PrewittVision: Discussion and Analysis

By Mitchell Carroll and Matthew To

Table of Contents

1.0 Introduction….………..………………………………………………………………………………………… 3

2.0 Workflow and Algorithms…….……………………………………………………………………………. 4

2.1 Grayscale Conversion.……………………………………………………………………………. 5

2.2 Edge Detection and the Prewitt Operator………………………………………………. 5

2.3 Image Dataset and Output……………………………………………………………………… 6

3.0 Analysis of Sequential and Parallel Algorithms…………………………………………………… 7

3.1 Grayscale Processing Algorithm……………………………………………………………… 7

3.1.1 Multi-threading performance with OpenMP………………………………. 8

3.1.2 Multi-threading performance with CUDA……………………………………. 9

3.2 Edge Detection Algorithm……………………………………………………………………….. 10

3.2.1 Multi-threading performance with OpenMP………………………………… 10

3.2.2 Multi-threading performance with CUDA…………………………………….. 11

3.3 Processing Multiple Images……………………………………………………………………… 11

3.4 Performance of Workflow and Amdahl’s law……………………………………………. 12

4.0 References…………………………………………………………………………………………………………… 13

***1.0 Introduction***

PrewittVision is an efficient image processing application that leverages parallel programming techniques using CUDA to perform accelerated edge detection on .png images. By implementing the Prewitt Operator on blocks of pixels in parallel, PrewittVision delivers high-performance gradient image generation, highlighting detected edges.

Edge detection is a type of algorithm used to determine the boundaries of objects within an image, or more technically, defined by sharp changes in brightness. This results in a simplified image output with objects resembling curves. The application of edge detection is numerous, with one of the main applications of this would be used for object detection. Another application of edge detection is detecting change. With a series of images applied with edge detection, we can extrapolate changes in an image by comparing changes to the pixel values.

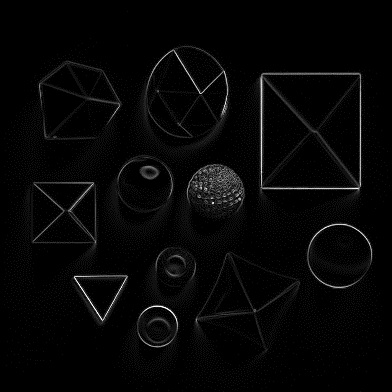


Figure 1: An image with a bunch of shapes (left), image after applying edge detection with the Prewitt operator (right)

In figure 1, we can see an example of edge detection in action. The original image is transformed by the edge detection algorithm using the Prewitt operator. We can that the algorithm is able to create white curves that illustrate the intensity of the changes in brightness. Shadows formed by the objects on the left images do not have outlines, because the shadows have a gradient effect that does not reflect a sharp change in shade or colour. However, certain shapes, for example the red pyramid in figure 1 is lightly highlighted, and the dark blue ball in the middle is not fully highlighted as the colour transitions in tandem with the black shadow. This can be worked around by using a threshold to select pixels that reach a certain brightness after edge detection. Despite its shortcomings and simplicity, edge detection is a powerful tool that can be used to illustrate objects and provides an output with a simpler dataset than if it was a full RGB dataset.

***2.0 Workflow and Algorithms***

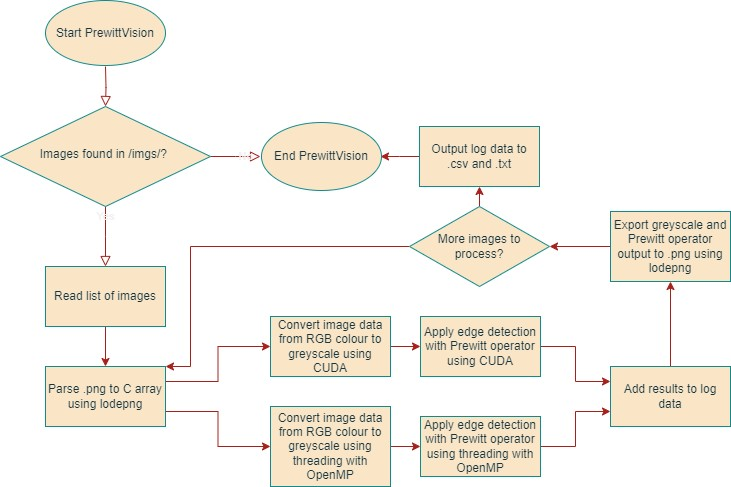


Figure 2: Workflow of PrewittVision

PrewittVision takes a list of images found within a path defined as /imgs for our project, iterates through each image (may be done in parallel with multi-threading), parses the image into a char[] array, converts the image to grayscale, and then performs the edge detection algorithm on the grayscale image. Finally, the image is saved in the output folder, and a log is generated.

*2.1 Grayscale Conversion*

When an image is converted from .png to a char[] array in C, the data is stored in a linear array that stores each pixel as a raw buffer formatted in sequences of RGBA (red, green, blue, alpha/opacity). A 2x2 image with pixels of the colour blue and no alpha would look like [0, 0, 255, 0, 0, 0, 255, 0, 0, 0, 255, 0, 0, 0, 255, 0]. The algorithm used to convert an image to grayscale works by taking a multiplier ratio of the brightness of each value of the colour channels of RGB and adding it up. For each pixel value, the conversion equals red \* 0.299 + green \* 0.587 + blue \* 0.114.

*2.2 Edge Detection and the Prewitt Operator*

The Prewitt operator works by applying a filter to each pixel two kernels that determine the gradient of both the horizonal and vertical directions to determine the change in brightness. The equation for calculating the horizontal gradient is , and the vertical gradient is , where A is the original image pixel. Finally, the gradient magnitude is calculated , and the result is applied to each individual pixel (Prewitt, 1970).

*2.3 Image Dataset and Output*

Table 1: A list of images used for testing input and output.

|  |  |  |  |
| --- | --- | --- | --- |
| File Name and Resolution | Original Image | Greyscale Output | Prewitt Output |
| objects,  1028x1028 |  |  |  |
| objects2,  1028x1028 |  |  |  |
| computer\_chip,  1536x768 |  |  |  |
| fruit\_basket, 1098x1200 |  |  |  |
| husky\_cardinal, 1024x1024 |  |  |  |
| network, 1536x768 |  |  |  |
| rooster, 1098x1200 |  |  |  |

***3.0 Analysis of Sequential and Parallel Algorithms***

Table 2: Specifications of the computer that was used to get these benchmark metrics

|  |  |
| --- | --- |
| OS | Ubuntu 22.04 on WSL 2 running on Windows 11 |
| CPU | Intel Core i7 13700k (8 performance cores + 8 efficiency cores) |
| RAM | 32 GB (DDR5 6000 mhz) |
| GPU | Nvidia GeForce RTX 4070 Ti (7,680 CUDA cores, 12 GB GDDR6X) |

PrewittVision has three areas where parallelism can be applied: when converting an image to greyscale (OpenMP or CUDA), when applying the edge detection algorithm with the Prewitt operator (OpenMP or CUDA), and calling multiple images to be processed using multi-threading with OpenMP. We will discuss in depth the performance of the algorithm used when converting the original image to greyscale, and the algorithm running the edge detection algorithm.

*3.1 Grayscale Processing Algorithm*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Performance (ms)** | **CPU/OpenMP by threads** | | | | **CUDA by block size** | | | | | |
| *File Name* | *1* | *2* | *6* | *16* | *4* | *8* | *16* | *32* | *64* | *128* |
| objects.png | 4.304 | 2.544 | 1.206 | 5.632 | 0.137 | 0.044 | 0.03 | 0.045 | 0.0016 | 0.0014 |
| objects2.png | 3.623 | 2.053 | 1.266 | 3.77 | 0.197 | 0.034 | 0.042 | 0.044 | 0.0014 | 0.0014 |
| computer\_chip.png | 3.254 | 2.118 | 1.267 | 9.108 | 0.124 | 0.2 | 0.4 | 0.025 | 0.0014 | 0.0014 |
| fruit\_basket.png | 3.674 | 2.145 | 1.19 | 3.069 | 0.145 | 0.031 | 0.05 | 0.262 | 0.0015 | 0.0014 |
| husky\_cardinal.png | 3.586 | 2.818 | 2.488 | 6.157 | 0.135 | 0.039 | 0.045 | 0.136 | 0.0014 | 0.0014 |
| network.png | 5.215 | 2.868 | 1.144 | 3.743 | 0.137 | 0.044 | 0.03 | 0.045 | 0.0014 | 0.0014 |
| rooster.png | 3.949 | 3.087 | 1.006 | 0.842 | 0.15 | 0.052 | 0.028 | 0.032 | 0.0014 | 0.0014 |
| Average | 3.944 | 2.519 | 1.367 | 4.617 | 0.146 | 0.063 | 0.089 | 0.084 | 0.0015 | 0.0014 |
| Parallel Overhead |  | 1.094 | 4.257 | 69.933 |  |  |  |  |  |  |
| Parallel Speedup |  | 1.566 | 2.885 | 0.854 | 26.932 | 62.173 | 44.168 | 46.868 | 2704.252 | 2791.768 |
| Parallel Efficiency |  | 0.783 | 0.481 | 0.053 |  |  |  |  |  |  |

Table 3: Performance metrics when applying grayscale conversion to different parallel implementations. Performance is measured in milliseconds (ms) from start to finish.

Converting an image to grayscale is a simple process that can be optimized with parallel programming as the data structure it is processing on is an array. For each operation to calculate each pixel, it must access three sequential array elements, and perform a simple addition and multiplication operation as discussed in section 2.1.

*3.1.1 Multi-threading performance with OpenMP*

From the benchmarks above in table 3, we can see that when using multi-threading with OpenMP, there’s an increase of performance up to 6 threads, however less efficient when running 2 threads. One possible reason for the reduction in efficiency is since the greyscale algorithm fetches the RGB values from the image array stored at the heap, and the memory bandwidth is limited its efficiency when using more threads.

At 16 threads, the performs perform worse than if the program was running sequentially with one thread. One possibility for this slowdown may be attributed to the parallel overhead when trying to use both the performance and efficiency cores of the Intel CPU used for the benchmark. This overhead can either be caused by background processes from the OS or another application, or the reduced performance of an efficiency core. It’s also worth noting that hyper-threading is a feature within performance cores which gives it two threads per core, however hyper-threading does not deliver twice the performance in one core, but rather allows two threads to more efficiently execute in tandem. Hyper-threading may be detrimental if two threads were to be used, as that could theoretically half the performance for two threads called from OpenMP.

By design, 8 of the cores are designed to be more performant than the other 8 efficiency cores, and the efficiency cores are usually delegated to background tasks. TechPowerUp benchmarked the difference between a performance core and an efficiency core, and found that performance cores outperformed efficiency cores by over 50% (W1zzard, 2021). In table 2, we see that rooster.png was converted faster with all 16 threads than with 1-6 threads, hinting that there is possibility of performance increase which may have been due to less overhead from background tasks, or better allocation of threads if hyper-threading is the culprit.

*3.1.2 Multi-threading performance with CUDA*

We’ve benchmarked the performance of various block sizes from 4 to 128. We can immediately see that the performance increase drastically when the block size is at 64 or 128. The block size is recommended to be at least 32 and a multiplier of 32 since each warp consists of 32 threads. This is expected, because selecting 4-16 blocks would result in 50-75% unused CUDA cores being processed and would be less efficient, however using a block size of 32 showed similar performance to 8 or 16 blocks if we assume the variations to be anomalous. This suggests that having a larger block size may be more efficient as there are more threads running concurrently, as the algorithms will be processing array sizes with over a million elements.

*3.2 Edge Detection Algorithm*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Performance (ms)** | **CPU/OpenMP by threads** | | | | **CUDA by block size** | | | | | |
| *File Name* | *1* | *2* | *6* | *16* | *4* | *8* | *16* | *32* | *64* | *128* |
| objects.png | 29.667 | 15.182 | 5.119 | 3.403 | 0.074 | 0.052 | 0.05 | 0.115 | 0.0014 | 0.0015 |
| objects2.png | 29.946 | 16.172 | 5.056 | 3.454 | 0.081 | 0.054 | 0.052 | 0.175 | 0.0015 | 0.0014 |
| computer\_chip.png | 34.52 | 17.524 | 5.983 | 3.521 | 0.134 | 0.047 | 0.049 | 0.159 | 0.0014 | 0.0014 |
| fruit\_basket.png | 38.477 | 19.135 | 6.635 | 4.334 | 0.069 | 0.035 | 0.043 | 0.106 | 0.0014 | 0.0014 |
| husky\_cardinal.png | 29.981 | 15.432 | 5.11 | 3.447 | 0.077 | 0.054 | 0.05 | 0.051 | 0.0016 | 0.0015 |
| network.png | 33.809 | 17.41 | 6.131 | 3.352 | 0.083 | 0.048 | 0.043 | 0.048 | 0.0015 | 0.0014 |
| rooster.png | 29.785 | 14.931 | 5.117 | 3.417 | 0.077 | 0.051 | 0.157 | 0.055 | 0.0015 | 0.0014 |
| Average | 32.312 | 16.541 | 5.593 | 3.561 | 0.085 | 0.049 | 0.063 | 0.101 | 0.0015 | 0.0014 |
| Parallel Overhead |  | 0.770 | 1.246 | 24.666 |  |  |  |  |  |  |
| Parallel Speedup |  | 1.953 | 5.777 | 9.074 | 380.143 | 663.299 | 509.426 | 319.020 | 21951 | 22297 |
| Parallel Efficiency |  | 0.977 | 0.963 | 0.567 |  |  |  |  |  |  |

Table 4: Performance metrics when applying edge detection with the Prewitt operator to different parallel implementations. Performance is measured in milliseconds (ms) from start to finish.

Similar to converting an image to grayscale, applying an edge detection algorithm with the Prewitt operator works by applying a gradient vector and convoluting it into a single pixel value. While the principle of implementing multi-threading is the same for grayscale and edge detection, edge detection is more intensive, as we can see that one thread takes approximately 3 seconds to convert to greyscale, while edge detection takes 30 seconds. Despite being more computationally intensive, it showed better scaling with regards to more threads and GPU programming, likely due to overhead with memory access.

*3.2.1 Multi-threading performance with OpenMP*

With multi-threading, performing edge detection with the Prewitt Operator is very efficient up to 6 threads with an efficiency of 96.3%, nearing almost 6 times the performance as if it was executed sequentially. At 16 threads, there is still noticeable performance gain, however each core only offers about 56% efficiency. As discussed in section 3.1.2, the loss of efficiency beyond 6 cores may be due to the parallel overhead of using efficiency cores; either due to the lower performance of the cores, or due to background processes consuming priority of the threads.

*3.2.2 Multi-threading performance with CUDA*

Edge detection with CUDA regardless of block size is several times greater than multi-threading with OpenMP due to the vast number of threads used with GPU computing. With a block size of 64 or greater (with a multiple of a warp size of 32), the performance is as great as 22,000 times faster than with sequential execution, despite the GPU used in the benchmark only having 7,680 CUDA cores. The increase in performance being greater than the number of cores in the GPU may be due to the memory bandwidth or latency being faster on the GPU than it is for the CPU trying to access the heap memory. Since CUDA performs memory access on blocks, it may also be the reason it is faster than with individual CPU cores performing memory access sequentially.

*3.3 Processing Multiple Images*

Another area where parallelism can be achieved is by processing images concurrently. For example, if there are 8 images in the /img folder, PrewittVision will be able to process each individual image and call CUDA to process each image instantaneously. Within each loop, the application converts the image to a char[] array using LodePNG, apply greyscale conversion and edge detection, add to logging, and save the image to a defined location.

*3.4 Performance of Workflow and Amdahl’s law*

We can apply Amdahl’s to determine how much of the application can be improved by parallelization. The entire application when running sequentially with one thread through the entire dataset of 7 images takes approximately 250ms with 35ms for processing greyscale and edge detection for each image. The remaining 5ms would be resources spent searching for images within the /img folder, converting images to and from a .png, creating logging, and the rest of the application. The time the code takes before and after processing an image is negligible (approximately 0.01ms), and can be discounted. Based on section 3.3, we can use OpenMP to use multi-threading. This means in theory, 245 out of 250ms can be parallelized down to 35ms. Using the performance metrics with CUDA in table 3 and 4, we can further increase the performance to a fraction of a millisecond, less than 0.01ms precisely. This leaves us with a theoretical max speedup (with 7 images) of 5ms, or 98% of the code is parallelizable.

***4.0 References***

Prewitt, J. M. S. (1970). Object Enhancement and Extraction. *Picture Processing and Psychopictorics*, 75–149.

W1zzard. (2021, November 19). *Intel Core i9-12900K E-cores only performance review*. TechPowerUp. https://www.techpowerup.com/review/intel-core-i9-12900k-e-cores-only-performance/2.html