PathFinder in CUDA

by

James Doyle

Submitted in partial fulfillment of the Honors Requirements
For the Computer Science Major
Dickinson College, 2010

Professor John P. MacCormick, Supervisor
Professor Timothy A. Wahls, Reader
Professor Barry A. Tesman, Reader

April 27, 2010

The Department of Mathematics and Computer Science at Dickinson College hereby accepts this senior honors thesis by James Doyle, and awards departmental honors in Computer Science.

John P. MacCormick (Advisor)	Date
Timothy A. Wahls (Committee Member)	Date
Barry A. Tesman (Committee Member)	Date
David S. Richeson (Department Chair)	Date

Department of Mathematics and Computer Science
Dickinson College

May 2010

Abstract

PathFinder in CUDA

by
James Doyle

Image segmentation into superpixels is a common early step in computer vision algorithms. Several algorithms already exist that produce good quality results. PathFinder is a competing image segmentation algorithm that creates high quality superpixels at faster speeds. Our goal is to transfer computation from the CPU to the GPU, which has the potential to execute parallel tasks much more efficiently. We investigate CUDA, a freely available platform for programming GPUs, and learn what kinds of operations benefit most from this approach. We find a 3 to 5 times speed up of the PathFinder algorithm overall, with an improvement of three orders of magnitude in some components of the algorithm.

Acknowledgements

I wish to thank my research advisor, Prof. John MacCormick, for presenting me with several great research topics, always being available to help me when I reached an impasse, and for all of his previous work on the PathFinder algorithm. My thanks also go to my major advisor, Prof. Tim Wahls, for encouraging me to do an honors research project and, along with Prof. Barry Tesman, agreeing to serve on my honors committee. Additionally, I thank Fabio Drucker, whose initial research on PathFinder's performance made my investigation viable. Finally, to anyone who has asked me about my research, kept me company through long hours in the lab, or otherwise supported me over the last year: thank you.

Table Of Contents

Chapter 1

Introduction

1.1 The Performance of Superpixel Algorithms

A superpixel is a region of pixels in an image that share roughly similar properties, indicating that they likely represent part of a single object. Before any intensive video or image processing, a superpixel algorithm can be used to divide the image into superpixels, thus simplifying any following tasks by reducing the number of objects to be analyzed or edited (Moore, Prince, Warrell, Mohammed & Jones, 2008). My research will focus on the optimization of the PathFinder superpixel algorithm. The primary objective of my research is to investigate opportunities for substantial performance increases through parallel computing on graphics processing units (GPUs), the powerful video cards attached to some computers.

Prof. John MacCormick, my honors advisor, has previously worked with students to compare the efficiency and quality of PathFinder with other established superpixel algorithms (Drucker, 2009), notably the Efficient Graph-Based Image Segmentation (EGBIS) algorithm (Felzenszwalb & Huttenlocher, 2004). Thus, the concrete objective of this thesis is to compare the speed of PathFinder's Java implementation to a new implementation of PathFinder on GPUs.

1.2 Parallel Computing and CUDA

In this project, our goal was to shift as much of the computation from the computer's central processor (CPU) to the graphics processor (GPU), which has a multi-cored

architecture designed for parallel computing (Fung & Mann, 2008; Halfhill, 2008; Nyland, Harris & Prins, 2008). Specifically, we investigated NVIDIA Corporation's Compute Unified Device Architecture (CUDA) which allows programmers to use a subset of the C programming language to implement “massive multithreading”: tens of thousands of threads concurrently executed by the GPU (Halfhill, 2008). Every graphics card manufactured by NVIDIA in the last several generations is compatible with CUDA (NVIDIA Corporation, 2009).

Chapter 2

Background

2.1 PathFinder and Other Image Segmentation Algorithms

PathFinder is one of many image segmentation algorithms. They vary in method, quality, and efficiency. PathFinder divides an image into superpixels by creating crossing paths between pixels with sufficient difference in color. The resulting regions inside these paths are the superpixels. First this is done with the original image so comparisons are made between horizontally adjacent pixels. Then the image is transposed (the equivalent of a 90 degree rotation), and adjacent pixels are again compared, though these are the vertically adjacent pairs.

Before the algorithm compares the adjacent pixels, a convolution filter is applied to the entire image, which makes each pixel the "average" of its neighbors. This is a common initial step in computer vision algorithms that slightly blurs the image, removing anomalies and providing more reliable results. PathFinder's convolution operation smoothes the image by assigning the average of the red, green, and blue (RGB) values of each pixel within a small radius to the RGB values of the target pixel. In the actual comparison of adjacent pixels, the differences in red, green, and blue values between the two convolved pixels are summed.

Utilizing the resulting data – dubbed edge strengths – grid paths are calculated through the image along the heaviest edge strengths, dividing it into many regions. EGBIS creates a similar result, though instead of drawing paths through the image, the algorithm only compares adjacent pixels to each other to determine if they belong in the same segment

(Felzenszwalb, & Huttenlocher, 2004).

A disadvantage of the superpixel approach is the irregularity and the loss of the grid-like properties that are inherent with a pixel representation. The Superpixel Lattice algorithm has attempted to address this shortcoming with a regular grid of superpixels that forces the superpixels into a more computationally friendly arrangement (Moore, Prince, Warrell, Mohammed & Jones, 2008). However, this approach has its own disadvantages, in particular that it forces an artificial structure on an image that does not necessarily contain any.

Of these algorithms, EGBIS is considered to offer the best quality segmentation based on the benchmarks of explained variation and mean accuracy. Explained variation is a human-independent metric that evaluates the quality of the segmentation by comparing each individual pixel to the color values of the superpixel it has become part of. Mean accuracy uses human-defined examples to assess how many of the pixels were assigned to the correct region (Moore, Prince, Warrell, Mohammed & Jones, 2008; Drucker, 2009). Based on these two measures, PathFinder produces slightly lower quality segmentations. However, PathFinder has been shown to be over an order of magnitude faster than the EGBIS algorithm, with the advantage increasing with the size of the image, an important consideration for any real-time application (Drucker, 2009).

2.2 NVIDIA's CUDA Platform and Computer Vision

NVIDIA introduced CUDA in February 2007 to provide a standardized framework for outsourcing computational tasks to the GPU. It uses C for CUDA, which is C with some extensions provided by NVIDIA. Thus, programming for CUDA requires installing the correct drivers, an SDK, and a toolkit, while running CUDA software requires the correct

drivers and a new enough GPU. Specifically, any NVIDIA GPU in the GeForce 8xxx series or newer is CUDA-enabled. The most recent releases of the CUDA SDK added support for Mac OS X (NVIDIA Corporation, 2009).

CUDA is best suited to tasks that are highly parallelizable. These typically involve algorithms that conduct a large number of independent and simple (or at least similar) calculations. For example, an ideal candidate for CUDA acceleration is an astrophysical simulation of n-bodies interacting with each other. This requires a brute-force all-pairs analysis to exactly determine each object's motion. These calculations can be threaded so that thousands of these pairs are analyzed at the same time using a GPU, instead of one, two or four at a time with an advanced CPU (Nyland, Harris & Prins, 2008).

We are not the first to attempt to apply the GPU and CUDA to the field of computer vision. In fact, more than one pre-CUDA framework was created specifically to enable computer vision work on the GPU (Babenko & Shah, 2008; Alluse, Horain, Agarwal & Saipriyadarshan, 2008). Since its release, CUDA has been shown to be a suitable platform for the development of efficient computer vision algorithms (Fung & Mann, 2008).

Chapter 3

Methods

3.1 Hardware Requirements and Experiment Environment

Before significant work could begin on the PathFinder project, we needed to determine an appropriate test system and procure one. Our sole requirement was an NVIDIA GPU from the GeForce 8xxx generation or newer. However, this elicited several other sub-requirements of the system to support this device. First, these cards interface with the computer through a PCI-Express slot on the motherboard, which replaced Accelerated Graphics Port (AGP) technology as the industry standard several years ago. Second, the system needed a chassis that was physically large enough to house the new GPU, as most high-end cards are full width with a dedicated cooling system that occupies as much space as a second PCI card. Third, because the power requirements for a high-end GPU and its cooling system can far exceed even the CPU, a power supply (PSU) with enough capacity is a necessity.

Fortunately, we were able to locate a machine that was high-end when it was manufactured several years ago. The model was a Dell XPS 600, originally intended for consumer gaming, which had recently been replaced in a language lab on campus. Its configuration was satisfactory to the extent that the only part requiring replacement was the GPU itself. For this we chose the NVIDIA GeForce GTX 275. Detailed specifications of the system and the graphics card are provided in Tables 3.1 and 3.2.

Table 3.1: Specifications for our testbed system.

CPU	Intel Pentium D 3.2 GHz
Motherboard	NVIDIA nForce4 SLI Intel Edition
Memory	2 GB (2x1 GB) 533 MHz
PSU	650 Watts
GPU	NVIDIA GeForce GTX 275
OS	Windows XP Pro SP2 32-bit

Table 3.2: Specifications for the NVIDIA GeForce GTX 275 and CUDA.

Core Clock	633 MHz
Stream Processors	240 Cores
Memory Clock	2268 MHz
Memory Size	896 MB DDR3
Minimum PSU Capacity	550 Watts
Driver Version	190.38
CUDA Version	2.3

3.2 Optimization of PathFinder

Our approach was to integrate CUDA accelerated code into the pre-existing Java code of the PathFinder project. We accomplished this with the Java Native Interface (JNI), which enables Java code to utilize libraries and code from different languages, including C.

We determined that four subroutines in the PathFinder algorithm were highly parallelizable. These included the previously mentioned tasks of image transposition, image convolution filtering, calculation of edge strengths, and calculation of grid paths. These components of PathFinder were chosen because each contains a simple operation that is executed on every pixel. Because each operation is executed thousands of times, the improvement due to parallelization should be significant, and because each operation is simple, the penalty for using the slower processor core of the GPU should not be. Thus we

identified which portions of the PathFinder algorithm are the best candidates for CUDA, and our task was to rewrite each of those portions as a function in C for CUDA, then call each function at the appropriate time using JNI.

The multithreading process in C for CUDA is fairly straightforward. First, we allocate sufficient memory on the GPU and copy the data to be manipulated into that space. Second, we write one piece of code called the "kernel" that instructs the processor how to perform the required task. Finally, we implement in C the host function that runs the CUDA kernel with a designated number of threads, which split the sections of memory amongst themselves (NVIDIA Corporation, 2009).

3.3 Anatomy of a CUDA Kernel

```
01:__global__ void transpose(int *odata, int *idata, int width, int height)
02:{
03:   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
04:   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
05:
06:   if (xIndex < width && yIndex < height)
07:   {
08:       unsigned int index_in  = xIndex + width * yIndex;
09:       unsigned int index_out = yIndex + height * xIndex;
10:       odata[index_out] = idata[index_in];
11:   }
12:}
```

The example above is the transpose operation, which we will analyze to demonstrate a few of the unique aspects of a CUDA kernel. In line 1 we see the `__global__` qualifier, which declares the function to be a kernel, which can only be executed on the GPU. `odata` and `idata` are each pointers to an array of integers. In this case we are storing the two-dimensional image as a one-dimensional array of integers. The memory for these arrays has already been allocated in the GPU's memory by the host C code and the `idata` is already filled with the original image data. In lines 3 and 4 we declare two integers, `xIndex` and

`yIndex`, which represent the x and y coordinates of the pixel. CUDA divides the data into blocks and then further splits those blocks into individual threads. Each individual kernel is guaranteed access to its identifying information: which block and thread it is. Thus we find the coordinate in each direction by multiplying the dimension of the block by the ID number of the current block, and finally adding the current thread ID number. The `if` statement in line 6 prevents any addressing errors from halting the program. In the remainder of the kernel we copy the data to the transposed position. After this function completes and returns to the host code, the `cudaThreadSynchronize()` function forces the code to wait for all threads to complete, then we copy the data from the GPU's memory back to the host.

3.4 Testing Procedure

Using System.nanoTime() method calls and the comparable timer functionality included in C for CUDA, we analyzed the performance of the original implementation of PathFinder and the most recent implementation, which outsources work to CUDA. These tests were performed with a variety of test image resolutions: 192x144, 400x304, and 512x480. Each test was repeated five times and the mean of the five trials is the data presented. We did not attempt any quality optimizations with regards to the produced segmentations, and thus do not need to perform quality-related analysis.

For the sake of this study, we did not include time required by JNI operations or memory allocation and copying in our measurements. This was a conscious decision to maintain the focus of our results on the advantages offered by CUDA instead of the disadvantages of JNI and an unoptimized implementation, which we will discuss in more depth in Chapter 5.

Chapter 4

Results

Through my initial analysis of the PathFinder algorithm, we determined how much processing time each step in the algorithm required. Figure 4.1 depicts our findings.

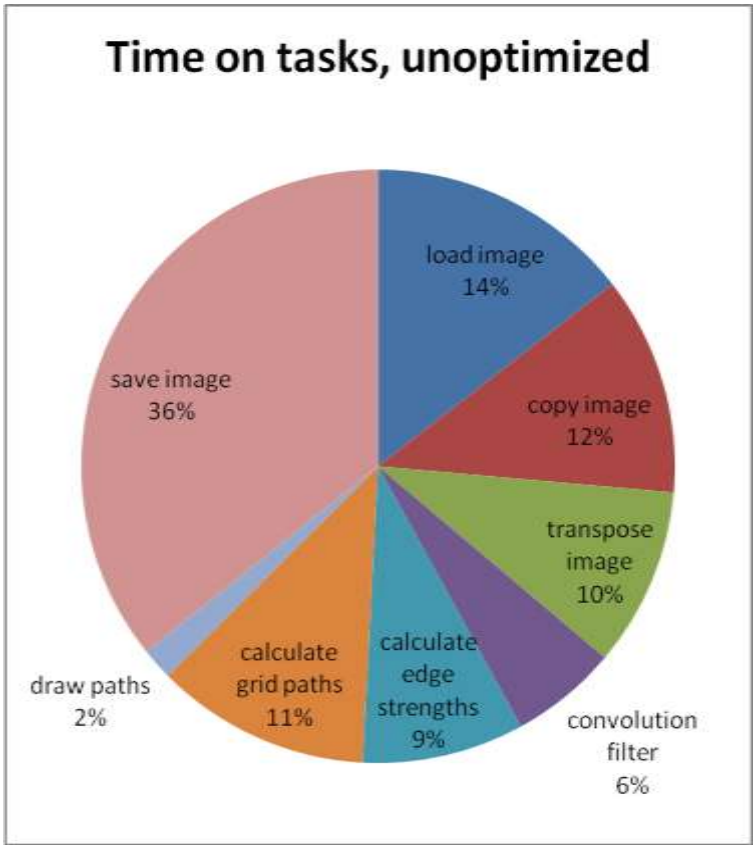

Figure 4.1: Each task performed by PathFinder algorithm, as percentage of runtime.

Almost two-thirds of the running time is devoted to file system operations (loading/copying/saving the image) or drawing the paths on the image, each of which we consider trivial overhead for the purposes of our investigation. This leaves the following four

tasks for our focus: performing an image transpose, applying a convolution filter, calculating edge strengths and calculating grid paths.

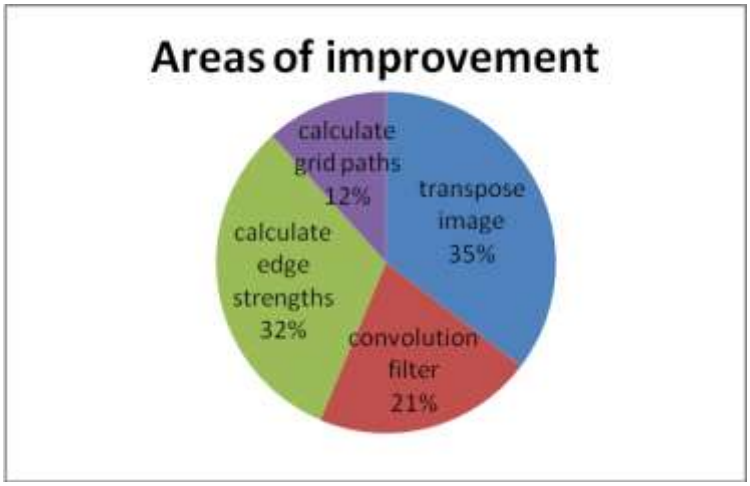

Figure 4.2: Each improved task performed by PathFinder algorithm, as relative percentages of time saved.

In practice, all four areas saw some runtime improvement and, with the exception of the grid paths calculation, the relative time savings was related to the amount of time originally required (Figure 4.2). That is, because an image transpose originally required more time than the edge strength calculation, it also saved more time when ported to CUDA.

The grid path calculation did not improve as much as the others, even though it used more runtime. This is because the grid path calculation can not be executed on each pixel of the image concurrently, unlike the transpose, convolution filter, or edge strength operations. The calculation at each scanline (the horizontal or vertical line of pixels being analyzed) depends on the results of the scanlines above or below it. Furthermore, the pixels at the edge of the image are special cases that had to be handled outside of CUDA in the host C code. Thus, instead of the simple process described in section 3.2 in which memory is allocated, copied, and returned once and the kernel is called once, memory was copied and returned and

the kernel called as many times as there were scanlines of pixels in the image to be analyzed. This has the dual effect of creating unavoidable overhead and reducing the potential for optimization, which leads to the result shown in Figure 4.3.

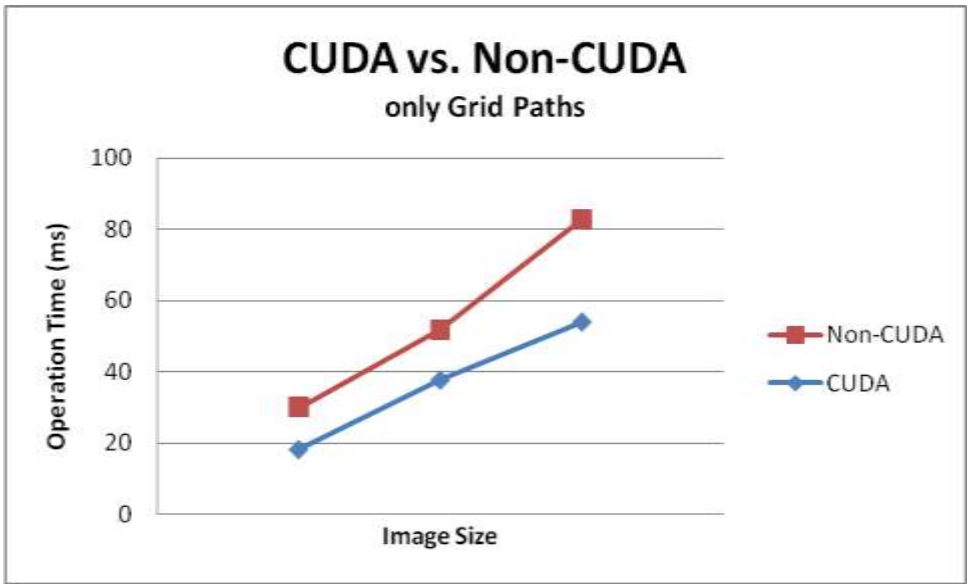

Figure 4.3: Time required to execute the grid paths calculation by CUDA and by the original Java code.

Overall, we achieved a significant runtime improvement over the original Java implementation. Including the grid paths calculation, this advantage is small (Figure 4.4). However, when we disregard the grid paths calculation the runtime improves by a factor of 100 (Figure 4.5).

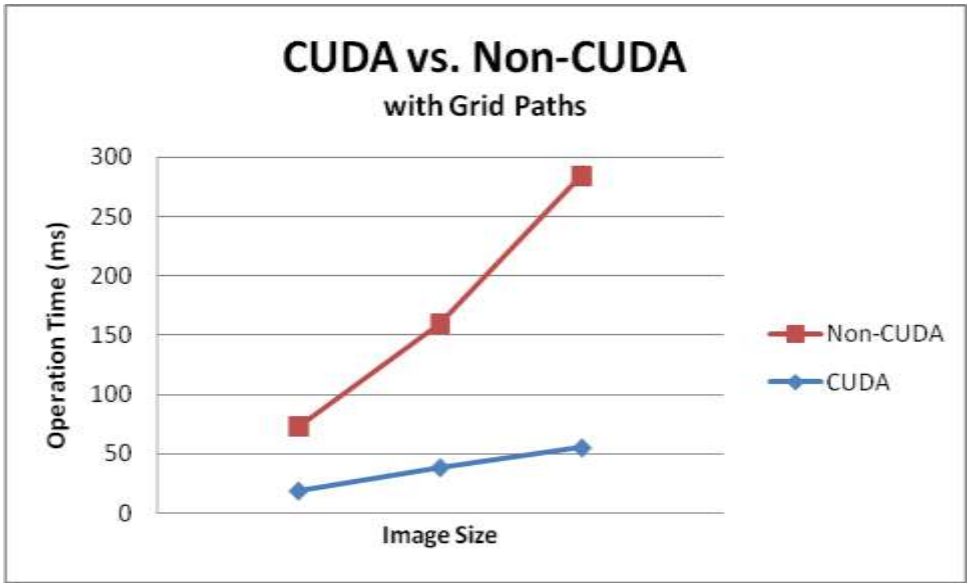

Figure 4.4: Summation of time required to execute the re-implemented segments of PathFinder by CUDA and by the original Java code. Includes the grid paths calculation.

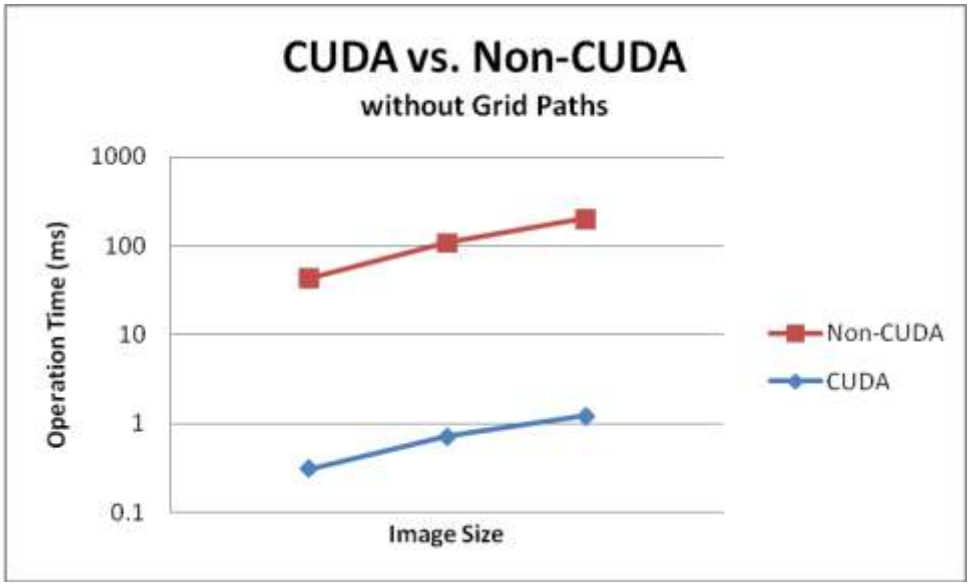

Figure 4.5: Summation of time required to execute the re-implemented segments of PathFinder by CUDA and by the original Java code. Does not include the grid paths calculation. Note the logarithmic scale of the y-axis.

Yet another way to see the difference before and after optimizations is to stack each part of the algorithm on top of the other. The taller the stack, the more runtime each segment

requires. As you can see in Figure 4.6, the CUDA-optimized transpose, convolution filter, and edge strength operations are an order of magnitude faster than their unoptimized versions or the grid paths calculation. With optimizations, all four parts of the algorithm are executed in the same or less time than the original transpose operation.

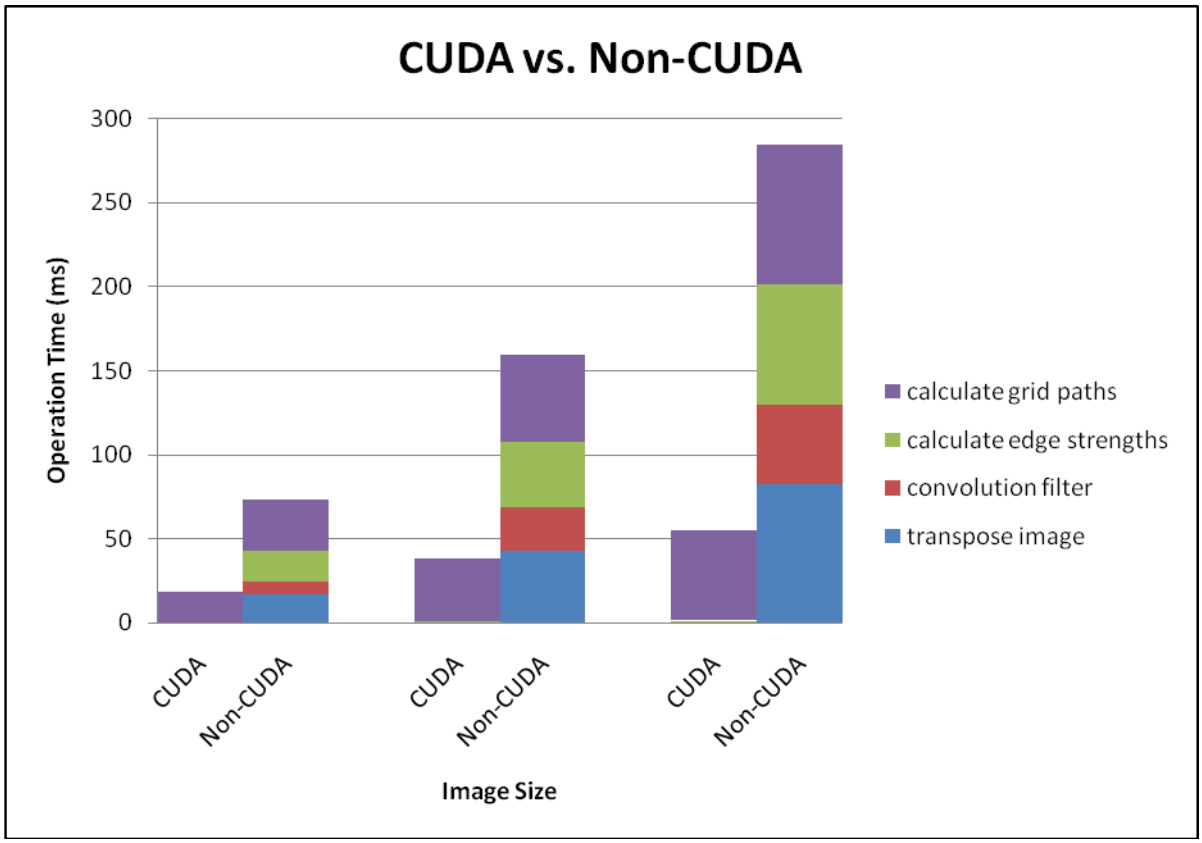

Figure 4.6: Comparisons of runtime of each task in CUDA and before optimizations. Note the CUDA executions of the edge strengths, convolution filter, and transpose operations are too fast to be visible on this timescale.

With these improvements, the transpose, convolution filter, and edge strength calculations that originally accounted for 25% of the original algorithm's runtime (see Figure 4.1) are now less than 1% of the entire algorithm (Figure 4.7). The grid paths calculation, though its runtime was improved, remains at 11%. As the sample raw data in Table 4.1 demonstrates, individual operation improvements of 400 times are possible. Overall, we

achieved a speed up of over 100 times with fully parallelizable operations and a 30% speed improvement of the more complex grid paths calculation. The result is a 3-5 times speed up across the four improved operations.

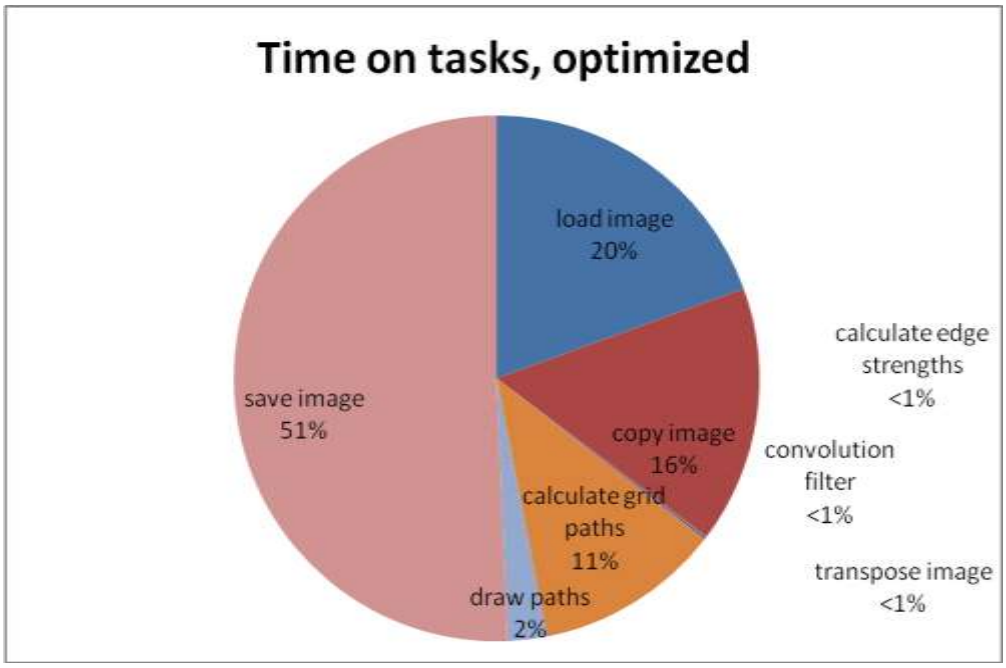

Figure 4.7: Each task performed by PathFinder, as percentage of runtime post-CUDA enhancements.

Table 4.1: Raw data from analysis of original Java PathFinder and PathFinder in CUDA. Data collected with a 400x304 resolution sample image. All times in milliseconds.

Task	Original PathFinder	CUDA-Optimized	Improvement Factor	Difference
load image	64.0	64.0		
copy image	52.7	52.1		
transpose image	43.1	0.11	406.9	43.0
convolution filter	25.9	0.51	50.8	25.4
calculate edge strengths	39.0	0.09	432.5	38.9
calculate grid paths	51.7	37.6	1.38	14.2
draw paths	7.6	7.8		
save image	158.0	166.7		
Total time	442.2	328.9		121.5

Chapter 5

Discussion and Future Work

5.1 Success of Project

As a proof-of-concept, this project has been a great success. We have shown that parallelizable code can be re-written to utilize an NVIDIA GPU via the CUDA platform with a significant performance advantage. Our overall result of an improvement factor of 100 is in line with previous CUDA projects (NVIDIA Corporation, 2010). We have also learned the best candidates for this process are simple code segments operating on a single block of memory. More complex code segments or operations that require distinct memory fragments do not benefit as much from parallelism in CUDA.

5.2 Limitations of the Java Native Interface

The JNI facilitated our study by allowing us to re-implement portions of the PathFinder algorithm to utilize CUDA within the time restrictions of this research project. The alternative approach would have been to re-write the entire algorithm, a lengthy and impractical task. Given that the goal of our research is increased performance, we have learned JNI is not a practical solution as we move forward.

Throughout our research, our benchmarking timers constantly showed a discrepancy between the runtime of our C code and the host Java code that called it. We identified this as overhead due to JNI. As shown in Figure 5.1, this overhead would typically be many times the actual runtime of the CUDA operation, nullifying any performance gains. At this time we see no means of circumventing this limitation of JNI, so a future step in the project will be

the complete removal of JNI. To accomplish this, a complete re-implementation of the PathFinder algorithm will likely be required. When we began this project our only option would have been a re-implementation in C, but a recent update to NVIDIA'S CUDA Toolkit has added C++ support (Ramey, 2010). Alternatively, there are several projects underway that could allow CUDA code to be called directly from Java (JCuda.org, 2010; Heusel, 2010), in addition to the possibility that NVIDIA will add support for Java to CUDA in the future.

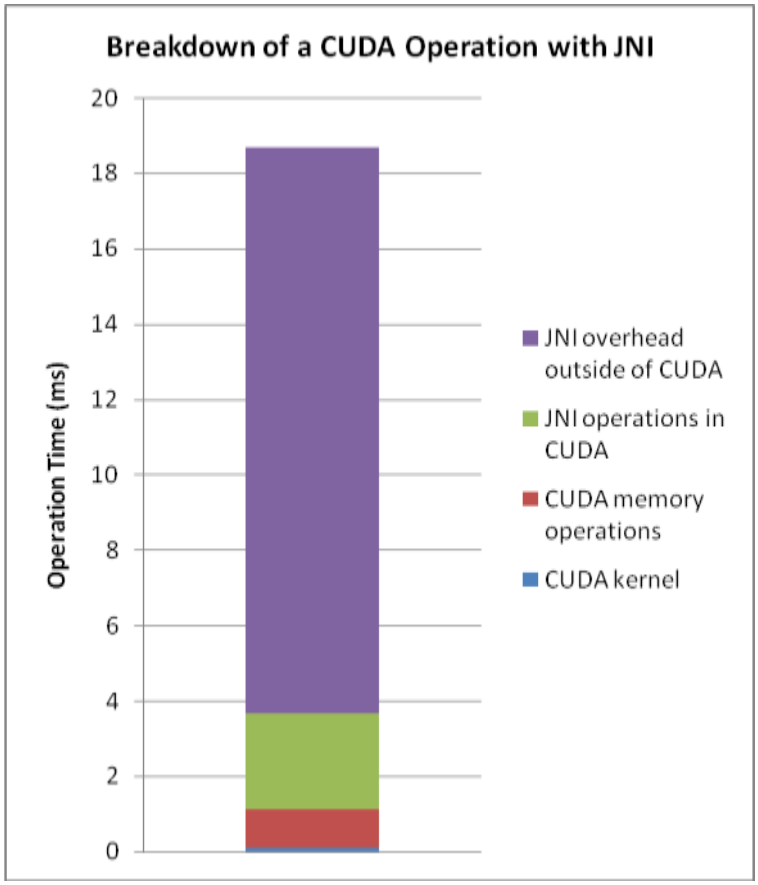

Figure 5.1: The benchmarks of each subsection of the transpose operation on a sample 400x304 image. The CUDA kernel, the sliver at the bottom, is the data presented in this paper. The CUDA memory operations are necessary, though unoptimized. The JNI operations and overhead will be removed entirely in future work.

5.3 Future Work

In addition to the removal of JNI, there exist more opportunities to improve the performance of PathFinder. The CUDA memory operations (see Figure 5.1) are likely candidates for optimization with the use of techniques such as memory coalescing. Further work must also be done to determine the best approach to optimizations of the grid paths calculation. While the current implementation is a slight improvement, there may be an alternative implementation that would benefit more from CUDA acceleration. There also remains the potential for improvements in the quality of the PathFinder algorithm. Finally, it would be interesting to investigate the performance and quality differences of a re-implemented PathFinder and the original EGBIS algorithm, similar to the recent work by Drucker (2009) with the original PathFinder and a re-implemented EGBIS algorithm.

The ultimate goal for the PathFinder algorithm remains to increase its performance until it can be utilized to conduct real-time analysis of video. At a rate of 20 frames per second, this allows 50 ms for each frame. Disregarding the question of file system operations, our work on this project has already reached this target, even for our largest test image (see Figure 4.6). Further improvements, as discussed above, will increase the practicality of this goal.

References

Moore, A., Prince, S., Warrell, J., Mohammed, U., & Jones, G. (2008). Superpixel Lattices. *IEEE Conference on Computer Vision and Pattern Recognition*, 1-8.

Drucker, F. (2009). Report on Research Project on the PathFinder image segmentation algorithm.

Fung, J., & Mann, S. (2008). Using graphics devices in reverse: GPU-based Image Processing and Computer Vision. *2008 IEEE International Conference on Multimedia and Expo*, 9-12.

Halfhill, T. (2008). Parallel Processing With CUDA. *Microprocessor Report*, 1.

NVIDIA Corporation (2009). CUDA Zone - Learn about CUDA. Retrieved September 28, 2009, from http://www.nvidia.com/object/cuda_home.html.

NVIDIA Corporation (2010). CUDA Zone – CUDA Community Showcase. Retrieved March 25, 2010, from http://www.nvidia.com/object/cuda_apps_flash_new.html.

Vineet, P. & Narayanan, P.J. (2008). CUDA cuts: fast graph cuts on the GPU. *GVGPU08*, 1-8.

Babenko, P. & Shah, M. (2008). MinGPU: A minimum GPU library for Computer Vision. *Journal of Real-Time Image Processing*, 3(4), 255-268.

Alluse, Y., Horain, P., Agarwal, A. & Saipriyadarshan, C. (2008). GpuCV: an open source GPU-accelerated framework for image processing and computer vision. *Proceedings of the 16th ACM international conference on Multimedia*, 1089-1092.

Nyland, L., Harris, M. & Prins, J. (2008). Fast N-Body Simulation with CUDA. In H. Nguyen (Ed.). *GPU Gems 3* (pp. 677-695).

Felzenszwalb, F. & Huttenlocher, D. (2004). Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 2(59), 167-181.

JCuda.org (2010). Retrieved March 23, 2010, from http://www.jcuda.org/.

Heusel, A. (2010). Project Jacuzzi. Retrieved March 23, 2010, from http://sourceforge.net/apps/wordpress/jacuzzi/.

Ramey, W. (2010). GPGPU Developers Get Boost from New CUDA Toolkit 3.0. Retrieved March 23, 2010, from http://blogs.nvidia.com/ntersect/2010/03/gpgpu-developers-get-boost-from-new-cuda-toolkit-30.html.

NVIDIA Corporation (2009). NVIDIA CUDA – Programming Guide. Retrieved October 10, 2009, from http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.