Image Convolution

Luke Brom

October 1, 2018

1 Strategy

1.1 Partitioning the Data

I decided to partition the data strictly by number of pixels, as opposed to by row, column, or block. For example, if 4 cores are working on a 16x16 image (256 pixels total), each core would be responsible for processing 64 pixels, and since the image is represented in a 1-D array, we can make sure that all of those pixels will be next to each other in memory (i.e. core 0 processes the first 64 pixel entries in the array). This offers two distinct advantages: first, it takes advantage of cashing since the pixels are mostly next to each other, and second, it guarantees near-perfect load balancing for each core.

1.2 Experimental Parameters

I have selected three images as outlined below:

Image Name	Image Size	Image Dimensions
Small	76.7 KB	200x200
Medium	40.5 MB	5760x3240
Large	117.5 MB	10800x7200

The cache configuration of the lab machine:

Cache	Size	Distribution
L1	256 KB	Individual
L2	1024 KB	Individual
L3	8192 KB	Shared

The Core configuration of the lab machine:

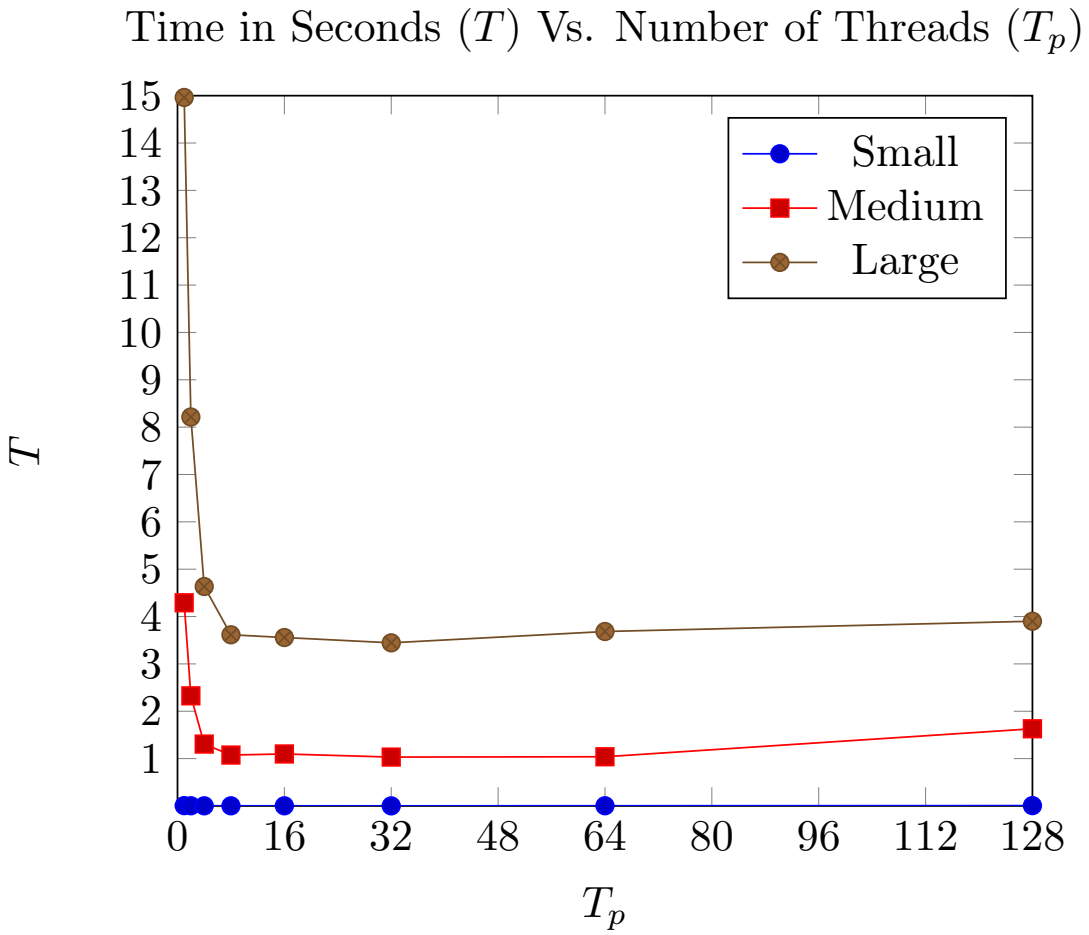
Physical cores	4
Hyper-threads	8

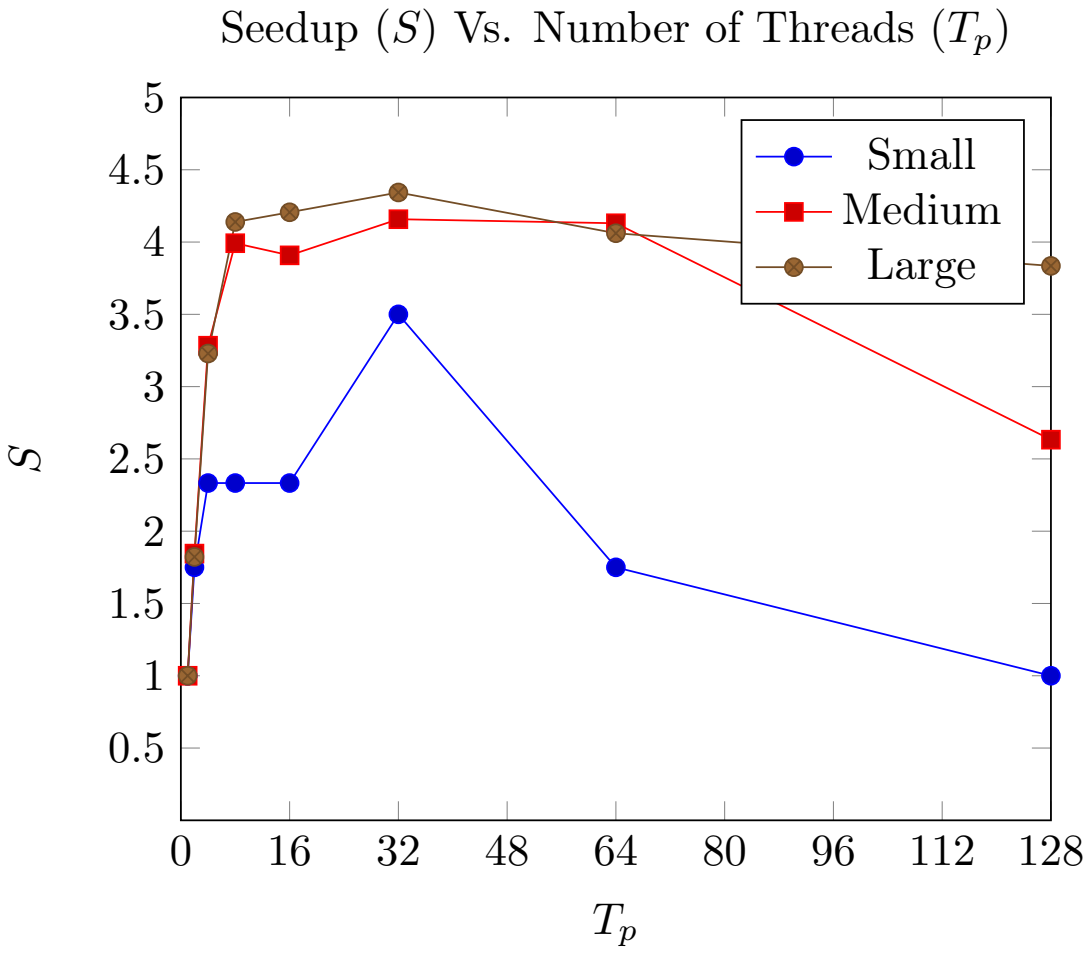
Based on the above information, the images that I have selected do satisfy the requirements of this assignment. The Small image fits in its entirety in the L1 cache of a single core, while the Large image will not.

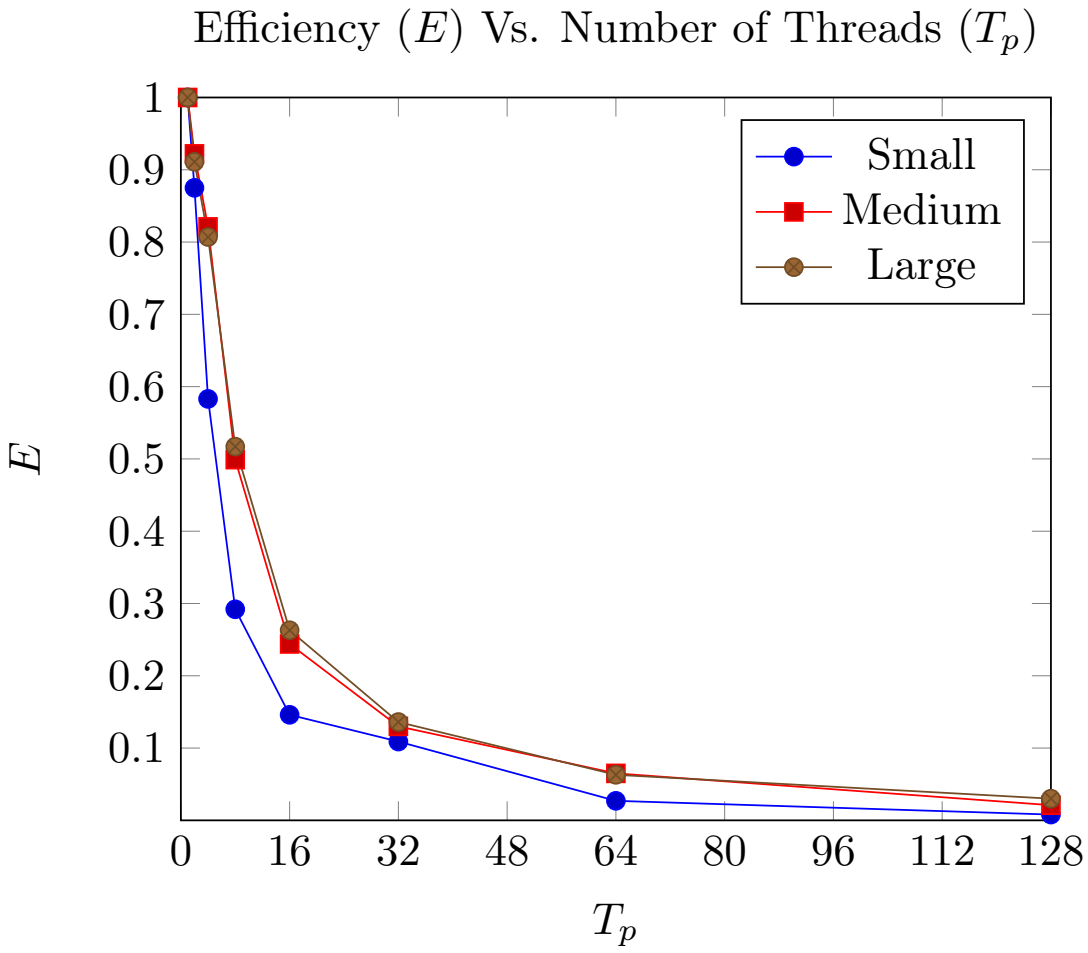
2 Data

Image	p	$T_p(s)$	s	e
Small.png	1	0.007	1.000	1.000
	2	0.004	1.750	0.875
	4	0.003	2.333	0.583
	8	0.003	2.333	0.292
	16	0.003	2.333	0.146
	32	0.002	3.500	0.109
	64	0.004	1.750	0.027
	128	0.007	1.000	0.008
Medium.png	1	4.291	1.000	1.000
	2	2.326	1.845	0.922
	4	1.307	3.283	0.821
	8	1.075	3.992	0.499
	16	1.098	3.908	0.244
	32	1.032	4.158	0.130
	64	1.039	4.130	0.065
	128	1.630	2.633	0.021
Large.png	1	14.962	1.000	1.000
	2	8.214	1.822	0.911
	4	4.634	3.229	0.807
	8	3.615	4.139	0.517
	16	3.557	4.206	0.263
	32	3.445	4.343	0.136
	64	3.684	4.061	0.063
	128	3.901	3.835	0.030

3 Graphs


Time in Seconds (T) Vs. Number of Threads (T_p)
T
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
16
32
48
64
80
96
112
128
T_p
Small
Medium
Large


Seedup (S) Vs. Number of Threads (T_p)
Small
Medium
Large
S
T_p
0
16
32
48
64
80
96
112
128
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5



4 Analysis

The results were largely as expected. With the small image, there was the smallest speedup and efficiency, and with the large image, there was (in general) the largest speedup and efficiency. For each image, the speedup starts to drop off after around 32 cores, and the efficiency is consistently going down.

There were a few surprising results that were obtained. The first, as mentioned above, is that the speedup doesn't start to drop off until after 32 cores, which is surprising because the lab machine only has 4 physical cores and 8 hyper-threads. Given the specs of the lab machine, I would expect the speedup to start dropping off sooner, probably around 16 cores, however that was clearly not the case. Since this was a consistent occurrence, my only conclusion would be that the architecture must be made to handle multiple threads very well, even when they are running on the same core.

The second surprising result consists of a few different anomalies in the data. The speedup for the medium image with 16 cores decreases from 8 cores instead of increasing, and the speedup for the small image spikes drastically at 32 cores. While these are anomalies that go against the expected trend, I think they can be explained away by the fact that I didn't average the data. If I had recorded 100 result for each data point and averaged them together I suspect that these anomalies would disappear.

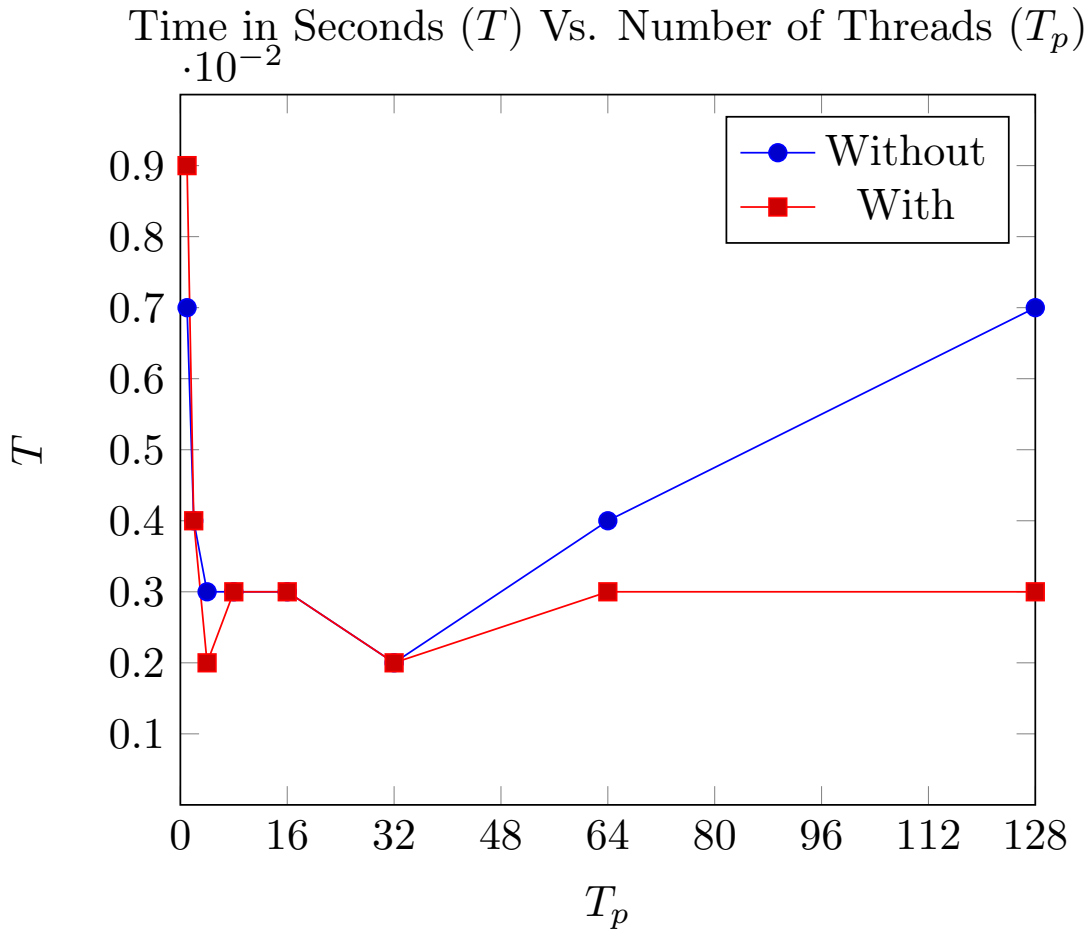
My partitioning scheme is described at the start of this document. Given the scheme that I went with, I am very pleased with my results and I would choose the same scheme again.

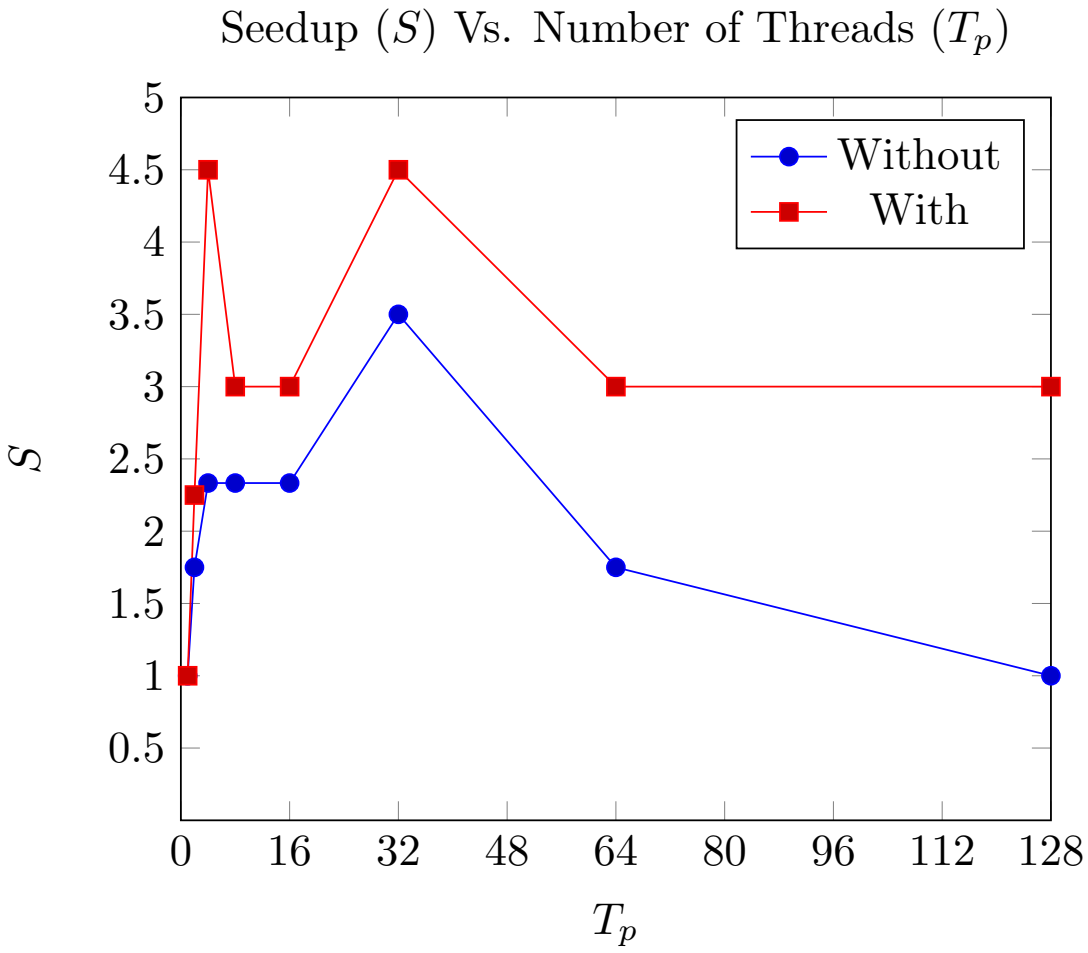
The best evidence of caching in my data lies in the speedup of the medium and large images at 64 cores. At this point, the speedup of the medium image passes the speedup of the large image by a small amount, and one explanation of this could be cashing. The medium image, split into 64 partitions, actually does fit into the L2 cache (coming in at 648 KB), whereas the large image does not (1880 KB). Having an entire partition fit into the cache seems to bring greater speedup

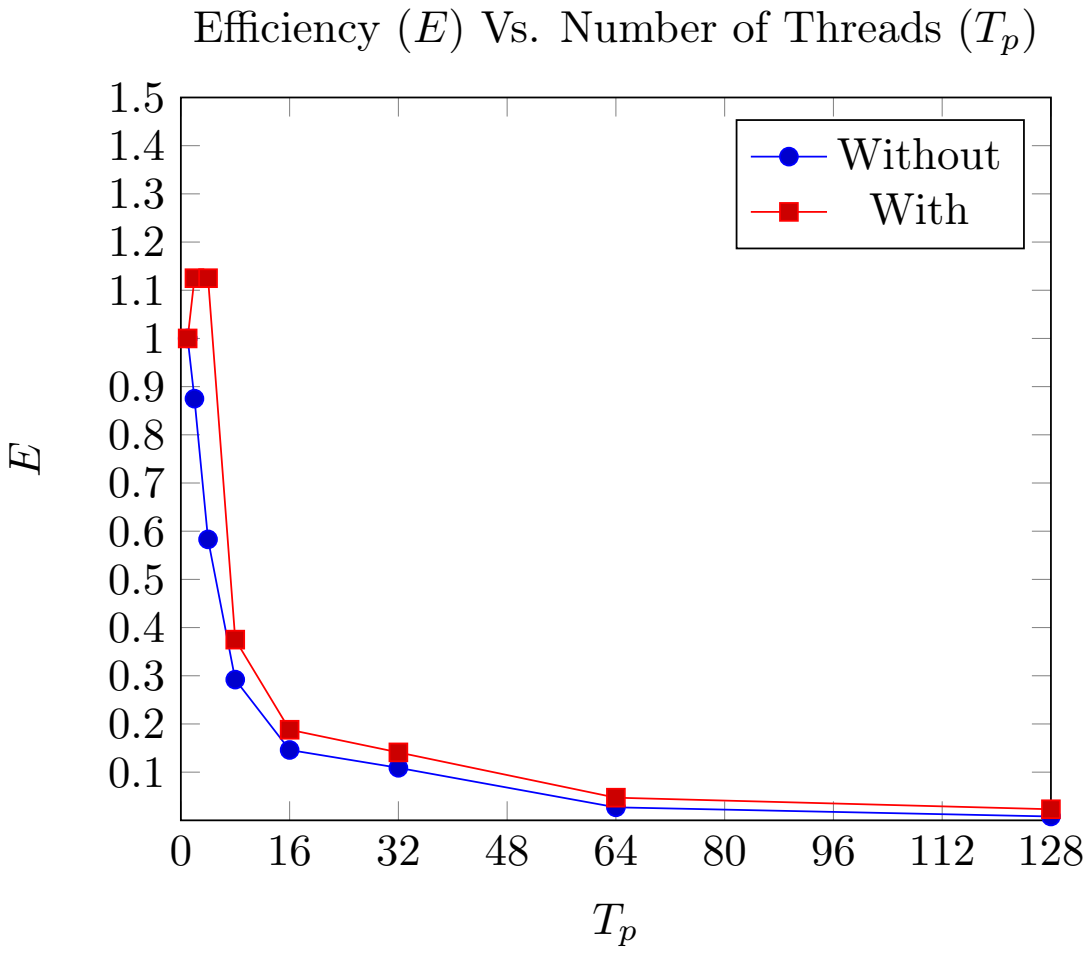
than if it doesn't.

5 Adding Edge Handling

Image	p	$T_p(s)$	s	e
Without	1	0.007	1.000	1.000
	2	0.004	1.750	0.875
	4	0.003	2.333	0.583
	8	0.003	2.333	0.292
	16	0.003	2.333	0.146
	32	0.002	3.500	0.109
	64	0.004	1.750	0.027
	128	0.007	1.000	0.008
With	1	0.009	1.000	1.000
	2	0.004	2.250	1.125
	4	0.002	4.500	1.125
	8	0.003	3.000	0.375
	16	0.003	3.000	0.188
	32	0.002	4.500	0.141
	64	0.003	3.000	0.047
	128	0.003	3.000	0.023




Seedup (S) Vs. Number of Threads (T_p)
Without
With
S
T_p
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
16
32
48
64
80
96
112
128



5.1 Analysis

Oddly enough, the performance of my edge-handling code actually seems to be a little better than my non-edge-handling code. I have a few guesses as to why. First, I did my testing on a lab machine on two different days. Given that I am not the only person who uses the lab machines, I am willing to bet that something changed about the state of the machine between the two tests. Second, because the data is not averaged from multiple runs, these results could be anomalies. Third, in my original run, I had my code checking for the edge and just continuing the loop if it was on an edge, this action might of been just as costly, if not more costly, as my edge-handling code. Any of the above, or a combination of each, is a possible explanation for my results.