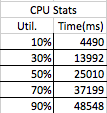
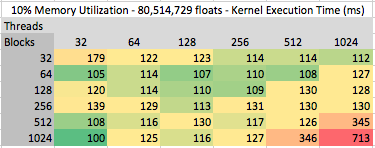
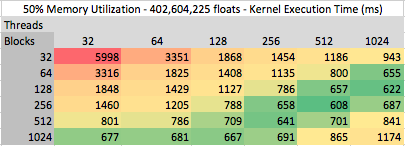
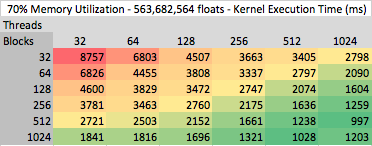
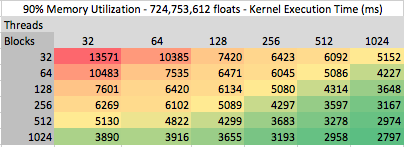
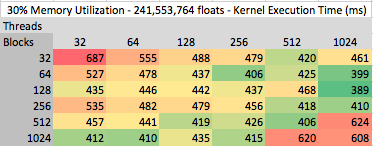
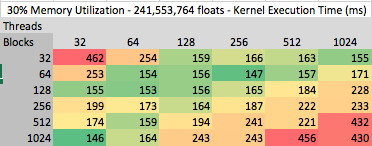
Matrix Transpose

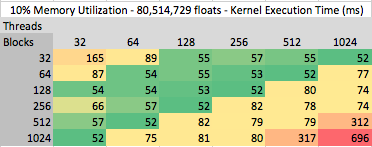
CPU Transpose Results: Global Transpose Results:

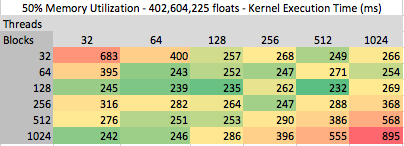


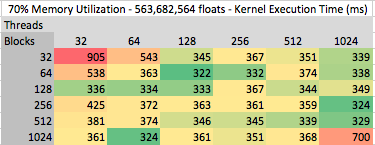


Shared Memory Transpose Results:

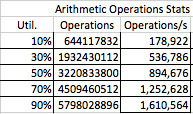
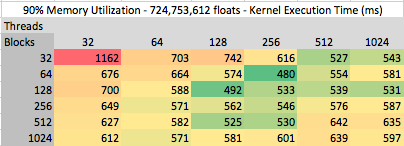








Arithmetic Table:



Conclusion:

The tables above are colored from a red to 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. The shared memory performed significanly faster with an average of <I need this stat>% better than global memory transpose. The speed up from global to shared, is noticeable. Both transpose implementation did 8 arithmetic operations per cell. The memory location of data to have arithmetic operations is a deciding factor in performance.

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.

To compare with the GPU vs CPU, GPU will out perform the CPU, if 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, and in all those cases, the GPU is signficantly faster in both the shared memory and global memory cases.

We do differing implementations on the global verse shared in memory access. We have a coalesced memory access of tiling and interleaving for shared transpose, while our global tranpose does not have that ability. For future work, I would like to try differing transpose methods, we implemented the generic transpose with no optimizations.