Parallel Image Convolution

Ryan Jones

2019 - 09 - 30

For this project we were tasked with performing several versions of an image convolution using different filter kernels. There were two main parts to solving this problem:

1. Partitioning the image data
2. Parallelizing the convolution of the image along these partitions

All code and images used for testing can be found on my Github.

1 Image Details and Cache Setup

For this project I found three `.png` images to demonstrate different caching performances based on the size of the images.

My test machine had an Intel Xeon E3-1240 v5 CPU, whic has 4 cores and 8 hyperthreads. Cache-wise, I was working with an L1, L2, and L3 cache of `256 KiB`, `1 MB`, and `8 MiB`, respectively. These are the parameters I considered for my three test images.

I chose one image that was small enough to fit into a single L1 cache, another that would fit somewhere between the L1 and L3 caches, and a final image that was too large to fit in my L3 cache:

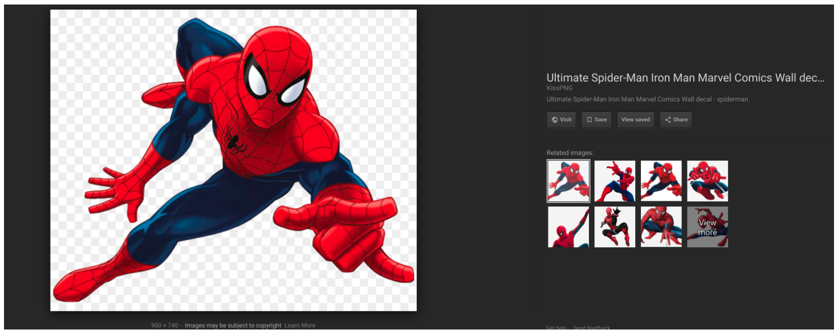
`Image 1:  small.png` (Fits into L1 Cache)
`Resolution:  320 x 180 pixels`
`Image Size:  119.1 kB (121,958.4 Bytes)`

```
Image 2:  medium.png (Fits between L1 and L3 Caches)
Resolution:  2196 x 1332 pixels
Image Size:  4.6 MB (4,823,449.6 Bytes)

Image 3:  large.png (Larger than L3 Cache)
Resolution:  5000 x 5000 pixels
Image Size:  8.6 MB (9,017,753.6 Bytes)
```

Each of these images satisfies my above requirements based on their file size. Finding images of a larger size was more difficult than I had first thought for two reasons. First, full-size `.png` images stored on web servers weren't readily available when following their links on Google images since Google removed the "View Image" functionality. I then tried using the Bing search engine to look for my images, however Bing did not return the kind of results I was looking for.

Second, searching "`subject png`" on the web returned many results that had empty alpha pixels around the subject. This was fit for placing the image into another without having to do a lot of removal of background, but was not ideal for this project because of testing filters like edge detection with backgrounds and other objects. This also meant I would be unable to test code to handle the edges of the image:


Ultimate Spider-Man Iron Man Marvel Comics Wall dec...
KissPNG
Ultimate Spider-Man Iron Man Marvel Comics Wall decal - spiderman

2 Data

Table 1: Data for small image on multiple cores

Image	p	t_p	s	e
small.png	1	0.010	1.0	1.0
	2	0.005	2.0	1.0
	4	0.004	2.5	0.8
	8	0.002	5	0.63
	16	0.002	5	0.31

Graph 1: Small image processing

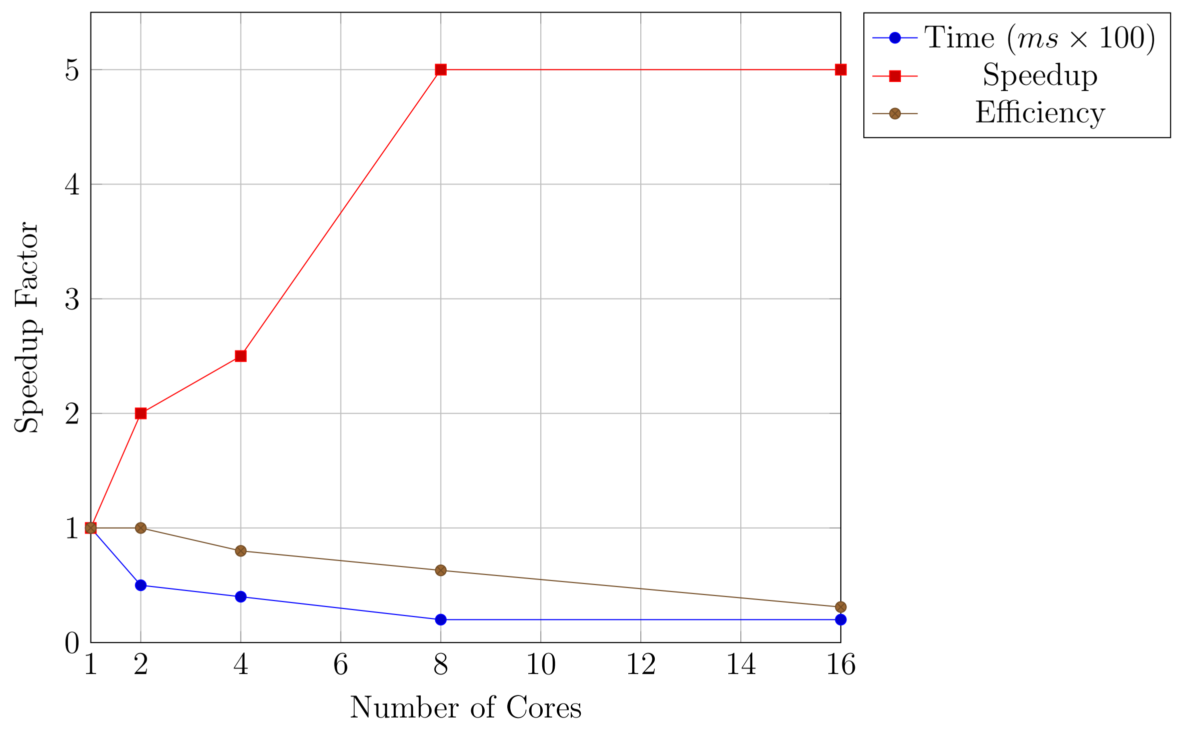


Table 2: Data for medium image on multiple cores

Image	p	t_p	s	e
medium.png	1	0.442	1.0	1.0
	2	0.223	1.98	0.99
	4	0.115	3.84	0.96
	8	0.109	4.01	0.51
	16	0.114	3.87	0.24

Graph 2: Medium image processing

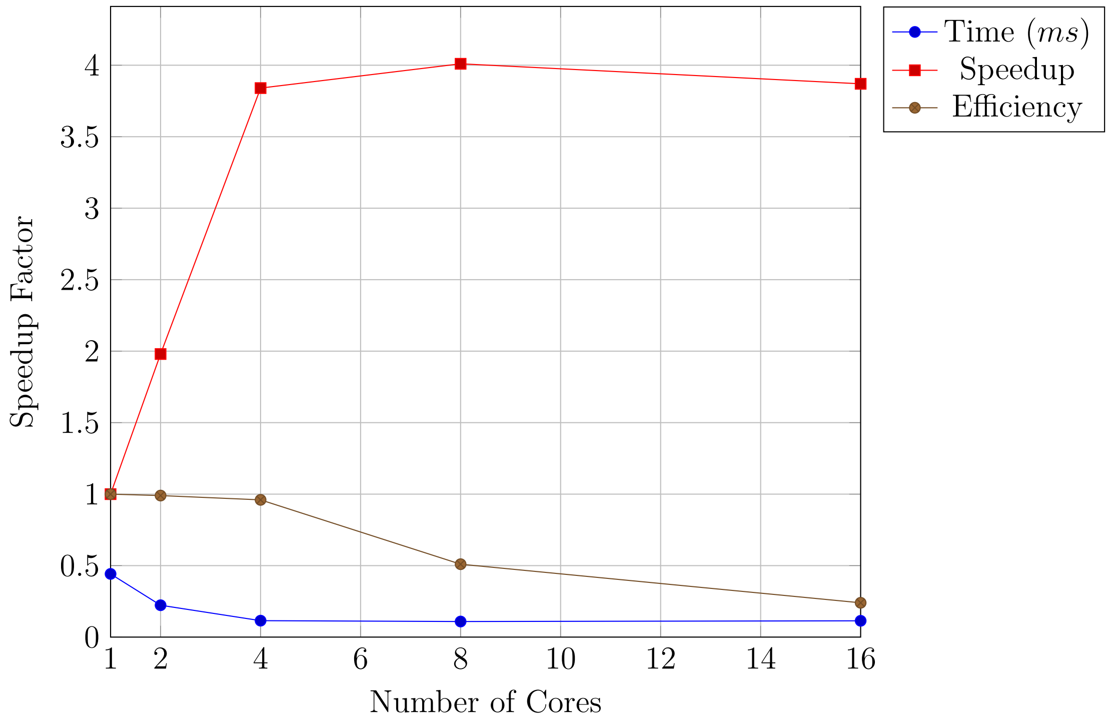
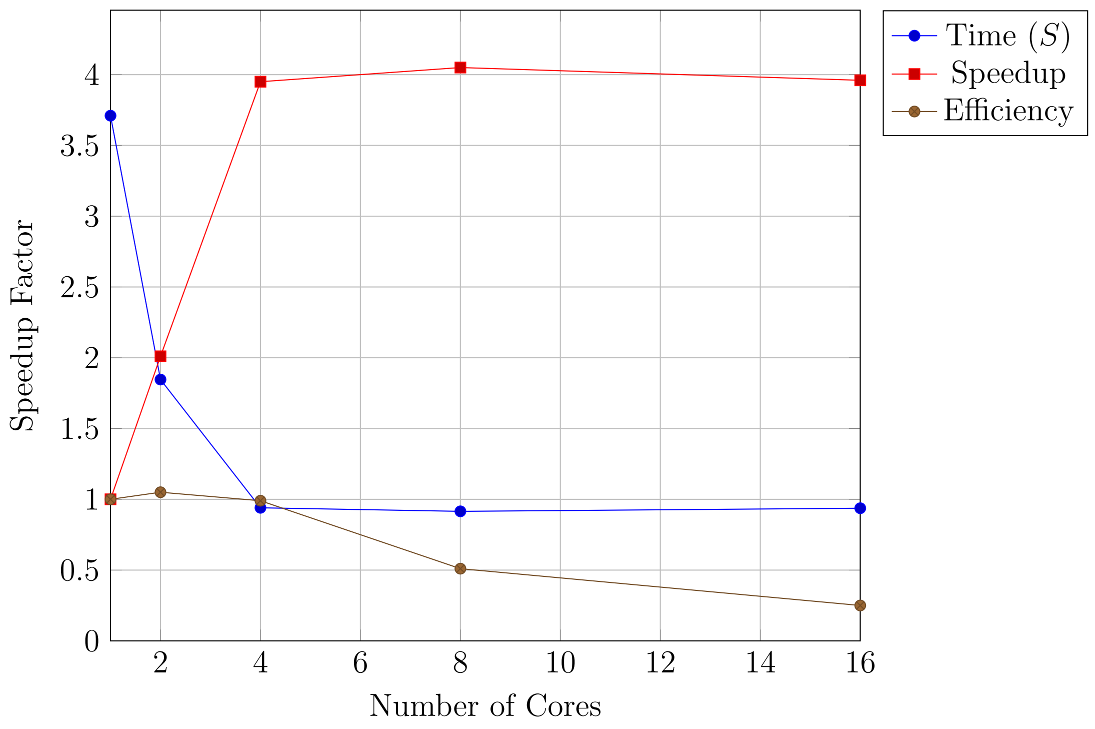


Table 3: Data for large image on multiple cores

Image	p	t_p	s	e
large.png	1	3.710	1.0	1.0
	2	1.846	2.01	1.05
	4	0.940	3.95	0.99
	8	0.915	4.05	0.51
	16	0.937	3.96	0.25

Graph 3: Large image processing



3 Interpretation

My partitioning method depended on assigning specific rows of the image to individual processors. After completing a row, the processor would move down p rows and continue the image filtering, where p is the number of processors running the program. After seeing the results, I think a better method of partitioning would have been a method I discussed with Luke Brom, and which he implemented in his code, as I understand. The alternate method involved viewing the entire image as a one-dimensional array and splitting up that array into p parts, similar to the solution for the Genome project. This would have allowed for better use of caching for each thread on our shared memory system because as the kernel looked around it could reference data that had already been brought into lower levels of cache by other cores.

Across the board, it appeared as though performance and efficiency were constantly improved as the number of cores increased. For each image, speedup increased consistently until passing the 8 core mark, where speedup tended to decrease. I attribute this to the max core count on the lab machines being 8 cores with hyperthreading. After 8, threads were having to multitask which decreased functionality and efficiency.

For each image, time also increased passed the 8-core mark due to the overhead of multitasking on each thread. Peak efficiency for the images appeared to occur between a 4 and 8 thread count. What the data told me about the CPU's caching behavior is that better results came when the CPU's were able to execute instructions on longer rows. This leads me to believe that the alternate method of partitioning described above would be superior to my current algorithm, and why I had some strange results with 4-core runs as described below.

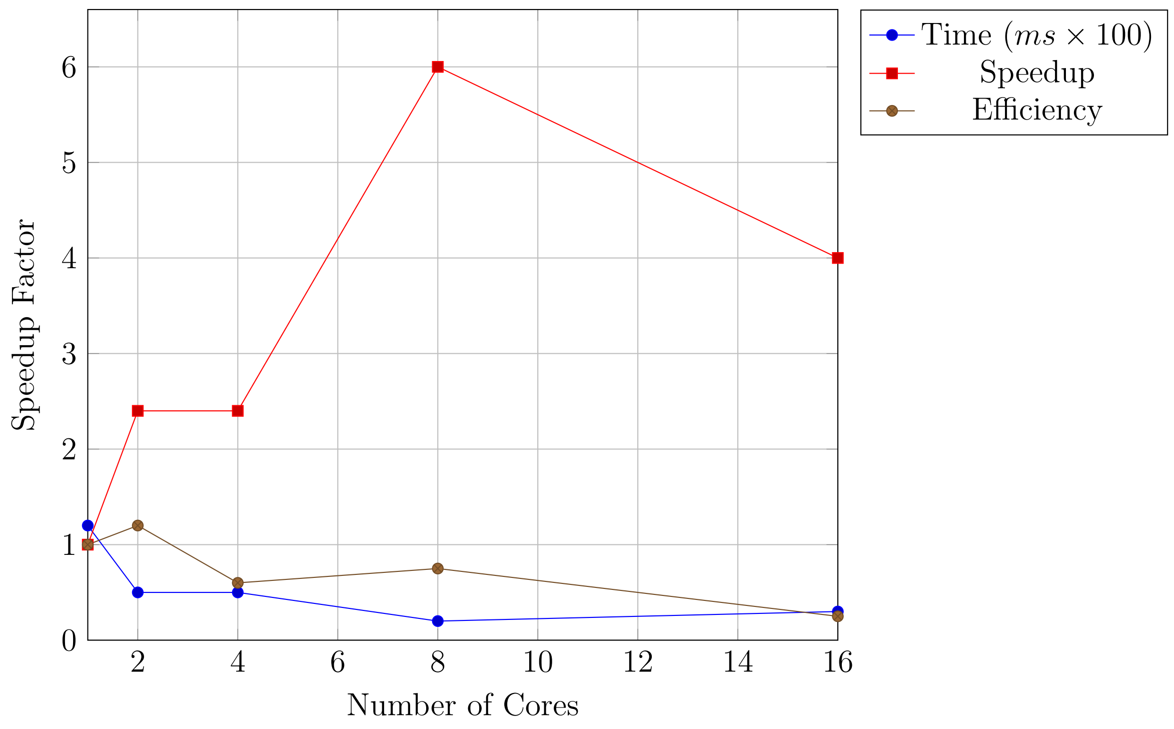
One of the most surprising results was the handling of the small `.png` file, which had some bizzare speedup results. Subsequent runs consistently showed poor efficiency on 4-core tests. This is better exemplified by Graph 4. I am thinking that this may be an architecture problem, as I believe the CPU's cache setup should have had more performance decrease by increasing the thread count from 1 to 2. This is because the entire image is capable of fitting in a single core's L1 cache, so I believed that the first split might have a more significant drop in efficiency than it did. The data appears to show that sharing such a small file among more is not very beneficial, even on a shared memory system.

4 Bonus Functionality

To handle edge detection, I had a nested `if()` statement that checked to see if the program was currently checking an edge. An inner nest of if statements then also checked to see if the current position was a corner, which then substituted a new `row` and `column` value accordingly.

Based on my run on `small.png`, it appears that handling the edges and corners adds some computations to the runtime that were not present in previous versions. Computing all of the edges adds $2(row + column) - 4$ pixels to the program's runtime. Based on my tests, this had the most impact on smaller images, which is consistent with my observations about the cache performance of the image. Because the added rows were still short, the image convolution did not perform well cache-wise

Graph 4: Small image processing (with edge handling)


Time ($ms \times 100$)
Speedup
Efficiency
Speedup Factor
Number of Cores