Unity3d can be found at the following link

https://drive.google.com/file/d/0B49gTqY3DhdmdVdNZ1p4YlRxRWc/view?usp=sharin
g.

APPENDIX J

MATLABCOMPILE.SH

```
#!/usr/bin/bash

#MATLAB=mex_matlab;
MATLAB=mex;
OUTPUT_DIR=runTest/build/;

cd objects;
./compile.sh;

cd ..;

$MATLAB matlabChildDriver.cpp MatlabToCppArgument.cpp \
../CMakeFiles/child-shared.dir/tInputFile/tInputFile.cpp.o \
../CMakeFiles/child-shared.dir/Erosion/erosion.cpp.o \
../CMakeFiles/child-shared.dir/MeshElements/meshElements.cpp.o \
../CMakeFiles/child-shared.dir/tMesh/ParamMesh_t.cpp.o \
../CMakeFiles/child-shared.dir/tMesh/TipperTriangulator.cpp.o \
../CMakeFiles/child-shared.dir/tStreamMeander/tStreamMeander.cpp.o \
../CMakeFiles/child-shared.dir/tVegetation/tVegetation.cpp.o \
../CMakeFiles/child-shared.dir/tLNode/tLNode.cpp.o \
../CMakeFiles/child-shared.dir/tStorm/tStorm.cpp.o \
../CMakeFiles/child-shared.dir/tWaterSedTracker/tWaterSedTracker.cpp.o \
../CMakeFiles/child-shared.dir/tTimeSeries/tTimeSeries.cpp.o \
../CMakeFiles/child-shared.dir/ChildInterface/childDriver.cpp.o \
../CMakeFiles/child-shared.dir/ChildInterface/childInterface.cpp.o \
../CMakeFiles/child-shared.dir/tListInputData/tListInputData.cpp.o \
../CMakeFiles/child-shared.dir/tUplift/tUplift.cpp.o \
../CMakeFiles/child-shared.dir/tEolian/tEolian.cpp.o \
../CMakeFiles/child-shared.dir/tFloodplain/tFloodplain.cpp.o \
../CMakeFiles/child-shared.dir/globalFns.cpp.o \
../CMakeFiles/child-shared.dir/tIDGenerator/tIDGenerator.cpp.o \
../CMakeFiles/child-shared.dir/tLithologyManager/tLithologyManager.cpp.o \
../CMakeFiles/child-shared.dir/tRunTimer/tRunTimer.cpp.o \
../CMakeFiles/child-shared.dir/Predicates/predicates.cpp.o \
../CMakeFiles/child-shared.dir/Mathutil/mathutil.cpp.o \
objects/*.o;

mv matlabChildDriver.mex* $OUTPUT_DIR;
```

APPENDIX K

CHILD MATLAB PATCH

```
diff -rupN child/Code/errors/errors.cpp CHILD_MATLAB/Code/errors/errors.cpp
--- child/Code/errors/errors.cpp         2015-07-24 20:36:09.639036529 -0400
+++ CHILD_MATLAB/Code/errors/errors.cpp       2015-07-24 20:06:59.007029043 -
0400
@@ -16,6 +16,10 @@
 #define CHILD_ABORT_ON_ERROR "CHILD_ABORT_ON_ERROR"
 #define CHILD_ABORT_ON_WARNING "CHILD_ABORT_ON_WARNING"

+#ifdef MATLAB
+        #include <mex.h>
+#endif
+

/*********************************************************************
******\
 **
 **  ReportFatalError:  This is an error-handling routine that prints the
@@ -30,6 +34,10 @@ void ReportFatalError( const char *errMs
 {
   std::cout << errMsg <<std::endl;

+#ifdef MATLAB
+        mexErrMsgTxt(errMsg);
+#endif
+
   std::cout << "That was a fatal error, my friend!" <<std::endl;
   if (getenv(CHILD_ABORT_ON_ERROR) != NULL)
     abort();
diff -rupN child/Code/matlabInterface/matlabChildDriver.cpp
CHILD_MATLAB/Code/matlabInterface/matlabChildDriver.cpp
--- child/Code/matlabInterface/matlabChildDriver.cpp        1969-12-31
19:00:00.000000000 -0500
+++ CHILD_MATLAB/Code/matlabInterface/matlabChildDriver.cpp       2015-07-24
20:06:59.630355966 -0400
@@ -0,0 +1,57 @@
+/********************************************************************
****/
+/**
+**  matlabChildDriver.cpp: This file modifies the childRDriver.cpp example
+**  file which uses the interface functions.  This file wraps those calls
+**  for use with matlab.
+**
```

```
+**  October 2013
+**
+**  For information regarding how to use the matlab wrapper functions,
+**  please contact Forrest Flagg at:
+**        raymond.flagg@maine.edu
+**
+**  For information regarding this program, please contact Greg Tucker at:
+**
+**     Cooperative Institute for Research in Environmental Sciences (CIRES)
+**     and Department of Geological Sciences
+**     University of Colorado
+**     2200 Colorado Avenue, Campus Box 399
+**     Boulder, CO 80309-0399
+**
+*/
+/*******************************************************************
****/
+
+#include "../ChildInterface/childInterface.h"
+#include "MatlabToCppArgument.h"
+#include "mex.h"
+
+void mexFunction( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
+	childInterface myChildInterface;
+
+	MatlabToCppArgument* arguments = new MatlabToCppArgument(nrhs, prhs,
"test");
+
+	//myChildRInterface.Initialize( argc, argv );
+	myChildInterface.Initialize( arguments->getNumArguments(), arguments-
>getCArguments() );
+
+	if(1) // make this zero to use "example 2" below
+	{
+		// Example 1: using "Run" method, and setting run duration to zero so
model reads duration from input file
+		myChildInterface.Run( 0 );
+	}
+	else
+	{
+		// Example 2: using "RunOneStorm"
+		double mytime = 0;
+		double myrunduration = 100000;
+
+		while( mytime<myrunduration )
+		{
```

```
+                               mytime = myChildInterface.RunOneStorm();
+                       }
+           }
+
+           // Note that calling CleanUp() isn't strictly necessary, as the destructor will
automatically clean it
+           // up when myChildInterface is deleted ... but it's nice to be able to do this at will
(and free up
+           // memory)
+           myChildInterface.CleanUp();
+}
diff -rupN child/Code/matlabInterface/matlabCompile.sh
CHILD_MATLAB/Code/matlabInterface/matlabCompile.sh
--- child/Code/matlabInterface/matlabCompile.sh     1969-12-31 19:00:00.000000000 -
0500
+++ CHILD_MATLAB/Code/matlabInterface/matlabCompile.sh   2015-07-24
20:06:59.637022565 -0400
@@ -0,0 +1,38 @@
+#!/usr/bin/bash
+
+#MATLAB=mex_matlab;
+MATLAB=mex;
+OUTPUT_DIR=runTest/build/;
+
+cd objects;
+./compile.sh;
+
+cd ..;
+
+$MATLAB matlabChildDriver.cpp MatlabToCppArgument.cpp \
+../CMakeFiles/child-shared.dir/tInputFile/tInputFile.cpp.o \
+../CMakeFiles/child-shared.dir/Erosion/erosion.cpp.o \
+../CMakeFiles/child-shared.dir/MeshElements/meshElements.cpp.o \
+../CMakeFiles/child-shared.dir/tMesh/ParamMesh_t.cpp.o \
+../CMakeFiles/child-shared.dir/tMesh/TipperTriangulator.cpp.o \
+../CMakeFiles/child-shared.dir/tStreamMeander/tStreamMeander.cpp.o \
+../CMakeFiles/child-shared.dir/tVegetation/tVegetation.cpp.o \
+../CMakeFiles/child-shared.dir/tLNode/tLNode.cpp.o \
+../CMakeFiles/child-shared.dir/tStorm/tStorm.cpp.o \
+../CMakeFiles/child-shared.dir/tWaterSedTracker/tWaterSedTracker.cpp.o \
+../CMakeFiles/child-shared.dir/tTimeSeries/tTimeSeries.cpp.o \
+../CMakeFiles/child-shared.dir/ChildInterface/childDriver.cpp.o \
+../CMakeFiles/child-shared.dir/ChildInterface/childInterface.cpp.o \
+../CMakeFiles/child-shared.dir/tListInputData/tListInputData.cpp.o \
+../CMakeFiles/child-shared.dir/tUplift/tUplift.cpp.o \
+../CMakeFiles/child-shared.dir/tEolian/tEolian.cpp.o \
```

```
+../CMakeFiles/child-shared.dir/tFloodplain/tFloodplain.cpp.o \
+../CMakeFiles/child-shared.dir/globalFns.cpp.o \
+../CMakeFiles/child-shared.dir/tIDGenerator/tIDGenerator.cpp.o \
+../CMakeFiles/child-shared.dir/tLithologyManager/tLithologyManager.cpp.o \
+../CMakeFiles/child-shared.dir/tRunTimer/tRunTimer.cpp.o \
+../CMakeFiles/child-shared.dir/Predicates/predicates.cpp.o \
+../CMakeFiles/child-shared.dir/Mathutil/mathutil.cpp.o \
+objects/*.o;
+
+mv matlabChildDriver.mex* $OUTPUT_DIR;
diff -rupN child/Code/matlabInterface/matlabDefines.h
CHILD_MATLAB/Code/matlabInterface/matlabDefines.h
--- child/Code/matlabInterface/matlabDefines.h	1969-12-31 19:00:00.000000000 -
0500
+++ CHILD_MATLAB/Code/matlabInterface/matlabDefines.h	2015-07-24
20:06:59.630355966 -0400
@@ -0,0 +1 @@
+#define MATLAB	1
diff -rupN child/Code/matlabInterface/MatlabToCppArgument.cpp
CHILD_MATLAB/Code/matlabInterface/MatlabToCppArgument.cpp
--- child/Code/matlabInterface/MatlabToCppArgument.cpp 1969-12-31
19:00:00.000000000 -0500
+++ CHILD_MATLAB/Code/matlabInterface/MatlabToCppArgument.cpp	2015-
07-24 20:06:59.630355966 -0400
@@ -0,0 +1,83 @@
+#include <iostream>
+#include <string>
+#include <cstring>
+#include "MatlabToCppArgument.h"
+#include "mex.h"
+
+MatlabToCppArgument::MatlabToCppArgument(int nrhs, const mxArray* prhs[],
string n) {
+	// set the name
+	name = n;
+
+	// create the argument array
+	createArgumentList(nrhs, prhs);
+
+	// create the c style char** array from the string array
+	convertStringPtrToCharPtrPtr();
+}
+
+MatlabToCppArgument::MatlabToCppArgument(int nrhs, const mxArray* prhs[],
const char* n) {
+	// set the name
```

```
+	name = n;
+
+	// create the argument array
+	createArgumentList(nrhs, prhs);
+
+	// create the c style char** array from the string array
+	convertStringPtrToCharPtrPtr();
+}
+
+MatlabToCppArgument::~MatlabToCppArgument() {
+	// clean up the allocated memory for the argument array
+	delete[] argv;
+
+	// clean up memory allocated for the c string version
+	// of the argument array
+	for (int i = 0; i < argc; ++i) {
+		free(cargv[i]);
+	}
+
+	// finish cleaning up the memory allocated
+	free(cargv);
+}
+
+void MatlabToCppArgument::createArgumentList(int nrhs, const mxArray* prhs[]) {
+	// add one to the number of arguments to allow
+	// for the first argument to be the program name
+	argc = nrhs + 1;
+
+	// allocate memory for the argument array
+	argv = new string[argc];
+
+	// set the first element of the array to be
+	// the passed in program name
+	argv[0] = name;
+
+	// convert each mxArray element to the char type
+	// and store the resulting string in the argument
+	// array.
+	for (int i = 1; i < argc; ++i) {
+		char* str = mxArrayToString(prhs[i-1]);
+
+		argv[i] = str;
+
+		// mxArrayToString allocates memory for the char*
+		// and it must be freed, usually as soon as it
+		// is no longer needed
```

```
+			mxFree(str);
+		}
+}
+
+void MatlabToCppArgument::convertStringPtrToCharPtrPtr() {
+
+		// allocated memory for the char** cargv
+		cargv = (char**)malloc(argc * sizeof(char*));
+
+		// for each element of cargv, allocate the necessary ammount of
+		// memory to hold the string
+		for (int i = 0; i < argc; ++i) {
+			cargv[i] = (char*)malloc((argv[i].length() + 1) * sizeof(char));
+			strcpy(cargv[i], argv[i].c_str());
+		}
+
+}
+
diff -rupN child/Code/matlabInterface/MatlabToCppArgument.h
CHILD_MATLAB/Code/matlabInterface/MatlabToCppArgument.h
--- child/Code/matlabInterface/MatlabToCppArgument.h	1969-12-31
19:00:00.000000000 -0500
+++ CHILD_MATLAB/Code/matlabInterface/MatlabToCppArgument.h	2015-07-24
20:06:59.630355966 -0400
@@ -0,0 +1,86 @@
+#include <iostream>
+#include <string>
+#include "mex.h"
+
+using namespace std;
+
+class MatlabToCppArgument {
+		public:
+			/*
+				Constructor setting name using c++ string class
+				prhs is a mxArray of matlab arguments
+				nrhs is the number of elements in prhs
+			 */
+			MatlabToCppArgument(int nrhs, const mxArray* prhs[], string n);
+
+			/*
+				Constructor setting name using c style char*
+				prhs is a mxArray of matlab arguments
+				nrhs is the number of elements in prhs
+			 */
+			MatlabToCppArgument(int nrhs, const mxArray* prhs[], const char* n);
```

```
+
+		/*
+			Destructor that cleans up allocated memory for the
+			argument string array
+		 */
+		~MatlabToCppArgument();
+
+		/*
+			Method to return the name set with the constructor.
+			The name is automatically set as the first element
+			of the argument string array.
+		 */
+		string getName() { return name; }
+
+		/*
+			Method to return the argument of the given index.
+			The index applies to the argument string array, not
+			the mxArray.  Therefore, to get the first argument
+			passed to the matlab function, use index 1 and not
+			index 0 as the first index is, in the c/c++ style,
+			the name of the program.
+		 */
+		string getArgument(int index) {return argv[index]; }
+
+		/*
+			Method to return a pointer to the argument string
+			array.
+		 */
+		string* getArguments() { return argv; }
+
+		/*
+			Method to return a pointer to the c style char**
+			argument array.
+		 */
+		char** getCArguments() { return cargv; }
+
+		/*
+			Method to return the number of arguments in the
+			string array, not the number of arguments passed
+			to the matlab function.  Therefore, if the matlab
+			function was passed no arguments this method will
+			return 1 and not zero as the first element of
+			the argument string array is the name of the
+			program as is the c/c++ style.
+		 */
+		int getNumArguments() { return argc; }
```

```
+
+	private:
+		string name;
+		string* argv;
+		int argc;
+		char** cargv;
+
+		/*
+			Function to create the argument string array from the
+			mxArray.
+		 */
+		void createArgumentList(int nrhs, const mxArray* prhs[]);
+
+		/*
+			Function to create the c style char** argument array from
+			the string array.
+		*/
+		void convertStringPtrToCharPtrPtr();
+};
diff -rupN child/Code/matlabInterface/objects/compile.sh
CHILD_MATLAB/Code/matlabInterface/objects/compile.sh
--- child/Code/matlabInterface/objects/compile.sh	1969-12-31 19:00:00.000000000 -
0500
+++ CHILD_MATLAB/Code/matlabInterface/objects/compile.sh	2015-07-24
20:06:59.630355966 -0400
@@ -0,0 +1,4 @@
+#/bin/bash
+
+#mex_matlab -c ../../errors/errors.cpp ../../tMesh/TipperTriangulatorError.cpp
../../tOption/tOption.cpp ../../tStratGrid/tStratGrid.cpp ../../tStreamMeander/meander.cpp
../../tStreamNet/tStreamNet.cpp CFLAGS="-fPIC -DMATLAB"
+mex -c ../../errors/errors.cpp ../../tMesh/TipperTriangulatorError.cpp
../../tOption/tOption.cpp ../../tStratGrid/tStratGrid.cpp ../../tStreamMeander/meander.cpp
../../tStreamNet/tStreamNet.cpp CFLAGS="-fPIC -DMATLAB"
diff -rupN child/Code/tInputFile/test_input_file.cpp
CHILD_MATLAB/Code/tInputFile/test_input_file.cpp
--- child/Code/tInputFile/test_input_file.cpp	2015-07-24 20:39:26.533699667 -0400
+++ CHILD_MATLAB/Code/tInputFile/test_input_file.cpp	2015-07-24
20:06:59.003695744 -0400
@@ -25,6 +25,10 @@ int main( int argc, char** argv )
   if( argc<2 )
   {
     cout << "Must include name of input file on the command line\n";
+#ifdef MATLAB
+	const char* c = "";
+	mexErrMsgTxt(c);
```

```
+#endif
    exit(0);
   }

diff -rupN child/Code/tMesh/TipperTriangulatorError.cpp
CHILD_MATLAB/Code/tMesh/TipperTriangulatorError.cpp
--- child/Code/tMesh/TipperTriangulatorError.cpp   2015-07-24 20:36:16.465633747 -
0400
+++ CHILD_MATLAB/Code/tMesh/TipperTriangulatorError.cpp  2015-07-24
20:06:59.017028941 -0400
@@ -15,6 +15,10 @@

 void tt_error_handler(void){
 #if defined(TIPPER_TEST)
+         #ifdef MATLAB
+                   const char* c = "";
+                   mexErrMsgTxt(c);
+         #endif
   exit(1);
 #else
   ReportFatalError( "Fatal error in Tipper Triangulator" );
diff -rupN child/Code/tOption/tOption.cpp CHILD_MATLAB/Code/tOption/tOption.cpp
--- child/Code/tOption/tOption.cpp    2015-07-24 20:38:57.963990440 -0400
+++ CHILD_MATLAB/Code/tOption/tOption.cpp  2015-07-24 20:06:59.617022771 -
0400
@@ -17,6 +17,10 @@
 #include "tOption.h"
 #include "../Definitions.h"

+#ifdef MATLAB
+         #include <mex.h>
+#endif
+
 tOption::tOption(int argc, char const * const argv[])
   : exeName(argv[0]),
     silent_mode(false), checkMeshConsistency(true), no_write_mode(false),
@@ -30,6 +34,10 @@ tOption::tOption(int argc, char const *
   }
   if (inputFile ==NULL){
     usage();
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
    exit(EXIT_FAILURE);
   }
```

```
 }
@@ -59,6 +67,10 @@ void tOption::ProcessOptionsFromString(
    parseOptions( s );
  if (inputFile ==NULL){
    usage();
+#ifdef MATLAB
+        const char* c = "";
+        mexErrMsgTxt(c);
+#endif
    exit(EXIT_FAILURE);
  }
 }
@@ -83,18 +95,34 @@ int tOption::parseOptions(char const * c
  }
  if (strcmp(thisOption, "--help") == 0){
    usage();
+#ifdef MATLAB
+        const char* c = "";
+        mexErrMsgTxt(c);
+#endif
    exit(EXIT_SUCCESS);
  }
  if (strcmp(thisOption, "--version") == 0){
    version();
+#ifdef MATLAB
+        const char* c = "";
+        mexErrMsgTxt(c);
+#endif
    exit(EXIT_SUCCESS);
  }
  if (thisOption[0] == '-') {
    usage();
+#ifdef MATLAB
+        const char* c = "";
+        mexErrMsgTxt(c);
+#endif
    exit(EXIT_FAILURE);
  }
  if (inputFile != NULL)  {
    std::cerr << exeName << ": Several input files given." << std::endl;
+#ifdef MATLAB
+        const char* c = "";
+        mexErrMsgTxt(c);
+#endif
    exit(EXIT_FAILURE);
  }
```

```
@@ -119,18 +147,34 @@ int tOption::parseOptions(string thisOpt
   }
   if (thisOption.compare("--help") == 0){
     usage();
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
     exit(EXIT_SUCCESS);
   }
   if (thisOption.compare("--version") == 0){
     version();
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
     exit(EXIT_SUCCESS);
   }
   if (thisOption[0] == '-') {
     usage();
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
     exit(EXIT_FAILURE);
   }
   if (inputFile != NULL)  {
     std::cerr << exeName << ": Several input files given." << std::endl;
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
     exit(EXIT_FAILURE);
   }

diff -rupN child/Code/tStratGrid/tStratGrid.cpp
CHILD_MATLAB/Code/tStratGrid/tStratGrid.cpp
--- child/Code/tStratGrid/tStratGrid.cpp        2015-07-24 20:40:50.582844125 -0400
+++ CHILD_MATLAB/Code/tStratGrid/tStratGrid.cpp       2015-07-24
20:06:59.617022771 -0400
@@ -26,6 +26,11 @@

 #include <iostream>

+#ifdef MATLAB
```

```
+         #include <mex.h>
+#endif
+
+

/***********************************************************************
***\
  **                                                           tSTRATGRID
  **  @fn tStratGrid( tInputFile &infile, tMesh<tLNode *mp)
@@ -549,6 +554,10 @@ void tStratGrid::CheckSectionBase(int mo
          }
          std::cout<<"Number of layers is "<<numlayers<<", Total thickness is:
"<<totalthickness<<std::endl;
          if(totalthickness > 0.0){
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
            exit(1);
          }
      }
@@ -1322,6 +1331,10 @@ double tStratNode::getAgeAtDepth( double
          std::cout<<"l-1= "<<l-1<<" Rtime = "<<getLayerRtime(l-1)
             <<"Ctime= "<<getLayerCtime(l-1)<<" "<<std::endl;
          std::cout<<"Rsed = "<<Rsed<<std::endl;
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
          exit(1);
      }

@@ -1664,6 +1677,10 @@ void tStratNode::EroDepSimple( int l, tA
            else if(remainder <= maxregdep+10.){
              std::cout<<"Loads of deposition at " << x <<' '<< y <<" time= " <<tt<< '\n';
              std::cout<<"Make extra layers ??? \n";
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
             exit(1);
           }

@@ -1798,6 +1815,10 @@ void tStratNode::EroDepSimple( int l, tA
      std::cout<<"z-columnheight= "<<z-columnheight_after<<"diff = "<<
(sectionBase_after)- (z-columnheight_after)<<std::endl;
```

```
      std::cout<<" "<<std::endl;
      std::cout<<" "<<std::endl;
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
      exit(1);
    }
  }
diff -rupN child/Code/tStreamMeander/meander.cpp
CHILD_MATLAB/Code/tStreamMeander/meander.cpp
--- child/Code/tStreamMeander/meander.cpp2015-07-24 20:41:26.422479260 -0400
+++ CHILD_MATLAB/Code/tStreamMeander/meander.cpp         2015-07-24
20:06:59.613689472 -0400
@@ -17,6 +17,10 @@
 #include "meander.h"
 #include "../Definitions.h"

+#ifdef MATLAB
+         #include <mex.h>
+#endif
+
 #define integer int
 #define doublereal double

@@ -483,6 +487,10 @@ void getcurv_(const integer *stnserod, c
          b = dels[s - 1];
          if (a == 0. || b == 0.) {
            std::cout << "dels(s) or dels(s-1) equals zero" << std::endl;
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
            exit(1);
          }
          c__ = sqrt((delx[s - 1] + delx[s]) * (delx[s - 1] + delx[s]) + (dely[
diff -rupN child/Code/tStreamNet/tStreamNet.cpp
CHILD_MATLAB/Code/tStreamNet/tStreamNet.cpp
--- child/Code/tStreamNet/tStreamNet.cpp    2015-07-24 20:36:27.028859616 -0400
+++ CHILD_MATLAB/Code/tStreamNet/tStreamNet.cpp  2015-07-24
20:06:59.697021948 -0400
@@ -21,6 +21,10 @@
 #include "../errors/errors.h"
 #include "tStreamNet.h"

+#ifdef MATLAB
```

```
+         #include <mex.h>
+#endif
+
 tStreamNet::kChannelType_t tStreamNet::IntToChannelType( int c ){
   switch(c){
     case 1: return kRegimeChannels;
@@ -1523,6 +1527,10 @@ void tStreamNet::FlowDirs()
          std::cout<< "Voronoi area <  0.0 at \n";
          std::cout<< curnode->getX() <<' '<<curnode->getY()<<std::endl;
          std::cout<< "Area= "<<curnode->getVArea()<<std::endl;
+#ifdef MATLAB
+         const char* c = "";
+         mexErrMsgTxt(c);
+#endif
          exit(1);
        }
```

BIOGRAPHY OF THE AUTHOR

Raymond Forrest Flagg III was born in Augusta, Maine on June 6, 1989. He received his high school education from Presque Isle High School located in Presque Isle, Maine in 2007. He graduated magna cum laude from the University of Maine in 2012 with a Bachelor of Science degree in Computer Engineering.

Raymond is a candidate for the Master of Science degree in Computer Engineering from the University of Maine in August 2015.