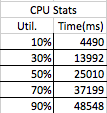
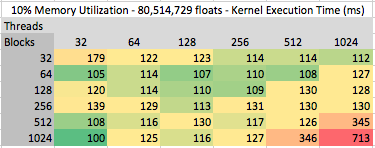
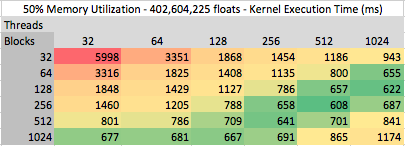
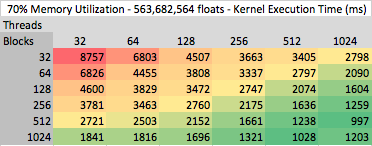
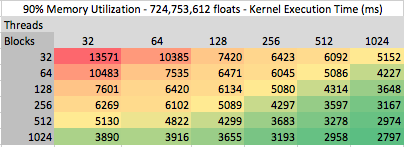
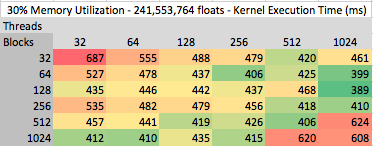
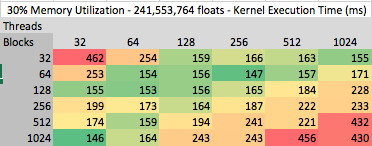
**Matrix Transpose**

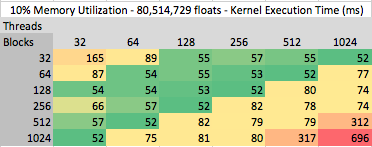
CPU Transpose Results: Global Transpose Results:

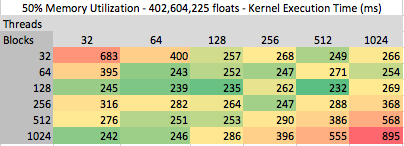


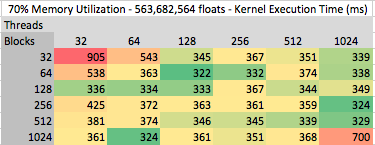


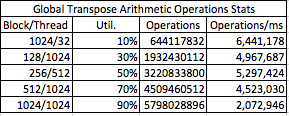
Shared Memory Transpose Results:

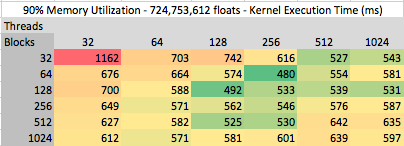
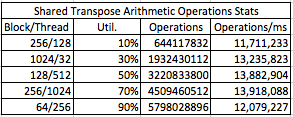








**** Arithmetic Tables:

****

**Conclusion:**

Matrix transposition is crucial to every industry and especially numerical analysis software solutions. Thus speed in computing the transpose is of utmost importance for applications. GPUs can quickly do hundreads of thousands of numerical computations. Our implementation differs between the global and shared memory versions; both use a data partition scheme, but the access pattern is different. The global tranpose has each thread work on a chunk of the incoming matrix to transpose up to a bound. The shared transpose utilizes interleaved acesss and uses a shared memory location as a staging area of the incoming matrix to decrease memory access time compared to the global transpose. With the given constraint of only mono-dimensional access, we calcuate from the given thread index and width the x and y coordinates that pulls from input matrix and then calculate the index to store into the output matrix.

The tables above are colored from a red to green gradient to indicate kernel runtime. Red indicates a poor runtime performance and, conversly, green indicates a faster execition time. These tables allow us to figure out the best kernel parameters for both tranpose kernels. The shared memory kernel performed significanly better, with a 66.59% decrease in execution time compared to the global memory kernel. With less time spent waiting on memeory transfer, the number of operations per second increases as well in the shared kernel. Both transpose implementations did eight arithmetic operations per cell. To get the best performance, you must test every possible combination of configuration parameters. A general trend for performance is the larger the data, the more blocks and threads are needed for processing, but using too many will cause a performance decrease. The range of block sizes and thread count from 128 to 512 provide the best overall performance. Those appear to have consistent runtimes compared to the extreme block sizes and thread count of 1024 and 32. To compare the GPU vs CPU implementations, the GPU will out perform the CPU when the number of numerical calculations is large enough. If you have smaller datasets (under 5% in our testing), the CPU will have the advantage. We compared the CPU with the 10% to 90% memory of the GPU, and in all those cases, both GPU implementations are signficantly faster.

For future work, I would like to try differing transpose methods, we implemented the generic transpose with no optimizations. Our implementation does double the number of arithmetic operations. We are not doing the upper triangle method, thus losing out on the performance increase and runtime decrease of proportionally half.