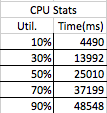
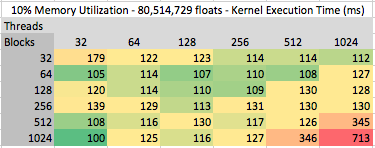
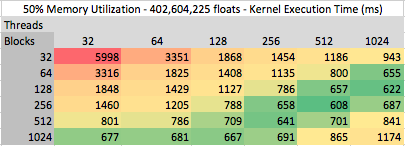
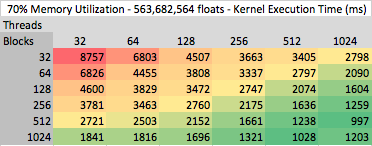
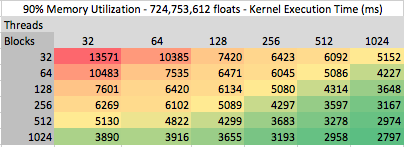
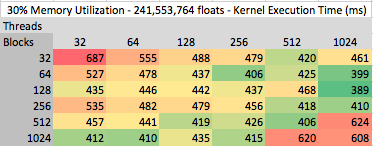
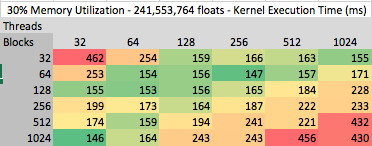
**Matrix Transpose**

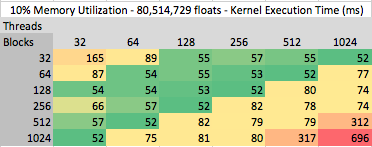
CPU Transpose Results: Global Transpose Results:

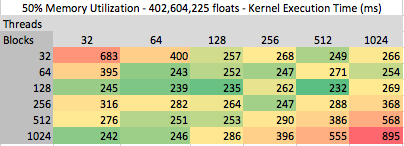


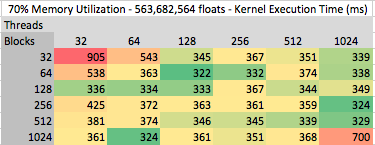


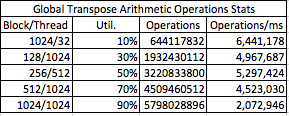
Shared Memory Transpose Results:

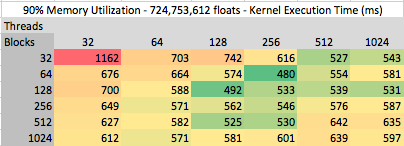
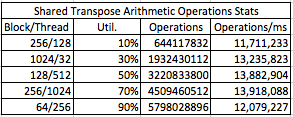








**** Arithmetic Tables:

****

**Conclusion:**

Matrix transposition is crucial to every industry and especially numerical analysis software solutions. Thus speed in computing the transpose is a upmost importance for applications. GPU can do hundreads of thousands numerical computations fast. Our implementation is different between the global and shared memory, both use a data partition scheme, but the access is different. The global tranpose has each thread work a chunk of the incoming matrix to transpose up to a bound. As does the shared transpose, but we use the shared memory location as a staging area of incoming matrix to decrease access time compared to global access. 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 shared memory does the same index calculation, but the memory access is coalesced tiling and interleaving for shared the transpose.

The tables above are colored from a red to a green gradient to indicate runtime performance. A color that is red, is poor runtime performance compared to the others, and conversly green is good runtime. Our tables allow us to figure out the best kernel parameters for both tranpose kernels block and thread. The shared memory performed significanly faster with an average of 66.59% better than global memory transpose at every test point. The speed up from global to shared, is noticeable, even the number of arithmetric operations increased per second. Both transpose implementation did 8 arithmetic operations per cell. To get the best performance, you must test every possible combinations. A general trend for performance is the larger the data, the more blocks and threads are needed for processing, but not too many. The range of block sizes and thread count are 128 to 512 provide the best overall performance. Those seem to have consistent runtimes compared to the kernel parameters of the extremes of block sizes and thread count of 1024 and 32. This is highlighed between the shared memory tables and global memory tables. The global memory tables can handle those outliers with decent performance, but the shared memory struggles. To compare with the GPU vs CPU, the GPU will out perform the CPU when the number of numerical calculations is large enough. If you have smaller datasets under %5, the CPU will have the advantage. We compare the CPU with the 10% to 90% memory of the GPU, and in all those cases, the GPU is signficantly faster in both the shared memory and global memory cases.

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.