Parallel vs. Serial Performance Analysis of Image Convolution

Eddie Shim
MPCS 52060 Parallel Programming
eddieshim@uchicago.edu

05/27/2020

Overview

In this assignment, I will be testing the serial and parallel versions of an image filtering program I wrote, `editor.go`. This program uses image convolution in order to produce a variety of filtering effects given by the user. As input, the user provides instructions through JSON Stdin tasks which detail a PNG image's input filepath, output filepath, and array of effects (sharpen, edge-detection, blur, or greyscale).

Each of these effects transforms a given image by using convolution with a unique kernel. For more information about the convolution, please see this reference:[1]. In essence, it's a pixel value calculation that involves summing and multiplying each of the image's pixel values by a given matrix (kernel).

The script I wrote performs image processing in either parallel or serial execution, where the user can specify the number of threads to use as a command line argument using the flag `-p=numThreads`. In the parallel version of this program, `editor.go` will first spawn a number of reader threads ($n = \lceil numThreads * .2 \rceil$) that take read in JSON arguments from Stdin in $blockSize = 2$ chunks each repetition. Each reader can work independently and in parallel. From here, each reader then spawns a single worker thread which utilizes a take-and-repeat pipeline structure to process each individual effect. Since the output effect is dependent

[1] http://www.songho.ca/dsp/convolution/convolution2d_example.html

on the past effect, they must run in order serially. Within each pipeline effect, however, is another level of parallelism. To apply a single effect we break down the given image into $n = numThread$ subimages, where each subimage is a section of the original image that's been sliced horizontally n times. We spawn n threads to process convolution on each subimage independently, then rejoin all the subimages before moving onto the next effect. Finally, after all effects have been processed we spawn a writer thread to write our image to our given output filepath.

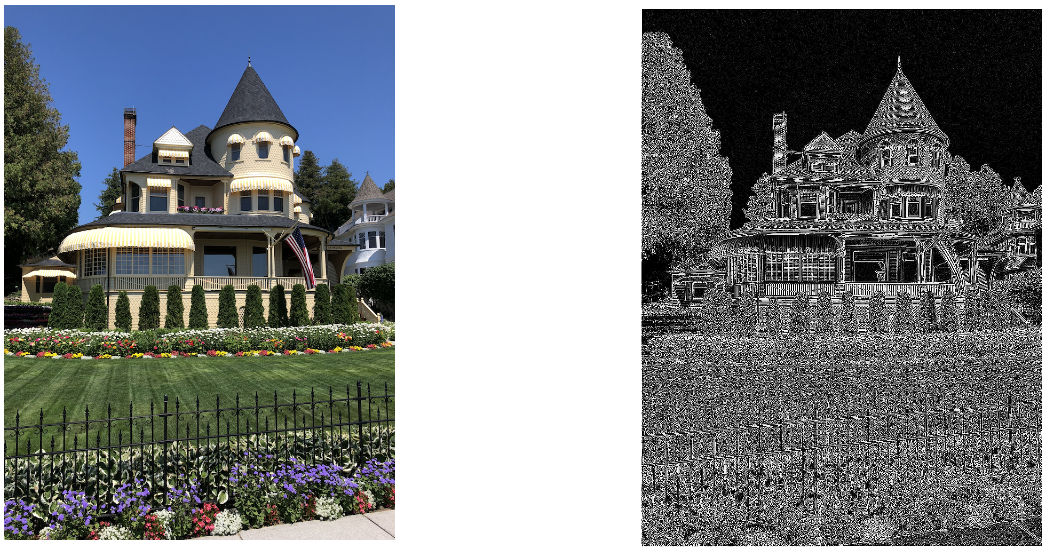

Figure 1: Example of an image filtered with Greyscale, Sharpen, and Edge-detection

Below is a rough diagram of our structure:

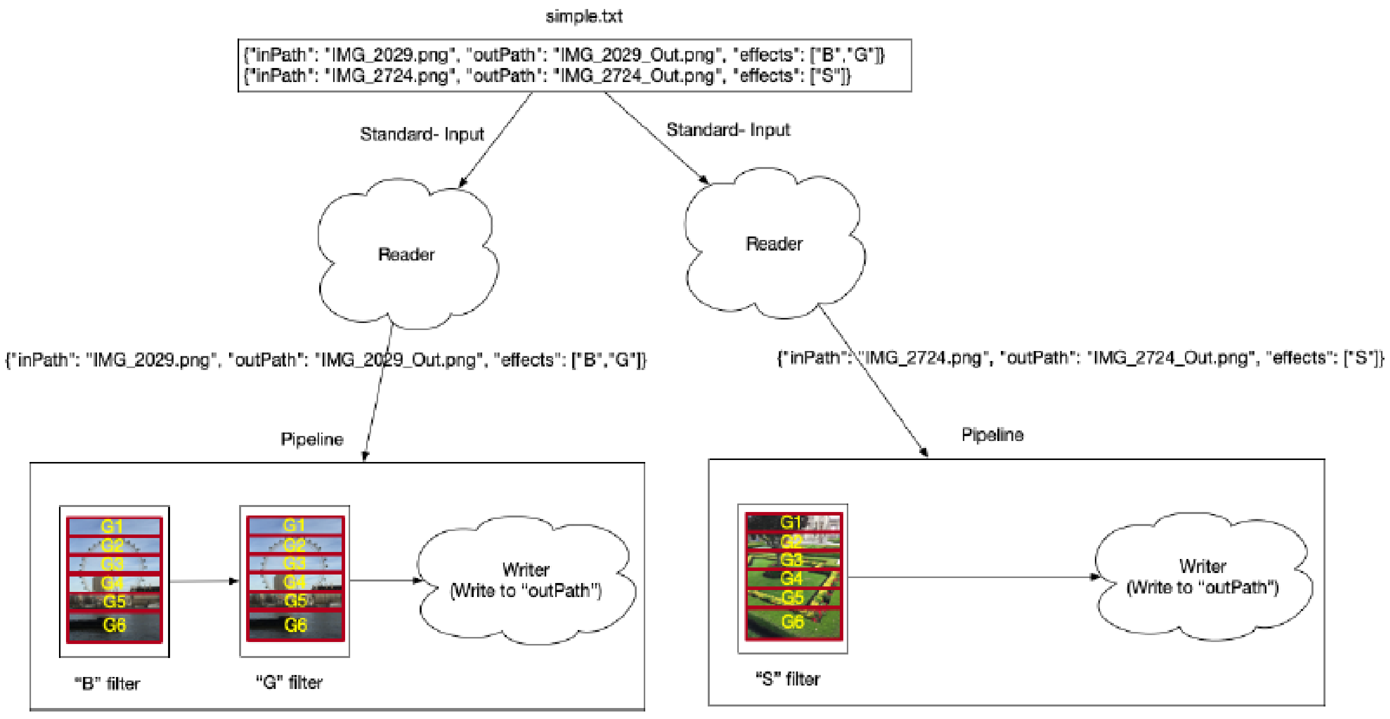
simple.txt
{"inPath": "IMG_2029.png", "outPath": "IMG_2029_Out.png", "effects": ["B","G"]}
{"inPath": "IMG_2724.png", "outPath": "IMG_2724_Out.png", "effects": ["S"]}
Standard- Input
Standard- Input
Reader
Reader
{"inPath": "IMG_2029.png", "outPath": "IMG_2029_Out.png", "effects": ["B","G"]}
{"inPath": "IMG_2724.png", "outPath": "IMG_2724_Out.png", "effects": ["S"]}
Pipeline
Pipeline
G1 G2 G3 G4 G5 G6
"B" filter
"G" filter
"S" filter
Writer
(Write to "outPath")
Writer
(Write to "outPath")

Figure 2: Structure of editor.go

Testing Machine

Execution performance was measured on a Lenovo Thinkpad laptop with the following specifications:

- Processor: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz
- Number of Cores: 4
- Operating System: Windows 10 64-bit
- RAM: 16.0 GB

Performance Analysis

Using the Python notebook I wrote (`convolution_performance_analysis.ipynb`) we'll be testing the performance of our parallel results by running the program on a set of 10 images. The first set of images are stored in `file1` and are roughly 2 MB each. The second set of images are stored in `file2` and are roughly 10 MB each. The third set of images are stored in `file2` and are roughly 15 MB each. Each set of images will be tested with $numThreads = \{0, 1, 2, 4, 6, 8\}$, where 0 threads represents a serial execution. Each timing measured is an average of 5 executions in order to prevent outliers and ensure results are more accurate. From each timing, we can calculate the speedup which is defined as:

$$speedup = \frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

As we can see in figure 3, there is a clear benefit to using the parallel version with more threads. We're able to achieve upto a nearly 3 times speedup when using 8 threads. There seems to be a plateau past 6 threads, however, which could be due to the fact that my testing machine limits the number of threads able to be efficiently used.

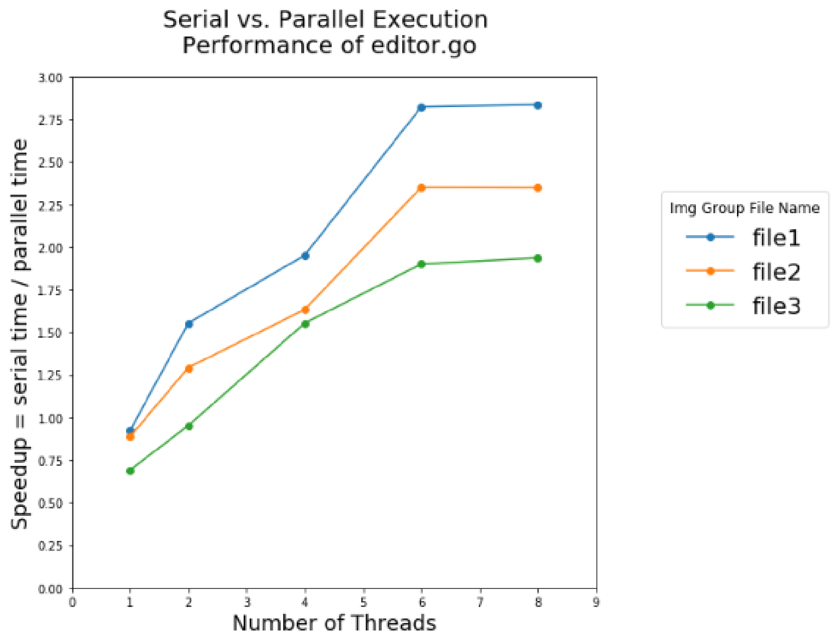

Figure 3: Performance Results

Questions

- What are the hotspots and bottlenecks in your sequential program? Were you able to parallelize the hotspots and/or bottlenecks in your parallel version?

  Hotspots are defined as areas in the program doing the most work, whereas bottlenecks are areas of the program causing the slow down in performance. As the images get bigger and bigger, we can clearly see that performing the matrix computation across each pixel demands the most work thus it's both a hotspot and a bottleneck. By utilizing image decomposition and computing the convolution on subimages, we see a dramatic speed increase. In our parallel version, another bottleneck is the reader's ability to process JSON inputs from Stdin. Since multiple readers must access Stdin in a thread safe manner, we use a coarse-grained lock synchronization where we lock and unlock the Stdin stream each time a reader pulls tasks, which forces it to work sequentially. Across each of our methods, worker pipeline contains the most computationally heavy portion of the overall work and thus is a hotspot for the parallel version of our program.

- Describe the granularity in your implementation. Are you using coarse-grained or fine-grained granularity? Explain.

  Since there are multiple levels of parallelism going on in this project, it's a mixture of both fine-grained and coarse-grained granularity. As we increase the number of threads we split our image into more fine-grained tasks. However, since each worker is a single go routine that works in coarse grain granularity in order to process each effect discretely and sequentially through an effects pipeline.

- Does the image size being processed have any effect on the performance?

  Yes, we can see that the larger the image file size is, the more damp the performance increase is across the board. In the graph, the green line roughly maintains the same shape but is vertically shifted downward by roughly 50%.