Team Sulley

Where's a Pattern?

CUDA C/C++ Pattern Recognition

Kenny Ballou, Abe Oros, Jesse Riggs,
Sasa Rkman, Kristin Van Andel
COMPSCI 354
Fall 2012
December 15, 2012

Contents

List of Figures

List of Tables

1 Introduction

This project utilized the massively parallel capabilities of NVIDIA GPU's through CUDA C/C++ to efficiently detect pre-defined patterns of pixels in images. The project involved learning about image processing (pattern recognition) and parallel processing as well as the syntax and capabilities of CUDA C/C++.

The ideal goal of this project was to dynamically detect "Waldo" in a standard "Where's Waldo?" image. However, due to the complexities of the typical "Where's Waldo?" image, the variability of "Waldo", as well as the challenges of dynamic pattern detection, it was anticipated at the outset this goal would probably never be achieved within the timeframe of the project. Instead, a series of increasingly difficult pattern detection milestones were set, with the flexibility to modify the milestones based on the successes and/ or challenges encountered at each step. The actual series of milestones that were undertaken and completed in this project are as follows:

1. Create a simple grey-scale image containing a single box composed of one grey-scale value and a background of one or more different grey-scale values. Detect all of the pixels that constitute the box and "white-out" any other pixels.
2. Repeat step 1, but using a blue box and a red background. Color processing, because it uses three channels (Red, Blue, and Green) instead of one, was expected to be more difficult than grey-scale processing.
3. Repeat step 1, but modify the program to compare pixels to detect the edge of the box and draw a border around it, rather than "white-out" the remaining / unmatched portion of the image.
4. Select a simple 16 x 16 pixel pattern in an image and update the program to detect this pattern and "white-out" all other pixels in the image.
5. Repeat step 4, but modify the program to draw a border around the detected pattern, rather than "white-out" the remaining / unmatched portion of the image.
6. Define a static pattern that constitutes "Waldo" in a specific "Where's Waldo?" image and modify the program to detect Waldo's pattern.

The following milestone was not achieved because a static approach was used for Milestone 6, which successfully identified a unique "Waldo" pattern in a color image with no false positives:

- Filter out false positives for "Waldo" that are detected by the program developed in step 6.

2 Approach

This project used CUDA C/C++ to perform massively parallel processing on the GPU to detect a specified pattern of pixels in a 1024 x 1024 pixel image. This approach was anticipated to be far more efficient than attempting to perform the same task in non-parallel processing on the CPU using traditional programming languages when multiple comparisons between pixels are required. Massively parallel processing is exactly what CUDA C/C++ is designed to do and image processing is a common application for this language.

Because of the combination of the challenges of learning the syntax specific to CUDA C/C++, the indexing of blocks and threads, and parallel programming, we took an evolutionary/additive approach to this project. We started with small programs to detect and mutate very simple patterns and worked up to larger programs to handle more complex patterns. The patterns ranged in complexity from individual pixel screening to a pre-defined 16 x 16 pixel pattern.

The individual pixel screening was performed on both color and grey-scale images. In these exercises, a large box of one color was placed in the center of the image with another color for the background. Pixels that did not have the color value of the pixels in the box were identified as background pixels and "whited-out"; that is, their color was changed to white. Although these exercises could be performed more efficiently in non-parallel, CPU processing (due to the expense of copying the array of pixels to the GPU and back in CUDA), these exercises were valuable in laying the groundwork for reading and writing image files and in learning how to index color and grey-scale pixels.

The edge detection exercise was conducted using the image from the grey-scale individual pixel screening exercise. The edge of the box was detected by comparing individual pixels to their neighbors and detecting a difference in pixel colors. The goal of this exercise was to develop further expertise in indexing pixels and to learn to make comparisons between pixels in different comparison of pixels in different blocks. The exercise also prompted the development of a strategy to allow comparison of pixels between different blocks and threads without causing synchronization issues.

In the 16 x 16 pixel pattern detection exercises, a known 16 x 16 pattern embedded in an image was detected by comparing each pixel in the 16 x 16 pattern to groups of pixels in the image. This process required comparisons between large groups of pixels with complicated offsets and indexing. Techniques from all of the previous milestones were used in these exercises to accomplish the whiting out of non-pattern pixels, outlining the pattern edge, and making numerous pixel comparisons between blocks and threads.

The stepwise, modular approach to this project worked well because it took into account the steep learning curve associated with learning CUDA C/C++ and parallel processing. Although most of the code is written in C/C++, or is very similar to C/C++, the notion of blocks, threads, and parallel/concurrent processes was very foreign at the beginning of this project. The challenge of this project was not so much a matter of learning new syntax and constructs as it was the difficulty of learning to look at programming in a whole new fashion. The discrete incremental approach applied to this project allowed us to start with simple programs and tasks; then build on each of those programs based on lessons learned and knowledge gained in previous tasks. This, in turn, made the project much less intimidating and allowed us to

measure successes and celebrate progress throughout the project.

3 Methodology

3.0.1 Milestone 1: Simple Grey Box

The first milestone was completion of a small program that would take as input a 1024 x 1024 image with a filled dark grey box on a light grey background. The program then used individual pixel screening to identify pixels as either part of the dark grey box or not, whiting out the latter. The program then created a new output image file and transferred the converted pixel pattern to the new file. The input image is shown in 1. The goal of Milestone 1 was to learn how to create, read, and write a portable grey-map (PGM) image file and become familiar with indexing blocks and threads in CUDA C/C++.

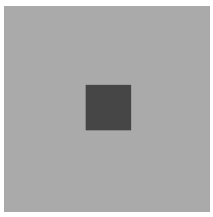

Figure 1: Sample Grey Box Input Image

IrfanView, a free-ware program available on the Internet, was used to convert bitmap (BMP) or portable network graphics (PNG) files to PGM files. The advantage of using PGM files for input and output is that the file format is quite simple, making parsing out pixel data incredibly simple. The file type simply contains 2 rows of metadata and some rows and some rows of comments preceded by a `#` symbol. The remainder of the file consists of binary data for each of the pixels in the image. The data in the PGM file is continuous and constitutes a flattened (one-dimensional) representation of the image. In a PGM file, there is only one data point (1 byte) for each pixel. Sample code provided by the instructor for reading the pixel color value data from a color image to an array was adapted for grey-scale images(PGM's) in this phase of the project. The code was further adapted to include writing of pixel data to a new PGM file.

The CUDA kernel is a subprogram that runs on the GPU. Th kernel includes all of the actions that must be performed on the blocks and threads on the GPU. Indexing threads and blocks in the first milestone was relatively straightforward. The thread capacity per block is specific to the architecture of the specific NVIDIA GPU a user's computer. In order to ensure that the code developed in this project would run on any CUDA-capable card, 128 threads per block was chosen as the standard for this project. An image size of 1024 x 1024 pixels was used throughout the project, which neatly divides the image into 8 blocks per row of 1024 pixels (128 threads per block) for a grey-scale image. Since every single pixel value needed to be examined in this program, no multipliers were needed to limit the threads and blocks to be examined.

The grey-scale values for the pixels that compose the box and the pixels that constitute the background were hard-coded for this first milestone. However, the program could be easily modified for images with other grey-scale values by changing the value of a `#define` value near the beginning of the code. This program will also work for an image with any type of shape, as long as the grey-scale value in the program is updated for the new image. This program was tested on multiple images and shapes to ensure that the success of the algorithm was not specific to the originally tested image and shape.

3.0.2 Milestone 2: Simple Color Box

The second milestone was essentially just an expansion of the program completed in the first milestone. This time a color image was used as the input, rather than a grey-scale image (see 2). The goal of this milestone was to learn to read , process, and write color image files as portable pixel-maps (PPM's). This exercise was expected to be more challenging than the first milestone because of the added complexities imposed by color images' multiple channels (Red, Green, and Blue or RGB) per pixel.

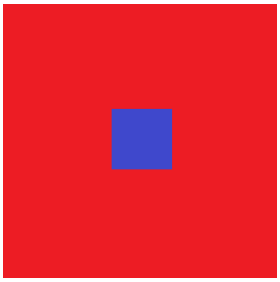

Figure 2: Sample Color Box Input Image

One of the primary concerns for this milestone was that the number of threads per block (128) is not divisible by three. This means the color channels for certain pixels would cross the boundary between blocks. This created concerns about synchronization of threads during processing, specifically access/ write violations. To alleviate this concern, a fourth "color" channel was inserted for each pixel in the image array. This fourth channel is more commonly referred to as the "alpha" channel. The alpha channel was assigned a value of zero as a default when reading the image data into a pixel array. Beyond reducing indexing complexities, it was anticipated the alpha channel may prove useful in later milestones.

In order to create the alpha channel for each pixel, the image file read and write portions of the code from the first milestone needed to be modified. Rather than reading all of the pixel directly into the array in one step, the pixel data was read three bytes at a time, with each byte being assigned to a separate index in the array. This was accomplished using a `for` loop, which was incremented by four on each iteration to allow space for the four color channels. The process was reversed when writing to the output file. A `for` loop, incremented by four, was also used. However, the output array was only three channels and because the alpha channel wasn't really used, the alpha channel was safely ignored.

The challenge in this color exercise was to determine how to access the pixel color values in groups of four, rather than one at a time as in the first milestone. We found that this could be accomplished by using a multiplier of four in the assignment expression for the thread ID's (`tid`'s) to be processed, which limited computation to every fourth thread on the GPU (beginning with thread 0). The first color channel for a pixel can be thought of as the pixel "base", with the other channels incorporated by reference from the pixel base in the calculations. This allowed all of the color channels for a single pixel to be evaluated and altered within the same set of calculations, which helped prevent synchronization issues between threads.

The color values for the pixels that compose the box and the pixels that constitute the background were hard-coded for this second milestone. However, the program can be easily modified for images with other color values by changing the values of certain `#define` values

near the beginning of the code. This program will work for an image with any type of shape, as long as the RGB values of the shape are unique in the image and the hard-coded RGB values in the program are updated for the new image. This program was tested on multiple images and shapes to ensure that the success of the algorithm was not specific to the originally tested image and shape.

3.0.3 Milestone 3: Grey Box Edge Detection

The third milestone expands on the program completed in the first milestone, but instead of whiting or filtering out the non-matching pixels, an outline is to be drawn around the matching pixels. The goal of this milestone was to begin making comparisons between pixels, rather than simply screening (accepting or rejecting) individual pixels. This program reuses the input image from 3.0.1 on page 5 (1 on page 5).

There are several ways to perform edge detection in an image. Some of these techniques are briefly described on Wikipedia and involve more complex comparisons involving gradient magnitudes or gradient rates of change. However, the common thread to all of these methods is comparisons between neighboring pixels. The approach used in this project also involves comparisons between neighboring pixels, but is highly simplified, dealing only with two hard-coded grey-scale values such that the edge falls wherever a neighboring pixel is of a differing color. The reason for this simplicity is because the goal of this milestone was to learn to make comparisons between pixels and not learn complex image processing techniques.

The approach to this milestone was developed with an eye toward future milestones. It would be easier to draw the outline along the inside edge of the box than along the outside edge of the box, because no special comparisons would be required to fill in the corners of the outline. Each pixel would only need to be compared to its four immediate neighbors in the horizontal and vertical directions. However, we were concerned that this approach would be problematic in future milestones when we were to outline small 16 x 16 pixel patterns. Overwriting the interior edge of the box would cause the loss of a fairly significant portion of the pattern and would not really achieve the goal of outlining the pattern. This would be particularly problematic if thickness of the outline were to be increased in future stages of the project. Therefore, we decided to take an "outside-in" approach to the pixel comparisons to avoid synchronization issues and ensure that the contents of the box would not be overwritten.

The "outside-in" approach was accomplished by checking non-box colored pixels for immediately adjacent neighbors that matched the color of the box. Comparisons were made in eight directions: horizontally in two directions, vertically in two directions, and diagonally in four directions. For this program the base pixel in the comparison was changed to white if its color was a match to the light grey background color and one of its neighboring pixels matched the dark grey color of the box this effectively draws an outline around the edge of the box, leaving the pixels within the box unchanged. Since the outline is only one pixel wide and diagonal comparisons were included, the corners of the box are successfully closed. Since only the base pixel was changed and its new color did not match either the background color or the box color, synchronization issues between blocks and threads were avoided.

One major concern in this milestone was the possibility of attempting to make pixel comparisons outside the boundaries of the image (out of bounds of the pixel array). There were two approaches that we could take to ensure this cannot occur. One option would be to limit the

range of thread index ID (`tid`) values (pixel indexes) that could be compared. However, there was concern that this could be increasingly complicated with thicker outlines and additional limitations to `tid`'s in the future milestones. So instead, rows of "buffer" pixels were added to both ends of the image to simplify the `tid` range restrictions. Since the outline had a thickness of one and diagonals were bing searched, the largest offset from the base pixel in the comparison was one full row of pixels plus one single pixel (diagonally up and left or diagonally down and right from the base pixel. Two additional buffer rows were added to the top and bottom of the image, which would more than cover the worst case offset, while maintaining relatively simple range checking for the `tid`'s.

Although the grey-scale values for the pixels that compose the box and the pixels that constitute the background were hard-coded in this program, the program could be easily modified for images with other grey-scale values by changing the value of the `#define` value near the beginning of the code. This program will work for an image with any type of shape, so long as the grey-scale values in the program are updated for the new image. This program was tested on multiple images and shapes to ensure that the success of the algorithm was specific to the originally tested image and shape.

3.0.4 Milestone 4: Grey-Scale 16 x 16 Pixel Pattern Detection

Milestone 4 involved finding a specific 16 x 16 pixel grey-scale pattern within a 1024 x 1024 grey-scale image. Although many of the techniques utilized in the earlier milestones were also used in this program, this milestone constituted a great leap forward in terms of the number of pixel comparisons to be performed. In this program, once the specified pattern is identified within the image, any image pixels that are not of the pattern are whited-out (changed to white). A sample pattern is shown in 3. A sample input image with embedded pattern is shown in 4.

Figure 3: Sample Grey-Scale Pattern Pattern

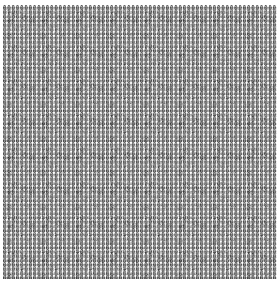

Figure 4: Sample Grey-Scale Input Image with Embedded Pattern

Numerous techniques exist for detecting patterns in images. A computer can be "trained" to recognize a pattern using a set of training data containing multiple occurrences of a specific type of object or pattern, such as a boat. Once the computer has identified a pattern that characterizes that object, the computer can search any image for occurrences of that object. However, such techniques often require the use of artificial intelligence techniques such as neural networks. The goal of this project was to advance our understanding and experience with

CUDA C/C++ programming, not to learn techniques of artificial intelligence. Therefore, pattern detection in this project was simplified to include only a static (fixed) pattern specific to one image. The pattern is then detected by selecting each pixel in the image once as a base pixel, then comparing it and its near neighbors sequentially to the 16 x 16 pattern. A 16 x 16 pixel pattern contains 256 pixels, which means that in order to determine if a pixel is part of the pattern, comparisons must be made to 256 of its near neighbors. Although this is a very naive and brute force method of pattern detection, it was a technique that could be implemented within the project timeframe, did not require learning advanced image processing techniques, and involved numerous calculations to showcase the efficiency of GPU processing in CUDA C/C++ versus CPU processing in straight C/C++.

Another approach would have consisted of using the mathematical formulas that define specific shapes to specific shapes in the image. For instance, finding a triangle of pixels might be possible if the formula for the outline of a triangle had been provided as a search criteria. However, this approach was considered too complicated for the timeframe of this project because of the types of comparisons that would be required between pixels and colors.

The pattern detection process was broken down into three steps: identify and flag the base pixel in the image (match to the upper leftmost pixel in the pattern), flag the other 255 pattern pixels in the image, and white-out all unflagged pixels in the image by changing their color to white. In order to avoid synchronization issues between blocks and threads, a separate kernel was used for each step. The second kernel is called when all blocks and threads have completed the calculations in the first kernel. Likewise, the third kernel is only called when the second kernel has completed its work. This ensures that the base pixel has been flagged before any attempt is made to identify the other 255 pattern pixels in the image and that all 256 pattern pixels have been flagged in the image before whiting out the non-pattern pixels. These three steps constitute three passes through the pixel data on the GPU and the kernels were named accordingly. An alpha channel was added to each of the image's grey-scale pixels to provide a place to flag each pixel as a base pixel or pattern pixel.

In the first pass, each pixel in the image is tested to determine if it matches the base pixel (the upper left-most pixel) in the 16 x 16 pattern. First, a flag variable is initialized to a value of 1 for the test pixel. Then nested for loops are used to iterate through each of the next 16 pixels (to the right of the pixel) in the test pixel's row, then through the same 16 columns in the next row of the image and so on until all pixels through the 16^{th} row and 16th column from the test pixel have been tested. If at any point in the nested loops a pixel is found that does not match the corresponding pixel in the pattern, the flag variable is set to 0 and two break statements are used to drop out of the nested loops, thus eliminating further unnecessary pixel comparisons. If the test pixel's flag variable still has a value of 1 at the completion of the nested loops, then the test pixel has met the criteria to be flagged as a base pixel in the pattern within the image. The value of the alpha channel for this pixel is then set to 1. Detection of more than one base pixel will occur if the pattern is embedded more than once in the image.

In the second pass, each pixel in the image is tested to determine if it falls within the pattern's boundary within the image. This is determined by testing each non-base pixel in the image to determine if it falls within the 16 x 16 columns and rows to the right and down from a base pixel in the image. These tests were again performed using nested for loops to iterate through the whole 16 x 16 range to the left and up from the test pixel that might contain a base pixel. If a base pixel is detected within the 16 x 16 boundary from the test pixel, the test

pixel is identified as being a member of the pattern. The pixel's alpha channel is then set to a value of 2, to indicate that it was identified as a pattern pixel within the image on the second pass. A 2 was used rather than a 1 to avoid synchronization issues between blocks and threads which might result in the pattern pixel being mistaken as a base pixel.

In the third pass, each pixel in the image is tested to determine if it is part of the pattern within the image. If the pixel's alpha channel value is 0, the pixel is not part of the pattern and its grey-scale value is changed to white. This effectively whites out all non-pattern pixels in the image, leaving the detected pattern easily identifiable in the output image.

3.0.5 Milestone 5: Grey-Scale 16 x 16 Pixel Pattern Detection with Outline

The fifth milestone expands on the program completed in the fourth milestone, but in this evolution, the pixels that do not compose the pattern are not whited out. Instead, a black outline is drawn around the edge of pattern. Unlike the outline in the third milestone, the outline in this program has a thickness greater than one and the thickness can be easily modified by changing a single `#define` value representing the thickness of the outline. The goal of this milestone was to improve upon the program created in 3.0.4 on page 8 by allowing the pattern to be demarcated within the context of the original image and to gain additional experience with even more complex pixel (block/thread) indexing. We decided to use an outline with a thicker outline than in 3.0.3 on page 7 because in complex images it would be difficult to see the outline and spot the pattern if the outline was only one pixel thick. Since it was not known what thickness would be ideal for spotting the outlined pattern in an image, we chose to write the code in such a way that it would be very easy to modify the thickness of the outline to achieve the desired effect. This program reuses the input image and pattern from 3.0.4 on page 8 (see 4 on page 8 and 3 on page 8 respectively).

Another approach that could have been taken for this milestone would be to alter pixel shades of the larger image that are not within the 16 x 16 pattern, rather than outlining the pattern. Making these pixels lighter and keeping the original shade on the embedded pattern would in essence, "bring out" the embedded pattern within the image. However, we decided we would rather improve on the earlier implementation of the outline by finding a way to increase the thickness of the outline by changing a single variable in the program.

The pattern detection and flagging process for this program is the same as in 3.0.4 on page 8. The first and second pass kernels did not change substantially from Milestone 4 (3.0.4 on page 8). However, the third pass kernel was changed to create the outline around the pattern, rather than whiting out non-pattern pixels. Many of the issues that were encountered in the third milestone also applied here such as avoiding synchronization issues, determining how to close the corners of the outline, and the need for buffer rows at the top and bottom of the image to allow diagonal comparisons through the first and last image rows. The number of buffer rows added to the beginning and end of the image was equal to the target thickness of the outline plus one. The "plus one" rows ensured that diagonal comparisons in the first and last rows of the image would stay within the boundaries of the pixel array.

The biggest challenge in this milestone was determining the algorithm to close the corners of the outline without a fixed value for thickness. However, after drawing out the first few rows and columns of the outline on paper and calculating the relative indexes of the pixels in the outline, a pattern of offsets emerged. Nested for loops were used for the comparisons, iterating

through an offset range of rows and columns equal to the target thickness of the outline. As in the third milestone, an "outside-in" approach was taken for the pixel comparisons to avoid synchronization issues and ensure that the contents of the pattern would not be overwritten. However, in this program the alpha channels of the pixels were compared where pixels with an alpha channel value of 0 were the "outside" pixels and pixels with an alpha channel value greater than 0 were the "inside" pixels.

3.0.6 Milestone 6: 16 x 16 "Waldo" Pattern Detection

The sixth milestone is essentially just an expansion of the program completed in the fifth milestone. This time, the program was designed to find and outline a 16 x 16 color pattern in a color image, rather than a grey-scale pattern and grey-scale image (see 5 and 6). This milestone was envisioned as the culmination of all of the previous milestones. In fact, this program utilizes nearly all of the techniques developed in the first 5 milestones. The only technique from earlier milestones that is not used in this program is individual pixel screening.

Figure 5: Sample Color Pattern

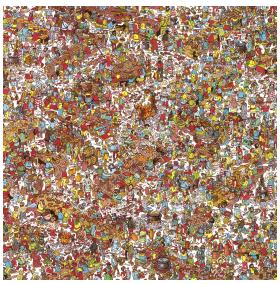

Figure 6: Sample Color Pattern Input Image with Embedded Pattern

Upon completion of the first color program (3.0.2 on page 6), we found the use of color for further project work a bit intimidating. However, after tackling more complex indexing issues and algorithms in the subsequent programs, working with color no longer seemed so intimidating at this phase of the project. The initial inspiration for this project was based around the popular "Where's Waldo" franchise and at this phase of the project we decided to proceed with a color implementation of the fifth milestone to be able to find "Waldo" in an image. However, we recognized that due to the limited project timeframe our approach would still need to utilize the naive, brute force pattern detection methods developed in the fourth and fifth milestones. Alternative approaches that were considered, but rejected due to timeframe, included a single pass approach with a new pattern detection algorithm, combining the values from the multiple color channels into a single unique "color" value, using a neural network to "train" the computer to characterize an object's pattern then search for it, or to use the alpha channel to represent the "weight" of the pixel in terms of its probability of being part of the pattern.

For our static approach to the "Where's Waldo?" problem to work, the "Waldo" pattern would be specific to a single image only as "Waldo" often varies in pose, shades of color, and/or scale between "Waldo" images. This meant that we would have to find "Waldo" in the image ourselves to crop the "Waldo" pattern for the program to use for the exercise. Visually detecting

"Waldo" in the image was, ironically, one of the bigger challenges of this milestone.

The only substantial change to the program written for the fifth milestone was the addition of two more color channels for each pixel. This required adjustments to the indexing in the program, but these adjustments were relatively simple after handling the complex indexing in the first two pattern programs. All of the pixel comparisons also needed to be expanded to include the two additional color channels. The pattern detection and outlining was still performed in three passes using only slightly modified versions of the three kernels used in the fifth milestone.

4 Results

The success of this project was measured against the established project milestones. Six of the seven projected milestones were completed. The seventh milestone was not completed because a different approach was taken to the sixth milestone, which resulted in no false positives to screen out for Milestone 7. The results of each of the six milestones are discussed below.

4.0.7 Milestone 1: Simple Grey Box

The output image generated from the Simple Grey Box exercise (3.0.1 on page 5) is shown in 7. The image shows a dark grey box on a white background, which demonstrates that all pixels that were not part of the dark grey box were successfully filtered out. This also demonstrates successful attainment of the secondary goals of learning the basics of image file processing and block and thread indexing in CUDA C/C++.

Figure 7: Simple Grey Box Output

4.0.8 Milestone 2: Simple Color Box

The output image generated from the Simple Color Box exercise (3.0.2 on page 6) is shown in 8. The image shows a dark blue box on a white background, which demonstrates that all pixels that were not part of the dark blue box were successfully whited out. This also demonstrates successful attainment of the secondary goals of learning the basics of color image file processing and block and thread indexing for multiple color channels in CUDA C/C++. However, at this phase in the project, dealing with the increased complexity of indexing multiple color channels while simultaneously increasing the complexity of pixel comparisons (for edge detection and pattern detection) was viewed as somewhat daunting. Therefore, it was decided that the next couple of milestones would be pursued using only grey-scale images.

Figure 8: Simple Color Box Output

4.0.9 Milestone 3: Grey Box Edge Detection

The output image generated from the Grey Box Edge Detection exercise (3.0.3 on page 7) is shown in 9. The image shows a dark grey box with a thin white outline on a light grey background, which demonstrates that the box was successfully outlined. This also demonstrates achievement of the secondary goals of learning the basics of edge detection in an image file and making comparisons between blocks and threads in CUDA C/C++.

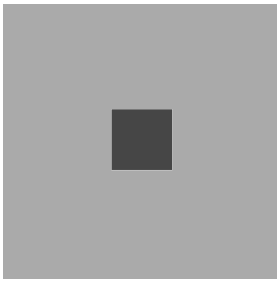

Figure 9: Grey Box Edge Detection Output

4.0.10 Milestone 4: Grey-Scale 16 x 16 Pixel Pattern Detection

The output image generated from the Grey-Scale 16 x 16 Pixel Pattern Detection exercise (3.0.4 on page 8) is shown in 10. The image shows a white background with a small 16 x 16 pattern, which demonstrates that the pattern was successfully isolated in the image. This also demonstrates the effectiveness of our simplistic approach to pattern detection using CUDA C/C++. This program was tested on multiple images and patterns to ensure that the success of the algorithm was not specific to the originally tested image and pattern.

Figure 10: Grey-Scale 16 x 16 Pixel Patern Detection Output

4.0.11 Milestone 5: Grey-Scale 16 x 16 Pixel Pattern Detection with Outline

The output image generated from this program (3.0.5 on page 10) is shown in 11 on the next page. The output image replicates the input image with the 16 x 16 pattern outlined with a thick black outline, which demonstrates that the pattern was successfully isolated in the image. The program was tested at least 3 times with different values assigned to the `#define` value T, which represents the thickness of the outline. In all of the tests the outline appeared in the image with the expected thickness and properly closed corners. This demonstrates the effectiveness of the algorithm for producing the thicker outline with an easily adjustable thickness. It also demonstrates our success in meeting the increased indexing challenges of this milestone. This program was tested on multiple images and patterns to ensure that the success of the algorithm was not specific to the originally tested image and pattern.

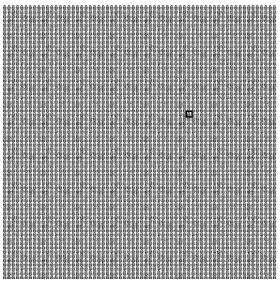

Figure 11: Grey-Scale 16 x 16 Pattern Detection with Outline Output

The Milestone 5 (3.0.5 on page 10) input image shown in 4 on page 8 was modified to include multiple embedded copies of the 16 x 16 pattern shown in 3 on page 8. The output image from the program with this new set of images is shown in 12. This output image demonstrates that the program can successfully detect multiple instances of the same embedded pattern within an image. In fact, all three of the pattern detection programs written for this project (3.0.4, 3.0.5, and 3.0.6) are capable of detecting and demarcating (through whiting-out non-pattern pixels or outlining the pattern pixels) multiple instances of the same embedded pattern. However, none of the programs are configured to accept more than one static pattern as an input.

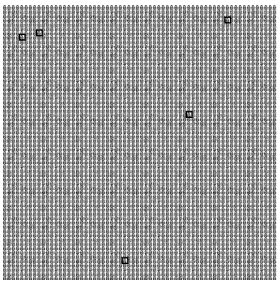

Figure 12: Grey-Scale 16 x 16 Pixel Pattern Detection with Multiple Instances of the Same Embedded Pattern

4.0.12 Milestone 6: 16 x 16 "Waldo" Pattern Detection

The output image generated from this program (3.0.6 on page 11) is shown in 13. The output image replicates the input image with the 16 x 16 pattern outlined with a thick green outline, which demonstrates that the pattern was successfully isolated in the image. This program was tested on multiple images and patterns to ensure that the success of the algorithm was not specific to the originally tested image and pattern.

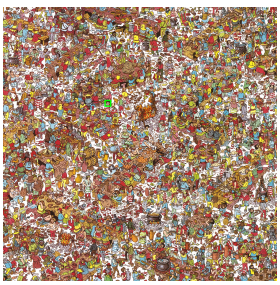

Figure 13: Color 16 x 16 Pixel Pattern Detection with Outline Output

4.0.13 Running Times

One of the major selling points of CUDA C/C++ is the potential to significantly decrease program running time through the use of parallel processing on NVIDIA GPUs. As a result, we were interested in comparing the running times of our CUDA C/C++ programs to equivalent programs in C. Converting the CUDA C/C++ programs to C was relatively simple as only a few syntax changes were required to convert the CUDA kernels to C functions and remove all of the data copying to and from the GPU.

It was anticipated that the first three programs (simple shape detection) might have longer running times in CUDA C/C++ than in C due to the large amount of data copying from the CPU to the GPU and back versus the relatively small number of computations. However, it was hoped that the pattern detection programs would run faster in CUDA C/C++ than in C due to the comparatively large number of calculations that would be performed in each kernel (or function). The running times of the programs were tested on Onyx on the computers in the MetaGeek lab to facilitate repeatability of the results. The results of the test are shown in 1.

Milestone	**CUDA Running Time**	**C Running Time**
Milestone 1 (3.0.1)	0.157 (secs)	0.052 (secs)
Milestone 2 (3.0.2)	0.257 (secs)	0.133 (secs)
Milestone 3 (3.0.3)	0.201 (secs)	0.065 (secs)
Milestone 4 (3.0.4)	0.303 (secs)	0.953 (secs)
Milestone 5 (3.0.5)	0.355 (secs)	1.227 (secs)
Milestone 6 (3.0.6)	0.620 (secs)	1.278 (secs)

Table 1: Average Running Time Comparison

The results of running time testing of both the CUDA C/C++ programs and the equivalent C programs were as anticipated. The first three programs (3.0.1, 3.0.2, and 3.0.3) that had very few computations in the kernels (or functions) were slower (overall) on the GPU than on the CPU. However, both versions had running times that could be measured in 10^{ths} of a second or less and were both relatively fast. The copying of data to and from the GPU is believed to be the cause of the longer CUDA C/C++ running times since the number of computations performed was relatively small. As hoped, all three of the pattern detection programs ran faster on the GPU with CUDA C/C++ than on the CPU with C. As the complexity of the pattern detection programs increased, the running time increased in both CUDA C/C++ and C and the ratio of the running time in CUDA C/C++ to the running time in C increased. The ratios for the programs in Milestone 4: Grey-Scale 16 x 16 Pixel Pattern Detection, Milestone 5: Grey-Scale 16 x 16 Pixel Pattern Detection with Outline, and Milestone 6: 16 x 16 "Waldo" Pattern Detection were approximately 32%, 30%, and 49%, respectively. If the programs were implemented for a series of images and/or patterns, which is a common application in image processing, the time savings using parallel processing on the GPU could be immense.

Large Images To really drive home the speed benefit of CUDA over C, we did a series of tests of our program from 3.0.6 on page 11, passing it an image approximately 7 Megapixels (MP or million pixels). To get a sense of this size, our image sizes throughout the duration of the project were only 1 MP. In the following table (2 on the following page), you can see the comparison of those tests. All tests were run on a node in the Onyx cluster.

	CUDA Running Times	**C Running Times**
	2.94 (secs)	7.90 (secs)
	2.94 (secs)	7.87 (secs)
	2.93 (secs)	7.90 (secs)
	2.93 (secs)	7.87 (secs)
	2.93 (secs)	7.89 (secs)
Average	2.93 (secs)	7.89 (secs)

Table 2: Large Image Running Times Comparisons

As we can see from the table, and image (15 on the next page) about 7 times larger than the previous tests, is searched for a 16 x 16 pattern (14) about 270% faster using CUDA C/C++ compared to standard C; this number is similar to the speed increase of CUDA over C in the previous table (1 on the previous page) as well.

The following images are the pattern that was searched and the image searched, respectively.

Figure 14: Large Image Test Pattern

Figure 15: Large Image Test

Notes We would have liked to test with much, much larger images (approximately 40 MP), however, due to the calculated block dimension launch size, we could not actually get CUDA to search the image. This is because *all* NVIDIA GPU's have a hardware limitation, limiting block dimension launch sizes to 65,535; the 40 MP image was attempting to launch with a block dimension about 3 orders bigger than this limitation.

5 Discussion

The goal of this project was not to learn complex image processing techniques, but to develop some expertise with simple GPU parallel programming with CUDA C/C++. Although more efficient and/or dynamic methods of pattern and edge detection exist, learning the computations, science, and/or complex artificial intelligence tools necessary to implement these methods was well beyond the timeline and scope of a single semester project. Therefore, we can't envision any other approaches we could have taken to the image processing algorithms used in this project. There is really nothing about this project that we would do differently given the timeframe of the project.

The tasks (milestones) defined for this project took into account the steep learning curve associated with learning CUDA C/C++ and parallel processing. The evolutionary, sequential approach to the project milestones also allowed us to take the lessons learned at each milestone to determine the approach for the next milestone. This approach made the project much less intimidating in the beginning and allowed us some flexibility to change the remaining milestones if the findings in an early milestone indicated an obstacle to completing a later milestone within the project timeframe. The downside of this approach was that the programming tasks could not really be divided up among the team members to allow more work to be performed within the project timeframe. In order to contribute to the work for the later milestones, each team member would need to have some active participation in the earlier programming tasks to understand and apply the methodologies and code to later milestones. However, the evolutionary, sequential approach to this project worked well overall.

Despite the relative inefficiency, naivety, and static nature of our image processing algorithms, we did see some reductions in the running time of our code on the GPU versus the CPU in the pattern recognition programs. The scope of our image processing was relatively small - a single 1024 x 1024 pixel image embedded with a single 16 x 16 pixel pattern. However, even with this relatively small scope, the running time of the final color pattern program (3.0.6 on page 11) in CUDA C/C++ on the GPU was approximately half the running time of an equivalent C program on the CPU. In real world image processing applications, numerous images may be processed in series either to define a pattern that characterizes as object or to detect or trace a pattern through the series of images. Even if more efficient image processing techniques were used for such an application, our results suggest that substantial time savings could be achieved by performing these tasks with CUDA C/C++ on the GPU. Overall our results demonstrate the promise of parallel processing in CUDA C/C++ for reducing the time required to perform numerous calculations on large volumes of data. However, it should be noted that due to the time cost of copying data to the GPU and back to the CPU outweighs the time savings of the parallel processing on the GPU for small data volumes and/or simple calculations. This was demonstrated by the running time results in the first three programs (3.0.1, 3.0.2, and 3.0.3) in this project.

6 Conclusion

Prior to undertaking this project, most of the members of the team had little to no experience with parallel processing. Several of the team members had some experience with C and/or C++, but most of the team had no experience in working with CUDA, despite its integration of C/C++. In the grand scheme of things, the steps taken in this project could be seen as baby steps towards learning CUDA C/C++, parallel processing, and image processing. However, CUDA C/C++ is still a relatively new language/application and the number of programmers with expertise or experience using CUDA C/C++ is relatively small. Therefore, the basic knowledge that we gained of both CUDA C/C++ and parallel processing could prove to be extremely useful in the future. If we had more than just a semester to complete this project, we would have liked to learn and implement more advanced image processing techniques for edge detection and pattern detection. We also would have liked to learn to implement a neural network to "train" the computer to characterize the pattern of a particular type of object ("Waldo", for example) and dynamically detect it in a series of images. Having seen the potential of parallel processing on the GPU through this project, several of the team members have been inspired to continue practicing and learning CUDA C/C++ after the semester is over, to the extent that the busy schedule of a computer science undergraduate permits.