Parallel Implementation of Facial Detection Using Graphics Processing Units

Russell Marineau
Marquette University

Recommended Citation
https://epublications.marquette.edu/theses_open/506
PARALLEL IMPLEMENTATION OF FACIAL DETECTION
USING GRAPHICS PROCESSING UNITS

by

Russell L. Marineau, B.S.

A Thesis Submitted to the Faculty of the Graduate School
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin
December 2018
This thesis proposes to study parallelization methods to improve the computational runtime of the popular Viola-Jones face detection algorithm. These methods employ multithreaded programming and CUDA programming approaches. The thesis provides a discussion of background information on all relevant topics, which is then followed by a presentation of the code architecture changes that are proposed. Specific implementation details are then discussed in more details followed by a discussion and comparison of results obtained through various tests.

This thesis first begins by presenting a history and description of the Viola-Jones algorithm. Detailed explanations of each step in the process used to detect a face are provided. Next, background information about parallel processing is provided. This includes both standard multithreaded program design as well as CUDA programming. New algorithm design methods that employ parallelization techniques will then be proposed to improve over the original Viola-Jones algorithm. These techniques include both multithreading and CUDA programming, whose potential advantages and disadvantages are discussed as well. Implementations of these new algorithms will be provided next as well as a detailed explanation of the functionality used.

Finally, this thesis will provide test results for all algorithm versions, including the original algorithm as well as a comparison and possible future improvements. Simulation results indicate that the multithreaded algorithm was able to provide a maximum of 7.8x speedup over the original version when running on 16 processing cores. The CUDA version algorithm was able to provide a maximum of 47x speedup over the original version. After exploring more detailed results and comparisons, it was determined that each version has advantages and disadvantages. The multithreaded version was much simpler to code and would run on a wider range of hardware, however the CUDA version was significantly faster. In addition, the CUDA version has much room for future optimizations to further increase the speed of the algorithm.
ACKNOWLEDGEMENTS

I would first like to thank my parents for supporting me and encouraging my passion for software engineering and computers in general. I would also like to thank my girlfriend for her support of my work and endless patience throughout the entire process. Secondly, I would like to thank my advisor, Dr. Cris Ababei without whom, this thesis would not have been possible. Finally, I would like to thank my committee members, Dr. Henry Medeiros and Dr. Richard Povinelli for reviewing my thesis and provide valuable feedback.
# TABLE OF CONTENTS

**ACKNOWLEDGEMENTS** .......................................................... i

**TABLE OF CONTENTS** .......................................................... ii

**LIST OF TABLES** ............................................................. iv

**LIST OF FIGURES** ............................................................ v

**CHAPTER 1** Problem Statement, Objective and Contributions .......................... 1

1.1 Problem statement ............................................................ 1

1.2 Objectives ................................................................. 2

1.3 Contributions ............................................................. 2

1.4 Thesis Organization ........................................................ 3

**CHAPTER 2** Background on Viola-Jones Face Detection Algorithm ...................... 4

2.1 Background Information .................................................. 4

2.2 Basics of Viola-Jones Algorithm ........................................ 5

2.2.1 The Integral Image .................................................. 5

2.2.2 Image Features .................................................... 7

2.2.3 Cascade of Classifiers ............................................. 9

2.2.4 The Sliding Window ................................................. 9

2.3 Related Work ............................................................ 11

2.4 Summary ................................................................. 14

**CHAPTER 3** Parallel Processing Techniques ....................................... 15

3.1 Basics of Parallel Processing ........................................... 15

3.2 Multithreading ........................................................... 15

3.3 GPGPU ................................................................. 17

3.4 CUDA Programming .................................................... 18

3.5 Comparison of CPU Processing and GPGPU ............................ 24

3.6 CUDA C vs Standard C++ .............................................. 24

**CHAPTER 4** Parallelization Approaches of the Viola-Jones Face Detection Algorithm ...................... 28
4.1 Original Design .............................................. 28
4.2 Design of Multithreaded Face Detection Algorithm .............. 32
4.3 Design of CUDA Face Detection Algorithm ....................... 33
4.4 Summary ................................................... 34

CHAPTER 5 Implementation Details ................................ 44
5.1 Implementation of Multithreaded Face Detection Algorithm ..... 44
5.2 Implementation of CUDA Face Detection Algorithm ............. 47
  5.2.1 Initial Considerations and Preparatory Work ................. 47
  5.2.2 Object Creation and Allocation Decisions ................... 50
  5.2.3 Kernel Implementation .................................. 51
5.3 Summary ................................................... 52

CHAPTER 6 Discussion of Results .................................. 54
6.1 Initial CUDA Test Results .................................... 55
6.2 Results from Tests for Different Image Resolutions with Constant
  Number of Faces ............................................... 60
6.3 Results from Tests on Images with Different Numbers of Faces and
  Constant Resolution .......................................... 63
6.4 Video Testing ................................................ 67
6.5 Further Observations ......................................... 68
6.6 Summary ................................................... 69

CHAPTER 7 Conclusion and Future Work ............................. 71
7.1 Conclusions ................................................ 71
7.2 Future Work ............................................... 74

REFERENCES .................................................. 76
LIST OF TABLES

6.1 Difference in processing speed for several test images. . . . . . . . . . . . 55
6.2 Time to process a single image vs the resolution of the image. . . . . . 60
6.3 Relative speedup obtained by each version of the program when pro-
    cessing differing image resolutions. . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
6.4 Time to process a single image vs the number of detected faces in the
    image. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.5 Speedup obtained by different implementations when testing differing
    numbers of detected faces in the image. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
## LIST OF FIGURES

2.1 Base image array with corresponding integral image array ............. 6  
2.2 Four different types of features. ......................................... 8  
2.3 Two different Haar-like features applied to a face ..................... 8  
2.4 Example of a cascade of classifiers. ...................................... 9  
2.5 Example of a scanning window in a test image .......................... 10  
3.1 Parallel vs. sequential processing. ....................................... 16  
3.2 CUDA programming structure showing Grids, Blocks, and Threads .... 19  
3.3 Simple kernel example. .................................................... 20  
3.4 Example of a simple kernel call ......................................... 21  
3.5 CUDA programming structure showing layout of memory on a GPU .... 22  
3.6 C++ sample code that computes the squares of the first 100 million integers. .................................................. 25  
3.7 CUDA sample kernel to compute the square of one array and save it into another array. ............................................. 26  
3.8 CUDA sample code that computes the squares of the first 100 million integers. .................................................. 27  
4.1 Basic structure of the face detection program ............................ 29  
4.2 Main program structure ..................................................... 29  
4.3 Structure of the face detection function. This diagram represents the process in the block labeled “Detect Faces Using Cascade Classifier” in Fig. 4.1 .................................................. 30  
4.4 Initial object detection structure ......................................... 31  
4.5 Structure of the classifier function. This diagram represents the process in the block labeled “Invoke Cascade Classifier” in Fig. 4.3 .................................................. 35  
4.6 Initial classifier structure ................................................... 36  
4.7 First proposal for a multithreaded optimization of the object detection function .................................................. 37
4.8 First proposal for a multithreaded optimization of the object detection function pseudocode. .................................................. 38

4.9 Number of steps performed by each thread. The variation is due to each iteration having a different scale factor. .......................... 39

4.10 Second proposal for a multithreaded optimization of the object detection function. .......................................................... 40

4.11 Second proposal for a multithreaded optimization of the object detection function pseudocode. .............................................. 41

4.12 Proposal for a CUDA replacement for the “invoke cascade classifier” function. ................................................................. 42

4.13 Proposed CUDA solution. ...................................................... 43

5.1 Original implementation of classifier invoker. ............................. 45

5.2 New implementation of classifier invoker. ................................... 45

5.3 Thread work method. ............................................................ 46

5.4 Copy data using cudaMemcpy. ................................................. 47

5.5 Copy data using unified memory. ............................................. 48

5.6 CUDA managed class. .......................................................... 49

5.7 CUDA constant variables defined. ............................................ 50

5.8 CUDA constant variables copied to. ........................................ 51

5.9 CUDA objects allocated in global memory. ............................... 51

5.10 CUDA kernel preparation and call. ........................................ 52

5.11 CUDA kernel implementation. .............................................. 53

6.1 Test image of a parade. Only a few faces detected most likely due to people not facing directly into the camera as well as wearing hats. . . 56

6.2 Test image of a family. 5 out of 6 faces detected. Most likely, the last face was not detected due to being tilted at an angle. .............. 56

6.3 Test image of a family. 8 out of 12 faces detected. Most likely, the last few faces were not detected due to being tilted at an angle. ....... 57

6.4 Test image of a family. 10 out of 19 faces detected. Most likely, only about half of the faces were detected due to the somewhat low quality of the image. ........................................ 57
6.5  Test image of a family. All 6 faces detected. .................................. 58

6.6  Test image of a family. All 8 faces detected with the addition of one false positive. .......................................................... 59

6.7  Time to process a single image vs the resolution of the image. ...... 62

6.8  Relative speedup obtained by each version of the program when processing differing image resolutions. ............................... 64

6.9  Time to process a single image vs the number of detected faces in the image. ................................................................. 66

6.10 Speedup obtained by different implementations when testing differing numbers of detected faces in the image. .................... 67
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU:</td>
<td>Central Processing Unit.</td>
</tr>
<tr>
<td>GPU:</td>
<td>Graphics Processing Unit.</td>
</tr>
<tr>
<td>GPGPU:</td>
<td>General Purpose Graphics Processing Unit.</td>
</tr>
<tr>
<td>SIMD:</td>
<td>Single Instruction Multiple Data.</td>
</tr>
<tr>
<td>MIMD:</td>
<td>Multiple Instruction Multiple Data.</td>
</tr>
<tr>
<td>CUDA:</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>TFLOP:</td>
<td>Teraflop - One trillion floating point operations per second</td>
</tr>
</tbody>
</table>
CHAPTER 1

Problem Statement, Objective and Contributions

1.1 Problem statement

This thesis proposes to improve facial detection speed using a Haar cascade face detection algorithm using CUDA programming. It will first provide background information on the advantages of parallel processing, specifically General-Purpose Graphics Processing Unit (GPGPU) code for speeding up parallelizable tasks. The remainder of the thesis focuses on how to meet the proposed performance goals. A description of the function of the current Haar cascade system is provided. Two new algorithms are then proposed to improve the performance of the existing system. The first is a multithreaded optimization exploiting the capabilities of current generation multicore CPUs implemented in C++14, and the second is a massively parallel algorithm using CUDA, programming language for Nvidia’s GPGPUs. The thesis will then analyze and compare the performance results of the algorithms and suggest future optimizations and improvements [1].

The primary problem this thesis attempts to solve is to improve the performance of an existing single threaded algorithm for face detection and demonstrate the increasing utility of GPGPU programming. Historically, most programs were written to only utilize a single thread. This was a reasonable approach at the time since most processors only contained a single hardware core
for processing instructions. Most current generation processors contain at least two cores, even in mobile devices such as phones or tablets. If a computer program is written such that it is executed by a single thread, it potentially would utilize only half or less of the performance potential of the modern processor. Therefore, in the present, an efficient program should be written for as many threads as possible in order to utilize the full potential of the processor and to gain computational speed.

1.2 Objectives

A main objective of this thesis is to provide a functional example of the performance potential of CUDA programming on graphics processing units. This program will be modified from an existing Haar cascade classifier face recognition algorithm known as the Viola-Jones algorithm. The original program was only single threaded, and a performance comparison will be provided between the single threaded version, a multi-threaded version, and a CUDA version. The CUDA version will provide a base onto which additional performance and functional improvements can be made since CUDA is a more recent development in comparison to standard multithreading.

1.3 Contributions

This thesis provides an insight into the considerations that must be taken when converting a single threaded algorithm to CUDA. The key contributions of this project are as follows:
• Provide open source multithreaded implementation of an existing single threaded face detection algorithm.

• Provide also an open source CUDA programming based implementation of the same face detection algorithm.

• Describe in details the parallelization process for both multithreaded and CUDA programming approaches.

• Provide a comparison of the performance achieved by the multithreaded and CUDA implementations.

• Discuss potential future improvements to the CUDA implementation.

1.4 Thesis Organization

Chapter 2 presents an overview of the Viola-Jones face detection algorithm and describes previous parallelization attempts. Chapter 3 provides background information about multithreading and CUDA programming as applicable to this project. Chapter 4 describes possible approaches for the improvement of the Viola-Jones algorithm processing speed through multithreading with multicore CPUs and CUDA on GPUs. Chapter 5 details the specific implementation of each of the two approaches. Chapter 6 gives a comparison of the results determined with each version of the program during testing. Chapter 7 draws conclusions from the provided results and provides suggestions for future work on the algorithm with CUDA.
CHAPTER 2

Background on Viola-Jones Face Detection Algorithm

This chapter provides background information on the Viola-Jones face detection algorithm. Increasing the speed of this algorithm from the provided baseline of a single threaded implementation is the main goal of this thesis. Previous attempts at improving the Viola-Jones algorithm are also discussed in this chapter.

2.1 Background Information

The Viola-Jones face detection algorithm was developed in order to provide a method of quickly detecting faces in a provided image. Previous attempts at object recognition were very computationally expensive and used RGB values at every pixel in the image. This algorithm was novel in that it uses an integral image to provide extremely fast detection. The algorithm relies on machine learning concepts to determine common features present in the object being detected, most commonly faces. An important note is that this algorithm does not implement facial recognition, only facial detection. This is useful as a first layer in a recognition algorithm in which the Viola-Jones algorithm is used to quickly detect possible faces before a second slower algorithm is used to identify the face.
2.2 Basics of Viola-Jones Algorithm

The Viola-Jones algorithm starts by creating an integral image. This image is then used to detect features present in a particular window. The window is then moved across the image to detect all possible features of that size. The window is then scaled to a different size and the process repeats. All of these steps are described in detail in the following sections.

2.2.1 The Integral Image

The Viola-Jones algorithm works by first creating an integral image, giving every pixel a value requiring only a minimal number of references to other locations in the image [1]. This integral image is also known as a summed area table. It consists of an array of integers of the same size as the array of pixels that make up the image. Each value in the integral image is equal to the sum of the intensity values of all the pixels above and to the left of it in the image as shown in equation 2.1. In this formula, $i$ is the value of the intensity at given coordinates and $I$ is the value of the integral image at given coordinates. The intensity value of a pixel lies on a scale of 0 (black) to 255 (white). An efficient single pass solution to create the integral image is shown in equation 2.2, which is used in the Viola-Jones algorithm.

$$I(x, y) = \sum_{x'\leq x} \sum_{y'\leq y} i(x', y')$$  \hspace{1cm} (2.1)
The advantage of using an integral image is that it allows the algorithm to calculate the sum of intensities in a particular rectangle in the image. An example of this is shown in Fig. 2.1. This example represents a 10-pixel by 10-pixel image. The desired sum is the rectangle highlighted in Fig. 2.1(a). To calculate the sum of the intensities of the given rectangle without an integral image, the algorithm would have to add up the intensities of each pixel as shown in equation 2.3. This calculation would require 42 references to locations in the image array as well as 41 math operations. After an integral image has been created, the sum can be calculated as shown in Fig. 2.1(b) and equation 2.4. As can be seen, this calculation only requires 4 references and 3 math operations making it significantly faster. When this type of sum operation is required to be

\[ I(x, y) = i(x, y) + I(x, y - 1) + I(x - 1, y) - I(x - 1, y - 1) \] (2.2)
performed many times for many different bounding rectangles, the difference in
processing time is significant.

\[ SUM = 23 + 24 + 25 + \ldots + 78 + 79 = 2142 \]  \hspace{1cm} (2.3)

\[ SUM = 2880 + 26 - 584 - 180 = 2142 \]  \hspace{1cm} (2.4)

2.2.2 Image Features

After the integral image is created, features are detected in a rectangular
window over a portion of the image. Feature examples are shown in Fig. 2.2.
The first two examples are features corresponding to two rectangles, one dark
and one light. The second and third represent three and four rectangle features
respectively. The features are simple representations of the difference between a
dark section of an image and a lighter section.

A simple example of what a feature might look like as detected by a
classifier in a real image is shown in Fig. 2.3. The feature shown in Fig. 2.3(c)
is used to estimate the difference in pixels’ gray levels between the area covering
the eyes and the area below the eyes. A face typically has pixels with lower
intensity (darker) along the eye line and pixels with higher intensity (brighter)
below. Fig. 2.3(b) shows another possible pattern in which white part of the
rectangle template covers the area between the eyes, which consist of brighter
pixels. The two black parts of the rectangular template cover the eye area, which consist of darker pixels [1].
2.2.3 Cascade of Classifiers

The detection of features, and thus a face, is accomplished by using a cascade of weak classifiers. A classifier is considered weak when it is computationally inexpensive and relatively inaccurate, but with an accuracy at least somewhat greater than random. These weak classifiers can then be cascaded together to form a complete face detection algorithm for a given window of an image with each additional classifier increasing the accuracy of the algorithm. Sub-windows of the image are rejected at each phase of the cascade, reducing the number of sub-windows that must be processed by the next stage as shown in Fig. 2.4. In this way, the algorithm can have a much shorter runtime than an algorithm with similar accuracy using a single strong classifier.

2.2.4 The Sliding Window

After features have been detected in a given window, the window is moved to a new location in a scanning pattern until all locations in the image have been
covered. This process is shown in Fig. 2.5. After this, the size of the window is changed and the process starts again in order to detect faces of different sizes.

After the cascade has processed all windows of all sizes, the windows with positive results for faces after passing all classifiers in the cascade have been added to a single list. This list is then processed to combine windows that are
determined to have detected the same face resulting in a final list of detected faces. The end result is a very fast method for face detection [1].

2.3 Related Work

The Viola-Jones face detection algorithm is still one of the more prominent face detection algorithms studied due to its simplicity. Several algorithms have now surpassed its speed and are in use in production environments, but a large body of research work is still conducted based on the Viola-Jones algorithm. Much of the research effort has been focused on improving the accuracy of the algorithm since it sacrificed a small amount of accuracy for a large speed gain over algorithms at the time. The authors of [2] attempted to improve the accuracy of the algorithm by passing images through a pre-processing filter before the Viola-Jones algorithm. Their results indicate that for the most part, it is harmful to apply any sort of filter to the image before processing through the Viola-Jones algorithm.

Another attempt at improving the accuracy of the algorithm focuses on pre and post processing using a different method [3]. The authors proposed to first filter out unimportant sections of the image using skin color filtering as well as salient object extraction. Skin color filtering simply selects the area of the image which most likely corresponds to a face by using the color in a particular area. Skin color detection suffers from a large number of false negatives however, so that method is combined with salient object extraction and the two results are combined. Salient object extraction chooses the areas of the image where the
most interesting features are located. The result is then passed through the
Viola-Jones algorithm and then once more through the skin color filter. This
resulted in a significant decrease in both the false negative rate and the false
positive rate bringing both below ten percent.

Although much work has been focused on the accuracy of the Viola-Jones
algorithm, there have been many attempts to increase its speed as well using
many different approaches. One of these approaches seeks to optimize the
algorithm for mobile devices by further simplifying the calculations required [4].
It approaches this problem by assuming that the program will run on weak
hardware and reducing the complexity of the algorithm to compensate. The
three initial types of optimizations were reducing the size of the image through
subsampling before detection, increasing the step size, and increasing the scale
factor all in an effort to reduce the number of steps required. In addition, the
authors set a minimum face size for detection. The authors were able to achieve
a massive increase in throughput implementing all of these optimizations on
reduced hardware.

There are also several approaches to increasing the speed of the
Viola-Jones algorithm through dedicated hardware. One such approach
synthesized a hardware design using Verilog HDL and ModelSim and were able
to achieve 52 frames per second using a 90nm architecture and a 500 MHz clock
cycle [5]. Another design using Verilog HDL as well but implemented in a
physical FPGA for testing was able to achieve 16 frames per second with an 8
stage classifier [6]. While some of these hardware implementations are very fast,
they are not particularly portable. They only operate on the hardware they were
designed for putting them at a disadvantage compared to optimized CPU or
GPU algorithms.

There have been several attempts to improve the Viola-Jones algorithm
through incorporating GPU processing. One of the most successful attempts was
able to get an average speed up of 23x across various resolutions ranging from
340x240 to 1280x1024 [7]. The method that was used was to assign one detection
window to one GPU thread. As with many other implementations, this one used
several functions present in the OpenCV open source library including the
pre-trained frontal-face cascaded classifiers [8]. This approach also implemented
the skin color filtering method mentioned earlier. Another similar implementation
was able to achieve a maximum speedup of 22x for a 640x480 image using a
Nvidia Tesla K40 [9]. This implementation also incorporated diagonal features as
well as the simple features originally proposed in the Viola-Jones algorithm to
increase accuracy. A third implementation also on GPUs was able to achieve a
12.615x speedup over a high-end Intel Xeon CPU using a Tesla C2050 [10]. Most
of these acceleration attempts were done several years ago meaning that the
CUDA versions and capabilities of the GPUs are significantly out of date. We
believe that there is significant room for improvement with more modern CUDA
programming techniques as well as modern GPUs.

While many of these speed increases can sound dramatic, many previous
works do not specify details about the configuration of the cascade classifier
beyond simple parameters. The number of classifiers in the cascade as well as the
configuration of the cascade can dramatically impact performance. Because of this, it is hard to draw a direct comparison among the performance results of individual works as well as the performance gained by our implementation.

2.4 Summary

In this chapter, we discussed the background of the Viola-Jones algorithm which was designed to increase the speed of face detection while sacrificing very little accuracy. This algorithm was then described in detail, explaining how each step of the algorithm was processed by the CPU. Some related works on improving the speed and accuracy were also explored to provide a reference point for our improvements to the algorithm.
CHAPTER 3

Parallel Processing Techniques

This chapter provides an overview of parallel processing to provide a framework for understanding the design concepts discussed later in this thesis. A basic overview of parallel processing as a whole will be provided followed by an in-depth explanation of the different types of parallel processing as well as their advantages and disadvantages.

3.1 Basics of Parallel Processing

Many computational problems in today’s world can be split up into many smaller problems that can all be solved simultaneously. Historically, computers have used one processor to solve all of these problems sequentially. With the advent of multicore processors, GPUs, and other types of parallel processors, it is now possible for a computer to execute all of these smaller sub-problems at the same time, increasing the efficiency of the program, thereby decreasing the execution runtime.

3.2 Multithreading

The simplest way to allow a program to take advantage of multiple processing cores is to simply split different tasks up manually and run them on different threads on the CPU. A simple and common example is running
computationally expensive tasks on a separate thread than the user interface in order to keep the user interface responsive while allowing background processing to continue. This type of multithreading is relatively simple to accomplish and using each thread to its full potential allows for a linear speed increase in processing speed, up to the number of cores in a current generation desktop computer which ranges from two to eight, however high end processors are available with up to 32 physical cores [11][12].

The speedup inherent in parallel processing can be significant and becomes greater with the ability to split a process into many smaller tasks. This is shown in Fig. 3.1 in which a program is split into "Task 1" and "Task 2". In this case, the parallel version in which each task is given its own processing element runs twice as fast as the sequential version in which both tasks run on the same
processing element. This is an ideal scenario, however in other examples, the speedup will not be quite as dramatic. This example assumes both tasks have the same runtime and do not depend on each other in any way. This is not usually the case in most real-world examples since cross thread communication is usually needed, requiring some form of thread synchronization in which one thread must pause and wait for the result from another thread. When this model is expanded to three, four, or more threads, this problem is exacerbated resulting in slowly diminishing returns on the overall processing speed up for adding each thread. Therefore, efficient algorithms for parallelization are needed in order to obtain significant improvements in performance.

3.3 GPGPU

A GPU is a specialized processor used by a computer in order to accelerate graphics output. Graphics processing consists almost completely of highly parallelizable operations. Because of this, GPUs are designed with hundreds or thousands of simple stream processors which are classified as SIMD. SIMD processors can perform a single function on multiple sets of data at the same time. Since individual cores in a SIMD processor can be much simpler than a core in a standard CPU, more can be included in one processor occupying less space, consuming less power, and at a lower cost. Recently, GPGPU have become more commonly used to solve problems that are highly parallelizable even when the problems are not graphics related. Several programing techniques have been introduced to write code that will execute on GPUs to take advantage
of their highly parallelized nature. The most popular of these languages is CUDA programming, and OpenCL is another example.

3.4 CUDA Programming

Initially, GPU’s were only used for their graphics display capabilities. Around 2003, researchers started to use GPU’s to speed up parallelizable program execution. At the time, it was difficult for someone to use a GPU for general calculations since the program would have to be converted to use either Direct3D or OpenGL, which were the two available graphics APIs at the time. Nvidia and ATI, the two main GPU vendors at the time, saw this potential alternative use for their devices, and Nvidia’s solution was to release CUDA [13]. CUDA is an extension to the C++ programming language provided by Nvidia for programming their graphics processing units [14]. It consists of many basic functions allowing for direct control over the GPU as well as several libraries providing access to preprogramed operations commonly used in massively parallel operations.

The architecture of an Nvidia GPU as it relates to CUDA programming is shown in Fig. 3.2. A thread is the basic unit of execution as in standard multithreaded programs, however a thread in CUDA is very different then a thread that would be executed on the CPU. In CUDA, a thread consists of a single data element to be processed since all threads must execute the same instructions due to the limitation of the GPU being a SIMD device, whereas a CPU thread can execute different instructions than other running threads. The threads are created with several different levels of granularity. The first of these
is a warp which consists of 32 threads. A warp is the minimum number of threads that will execute simultaneously. This means that if the programmer only defines enough data for 10 threads, 32 threads worth of GPU power will still be consumed [13].

The next unit of execution is called a block, which can contain anywhere from 64 to 1024 threads, the number of which are dependent on the hardware being used. The number of threads in a block is configurable by the programmer within the hardware limitations. The final unit of execution is called a grid which is made up of multiple blocks. A grid contains all threads which will be executed by a single kernel. Each kernel must be called by the programmer from the host with a set of configuration parameters indicating the block size and the number of blocks [15].
The code shown in Fig. 3.3 demonstrates a very simple CUDA kernel example in C++. It looks very similar to a standard function with a few exceptions. The first notable exception is the use of the "__global__" identifier before the function definition. This indicates that the function is intended to and can only be used to launch a new kernel. Line 4 displays the next significant departure from standard C++. When a kernel is launched, one copy of vectorAdd is called for each data element provided. The element to be processed is determined by calculating the unique ID of the thread being executed. Line 4 accomplishes this through the use of the blockDim, blockIdx, and threadIdx values. The blockDim variable represents the size of the block and the blockIdx variable represents the location of the block within the grid. Multiplying these two variables together gives the id of the first thread in the block. The threadIdx variable represents the location of the thread within the block, so it is added to the previous product giving the unique thread id. Line 6 checks to ensure that the GPU does no calculations beyond the end of the given arrays. This is necessary because a GPU can only process threads in multiples of the warp size. If the data does not fit exactly in that number of threads, there will be some threads at the end of the kernel launch that have no data to process. Attempting
The code shown in Fig. 3.4 demonstrates an example of launching the defined kernel. The number of threads per block and blocks per grid must first be defined by the programmer. The optimal number of threads per block is highly dependent on the code running in the kernel and the hardware being used and will vary from kernel to kernel. The number of blocks per grid is simply the total number of elements divided by the threads per block plus one additional block for any remaining threads that do not fit exactly into increments of the block size. The kernel is then launched similarly to a regular function call except for passing in the block size and grid size as shown in line 3 from Fig. 3.4. The final line gets the status of the kernel after it has finished execution and will contain a success value if the kernel was successful or a description of whatever error was encountered.

Fig. 3.5 shows a schematic representation of the memory organization available to the GPU. Each type of memory has advantages and disadvantages as well as different intended uses. The lowest level memory available to the GPU are registers and local memory. These are only accessible by single threads within a kernel. They are the fastest memory available to the programmer and a limited amount is available per block. This means that if each thread needs a large
Figure 3.5: CUDA programming structure showing layout of memory on a GPU.

quantity of registers, fewer threads can be launched within a single block. Shared memory is shared amongst all of the threads within a block. This memory is useful for calculations in which the results or inputs must be shared amongst a block of threads but not amongst all blocks within a kernel. Registers and shared memory are the fastest memory available to a CUDA programmer and should be
used whenever possible. The main limitation is the relatively small quantity of registers and shared memory available [15].

There are also a number of other types of memory available. Global memory is one of these, which can be accessed by any thread and is also the easiest memory to store and retrieve values from. Global memory is the slowest of the other memory types, but also the largest in size. Global memory access can be somewhat optimized by coalescing memory accesses. This is accomplished by having a thread with index i access the element of an array with index i. This allows the GPU to access all memory for a given warp in one single instruction. If instead thread i needs data from location i+x, the memory locations are not all adjacent and the GPU must break up memory reads into a number of operations, significantly slowing down memory access.

Constant and texture memory both have specific use cases in which each is faster than global memory. Constant memory is best for when all threads within a warp must access the same data. If every thread needs access to a value x in the same instruction, it can be retrieved once and broadcast to all threads in the same operation. Texture memory is optimized for 2D spatial storage and is arguably the most complicated memory type to optimize correctly. Texture memory can coalesce memory accesses for values in a matrix that are close to each other. An example is if the values arr[x][y], arr[x+1][y], arr[x][y+1], and arr[x+1][y+1] are required by four different threads in a warp, texture memory may be able to combine those into a single memory access.
3.5 Comparison of CPU Processing and GPGPU

Standard CPUs and GPUs are very different in both design and intended function. CPUs are MIMD devices and are designed to be able to run a wide range of programs using a few high-speed processor cores. GPUs are SIMD devices that are designed to perform a single calculation on large amounts of data simultaneously. Each individual physical execution unit in the Tesla GPU is only as fast in terms of frequency as each in the Core i7 CPU, however the Tesla has over 700 times as many cores as the Core i7. In addition, the memory bandwidth of the Tesla is much higher and the overall Floating-Point Operations per Second (FLOPS) is over six times faster. These benefits are however, only enjoyed when executing a highly parallelizable task.

3.6 CUDA C vs Standard C++

The benefits of GPGPU can be shown through a simple piece of C++ code that computes the squares of the first 100 million integers stored in vectors. The first code segment shown in Fig. 3.6 is the implementation in standard C++. The next code segment shown in Fig. 3.7 and Fig. 3.8 performs the exact same calculation as the standard C++ code but utilizes the GPU through CUDA C.

Running on an i7 CPU at 4 GHz, the code took 4.58 seconds to execute. Running on a GTX 670 GPU at 915 MHz, the code took 1.58 seconds to execute. This means that for this particular calculation and implementation, the GPU was able to perform the calculation 3.5x faster than the CPU. The difference between
```c++
int _tmain(int argc, TCHAR* argv[])
{
    clock_t tStart = clock();
    // Print the vector length to be used, and compute its size
    int numElements = 100000000;
    size_t size = numElements * sizeof(double);
    printf("[Vector squares of %d elements]\n", numElements);
    // Allocate the host input vector A
    double *h_A = (double *)malloc(size);
    // Allocate the host output vector B
    double *h_B = (double *)malloc(size);
    // Verify that allocations succeeded
    if (h_A == NULL || h_B == NULL)
    {
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
    }
    // Initialize the host input vectors
    for (int i = 0; i < numElements; ++i)
    {
        h_A[i] = i;
    }
    // Calculate the squares
    for (int i = 0; i < numElements; ++i)
    {
        h_B[i] = pow(h_A[i], 2);
    }
    // Free host memory
    free(h_A);
    free(h_B);
    // Reset the device and exit
    printf("Time taken: %.2fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
    printf("Done\n");
    getchar();
    return 0;
}
```

Figure 3.6: C++ sample code that computes the squares of the first 100 million integers.

A CPU and GPU in terms of processing efficiency varies widely from program to program. Some programs will run slower when compiled on a GPU due to not being parallelizable and the GPU having slower cores. There are also cases when the GPU may execute code over 300x faster than a CPU would be able to. Because of this, determining if it is worthwhile to use GPGPU is dependent on the algorithm being used in the program.
Computes the square of the doubles stored in A. The 2 vectors have the same number of elements numElements.

```c
/**
 * CUDA Kernel Device code
 */
__global__ void vectorAdd(const double *A, double *C, int numElements)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < numElements)
    {
        C[i] = pow(A[i], 2);
    }
}
```

Figure 3.7: CUDA sample kernel to compute the square of one array and save it into another array.
/**
 * Host main routine
 */

int main(void)
{
    clock_t tStart = clock();
cudaError_t err = cudaSuccess;
int numElements = 100000000;
size_t size = numElements * sizeof(double);
printf("[Vector addition of %d elements]\n", numElements);
double *h_A = (double *)malloc(size);
double *h_B = (double *)malloc(size);
for (int i = 0; i < numElements; ++i)
    h_A[i] = i;
double *d_A = NULL;
err = cudaMalloc((void **)&d_A, size);
double *d_B = NULL;
err = cudaMalloc((void **)&d_B, size);
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
int threadsPerBlock = 128;
int blocksPerGrid = (numElements + threadsPerBlock - 1) / threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, numElements);
err = cudaGetLastError();
printf("Copy output data from the CUDA device to the host memory \n");
err = cudaMemcpy(h_B, d_B, size, cudaMemcpyDeviceToHost);
err = cudaFree(d_A);
err = cudaFree(d_B);
free(h_A);
free(h_B);
err = cudaDeviceReset();
printf("Time taken: %.2fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
printf("Done\n");
getchar();
return 0;
}

Figure 3.8: CUDA sample code that computes the squares of the first 100 million integers.
CHAPTER 4

Parallelization Approaches of the Viola-Jones Face Detection Algorithm

This chapter provides a description of the original Viola-Jones algorithm implementation. It also provides design concepts for several different new versions of the Viola-Jones face detection algorithm. These approaches include two different multithreaded versions as well as a CUDA version. These concepts provide the basis for the implementations in Chapter 5.

4.1 Original Design

The original design for the Viola-Jones face detection algorithm uses a Haar cascade classifier object detection filter to process a PMG format image. The program follows the structure shown in the diagram from Fig. 4.1. Pseudocode for the program structure is shown in Fig. 4.2 The program first loads cascade classifier parameters out of a text file. These parameters have been determined programmatically through training prior to use of the algorithm. The image is also loaded into the program from the same location. The input image is then processed through the algorithm with the result being rectangle objects consisting of four coordinate locations on the image representing a detected face. These rectangles are then drawn onto the image and the result is saved into a new PMG image file for viewing.
Figure 4.1: Basic structure of the face detection program.

```plaintext
void main()
{
    start timer;
    load image from disk;
    load cascade classifier parameters from disk;
    results = detectObjects(image, classifier);
    foreach (result in results)
    {
        draw rectangle around face;
    }
    save modified image to disk;
    free classifier;
    free image;
    stop timer;
    print time taken;
}
```

Figure 4.2: Main program structure.

The detect objects process runs as shown in the flowchart from Fig. 4.3.

Pseudocode for the detect objects process is shown in Fig. 4.4. The chart shows the loop that the detection algorithm runs in for the majority of the program’s
Figure 4.3: Structure of the face detection function. This diagram represents the process in the block labeled “Detect Faces Using Cascade Classifier” in Fig. 4.1.
detectObjects(image, classifier)
{
    create image arrays;
    for (factor = 1; factor *= scaleFactor)
    {
        calculate window size;
        set image sizes;
        compute integral images;
        set images for classifier;
        classifierInvoker();
    }
    group candidates into results;
    free all images;
    return results;
}

Figure 4.4: Initial object detection structure.

duration. The number of iterations is dependent on the scale factor which can be
decreased to improve accuracy at the expense of runtime or vice versa.

Depending on the scale factor, the outer loop from Fig. 4.3 will run
approximately 10 to 20 times. This makes it a good possibility to implement
CPU multithreading since the number of threads to be run for a performance
increase is limited by the number of cores in a processor.

The classifier runs as shown in Fig. 4.5. Pseudocode for the classifier is
shown in Fig. 4.6. This diagram shows the internal workings of the classifier
itself. Using the input values of the image arrays and window size values, the
classifier starts at location (0,0). At each location, the classifier is evaluated and
any results that are obtained are saved. The window is then moved to a new
location determined by the stepsize and the process is repeated. As can be seen,
the base process “Evaluate Classifier at Location” runs potentially thousands of
times over the course of algorithm execution considering the number of large
nested loops. This makes it potentially a good place to optimize using CUDA if
the necessary data can be provided to all threads simultaneously.

The main optimization opportunity here involves the running of the face detection function and classifier. At what level to optimize is highly dependent on the method of optimization chosen. Multithreaded programs run on a small number of powerful threads, meaning that the program only needs to be broken up into a few pieces to fully optimize the code for a multi-core CPU. CUDA however runs on graphics cards that have thousands of small processor cores resulting in the need for a more fine-grained division of work.

4.2 Design of Multithreaded Face Detection Algorithm

When considering the design for the multithreaded version, the first proposal involved multithreading the outer loop. This seemed to be a good design decision since thread creation is an expensive process. This type of multithreading would ideally create enough threads to result in a speedup but not so many that the cost of thread generation outweighed the performance gain. The corresponding version of the block diagram is shown in Fig. 4.7. The pseudocode is provided in Fig. 4.8. The functionality and processes shown in this diagram are meant to replace the functionality in Fig. 4.3.

After the initial consideration, this approach was abandoned due to the great variation in the time taken to run each iteration of the scale factor loop. The first created threads ran for exponentially longer time than the final threads created as shown in Fig. 4.9, and while this approach resulted in some speedup, it was far from the potential optimum performance. Ideally, all threads would
run at full usage for the entire runtime of the program and that was not the case with this solution.

The second considered solution involved creating a set of worker threads for each iteration and assigning them jobs to multithread the detecting of potential candidates. This approach, while a bit more complicated to implement, turned out to be vastly superior in terms of performance. This optimization is intended to replace the functionality in Fig. 4.3.

As shown in Fig. 4.10, each factor has its work split up into equal size chunks, the number of chunks being equal to the number of threads available in the system. This will allow the program to take full advantage of every core in the system while also ensuring that every thread has as close to an equal amount of work as possible. This design is slightly more complicated than the one displayed in Fig. 4.7, but still fairly simple to implement resulting in substantial performance gains for the extra effort involved. Pseudocode for this solution is provided in Fig. 4.11.

4.3 Design of CUDA Face Detection Algorithm

Converting an existing algorithm to work on a GPU is more complicated than simply multithreading it on a CPU. In order to effectively utilize a GPU, the program must perform the same calculation hundreds or, ideally, thousands of times on different data. Because of this, the logical place to implement a CUDA kernel in this project is in a similar place as the final solution for the multithreaded version.
As shown in Fig. 4.12, the overall design of the CUDA version does not have very many additional steps. Instead of using the number of threads as in the multithreaded version, a blocksize is set by the programmer. This implementation parameter will be tested with several values to find the optimal setting for this particular parallelization approach. The number of blocks is then simply the number of steps divided by the blocksize. In this proposal, one kernel launch will take place for each factor in the outer loop. This optimization is intended to replace the cascade classifier invoker shown in Fig. 4.5. The pseudocode for this solution is shown in Fig. 4.13.

4.4 Summary

In this chapter, we provided several design concepts for future implementation in Chapter 5. These included a multithreaded concept and a CUDA concept. The original design was also discussed to provide a framework for our improvements.
Figure 4.5: Structure of the classifier function. This diagram represents the process in the block labeled “Invoke Cascade Classifier” in Fig. 4.3.
classifierInvoker()
{
    get window size;
    for (x = 0; x < x2; x += step)
    {
        for (y = y1; y < y2; y += step)
        {
            set window location;
            run classifier;
            if result is present, add result to candidates list;
        }
    }
}

Figure 4.6: Initial classifier structure.
Figure 4.7: First proposal for a multithreaded optimization of the object detection function.
detectObjects(image, classifier)
{
    create image arrays;
    for (factor = 1; factor <= scaleFactor)
    {
        start thread =>
        {
            calculate window size;
            set image sizes;
            build image pyramid;
            compute integral images;
            set images for classifier;
            classifierInvoker();
        }
    }
    wait for all threads to complete;
    group candidates into results;
    free all images;
    return results;
}
Figure 4.9: Number of steps performed by each thread. The variation is due to each iteration having a different scale factor.
Figure 4.10: Second proposal for a multithreaded optimization of the object detection function.
detectObjects(image, classifier)
{
    create image arrays;
    for(factor = 1; factor *= scaleFactor)
    {
        calculate window size;
        set image sizes;
        build image pyramid;
        compute integral images;
        set images for classifier;
        calculate total number of steps in invoker;
        get total number of hardware threads;
        calcsPerThread = total steps / number of threads;
        for(i = 0; i < numThreads; i++)
        {
            threadStartLocation = i * calcsPerThread;
            threadStopLocation = (i + 1) * calcsPerThread;
            create and start thread(doThreadWork, classifier, threadStartLocation, threadStopLocation);
        }
        wait for all threads to finish;
    }
    group candidates into results;
    free all images;
    return results;
}
Figure 4.12: Proposal for a CUDA replacement for the “invoke cascade classifier” function.
classifierInvoker()
{
  get window size;
  calculate total number of steps in invoker;
  set the blocksize;
  numblocks = total steps / blocksize;
  start kernel with blocksize and number of blocks to run
classifier;
  wait for kernel to finish;
}

Figure 4.13: Proposed CUDA solution.
CHAPTER 5

Implementation Details

This chapter will describe in detail how each implementation was created. These implementations include a multithreaded version with the capability to run on any x86 architecture processor as well as a CUDA version able to run on any Nvidia CUDA capable GPU. This chapter will only cover changes made to the original code and not any aspects that were left the same as the original.

5.1 Implementation of Multithreaded Face Detection Algorithm

The implementation of the parallelization approach described in Chapter 4 is fairly simple in comparison with the CUDA version. The section of the original classifier invoker design is shown in Fig. 5.1, and the new design is shown in Fig. 5.2 and Fig. 5.3.

As can be seen in Fig. 5.2, the new implementation creates a vector of threads and bases the number of threads to be used on the hardware concurrency available in the system. It then creates threads one by one using the doThreadWork method and passing the correct work data values using i*calcsPerStep and (i + 1)*calcsPerStep. Lines 9-10 join each thread to the main thread preventing work on the main thread from continuing until all spawned threads have finished.

The doThreadWork method displayed in Fig. 5.3 executes its given chunk
```c++
int steps = 0;
for (x = 0; x < x2; x += step )
{
 for (y = y1; y < y2; y += step )
 { p.x = x;
   p.y = y;
   result = runCascadeClassifier( cascade, p, 0 );
   steps++;
   if( result > 0 )
   {
     MyRect r = {myRound(x*factor), myRound(y*factor), winSize.width, winSize.height};
     vec->push_back(r);
   }
 }
}
```

Figure 5.1: Original implementation of classifier invoker.

```c++
int steps = x2 * (y2 - y1);
int numthreads = std::thread::hardware_concurrency();
int calcsPerStep = steps / numthreads;
std::vector<std::thread> threads;
for (int i = 0; i * calcsPerStep < steps; i++)
{
   threads.push_back(std::thread(doThreadWork, x2, cascade, factor,
                             i*calcsPerStep, (i + 1)*calcsPerStep, _vec, winSize, steps));
}
for (int i = 0; i * calcsPerStep < steps; i++)
   threads[i].join();
```

Figure 5.2: New implementation of classifier invoker.

of the work based on a lowerBound and an upperBound. Because the number of steps may not always be a multiple of the number of threads created, it must check if it has passed the total number of steps and exit the method if it has. An area of note is lines 14-16. Because this code is being executed on multiple threads simultaneously, it is possible to call _vec.push_back() multiple times at the same instant. In order to avoid this, a mutex must be used to only allow one thread to add a value to the vector at one time instance. Once a thread obtains a lock on the mutex, all other threads will suspend execution on the lock command.
void doThreadWork(int maxX, myCascade* _cascade, float _factor, int lowerBound, int upperBound, std::vector<MyRect>& _vec, MySize winSize, int totalSteps)
{
// initialization work
for (int i = lowerBound; i < upperBound; i++)
{
  if (i > totalSteps)
    return;
  p.x = i % maxX;
  p.y = (i - p.x) / maxX;
  result = runCascadeClassifier(_cascade, p, 0);
  if (result > 0)
    {
      MyRect r = {myRound(p.x*_factor), myRound(p.y*_factor),
      winSize.width, winSize.height};
      _vec.push_back(r);
      _vec.push_back(mutex.lock());
      _vec.push_back(mutex.unlock());
    }
}
}

Figure 5.3: Thread work method.

until the original thread has released its lock. This prevents cross-thread
exceptions in this implementation.

Another item of note is the simplicity of implementing this code. The
percentage of code required to be modified from the overall program was fairly
insignificant and the modification process was very straightforward. This
implementation resulted in a significant speedup as will be discussed in Chapter
6.
5.2 Implementation of CUDA Face Detection Algorithm

5.2.1 Initial Considerations and Preparatory Work

The CUDA implementation, despite being arguably simpler in design than the multithreaded version turned out to be significantly more complex. There were many more considerations involved when processing code on a GPU. The first of these considerations and arguably the most important is the fact that data that is stored in system RAM is not directly accessible by the GPU. When a kernel has to be launched, first it should be confirmed that all data has been transferred into some of the GPU memory types.

As shown in Fig. 5.4, copying data back and forth from the system RAM to the GPU global memory is a relatively complex process. Similar to how memory works in system RAM, space must first be allocated using cudaMalloc to which the data will be copied using cudaMemcpy. This process must be performed for every piece of data needed on the GPU [16]. Luckily, starting with CUDA version 6, there is a new method of copying data using unified memory as shown in Fig. 5.5.

As shown, using unified memory allows for the programmer to allocate
only once, which allocates memory on both the host and the device. The data is then copied automatically back and forth from host to device as necessary. Because of this simplicity in comparison to the original method of memory management, the method shown in Fig. 5.5 was chosen for this program.

In addition, a new feature implemented in CUDA 8 and only available on Pascal architecture GPUs is the Page Migration Engine [17]. Previously, allocated unified memory would all be moved as one large transaction before a kernel was launched. With the Page Migration Engine in Pascal, page faulting from CPU to GPU and back has been implemented allowing the GPU to request individual pages as necessary to be transferred instead of transferring all the data at once. This allows for more computation-memory transfer overlap when running multiple kernels, increasing performance. For example, Kernel A can be transferring data due to a page fault while Kernel B is using the GPUs processing power.

A final addition is the ability to request an asynchronous prefetch of data well before kernel launch. If it is known that Kernel A is going to need an entire dataset, it could potentially hurt performance to allow the GPU to page fault on every data access and have to wait for the data to be transferred hundreds of individual times. In this case, the memory can be prefetched to the GPU any

```c
int *data
cudaMallocManaged(&data, N);
// fill array here
// do something on device here
// use data on host here
cudaFree(data);
```

Figure 5.5: Copy data using unified memory.
```c
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }

inline void gpuAssert(cudaError_t code, const char *file, int line,
    bool abort = true)
{
    if (code != cudaSuccess)
    {
        fprintf(stderr, “GPUassert: %s %s %d\n”,
            cudaGetErrorString(code), file, line);
        if (abort) exit(code);
    }
}

class Managed
{
    public:
        void *operator new(size_t len)
        {
            void *ptr;
            gpuErrchk(cudaMallocManaged(&ptr, len));
            cudaDeviceSynchronize();
            return ptr;
        }

        void *operator new[](size_t len)
        {
            void *ptr;
            cudaMallocManaged(&ptr, len);
            cudaDeviceSynchronize();
            return ptr;
        }

        void operator delete(void *ptr)
        {
            cudaDeviceSynchronize();
            cudaFree(ptr);
        }
};
```

Figure 5.6: CUDA managed class.

time before the kernel is launched. This call is asynchronous meaning that CPU
execution continues after starting the transaction allowing CPU execution to
overlap with the memory transfer. After much testing, it was determined that in
our version of the algorithm, it provided the best performance to not prefetch
any data and allow the GPU to handle page faults and migration.

In order to facilitate ease of use for objects on both the host and CUDA
device, a Managed class was created as shown in Fig. 5.6. All non-primitive
types that need to be accessed at some point on the GPU and CPU inherit from this Managed class in the program. The first nine lines are present to allow for GPU error checking on every GPU command. The managed class overrides the new operators for both single objects and arrays of objects. This forces objects created from types that inherit from the Managed class to be allocated using CUDA unified memory by just using the standard new method. The delete operator is also overridden to facilitate freeing of unified memory.

5.2.2 Object Creation and Allocation Decisions

Despite the changes made to basic object allocation, there are still a few decisions that must be made before proceeding with kernel design. Primitive types and arrays still must be allocated manually. In addition to this, it must be considered how the memory will be accessed. If every kernel will use the same values, it is best to assign the data to constant device memory, but if the values must change, global memory must be used. How the variables are handled differs depending on the allocation choice made for each one.

Several arrays are used by every kernel over the course of program execution and are only read from by the kernels, never written to. These arrays

```c
#define NODES 3000
#define STAGES 100
__constant__ short dalpha1_array [NODES];
__constant__ short dalpha2_array [NODES];
__constant__ short dstages_thres_array [STAGES];
__constant__ short dweights_array [NODES*3];
__constant__ short dstages_array [STAGES];
__constant__ short dtree_thres_array [NODES];
```

Figure 5.7: CUDA constant variables defined.
are best assigned to constant memory as shown in Fig. 5.7. These arrays were originally int arrays in the initial version of the program. They were changed to shorts after confirming that it would not harm accuracy to allow all of the constant readonly data to fit in the constant memory available on the GPU.

These variables are copied to by using the cudaMemcpyToSymbol function as shown in Fig. 5.8. The scaled_rectangles_array is used many times over the course of execution and is therefore placed in global memory as shown in Fig. 5.9.

5.2.3 Kernel Implementation

The CUDA kernel functions are significantly different from the multithreaded implementation. Because there is no way to provide thread safe access to a vector in CUDA for depositing results, a different approach must be
\[
\text{int steps = } x2 \ast (y2 - y1);
\]
\[
\text{int blocksize = 128;}
\]
\[
\text{int numblocks = (int)steps / blocksize;}
\]
\[
\text{if (numblocks < 1)}
\]
\[
\text{numblocks = 1;}
\]
\[
\text{MyRect* rectList = new MyRect[numblocks * blocksize];}
\]
\[
\text{gpuErrchk(cudaDeviceSynchronize());}
\]
\[
\text{ShiftFilterCuda << <numblocks, blocksize >> >(x2, \ast cascade, factor}
\]
\[
\text{, winSize, rectList, scaled_rectangles_array);}
\]
\[
\text{gpuErrchk(cudaPeekAtLastError());}
\]
\[
\text{gpuErrchk(cudaDeviceSynchronize());}
\]
\[
\text{for (int i = 0; i < numblocks * blocksize; i++)}
\]
\[
\text{if (rectList[i].height != 0)}
\]
\[
\text{vec->push_back(rectList[i]);}
\]

Figure 5.10: CUDA kernel preparation and call.

taken. In this call, an array is passed in containing an empty space for every possible result. This array is then checked after the kernel has run in the last five lines in Fig. 5.10 and all valid results are pulled out and added to the vector. The final chosen blocksize was 128 threads in a block. This was used to calculate the total number of blocks needed. The kernel was then launched with the call to \text{cudaPeekAtLastError} in order to detect kernel errors.

The kernel itself, displayed in Fig. 5.11, is very simple. It obtains the proper index for thread calculation from the \text{blockIdx}, \text{blockDim}, and \text{threadIdx} as described earlier in Chapter 3. If a result is present after running the classifier on a particular index, the value is saved to the results array for later retrieval.

5.3 Summary

In this chapter, we covered the implementations for each of the design concepts discussed in Chapter 4. This includes a multithreaded version that can
```c
__global__ void ShiftFilterCuda(int maxX, myCascade &cascade, float factor, MySize winSize, MyRect* rects, int** scaled_rectangles_array)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    MyPoint p;
    p.x = i % maxX;
    p.y = (i - p.x) / maxX;
    int result = runCascadeClassifier(&cascade, p, 0, scaled_rectangles_array);
    if (result > 0)
    {
        rects[i].x = myRound(p.x * factor);
        rects[i].y = myRound(p.y * factor);
        rects[i].width = winSize.width;
        rects[i].height = winSize.height;
    }
}
```

Figure 5.11: CUDA kernel implementation.

run with as many cores as available in the hardware it is run on, as well as a CUDA version capable of running on any CUDA capable GPU. The performance of these implementations as well as the original will be explored in Chapter 6.
CHAPTER 6

Discussion of Results

After developing both the multithreaded and CUDA versions, several rounds of testing were performed using all three versions. The original version and the CUDA version were tested using one configuration after determining the optimal block size, while the multithreaded version was tested using multiples of two cores from 2-core up to 16-core. This test was performed on a 32-thread processor to reduce the possibility of results being skewed due to processing from other programs happening on one of the cores used for the test.

Each test was performed multiple times and the results averaged to get the result shown in each table. Two different tests were performed, the first of which is keeping the number of faces the same, one in this case, and varying the resolution to determine the effect of increasing resolution on each of the different versions. The second test keeps the resolution constant but changes the number of faces, showing the impact of increasing the number of observations on each version.

All tests were performed on a system with the following specifications:

- 4.0 GHz AMD Threadripper x1950 32-Thread processor
- 1500 MHz Nvidia Geforce GTX 1080 Ti GPU
- 745 MHz Nvidia Tesla K40 GPU
Table 6.1: Difference in processing speed for several test images.

<table>
<thead>
<tr>
<th>Image</th>
<th>GeForce 1080 Ti</th>
<th>Tesla K40</th>
<th>GeForce vs Tesla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parade</td>
<td>0.027</td>
<td>0.032</td>
<td>0.84x</td>
</tr>
<tr>
<td>Family1</td>
<td>0.027</td>
<td>0.032</td>
<td>0.84x</td>
</tr>
<tr>
<td>Family2</td>
<td>0.022</td>
<td>0.027</td>
<td>0.81x</td>
</tr>
<tr>
<td>Family3</td>
<td>0.027</td>
<td>0.032</td>
<td>0.84x</td>
</tr>
<tr>
<td>Family4</td>
<td>0.039</td>
<td>0.044</td>
<td>0.89x</td>
</tr>
<tr>
<td>Family5</td>
<td>0.030</td>
<td>0.036</td>
<td>0.83x</td>
</tr>
</tbody>
</table>

6.1 Initial CUDA Test Results

In order to test for each version’s accuracy as well as generate results for some real images. A few images with varying numbers of faces and resolutions were compiled and tested with each version. Each implementation produced exactly the same result as expected and simply took varying amounts of time to obtain that result. Each image after face detection is displayed in the following figures: Fig. 6.1, Fig. 6.2, Fig. 6.3, Fig. 6.4, Fig. 6.5, Fig. 6.6. The results for these test images are displayed in Table 6.1.

As can be seen, the Tesla K40 consistently performs 10-20% slower than the GeForce GTX 1080 Ti. This could be for any number of reasons. One of the most likely is simply the age of the architecture of the Tesla. Despite being a significantly more expensive GPU aimed at enterprise customers, it is only
Figure 6.1: Test image of a parade. Only a few faces detected most likely due to people not facing directly into the camera as well as wearing hats.

Figure 6.2: Test image of a family. 5 out of 6 faces detected. Most likely, the last face was not detected due to being tilted at an angle.
Figure 6.3: Test image of a family. 8 out of 12 faces detected. Most likely, the last few faces were not detected due to being tilted at an angle.

Figure 6.4: Test image of a family. 10 out of 19 faces detected. Most likely, only about half of the faces were detected due to the somewhat low quality of the image.

compute capability 3.5 meaning that it lacks many of the new features and performance enhancements present in newer GPUs. The GeForce card is a compute capability 6.1 GPU and since it is a Pascal architecture GPU, it
contains the new Page Migration Engine as mentioned earlier allowing support for dynamic page faults which are used as a performance enhancement in the CUDA implementation. Because the Tesla lacks support for this feature, it falls back to the old behavior of migrating all data before and after kernel launch. Another
possible reason for the performance difference is the single precision performance of both cards. The Tesla is significantly more focused on double precision throughput than the GeForce card is. Unfortunately, the algorithm being tested is completely single precision. The GeForce card provides 11.3 TFLOPs while the Tesla GPU only provides 4.3 TFLOPs [18] [19]. This is a greater than 50% difference in pure single precision performance and could be another reason for the Tesla being slower.

Because of the constant performance loss of the Tesla when compared to the GeForce, only the GeForce GPUs results will be displayed in future test results.
Table 6.2: Time to process a single image vs the resolution of the image.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>Original</th>
<th>2-Core</th>
<th>4-Core</th>
<th>6-Core</th>
<th>8-Core</th>
<th>10-Core</th>
<th>12-Core</th>
<th>14-Core</th>
<th>16-Core</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>480x640</td>
<td>0.283</td>
<td>0.173</td>
<td>0.111</td>
<td>0.090</td>
<td>0.077</td>
<td>0.069</td>
<td>0.065</td>
<td>0.062</td>
<td>0.061</td>
<td>0.016</td>
</tr>
<tr>
<td>960x1280</td>
<td>1.011</td>
<td>0.568</td>
<td>0.346</td>
<td>0.268</td>
<td>0.228</td>
<td>0.207</td>
<td>0.194</td>
<td>0.187</td>
<td>0.184</td>
<td>0.033</td>
</tr>
<tr>
<td>1440x1920</td>
<td>2.167</td>
<td>1.188</td>
<td>0.699</td>
<td>0.531</td>
<td>0.449</td>
<td>0.394</td>
<td>0.375</td>
<td>0.351</td>
<td>0.354</td>
<td>0.061</td>
</tr>
<tr>
<td>1920x2560</td>
<td>3.794</td>
<td>2.052</td>
<td>1.194</td>
<td>0.884</td>
<td>0.733</td>
<td>0.647</td>
<td>0.603</td>
<td>0.581</td>
<td>0.579</td>
<td>0.091</td>
</tr>
<tr>
<td>2400x3200</td>
<td>5.848</td>
<td>3.125</td>
<td>1.787</td>
<td>1.311</td>
<td>1.096</td>
<td>0.936</td>
<td>0.848</td>
<td>0.817</td>
<td>0.839</td>
<td>0.139</td>
</tr>
<tr>
<td>2880x3840</td>
<td>8.310</td>
<td>4.424</td>
<td>2.503</td>
<td>1.827</td>
<td>1.522</td>
<td>1.309</td>
<td>1.182</td>
<td>1.175</td>
<td>1.164</td>
<td>0.185</td>
</tr>
<tr>
<td>3360x4480</td>
<td>11.286</td>
<td>5.988</td>
<td>3.366</td>
<td>2.435</td>
<td>1.993</td>
<td>1.779</td>
<td>1.589</td>
<td>1.552</td>
<td>1.536</td>
<td>0.245</td>
</tr>
<tr>
<td>3840x5120</td>
<td>14.808</td>
<td>7.810</td>
<td>4.363</td>
<td>3.179</td>
<td>2.581</td>
<td>2.198</td>
<td>1.989</td>
<td>1.963</td>
<td>1.946</td>
<td>0.315</td>
</tr>
<tr>
<td>4320x5760</td>
<td>18.490</td>
<td>9.725</td>
<td>5.428</td>
<td>3.927</td>
<td>3.191</td>
<td>2.724</td>
<td>2.445</td>
<td>2.383</td>
<td>2.380</td>
<td>0.414</td>
</tr>
</tbody>
</table>

6.2 Results from Tests for Different Image Resolutions with Constant Number of Faces

As shown in Table 6.2, the resolution has a fairly significant impact on the time the program takes to run for all versions. The time taken to run seems to increase linearly with the number of pixels in the image, resulting in the curves shown in Fig. 6.7. The multithreaded version scales exactly as expected with each doubling of core count resulting in a fifty percent decrease in runtime plus some additional overhead as the core count grows higher. This overhead seems to overcome any significant additional speedup beyond 10 cores as the decrease in runtime is very small compared to the runtime of the program.

Converting the time taken to run the algorithm to a relevant frames per second measurement gives a baseline of 3.5 frames per second processed for the original implementation of the algorithm at 480x640 which has an equivalent number of pixels to 640x480 also known as VGA which is a common resolution used for security cameras and other video streams where face detection might be
required. The 16-Core multithreaded implementation provides a throughput of 17.68 frames per second based on its frame time which is more than the 15 frames per second average generally used by security cameras. The CUDA implementation is able to maintain 62.5 frames per second at 480x640 which is well above what most video streams contain. In fact, the CUDA implementation is able to maintain above 15 frames per second up to 1440x1920 which has more pixels than a standard 1080p image.

As shown in Fig. 6.7, the CUDA version has much better scaling with resolution increases than any CPU version of the software does. Even at the smallest resolution, the CUDA version has greater performance than even the 16-Core multithreaded version, and the gap in performance grows as the resolution does. An interesting item of note is that while profiling the CUDA version using Nsight, it was determined that the GPU was not being used to its fullest potential. Not enough work was being provided to the GPU to fully saturate both its bandwidth and processing power. If a future implementation of the program was to process many images in quick succession, the CUDA implementation may prove to have an even larger performance advantage over both CPU implementations.

Table 6.3 shows the speedup obtained by the multithreaded at 8-Cores and 16-Cores and the CUDA version compared to the original version of the program as well as a comparison of the 16-Core version to the CUDA version. Fig. 6.8 displays this information in graph form to more easily see the differences between the versions. As can be seen, the 16-Core multithreaded version gives a
maximum speedup of 7.77x over the original implementation. The CUDA version performs substantially better than that with a maximum of 47x the performance of the original.
Table 6.3: Relative speedup obtained by each version of the program when processing differing image resolutions.

<table>
<thead>
<tr>
<th>Resolution</th>
<th>8-Core vs Original</th>
<th>16-Core vs Original</th>
<th>CUDA vs Original</th>
<th>CUDA vs 16-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>480x640</td>
<td>3.68x</td>
<td>4.64x</td>
<td>17.69x</td>
<td>3.81x</td>
</tr>
<tr>
<td>960x1280</td>
<td>4.43x</td>
<td>5.49x</td>
<td>30.64x</td>
<td>5.58x</td>
</tr>
<tr>
<td>1440x1920</td>
<td>4.83x</td>
<td>6.12x</td>
<td>35.52x</td>
<td>5.8x</td>
</tr>
<tr>
<td>1920x2560</td>
<td>5.18x</td>
<td>6.55x</td>
<td>41.69x</td>
<td>6.36x</td>
</tr>
<tr>
<td>2400x3200</td>
<td>5.34x</td>
<td>6.97x</td>
<td>42.07x</td>
<td>6.04x</td>
</tr>
<tr>
<td>2880x3840</td>
<td>5.46x</td>
<td>7.14x</td>
<td>44.92x</td>
<td>6.29x</td>
</tr>
<tr>
<td>3360x4480</td>
<td>5.66x</td>
<td>7.35x</td>
<td>46.07x</td>
<td>6.27x</td>
</tr>
<tr>
<td>3840x5120</td>
<td>5.74x</td>
<td>7.61x</td>
<td>47.01x</td>
<td>6.18x</td>
</tr>
<tr>
<td>4320x5760</td>
<td>5.79x</td>
<td>7.77x</td>
<td>44.66x</td>
<td>5.75x</td>
</tr>
</tbody>
</table>

Table 6.3: Relative speedup obtained by each version of the program when processing differing image resolutions.

6.3 Results from Tests on Images with Different Numbers of Faces and Constant Resolution

When differing the number of faces, the results are heavily skewed in favor of the CUDA version as shown in Table 6.4. To attempt to show the largest variation in time between different numbers of faces and to ensure that all faces are composed of enough pixels to be detected, the highest resolution from the first test was used. The CUDA version is the fastest in the first few tests and maintains its lead throughout. This is most likely due to the increasing number of observations that must be sorted through by the program. This is even more obvious in Fig. 6.9. The processing time increases for each version at an exponential rate as the number of faces increases, but the CUDA version maintains a significant advantage. This makes the CUDA version a better choice when attempting to detect very large numbers of faces in a high-resolution image.

Table 6.5 shows the relative speed up obtained by each version of the program at differing numbers of faces. Fig. 6.10 shows this information in graph
form to more easily see the differences between the versions. This is a very interesting result when compared to the resolution speed up graph in Fig. 6.8. It seems that while the CUDA implementation begins with a massive lead in performance at small numbers of faces, this diminishes as the number of faces
Table 6.4: Time to process a single image vs the number of detected faces in the image.

<table>
<thead>
<tr>
<th>Faces</th>
<th>Original</th>
<th>2-Core</th>
<th>4-Core</th>
<th>6-Core</th>
<th>8-Core</th>
<th>10-Core</th>
<th>12-Core</th>
<th>14-Core</th>
<th>16-Core</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18.490</td>
<td>9.725</td>
<td>5.428</td>
<td>3.927</td>
<td>3.191</td>
<td>2.724</td>
<td>2.445</td>
<td>2.383</td>
<td>2.380</td>
<td>0.414</td>
</tr>
<tr>
<td>100</td>
<td>23.077</td>
<td>10.985</td>
<td>6.317</td>
<td>4.341</td>
<td>3.509</td>
<td>2.988</td>
<td>2.691</td>
<td>2.579</td>
<td>2.612</td>
<td>0.426</td>
</tr>
<tr>
<td>400</td>
<td>25.452</td>
<td>11.959</td>
<td>6.555</td>
<td>4.733</td>
<td>3.803</td>
<td>3.233</td>
<td>2.933</td>
<td>2.825</td>
<td>2.842</td>
<td>0.458</td>
</tr>
<tr>
<td>900</td>
<td>27.290</td>
<td>12.944</td>
<td>7.135</td>
<td>5.183</td>
<td>4.233</td>
<td>3.642</td>
<td>3.294</td>
<td>3.191</td>
<td>3.204</td>
<td>0.571</td>
</tr>
<tr>
<td>10000</td>
<td>79.422</td>
<td>65.331</td>
<td>63.273</td>
<td>57.342</td>
<td>58.689</td>
<td>60.769</td>
<td>60.812</td>
<td>60.337</td>
<td>60.594</td>
<td>28.455</td>
</tr>
</tbody>
</table>

Table 6.5: Speedup obtained by different implementations when testing differing numbers of detected faces in the image.

<table>
<thead>
<tr>
<th>Faces</th>
<th>8-Core vs Original</th>
<th>16-Core vs Original</th>
<th>CUDA vs Original</th>
<th>CUDA vs 16-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.79x</td>
<td>7.77x</td>
<td>44.66x</td>
<td>5.75x</td>
</tr>
<tr>
<td>100</td>
<td>5.58x</td>
<td>6.83x</td>
<td>54.17x</td>
<td>6.13x</td>
</tr>
<tr>
<td>400</td>
<td>6.69x</td>
<td>8.96x</td>
<td>55.57x</td>
<td>6.21x</td>
</tr>
<tr>
<td>900</td>
<td>6.45x</td>
<td>8.52x</td>
<td>47.79x</td>
<td>5.61x</td>
</tr>
<tr>
<td>1600</td>
<td>5.57x</td>
<td>6.98x</td>
<td>29.05x</td>
<td>4.16x</td>
</tr>
<tr>
<td>2500</td>
<td>4.4x</td>
<td>5.13x</td>
<td>15.88x</td>
<td>3.09x</td>
</tr>
<tr>
<td>3600</td>
<td>3.8x</td>
<td>4.28x</td>
<td>11.97x</td>
<td>2.8x</td>
</tr>
<tr>
<td>4900</td>
<td>3.2x</td>
<td>3.57x</td>
<td>9.15x</td>
<td>2.56x</td>
</tr>
<tr>
<td>6400</td>
<td>2.35x</td>
<td>2.52x</td>
<td>5.88x</td>
<td>2.34x</td>
</tr>
<tr>
<td>8100</td>
<td>1.85x</td>
<td>1.87x</td>
<td>4.05x</td>
<td>2.16x</td>
</tr>
<tr>
<td>10000</td>
<td>1.35x</td>
<td>1.31x</td>
<td>2.79x</td>
<td>2.13x</td>
</tr>
</tbody>
</table>

increases until the CUDA version is only 2.8x better than the original. A theory for why this might be the case is that while the detection process takes place on the GPU, once all the detected faces are compiled, the processing of those faces takes place on the CPU. With extremely large numbers of faces, the majority of the processing time is sorting through all of the detected results after running the algorithm. Luckily for the CUDA version in this case, detecting ten thousand
Figure 6.9: Time to process a single image vs the number of detected faces in the image.

faces in a single high-resolution image is a very unlikely scenario in the real world and most likely not worth optimizing the algorithm for.
Figure 6.10: Speedup obtained by different implementations when testing differing numbers of detected faces in the image.

6.4 Video Testing

A fourth test version of the application was created as a proof of concept for video face detection. It used a simple video stream from a webcam and libraries from OpenCV to allow for still images to be captured from the video
stream and the algorithm applied appropriately. This version used the multithreaded algorithm and was able to maintain an average frame time very close to the time it took to process one image.

A CUDA version was created as well, however this version was not a realistic representation of the algorithm’s performance. The issue was that the GPU must use its processing power to display the video with identified faces on the attached monitor. When a CUDA task to detect the faces was also provided to the GPU, degraded performance was apparent. The video stream, smooth with the multithreaded version, became choppy similar to a slideshow, and the time to process a frame increased drastically from the baseline provided by the CUDA single image version of the program. To truly process and display simultaneously, multiple GPUs must be used resulting in a much larger cost to the end user. This may be practical if processing many video streams on a single computer, but most likely would increase the cost of any non-enterprise product to a large degree. The CUDA version could potentially be useful to process batches of video files in an expedient manner the content of which would not be displayed on the screen, but instead output as files, but live display is impractical.

6.5 Further Observations

Even though the 16-core multithreaded version of the program is almost able to keep up in some tests with the CUDA version, it is worth noting that it is much more likely for a consumer to have a CUDA capable graphics card than a 16-core processor. Most consumer processors in desktops are still 8-core with
most of the market being fewer cores. Graphics cards capable of executing CUDA programs are relatively common, even ones almost as powerful as the ones used in our testing. The average consumer processor has 4-cores, so it is the most reasonable consumer comparison point. Taking this into account, the CUDA version appears even better than at first glance.

Unfortunately for CUDA, due to its rapid advancement and continual new features, support for programs written with different versions of CUDA can vary widely from GPU to GPU. In addition, performance is not constant across different generations of GPUs even with the same code. This was apparent when testing was done with different GPUs. Differing GPUs gave wildly different performance somewhat hampering the claim that CUDA is always better in any particular situation. Features in new versions of CUDA might speed up a program on newer architectures but slow it down or not function at all on older ones. The advantage of CPU multithreading in this case is the ability for the programmer to assume the program will run on all x86 processors with much less regard for age of architecture.

6.6 Summary

After many rounds of testing, it is clear that the CUDA implementation is superior in almost every way due to the vast performance improvements provided when compared to both the original implementation and the multithreaded implementation. The multithreaded version when running with the full 16 cores was able to obtain a maximum performance improvement of 7.77x over the
original implementation. The CUDA version however was able to display increased performance in every test up to 47x and only decreasing to a minimum of 2.1x under the most unusual conditions while still remaining significantly ahead of the multithreaded implementation.
CHAPTER 7

Conclusion and Future Work

This chapter provides conclusions drawn from the results of testing on the implemented programs and describes future potential work in the design of CUDA accelerated face detection algorithms. There is significant future work that can be done to improve the CUDA program, especially as CUDA itself improves as a language. Every new version adds multiple new features many of which allow a developer to more quickly convert existing programs to a faster, massively parallel version.

7.1 Conclusions

The main purpose of this thesis was to improve the face detection speed of the Viola-Jones algorithm using CUDA programming. Chapter 4 developed concepts on which to build both a multithreaded version and a CUDA version of the Viola-Jones face detection algorithm. Chapter 5 provided implementations for the multithreaded version as well as the CUDA version. Chapter 6 displayed the results gained from testing by manipulating a wide variety of image parameters such as resolution and number of faces. Regarding contributions, this thesis provides source code for both a multithreaded optimization as well as a CUDA implementation of the Viola-Jones algorithm. Both of these implementations resulted in significant speedup over the original algorithm.
There are advantages to both the multithreaded version of the algorithm and the CUDA version. The CUDA version is faster at almost all tasks, sometimes significantly so, especially at high resolutions of the image. This gap could potentially be widened by further development and optimization. On the other hand, the multithreaded version still runs significantly faster than the original version and took a very small fraction of the time to develop in comparison to the CUDA version. In addition, the multithreaded version did not require any additional specialized hardware as the CUDA version did. Every computer system contains a CPU but not every system contains a CUDA capable GPU, much less one as powerful as the one used in testing. C++ as a programming language has been around for much longer than CUDA and there are numerous resources available for a programmer wishing to optimize an algorithm whether through multithreading or another method.

CUDA is relatively new in comparison and has significantly fewer resources available. New versions are released every few months with major new features in each release, meaning it is very hard to find up-to-date sources beyond very basic examples. For many functions, the only available information is the manual provided by Nvidia and sometimes the Nvidia developer forums. There were also many more configuration issues to overcome when writing the CUDA version. Simply getting the program to compile properly was sometimes a challenge resulting in a significant amount of time being spent on debugging and making file changes that simply were not necessary for the multithreaded version. In addition, major bugs were sometimes present in specific versions of CUDA due to the rapid pace of language development. An example is that in CUDA version
8, which was used for development for a period of time before updating to CUDA version 9.1, the GPU being used for testing did not work with any of the performance analysis tools provided by Nvidia. This presented a problem since these tools are extremely important for determining the proper settings for kernel launches and other parameters to obtain the best performance.

CUDA presented the additional challenge of being a very different style of programming and a different thought process was needed for development. Modern object-oriented programming languages like C++, Java, or C# differ mostly in syntax and some structural elements and thus are fairly easy for a programmer to learn if one of the other languages was studied previously. CUDA requires a different approach since each thread will always execute an instruction at the same time. A simple example is that an “if” statement is executed very differently in CUDA than in any of the previously mentioned languages. If an execution diverges along two possible paths, the kernel must execute instructions along each of those paths independently resulting in a potential doubling of runtime in the worst case. If more divergences are present as with multiple “if” statements, performance can suffer even more.

Because of all of the previously mentioned disadvantages inherent with programming using CUDA, if a project is fairly simple and already has a very short runtime it would seem that multithreading is superior to CUDA. However, if a project has a larger amount of data processing involved resulting in a long runtime or a very large number of repeated short runs, CUDA will be superior if a developer is willing to invest the time learning a new programming paradigm.
and dealing with the extra complication involved. This CUDA advantage only
grows larger as the dataset under test grows. Despite all of the downsides of
programing using CUDA, it has great potential for future use. In the short
timespan since this project began, there have been many new features
implemented in CUDA to make initial development simpler and new features
continue to be added and optimized at a rapid pace. Soon, programming using
CUDA may be almost as simple as programming in C++.

7.2 Future Work

There are several ways in which this program could be improved in the future.
Streams were implemented in the CUDA design which is a huge optimization
over standard synchronous kernel launches, however, this came with a few
disadvantages. The main disadvantage is that each kernel needed its own section
of memory allocated for its current working set. Values that remained unchanged
were allocated and moved onto the GPU at the beginning of execution, but each
stream needed additional memory allocated that only it was allowed to use. This
resulted in a much larger memory usage for the CUDA version than for the
multithreaded version, so much so that at very high resolutions, the program
would run out of memory on the GPU and crash. A simple future optimization
would be to keep track of the amount of memory present and in use and only
asynchronously launch streams until the GPU memory was full, then wait for
more memory to be freed as streams completed their kernels.

Finally, use of unified memory could be replaced with manual memory
management with CUDA. Using the managed memory system allows for faster
development by allowing the programmer to focus on algorithms and kernel
design and not memory management. Unfortunately, unified memory
performance is not as fast as the old system of manually allocating memory with
cudaMalloc and explicitly transferring data to the GPU and back. This is
partially because unified memory is a new development in the last few years and
has much room for improvement performance wise, but also because the
automated paging system will just never be as good at deciding when to transfer
data to the GPU as a developer who knows exactly when the data will be needed
by the GPU.
References


https://ark.intel.com/products/96900/
Intel-Xeon-Processor-E7-8894-v4-60M-Cache-2.40-GHz.

[13] F. Abi-Chahla, “Nvidia’s cuda: The end of the cpu?”, Tom’s Hardware, 2008, 


beyond-gpu-memory-limits-unified-memory-pascal/.

com/content/tesla/pdf/nvidia-tesla-kepler-family-datasheet.pdf.

geforce/products/10series/geforce-gtx-1080-ti/.


http://www.dotnetcurry.com/dotnet/1360/
concurrent-programming-dotnet-core.


unified memory access performance in cuda”, IEEE High Performance 