**Project 1**

**Vector Addition and Matrix Transposition**

**Will Starms & Matt England**

**Vector addition**

Vector addition is relatively simple, just add two vectors. There’s a lot of optimizations that can be made, though I didn’t get the chance to do many of them. Work was divided into chunks based on the number of total threads. If the total number of vector elements isn’t a multiple of the thread total, the final thread in the final warp is assigned a larger batch of work to make up for the rounding of the step size. This minimizes warp divergence to a single warp, but it’s also pretty bad since the entire kernel will be stuck waiting on a single thread. Instead of having the last thread do N extra additions, it would have been more efficient to have the first N threads do one extra addition (I wish I had thought of that before just now). There would still be a warp divergence, but it would be much less severe. During computation, the vector pointers are incremented in the loop and dereferenced instead of being indexed to remove the time needed to calculate index values.

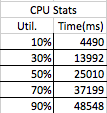
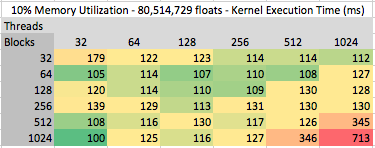
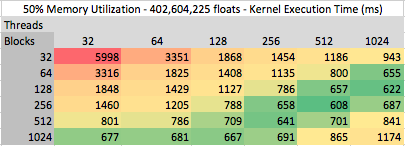
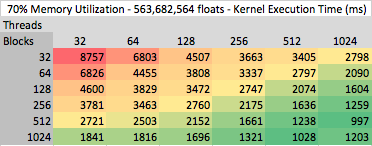
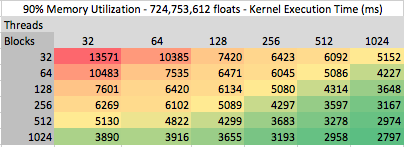
The next page shows all the data gathered. Colored charts show the time in milliseconds at different memory utilization level at different thread and block sizes (all vector sizes are the size of a single vector). There is also timing data for the memcpys needed to move data and the CPU version time. As the amount of work increases, the faster configuration range (the green band) moves farther down and to the right, as expected; more work requires more workers. The colors do lie a bit. While the green band was fastest, some are considerably more inefficient than others; they all took roughly the same amount of time, but some use considerably more total threads and are slower for it. What was interesting to me was that a block size of 32 was fastest for all utilization levels. More, smaller blocks performed better than fewer, larger blocks, like was said in class. This version works completely in global memory, and global memory is slow, not to mention my non-coalesced access pattern. The memcpys (two to the card, one back) were, looking at the green bands, about 3/4 of the overall execution time, a significant cost. Multi-GPU execution wasn’t very interesting since we have one device with two identical cards, so they have the same memory size and execution times.

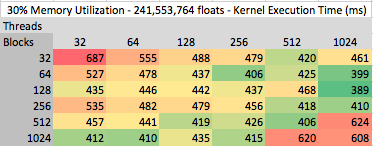
As for improvements, global memory use, the access pattern, and the memory transfer can all be greatly improved. As shown by the transpose times, switching to shared memory significantly improves execution time, even when it looks like it adds additional steps. Switching to an interleaved access pattern like the shared memory transpose kernel will also see improvements due to coalesced memory access. Switching to pinned memory should help with the transfer speed, though the amount of memory being locked may be an issue. With what I know right now, that’s about all I can think of to optimize vector addition, and these optimizations should really apply to any general CUDA code. Enabling compiler optimizations would obviously help, they were left off for this experiment, for better or for worse. One interesting experiment would be to see the effects of using smaller pinned segments and creating a pipeline of memcpys and kernel launches.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 10% Memory Utilization - 80,528,179 floats - Kernel Run Time (ms) | | | | | | |  |  |  |  |  |  |  |  |
| Threads | 32 | 64 | 128 | 256 | 512 | 1024 |  | Memcpy Stats | | | |  | CPU Stats | |
| Blocks |  | Util. | ms/Copy | MB | MB/s |  | Util. | Time(ms) |
| 32 | 139 | 95 | 79 | 125 | 154 | 160 |  | 10% | 111 | 307 | 2767.48 |  | 10% | 1879 |
| 64 | 94 | 77 | 128 | 142 | 158 | 149 |  | 30% | 333 | 922 | 2767.48 |  | 30% | 5251 |
| 128 | 82 | 119 | 137 | 156 | 166 | 156 |  | 50% | 676 | 1,536 | 2272.12 |  | 50% | 8802 |
| 256 | 136 | 146 | 143 | 162 | 149 | 155 |  | 70% | 786 | 2,150 | 2735.79 |  | 70% | 12742 |
| 512 | 128 | 147 | 154 | 149 | 153 | 236 |  | 90% | 938 | 2,765 | 2947.46 |  | 90% | 15727 |
| 1024 | 119 | 149 | 157 | 150 | 235 | 378 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 30% Memory Utilization - 241,584,537 floats - Kernel Run Time (ms) | | | | | | |  | 50% Memory Utilization - 402,640,896 floats - Kernel Run Time (ms) | | | | | | |
| Threads | 32 | 64 | 128 | 256 | 512 | 1024 |  | Threads | 32 | 64 | 128 | 256 | 512 | 1024 |
| Blocks |  | Blocks |
| 32 | 741 | 555 | 417 | 373 | 401 | 479 |  | 32 | 7060 | 4149 | 2322 | 1856 | 1232 | 848 |
| 64 | 560 | 441 | 360 | 400 | 447 | 432 |  | 64 | 4153 | 2305 | 1874 | 1107 | 739 | 666 |
| 128 | 467 | 365 | 390 | 433 | 407 | 403 |  | 128 | 2323 | 1876 | 1038 | 731 | 635 | 649 |
| 256 | 427 | 468 | 420 | 409 | 399 | 427 |  | 256 | 1740 | 1071 | 724 | 630 | 622 | 675 |
| 512 | 352 | 407 | 422 | 390 | 420 | 482 |  | 512 | 584 | 621 | 655 | 626 | 678 | 738 |
| 1024 | 341 | 384 | 402 | 426 | 477 | 483 |  | 1024 | 529 | 628 | 632 | 685 | 728 | 848 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 70% Memory Utilization - 563,697,254 floats - Kernel Run Time (ms) | | | | | | |  | 90% Memory Utilization - 724,753,612 floats - Kernel Run Time (ms) | | | | | | |
| Threads | 32 | 64 | 128 | 256 | 512 | 1024 |  | Threads | 32 | 64 | 128 | 256 | 512 | 1024 |
| Blocks |  | Blocks |
| 32 | 13805 | 9869 | 5506 | 4004 | 2698 | 1815 |  | 32 | 17780 | 15989 | 9405 | 6821 | 4679 | 2943 |
| 64 | 9809 | 5495 | 3992 | 2592 | 1229 | 886 |  | 64 | 15881 | 9349 | 6775 | 4585 | 2208 | 1287 |
| 128 | 5558 | 3963 | 2578 | 1061 | 892 | 842 |  | 128 | 9524 | 6824 | 4677 | 1729 | 1236 | 1078 |
| 256 | 3864 | 2581 | 1017 | 872 | 857 | 882 |  | 256 | 6631 | 4562 | 1674 | 1259 | 1061 | 1047 |
| 512 | 912 | 912 | 863 | 839 | 846 | 890 |  | 512 | 2586 | 1596 | 1247 | 1057 | 1086 | 1081 |
| 1024 | 746 | 847 | 847 | 878 | 883 | 989 |  | 1024 | 947 | 1073 | 1077 | 1086 | 1064 | 1083 |

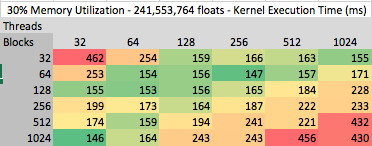
**Matrix Transpose**

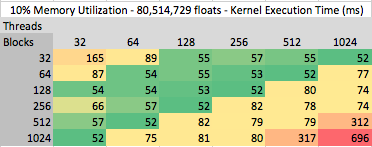
CPU Transpose Results: Global Transpose Results:

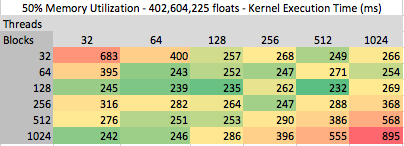


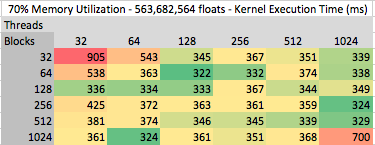


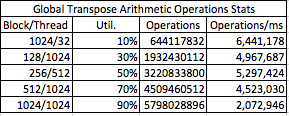
Shared Memory Transpose Results:

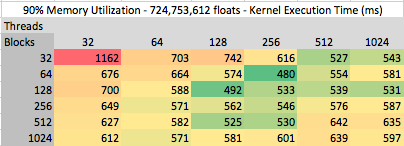
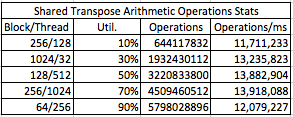








**** Arithmetic Tables:

****

**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.

**Device Info**

Our machine for testing has a GTX Titan Z, which is a two card device with 6144MB of global memory each. Code was compiled with CUDA 7.5 and GCC 4.9.2.

**Work Division**

I (Will) worked on the makefiles, the helper and prep code shared between programs, and the vector addition code. Matt worked on the implementation of the matrix transposition kernels. We both planned and debugged the kernels together and wrote our own relevant sections of this report.