20sdc2

Part One: Naïve Image Blurring

I've created and tested a parallelized method that utilizes a GPU device to blur an image. This is done using NVIDIA's CUDA for C. The concept is that we will create one thread per pixel and use the blurring technique from Project Two to calculate the new value of that pixel based on its neighbors. In this portion of the assignment, I did not make any branching-wise or pre-calculation optimizations. The radial parameter was set to 5%[1] . The one parameter tuned was the dimensionality of blocks within the kernel's grid in the execution. Dimensions tested were all squares, with lengths: 1, 2, 4, 8, 16, and 32. I witnessed large speed-ups as I increased the block size, which was to be expected. A graph of the results of these tests run on the redtree.jpg image is included here.

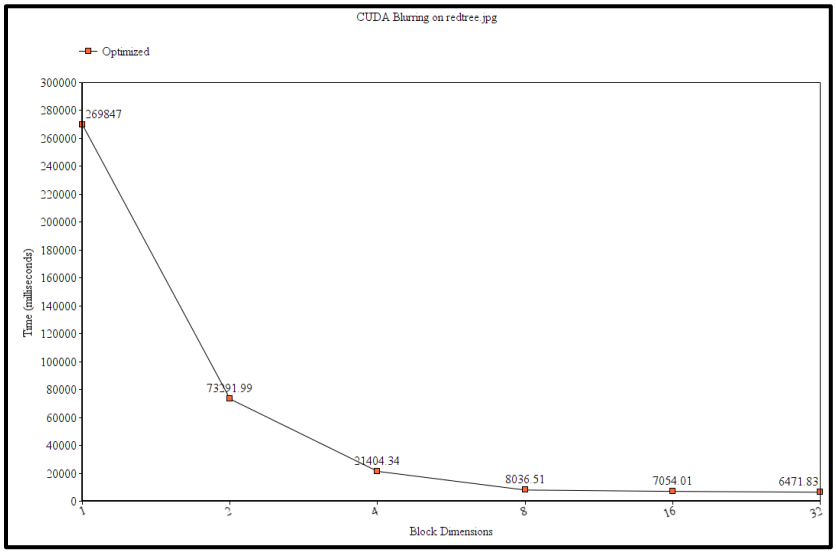

Figure 1 - Naive blurring

The speedup here is expected because as we increase the number of threads per block, we're able to increase the parallelism we obtain, yet this has limits and we see the speedups start to slow drastically after a dimensionality of 8 by 8. CUDA parallelizes work on the GPU by executing the same command on a warp of 32 threads within a block at once. If a block doesn't have enough threads to utilize this parallelism, however, then we don't get the benefits associated with this parallelism, as there aren't enough threads to chunk into warps for simultaneous execution. As noted, the parallelism slows down after a certain point, and that is likely because in this application, once the blocks get large enough to by split into warps, making the blocks larger doesn't create any additional benefits. Blocks can share memory, but we don't need that sort of shared memory in this application, so we're solely utilizing the warp-based instruction execution, which tops-out at 32 threads.

Part Two: Branching Optimization

One major pain point in Part One was that we did not have a mechanism for branching based on whether a pixel was a border pixel or an inner pixel. A border pixel is one that exists within a radial parameter's distance from one of the edges, such that at least some of its neighbors are invalid pixels, due to their being beyond the edges of the image. An inner pixel is a pixel which has no neighbors beyond the picture's edges. This caused an issue, however, because for every pixel, we had to conduct a logical check for every one of its neighbors to determine whether that neighbor was valid. This is especially problematic using CUDA, however, because CUDA relies on being able to send the same

[1] meaning we would consider any valid pixel within 5% of the selected pixel to be its neighbor

instruction to an entire warp at the same time. If each thread within a warp must make its own branching decisions based on which neighbors are valid, then there is no way to ensure that the device is able to operate in the expected SIMD (single instruction, multiple data) way.

In this section, I was able to fix this issue by having each thread check whether any of the threads within its block were border pixels. If a thread is in a block with border pixels, then the thread will operate as it would in Part One (checking the legality of each neighbor). However, if a thread is in a block with only inner pixels, it never needs to conduct logical checks when using its neighbors' values for blurring.

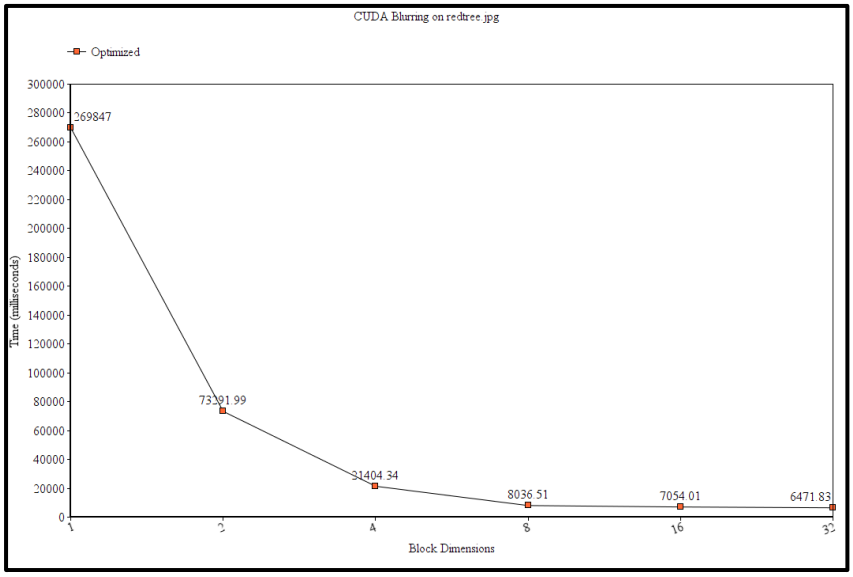

Figure 2 - Branching blurring

This is an efficient optimization because it guarantees that every thread within each block will follow the same initial branch[2]. In the case of inner pixel blocks, the optimization allows these blocks to continue executing in the desirable wrap-wise SIMD way, as there is no more branching when collecting neighbors' values. A graph of the results, run under the same conditions as in Part One, is included here. There is also a graph in Part Three which directly compares the output results from Part One, Part Two, and Part Three. We do note that we witnessed significant speedups compared to our tests from Part One, as expected. A breakdown of this data, as well as the data from Part One and Part Three, can be found in the Appendix.

Part Three: Pre-calculated Value Optimization

Part Two served as a major improvement compared to Part One. There was one component missing, however, which has been shown to improve execution times under standard thread-wise blurring programs[3]. That is: for inner blocks[4], their blurring divisor is fixed based on the radial parameter and doesn't need to be re-computed by each thread. Instead, we can compute a weight matrix and blurring divisor upfront, in the main method, and write the matrix to the device memory for thread access.

In theory, this allows us to reduce the number of necessary calculations in each thread, which should improve the performance relative to Part Two. A graph showing the results of this optimization run under the same conditions as in Part One and Part Two is included on the next page. It is worth noting, however, that this optimization *slowed* the execution time of the program, relative to Part Two, instead

[2] While different blocks may branch differently (based on whether they contain any border pixels), threads within a given block all take the same branch

[3] See Project Two

[4] Those blocks which only contain threads corresponding to inner pixels

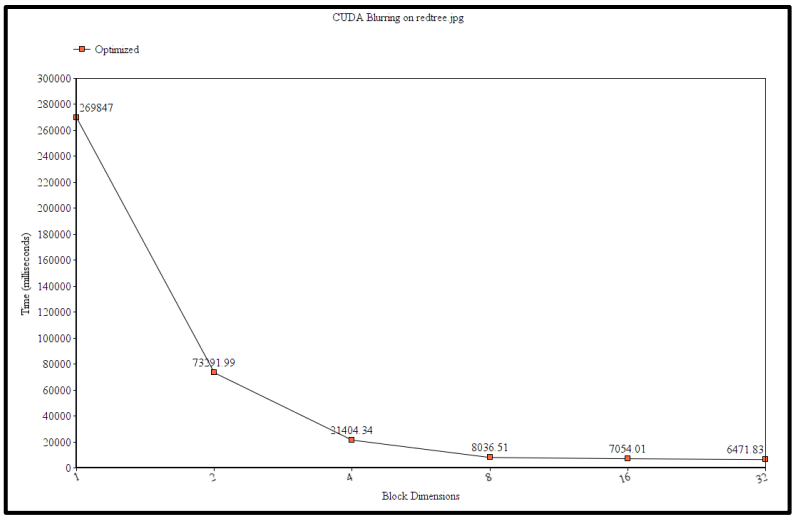

Figure 3 - Optimized blurring

of speeding it up. The execution was still improved relative to the performance of Part One. Another graph is included here which compares the execution results of Part One, Part Two, and Part Three.

It is perhaps not surprising, however, that the results from Part Three are worse than Part Two. This may be because our program is memory bound. That is to say, the bottleneck of our execution may be based on the ability to access memory, rather than by the amount of computation performed. With that in mind, the results would make sense. Our program is bound by its threads' ability to access memory, which stores the input and output pixel arrays. When a thread is executing, it needs to look up from memory the value of each of its neighbors, and then it needs to update its own value in the output array. This leads to an extremely large number of memory accesses. The optimization in Part Three indeed reduced the amount of computation necessary, but it also increases the number of memory accesses per thread. This is clear because for every neighbor of an inner pixel, the thread must lookup both the neighbor's values and now the neighbor's weight from the weight matrix. While we only need to look up the weight once per pixel (which we can then use three times: for the R, G, and B values), this still significantly increases the memory bandwidth of each thread in an inner block.

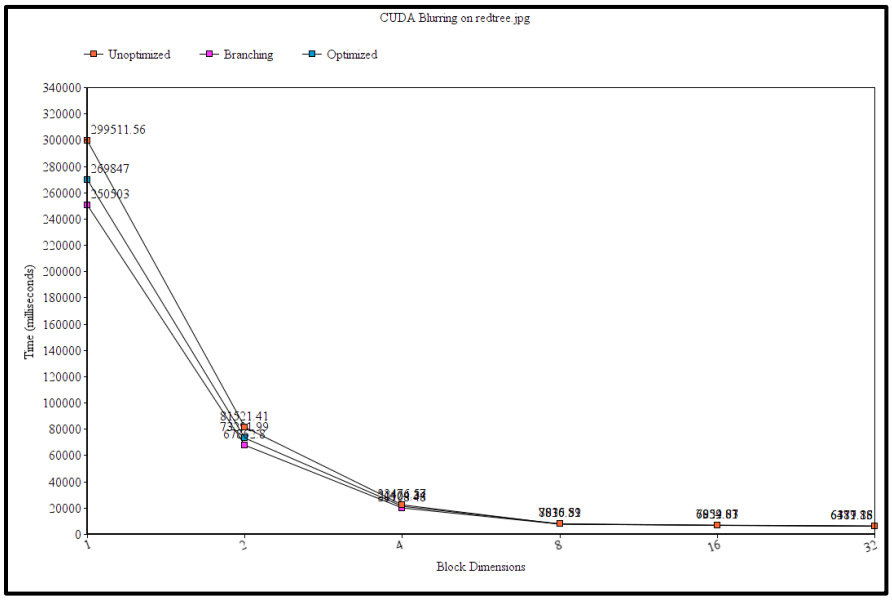

Figure 4 - Comparing Part One, Part Two, and Part Three

As noted, it is observed that Part One performed the worst due to its lack of branch control, Part Two performed the best with its optimized branching management, and Part Three performed in the middle, due to the increased memory bandwidth on an already memory-bound application. It is also noted that all three parts witness similar, converging benefits from increasing the block dimension, with similar topping-out patterns once a block of 8 threads by 8 threads is used. Graphs comparing the results of all three parts on two separate images (twotone.jpg and tree.jpg), as well as a table with all the results are including in the Appendix below.

Appendix

Time in ms*	twotone.jpg			tree.jpg			redtree.jpg		
Block Dimensions:	Unoptimized	Branching	Optimized	Unoptimized	Branching	Optimized	Unoptimized	Branching	Optimized
1	974.99	815.13	877.23	29923.59	24914.72	27418.06	299511.56	250503	269847
2	267.09	223.88	241.39	8146.17	6787.94	7333.04	81521.41	67862.8	73291.99
4	74.4	66.96	71.05	2237.88	2012.32	2138.95	22476.57	20108.48	21404.34
8	26.39	26.4	27.12	782.88	784.9	805.66	7810.89	7826.52	8036.51
16	23.52	23.25	24.01	698.85	686.49	709.6	6922.83	6829.87	7054.01
32	23.01	23.13	23.83	656.54	638.25	667.95	6319.18	6187.26	6471.83

Figure 5 - Table of all results from all three parts on three separate images

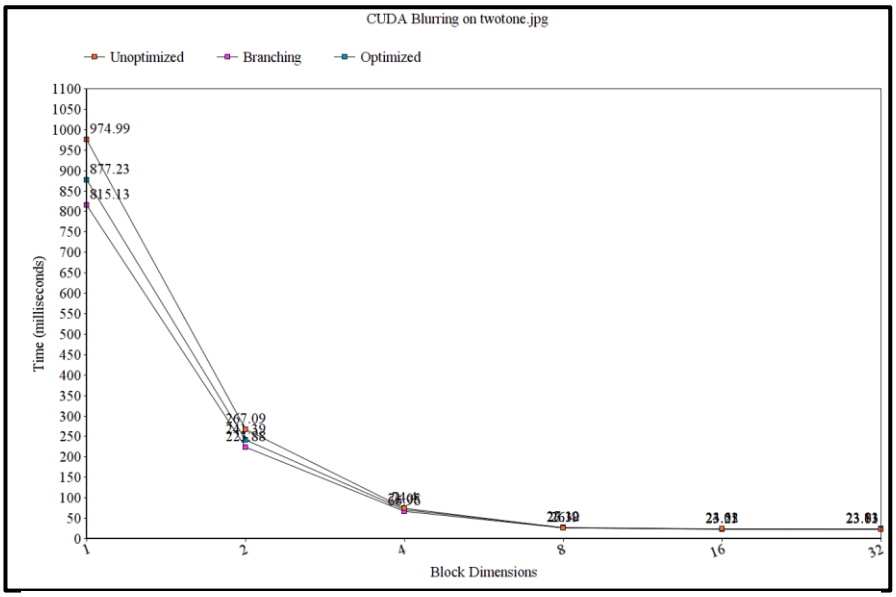

Figure 7 - Comparing Part One, Part Two, and Part Three on twotone.jpg

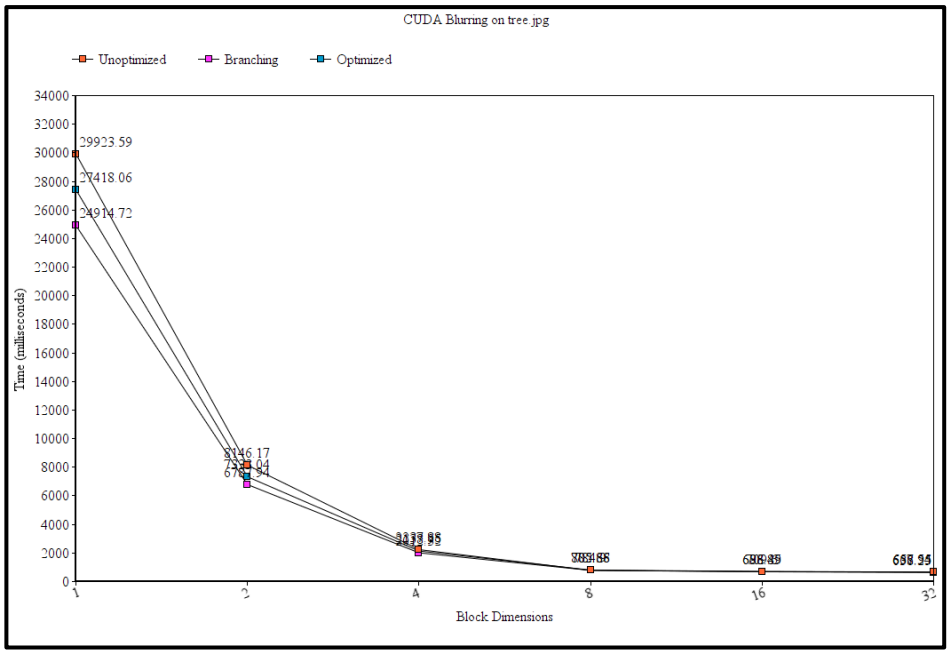

Figure 6 - Comparing Part One, Part Two, and Part Three on tree.jpg