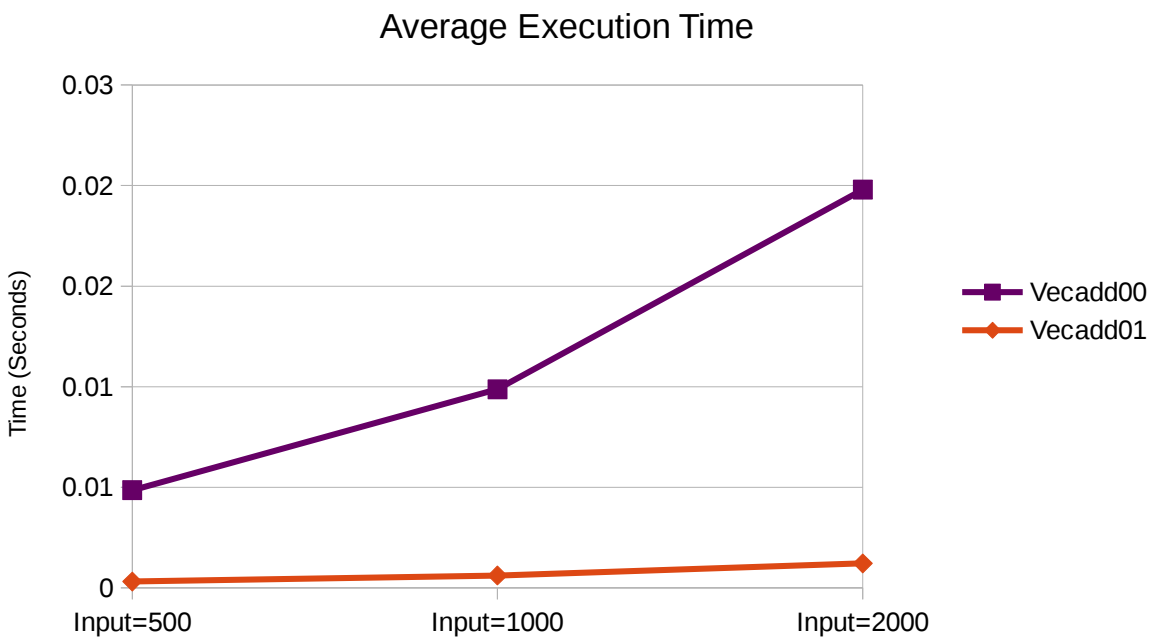
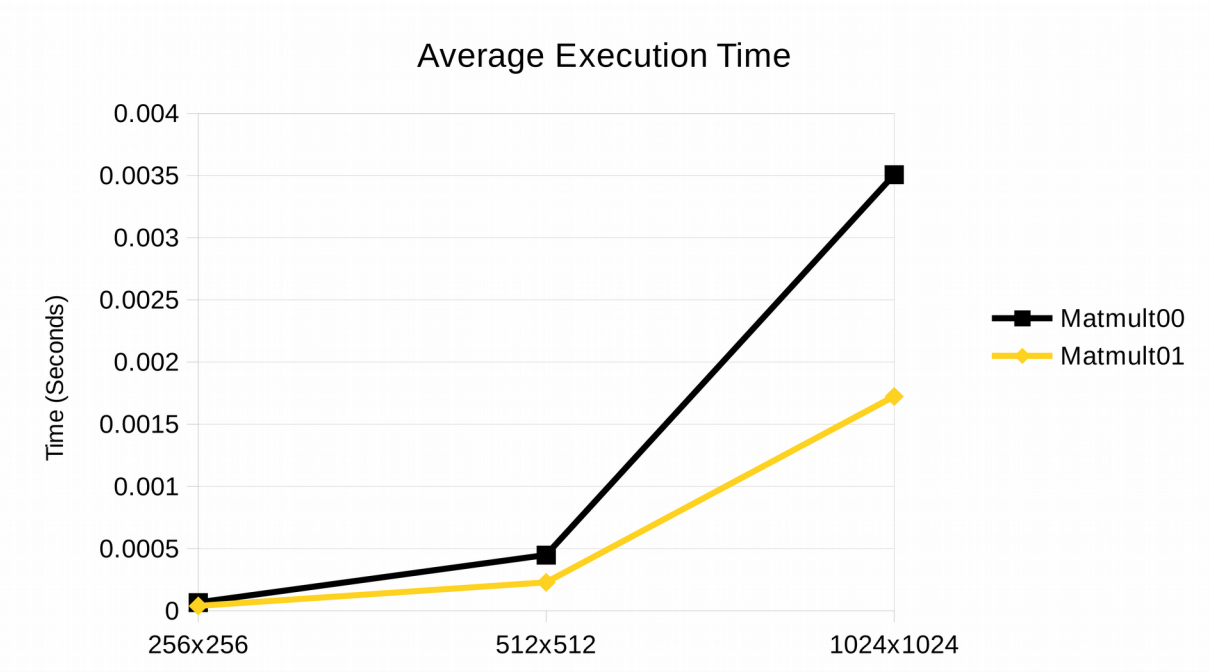
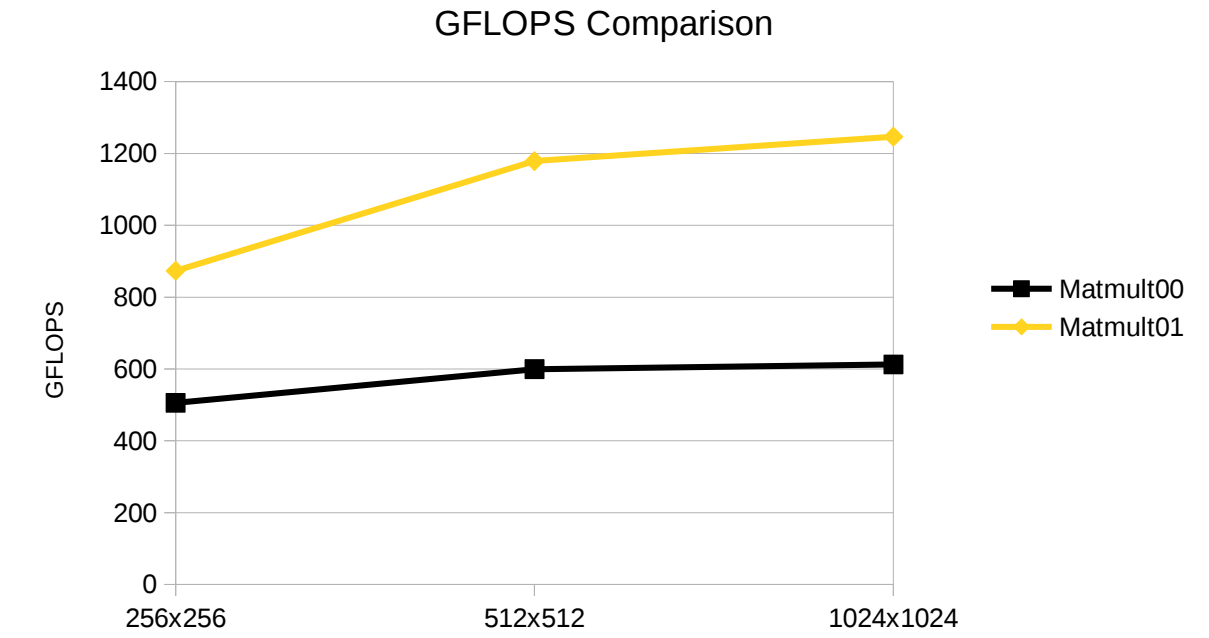
Tanner Pflager
CS 475
PA5 Report
11/14/16

The execution times between vecadd00 and vecadd01 varied greatly. Vecadd01 was approximately 1600% faster than vecadd00. Vecadd00 and vecadd01 scaled roughly the same, doubling the input size resulted in about twice the execution time. The graph below shows the executions times of both vecadd00 and vecadd01.



The execution times of matmult00 and matmult01 also differed, but not as significantly as the other pair of executables. The executions times with smaller inputs are relatively the same with matmult00 being the faster of the two. However, as the input values grew, the difference in execution times also grew. Looking at the graph below we can see that as the input size increases, the slopes of the lines increases exponentially but this is because the problem size is growing exponentially. The ratio between matmult00 and matmult01 appears to grow exponentially. This is not the case, we would eventually reach a point where matmult01 stops scaling and cause the ratio to be linear. We can see that this point will be reached in the graph labeled “GFLOPS Comparison”.





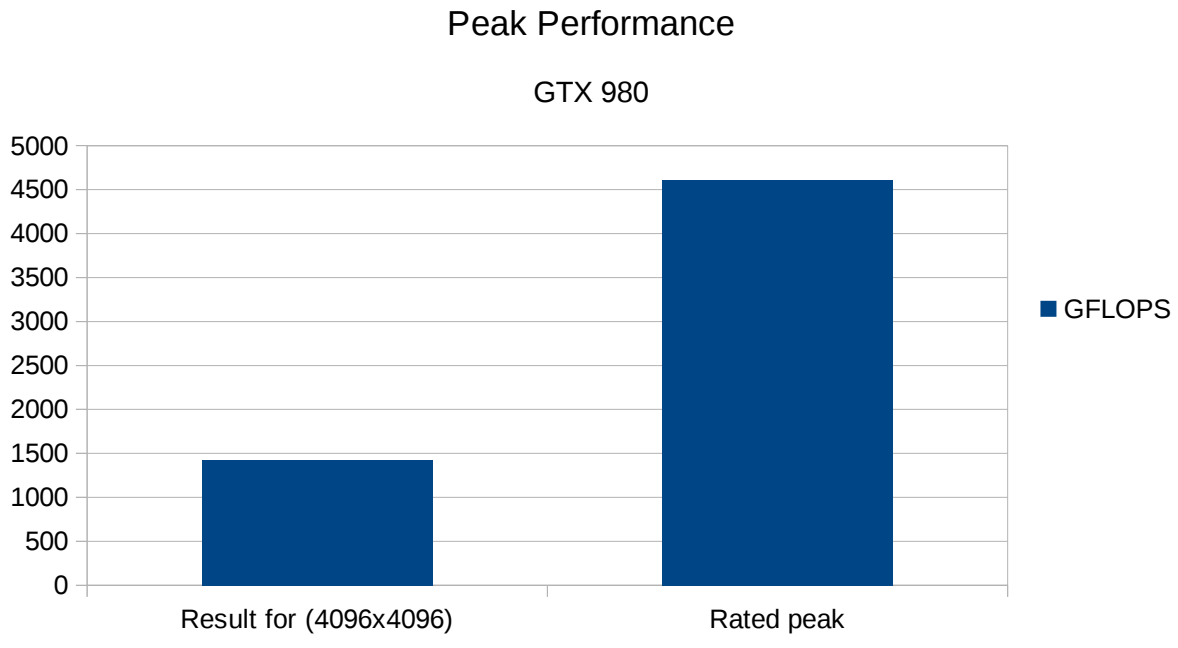
For matmult00 each thread computed a single element in matrix C. The grid dimensions were derived from the data dimensions and the fixed block dimensions. This scheme works well on small inputs. However, with larger inputs, this scheme performs poorly when the grid dimensions become large because blocks will have to wait for other blocks to finish.

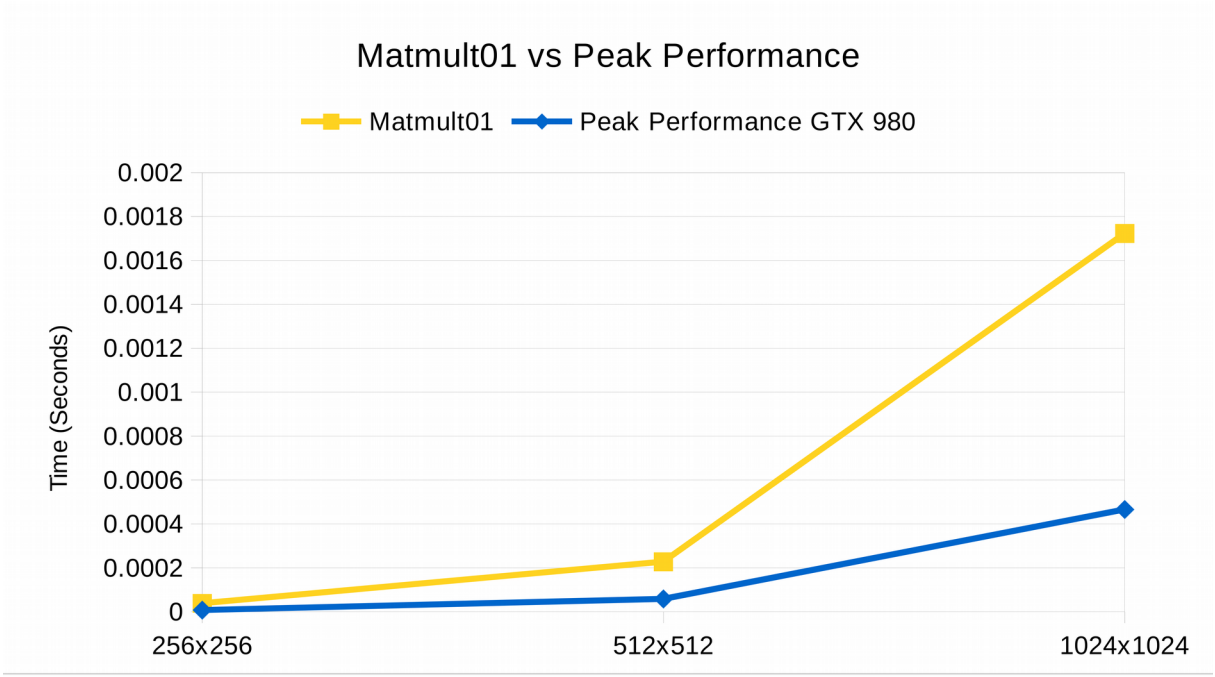
Matmult01 addresses this problem by reducing the grid size by a factor of 4. This results in each thread having to compute 4 values, thus the lifetime of thread, and its data, is longer. This also reduces overhead in swapping blocks in and out because there are less blocks to swap. There are potential drawbacks of matmult01. If the input is not large enough to achieve full occupancy (all cores are active), then the execution time will be lower than that of matmult00.

I've learned a few things when trying to obtain good performance when using CUDA. I think the two most important things I learned is using shared memory and coalescing reads and writes. The access times using shared memory is significantly faster than global memory. Coalescing reads and writes also contributes a significant portion to performance. I believe that optimizing by using shared memory and coalescing should be the starting point for better performance. After that, investigating the optimal number of threads to achieve full occupancy would be the next step.

The GFLOPs of vecadd00 and vecadd01 wasn't even close to the peak performance of the GTX 980. Vecadd00 saw about 0.77 GFLOPs which is no where near the 4612 GFLOPs. Vecadd01 wasn't much closer with an average of 12.62 GFLOPs. Vecadd00 and vecadd01 are bandwidth bound kernels, and I'm sure there can be some improvements, but no improvement (that I'm aware of) would lead to 4612 GFLOPs.

The GFLOPs of matmult00 and matmult01 were more respectable. Matmult01 had 300 to 600 more GFLOPs than matmult00 (see graph above), but still wasn't all that close to the peak performance of the GTX 980. The first graph below shows the maximum GFLOPs I obtained with matmult01 and the peak performance of the GTX 980. The second graph shows the execution times found with matmult01, and estimated execution times of a version of matmult that obtains 4612 GFLOPS. I think matmult01 could be improved. Perhaps a larger block size and adjusted grid size could perform better. I don't believe that peak performance would be reached by a matrix multiplication CUDA program due to bandwidth limiting performance.




Matmult01 vs Peak Performance
Matmult01
Peak Performance GTX 980
Time (Seconds)
0.002
0.0018
0.0016
0.0014
0.0012
0.001
0.0008
0.0006
0.0004
0.0002
0
256x256
512x512
1024x1024