Project 3: GPU CUDA Programing

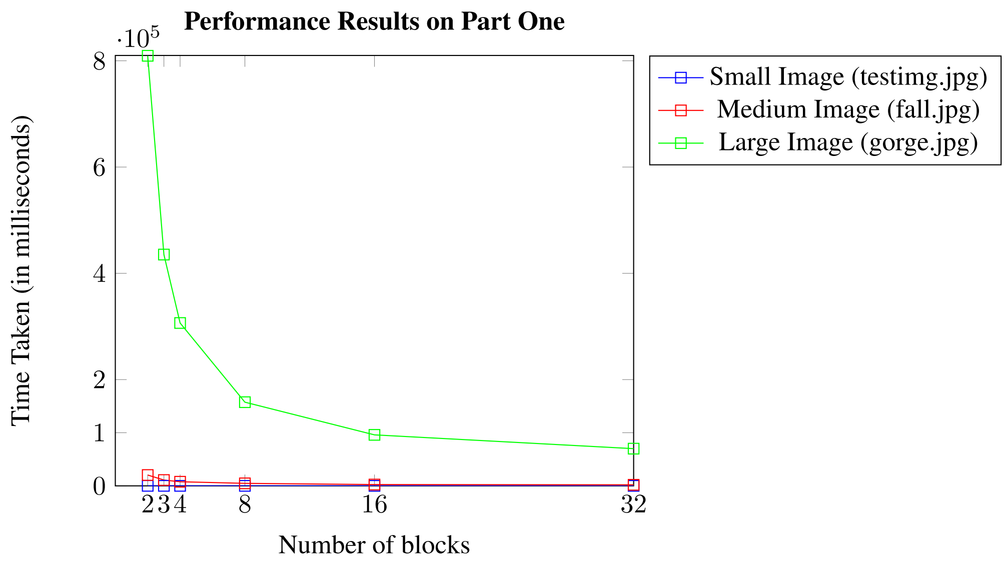
Daniel Woldegiorgis

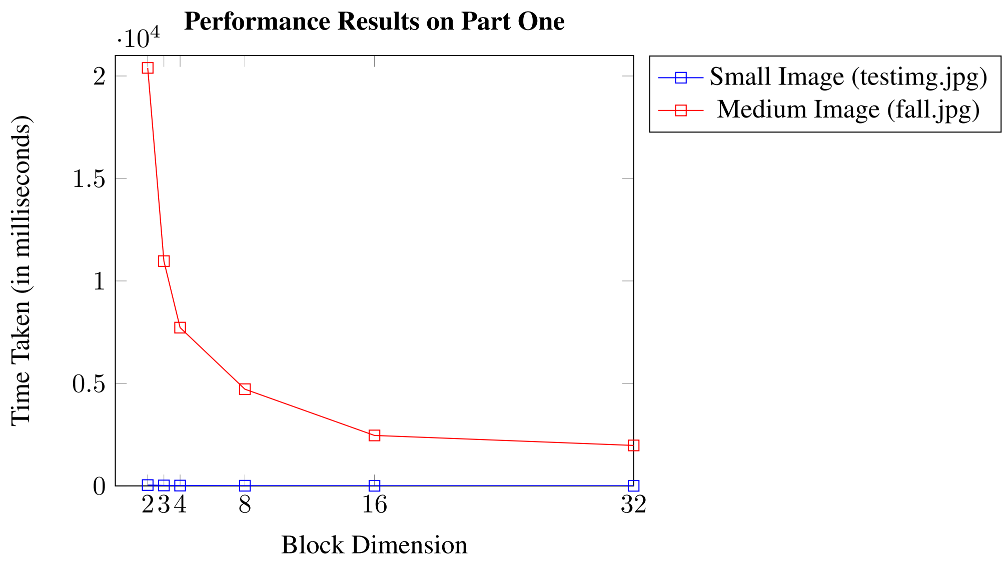
November 14, 2019

1 Simple Blurring

In this section, I implemented the basic, naive blurring algorithm as a CUDA kernel. I used 2 dimensional blocks and grids, and the blocks are equal in their x and y dimension (squares) for the testing results presented in this write up. The weights and division factor are not pre-cacluated for this particular section. One other thing to note here is that the timing information does not include memory transfer times to and from the kernel.
The results obtained using different block sizes 2,3,4,8, 16, and 32 (which is the maximum possible square block size because 32*32= 1024, which is the CUDA limit for number of threads), is tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified.

Part One Time Test Results (in milliseconds)			
Num. Blocks	Small Image	Medium Image	Large Image
2	39.783520	20395.83398	809403.750000
3	21.183456	10966.764648	435305.812500
4	14.831584	7722.006836	306510.500000
8	7.773184	3968.266357	157496.109375
16	4.808672	2461.310547	95966.039062
32	3.559456	1973.896973	70018.750000





We can see that there is generally an increase in performance with increasing block size. However, the rate of improvement continues to decrease as we increase the block size beyond 8.

We know that a block assigned to an SM is further divided into 32 thread units called warps. Each block will have (#threads in block/32) warps and we want to have as many warps as we can in order to avoid stalling. That is, there will likely be at least some warps that are ready to run at any given time when we have many such candidate warps. Therefore this increase in performance with increasing block size is expected, however given the total number of threads that the SM can handle is fixed, we cannot expect this growth to continue when it begins to reduce the number of blocks an SM can support.

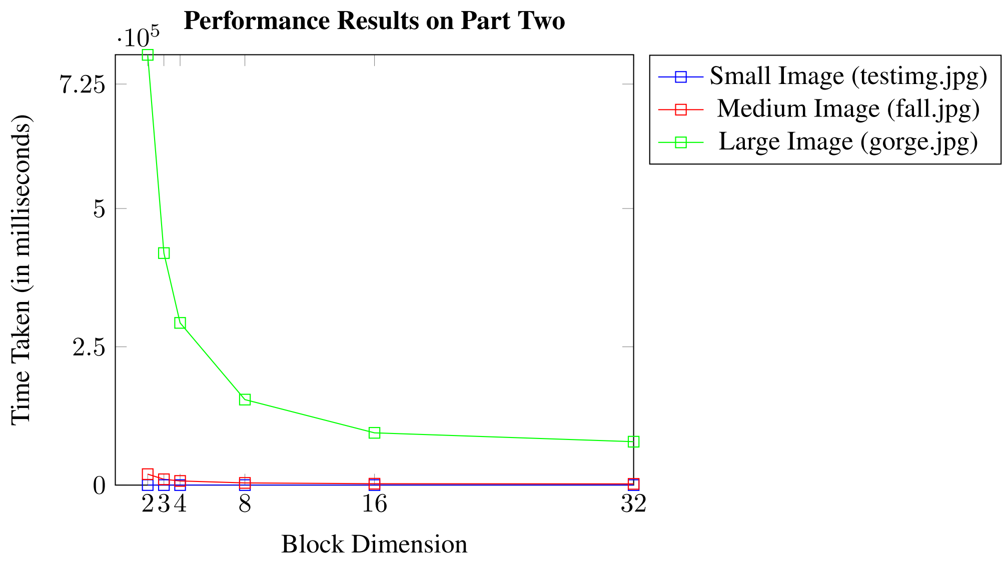
We generally want the block to be a multiple of 32(warp size), so that we will be making the most efficient use of the given threads accessing contiguous memory locations. We can see that this also holds true according to the graphs above.

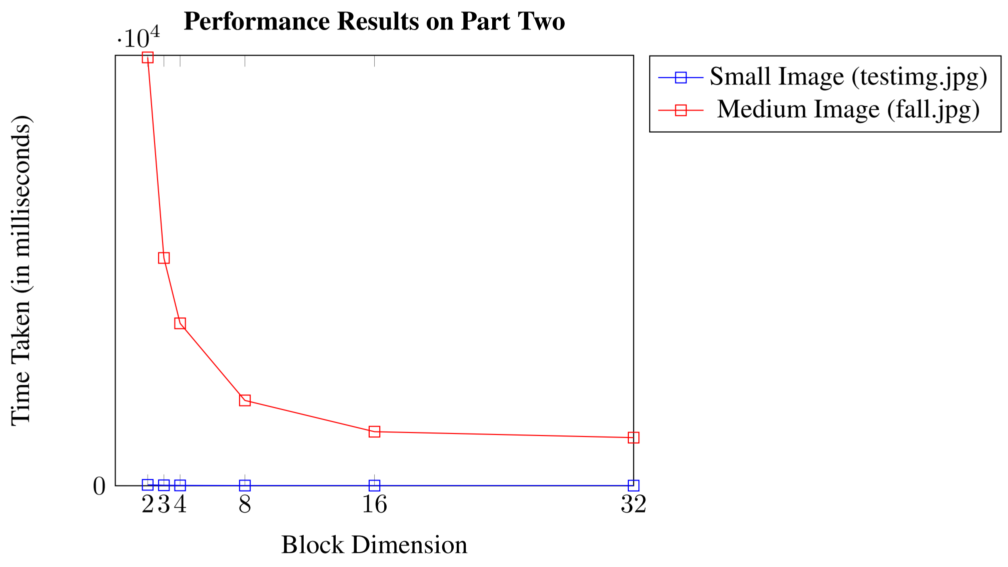
The results were obtained on a GeForce GT 730 GPU from NVIDIA.

2 Decreasing the impact of branches

In order to test the effect of branching, I removed all if statements from the doubly nested for loops that are used to blur the image. I ended up dividing the image into 9 regions: top left, top, top right, left, center, right, bottom left, bottom, bottom right in row major order. Threads therefore determined which region they would be working on and then proceeded to blur the image without having to check for the validity of the image pixels they were working with. The results obtained using different block sizes 2,3,4,8, 16, and 32, are tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified.

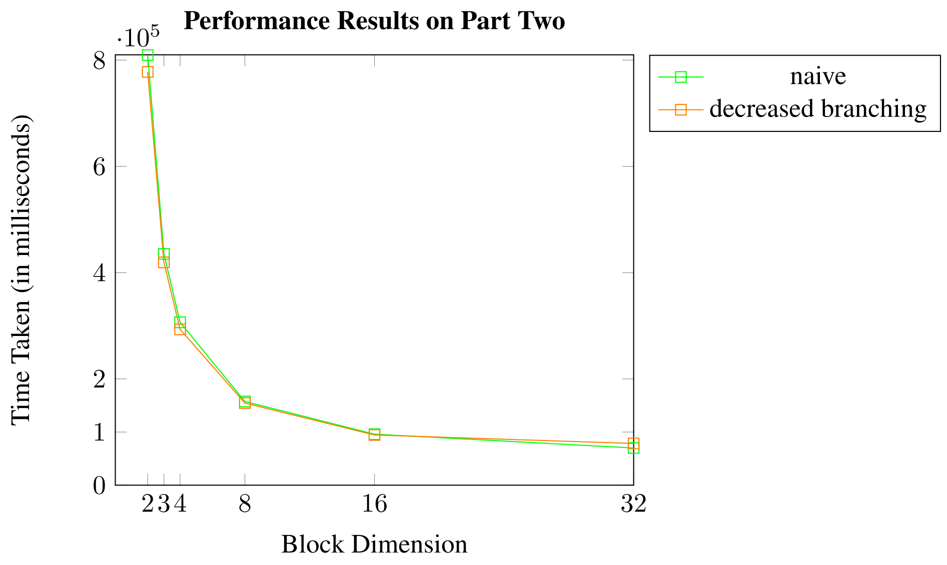
Part Two Time Test Results (in milliseconds)			
Num. Blocks	Small Image	Medium Image	Large Image
2	40.805473	19804.755859	777751.750000
3	21.144608	10531.153320	419391.843750
4	14.930080	7508.686523	293234.218750
8	8.153088	3944.388672	154555.718750
16	5.323744	2498.830078	94466.351562
32	4.791232	2223.920898	78479.390625





We can again see that there is generally an increase in performance with increasing block size. However, the rate of improvement continues to decrease as we continue to increase the number of blocks that are used.

To ease in comparing the performance of the naive blurring algorithm when compared to the blurring algorithm with the branching conscious code above, I have recreated their performance results on a large image (gorge.jpg) below.



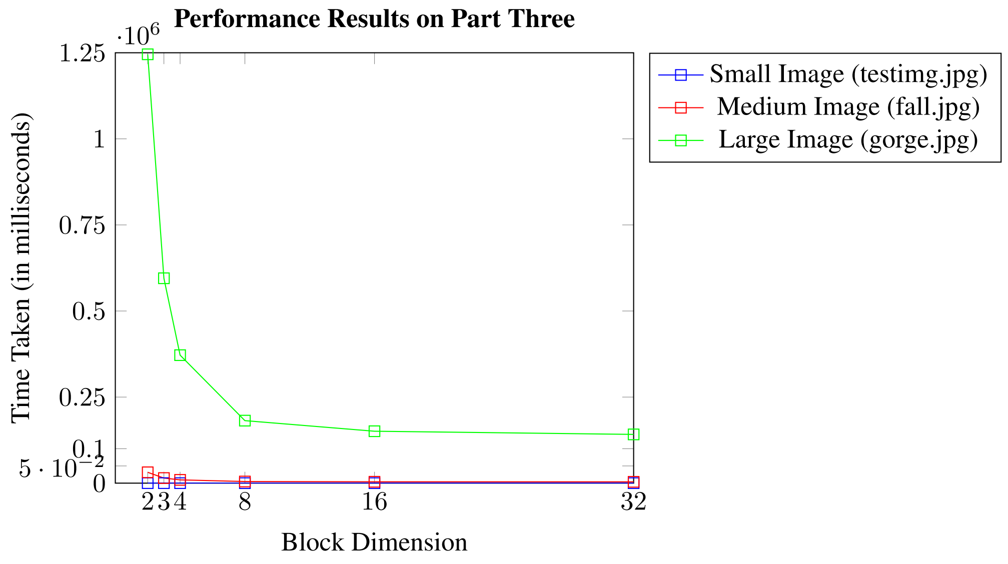
We can see that both the naive implementation and the one with reduced branching are comparable in terms of their performance. However, the latter slightly outperforms the naive version for smaller block size configurations.

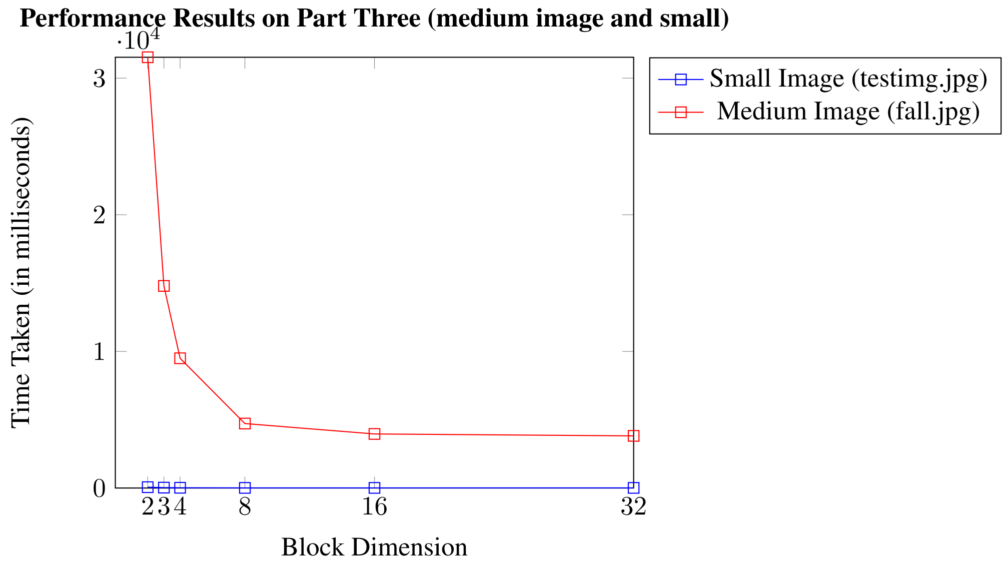
3 Precomputing weights and final division factor on host

In this section, the weights and final division factor were precalculated on the host and passed into the CUDA kernel. The division factor was then adjusted to include only the sum across valid pixels. The precomputation was done using the implementation in part 2, where the impact of branches was reduced by removing the if statements from the doubly nested blurring for loops.

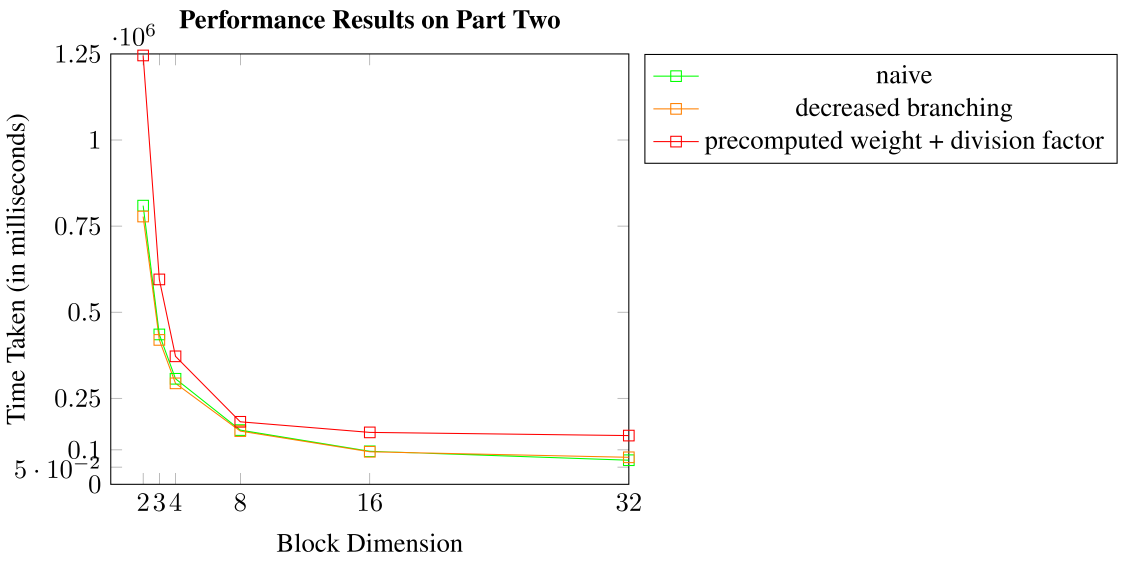
The results obtained using different block sizes 2,3,4,8, 16, and 32, are tabulated below and then presented visually on a graph. This is divided into three cases based on the size of the input image that was specified.

Part Three Time Test Results (in milliseconds)			
Num. Blocks	Small Image	Medium Image	Large Image
1	228.791199	118610.796875	4749894.000000
2	62.540798	31527.412109	1245528.625000
3	30.090273	14793.479492	595091.062500
4	19.546080	9495.391602	371741.187500
8	10.214304	4718.907715	181571.718750
16	9.175072	3957.512451	150808.609375
32	10.184672	3816.886719	141560.046875





Therefore as we can see as the block size increases there is an increase in performance however there is a slowdown in the rate of improvement with regard to increasing block sizes. As we can see from the table the same holds for the small image.
(it roughly halves the run-time of the algorithm when we increase the block size by powers of two.)
To ease in comparing the performance of the three different blurring implementations of the blurring algorithm, I have recreated their performance results on a large image (gorge.jpg) below.



We can see that the algorithm where we precompute the weight and division factor on the host is outperformed by the other two implementations. It would therefore be better to perform the computation of the weights and division factor in the device. This could perhaps be explained by the fact that the division factor has to be adjusted a significant amount in the cases where the pixels are within radius pixels of the border of the image.

3.1 Steps taken to Parallelize.

I first started out with getting a serial version of the blurring algorithm to cross check any results that would be obtained after beginning parallelization. After this, I started working on the naive blurring algorithm using a CUDA kernel. Instead of manually flattening the two dimensional array of values, I basically used the fact that C stores it's arrays using contiguous memory to simply come up with an indexing scheme to access specific entries within the kernel. (2 dimensional indexing is currently not supported in CUDA Kernels).
I then experimented with different 2 dimensional block sizes and observed a general trend of faster performance with increased number of blocks and so made the default launch configuration to be 32, unless otherwise changed.

For part 2, I basically removed all the if statements from the doubly nested blurring for loops and had each of the nine resulting sections of the image perform the blurring over their respective valid pixels. In order to reduce having to specify all of the 9 cases in the same level, I basically created a set of filters that made it so that a given thread will find the region it is supposed to work on after at most 3 decisions in the worst case. These decisions can be summarized as follows: Is (row,col) within radius of the boundary of the image? => Is it within radius of the left/right/top/bottom?

=> Is it within left/right/top/bottom of that specific section?. By the end of these three decisions, the decision tree will reach a leaf node and the thread will have found the data it is supposed to be processing in the image.

Later, I pre-calculated both the weights and division factor on the host and passed them into the kernel. In addition, since obtaining the numerator for the red, green and blue pixels are independent operations that do not interfere with each other, I chose to have execute them during a single iteration instead of doing multiple calculations. All three files are available in the cs338Blur.cu file and can be specified by using the PART_ONE, PART_TWO or PART_THREE directives respectively.