Matrix Multiplication Performance Analysis

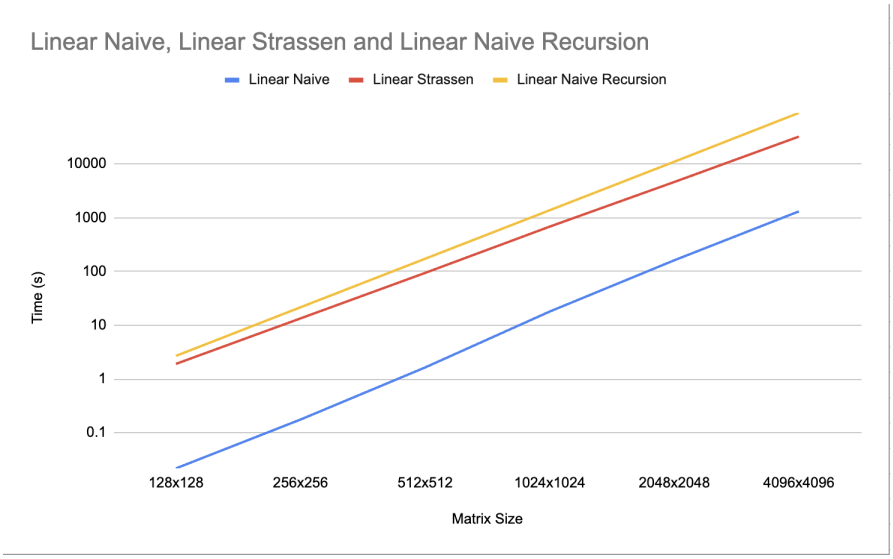
CSCE 435
Group 8

Kevin Dai
An Duong
Steven Mao
Matthew Wang

Control

Our single core linear matrix multiplication processing showed some expected and unexpected outcomes. Firstly, Strassen implementation correctly showed sub-cubic growth ($\log_2 7$) when compared to naive recursion. That is, when doubling the matrix size, it takes our Strassen algorithm roughly 7 times as long to finish computing, whereas our naive algorithm actually takes 8 times as long.

Surprisingly, the time it takes for our recursive algorithms to run on matrices of dimensions 128 to 4096 is around a 100 times slower than simply performing the basic nonrecursive algorithm. Given that the algorithms do not require communication time due to running without parallelization, we attribute this large difference in runtime to cache locality, both spatial and temporal. For every single recursive level, we are initializing new vectors to hold the same values, therefore not utilizing the benefits of fast memory access from our caches. Theoretically, there will be a crossover point in which our recursive Strassen's algorithm will ultimately be faster than the naive solution. The hypothetical matrix size where Strassen's algorithm runs faster than linear naive for our implementations is calculated to be in the millions.

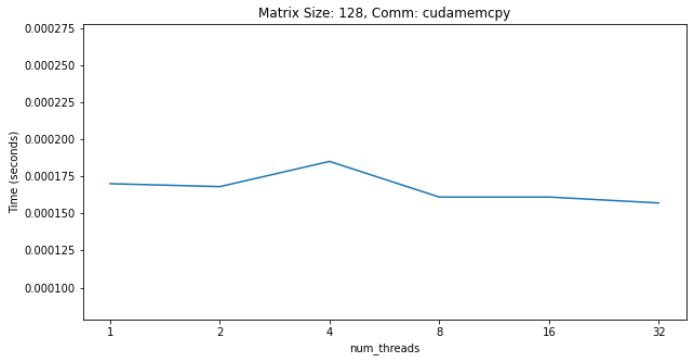


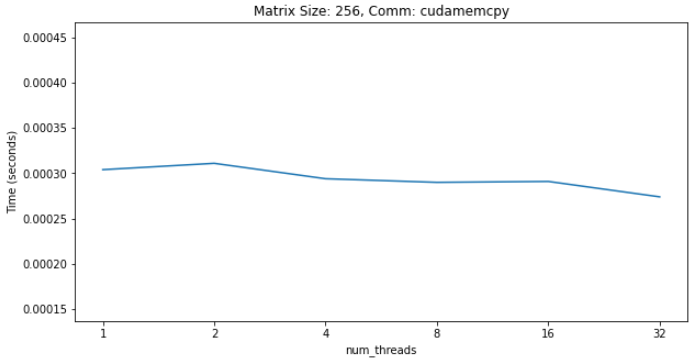
GPU (CUDA)

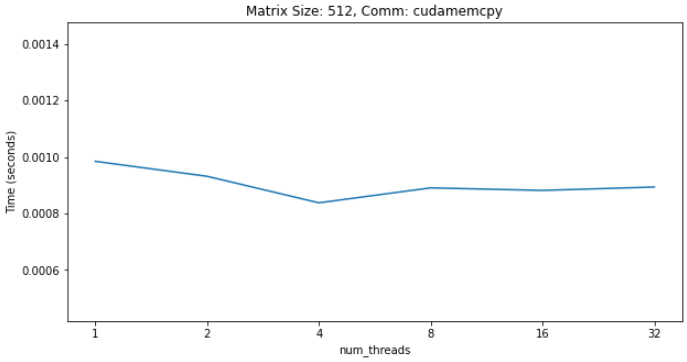
Naive

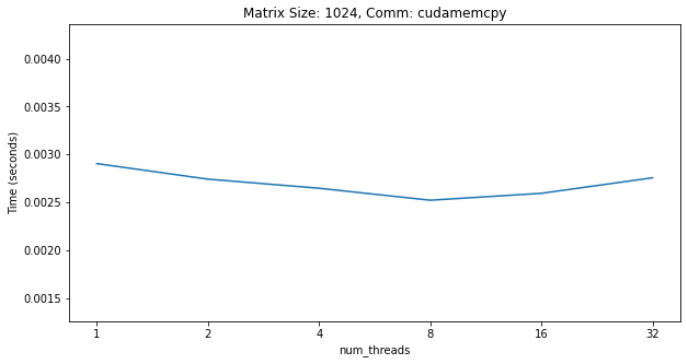
For CUDA GPU implementation, we can see from the graphs below that CUDAMemCopy time did not vary significantly between thread sizes when fixing matrix size. This is expected, since memcopy is not dependent on thread size, but rather the volume of memory it is transferring.

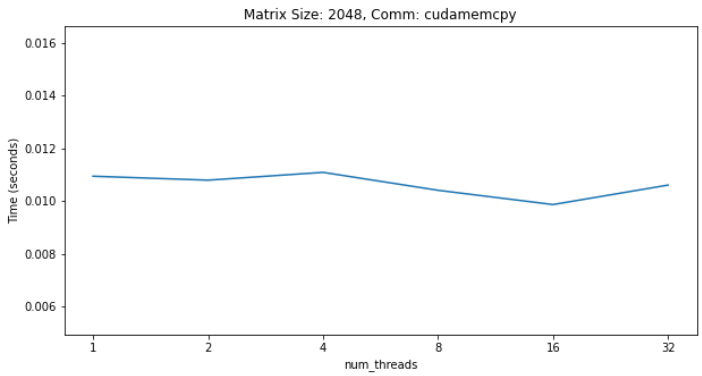
cudaMemcpy

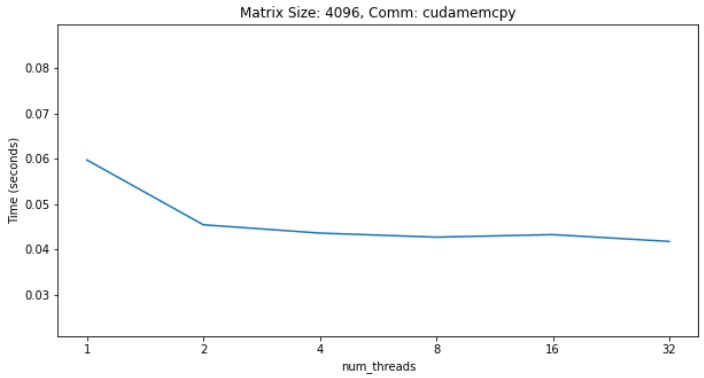








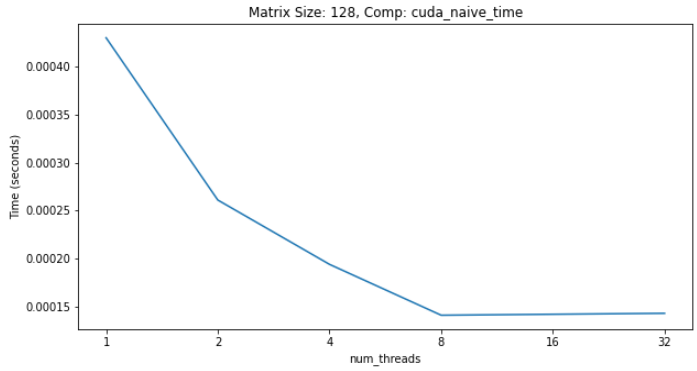


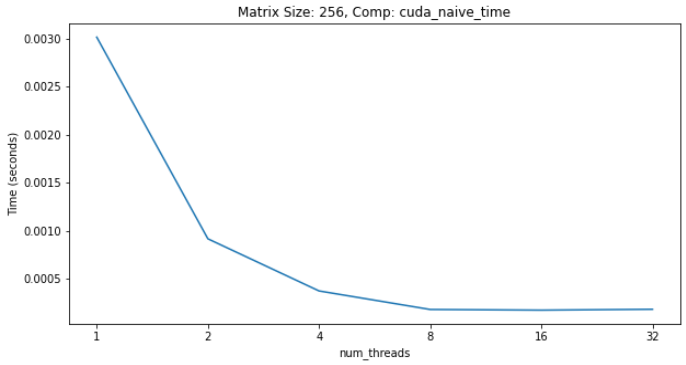


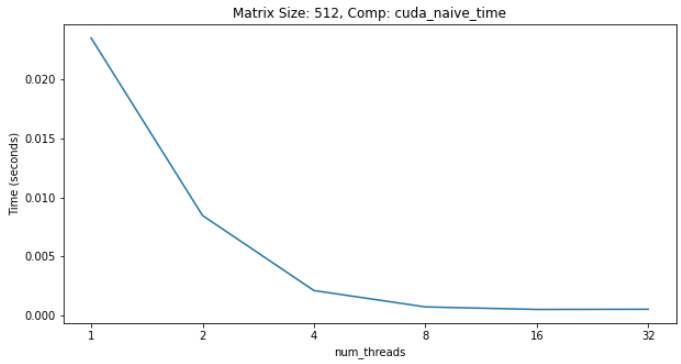
However, when comparing each plot against each other, it is evident that a higher matrix size will result in consistently slower communication time, which can be attributed to the higher parallelization overhead required for bigger matrices.

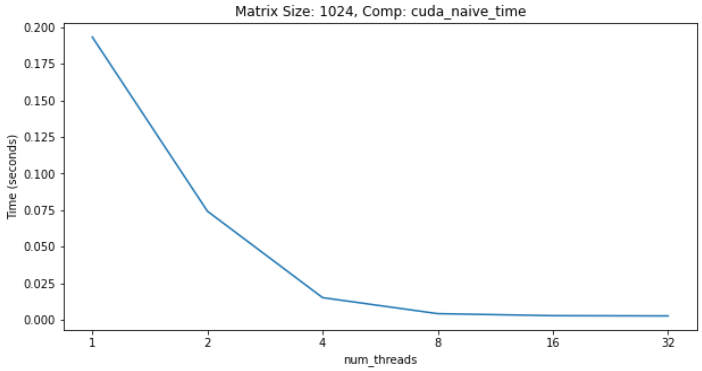
For total computation time, there is a steady decrease in runtime across an increase in thread size. Looking at the trend across graphs, we can see that there is approximately an eight-fold increase in runtime when doubling the matrix size, which tracks our experiment with running on a single core in CPU. The big change, this time, is that CUDA GPU runtime is vastly superior to simply running it sequentially. However, after a certain amount of threads, the performance didn't see any increase compared to few threads. This happens around the 8 thread mark.

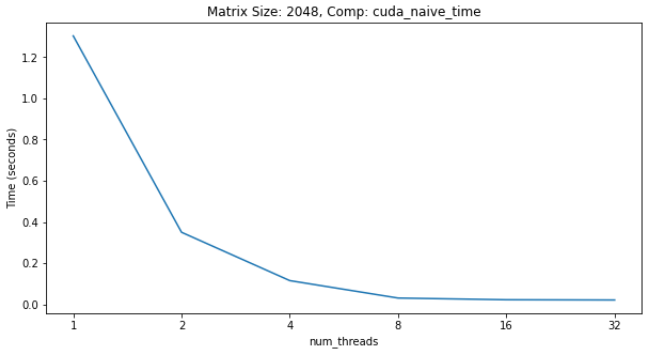
Cuda Naive Time

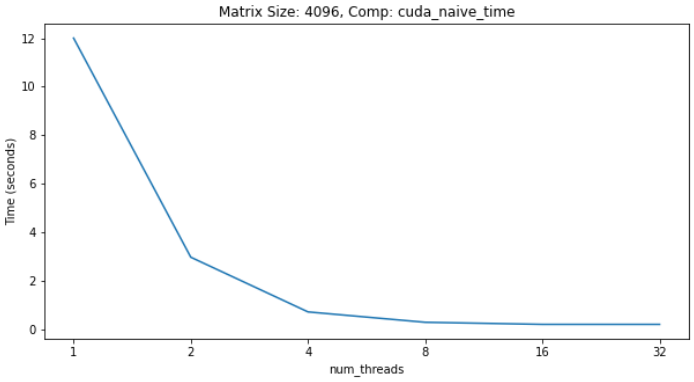








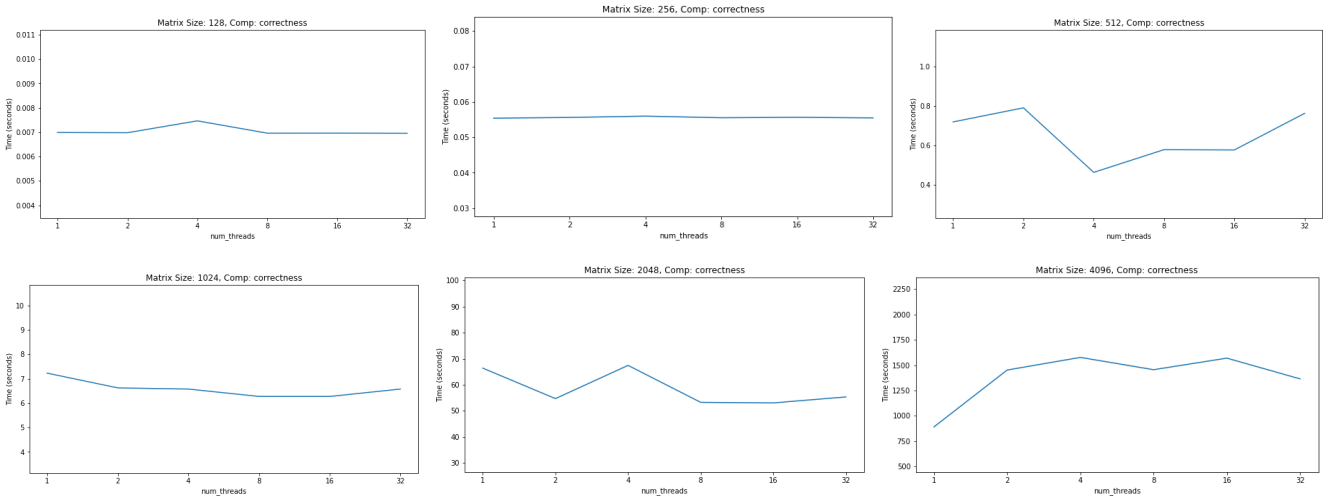




For correctness check runtime, we simply used our single core CPU algorithm (using no parallelization), and so the time didn't vary much across thread sizes. However, there was an outlier data

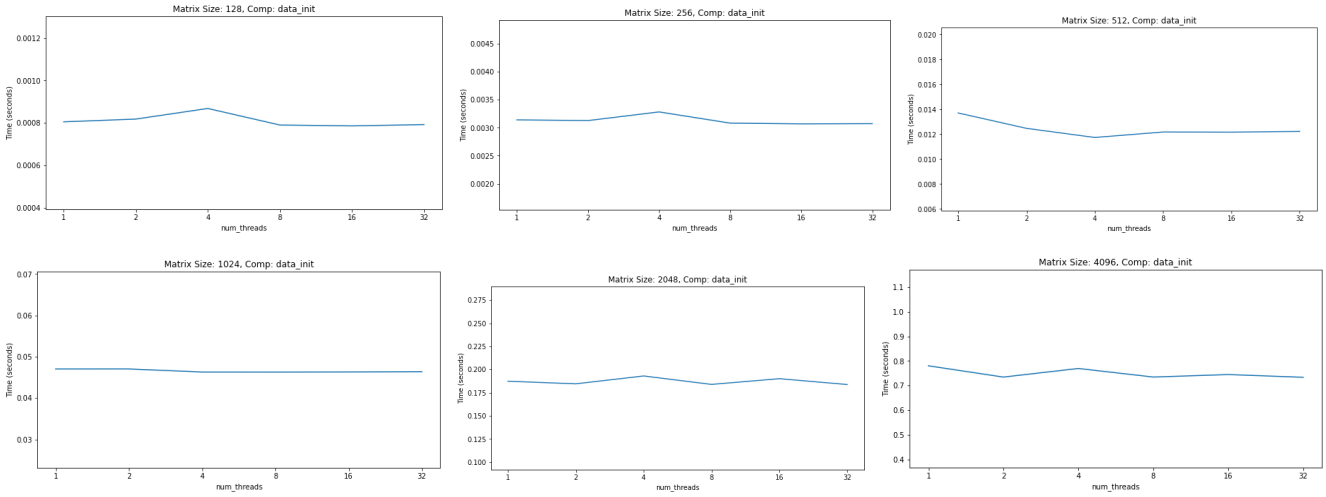
point for thread size of 1 with matrix size 4096. Our correctness check runtime there was significantly faster than the other runtimes, which we determined to be a special case of our node being not as occupied as usual.

Correctness



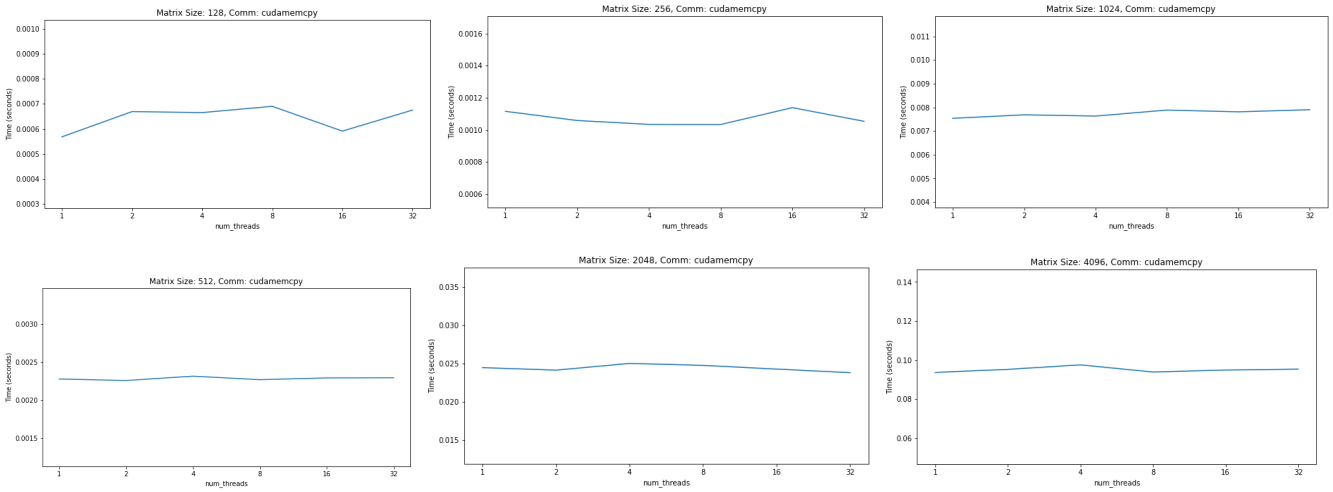
Similar to our correctness check, our data_init operation ran sequentially, and so the time across thread sizes didn't change by much. When doubling the matrix size, though, we can easily see a 4 times increase in time taken. This makes sense since a new matrix with double the dimension of another matrix holds four times as many values.

Data Init



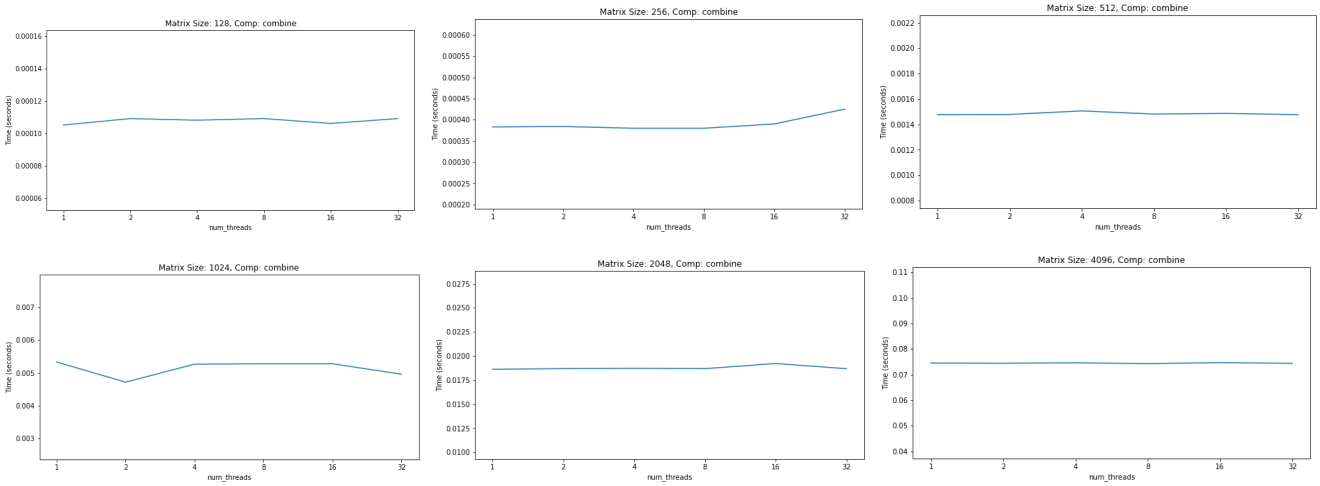
Strassen

cudaMemcpy

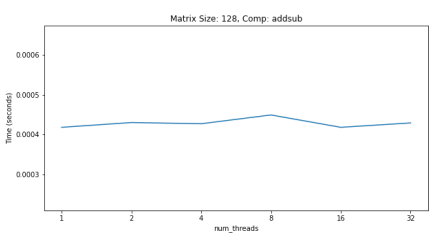


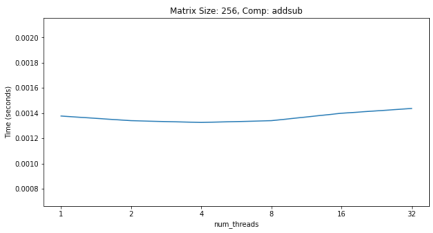
Generally speaking, trend across matrix and thread sizes was not different from that of Naive, in that the CUDAMemcopy time was roughly constant for fixed matrix sizes and larger matrix sizes took longer time. However, CUDAMemCopy times were persistently worse for Strassen when compared to the Naive counterpart. This might stem from the fact that Strassen uses multiple kernels to start the algorithm, thus having a higher overhead for parallelization.

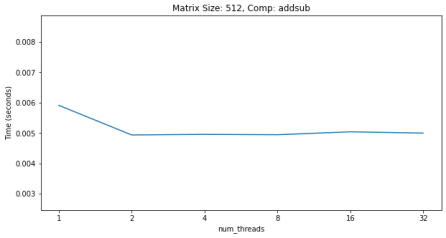
Combine

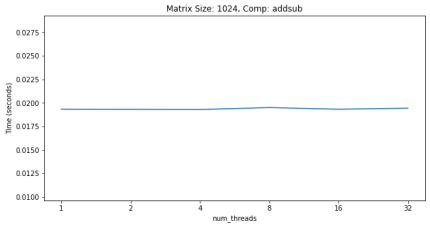


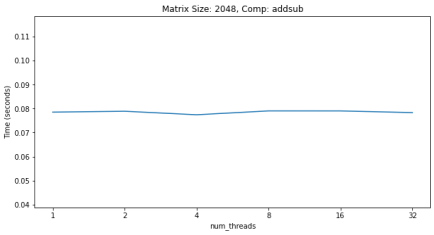
AddSub

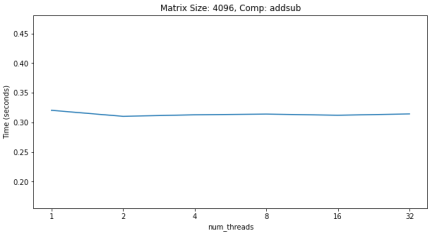




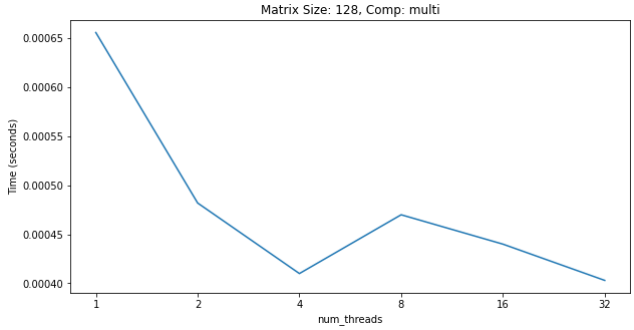


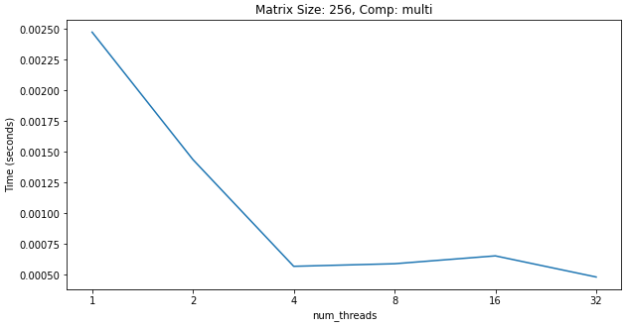


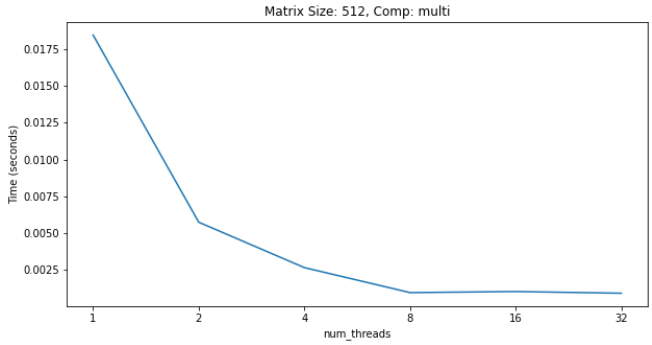


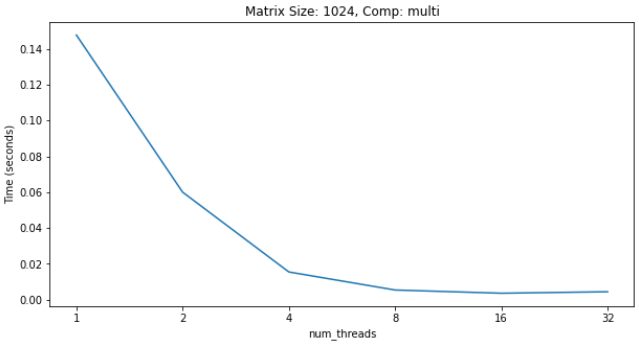


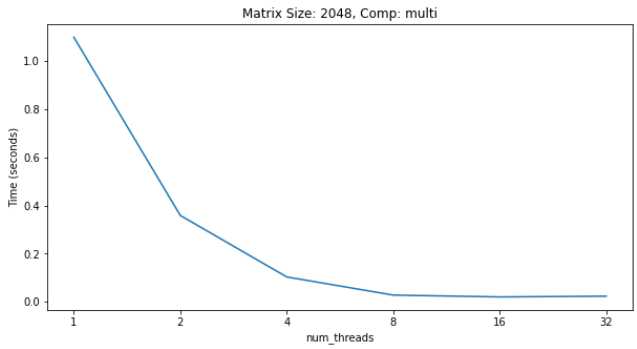
Multi

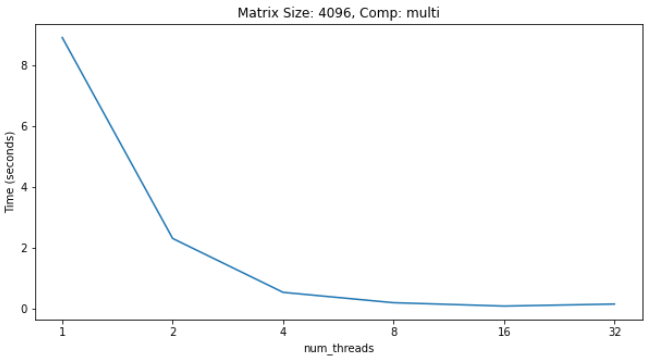




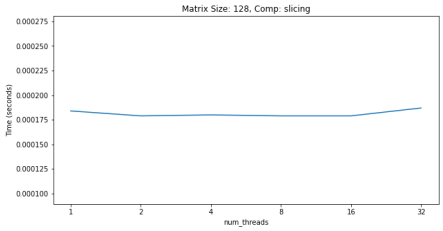


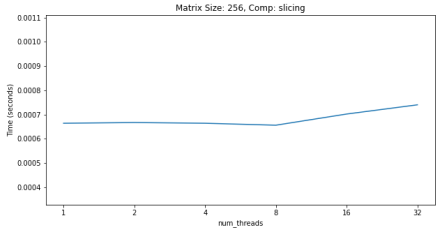


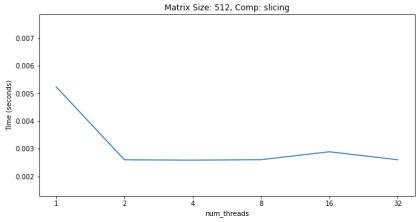


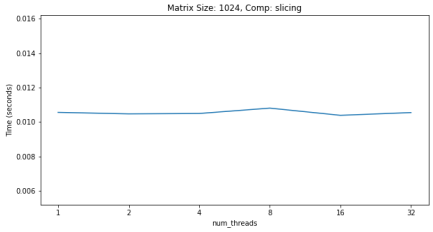


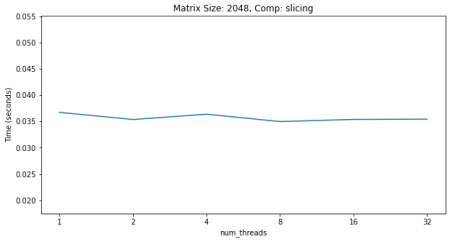
Slicing

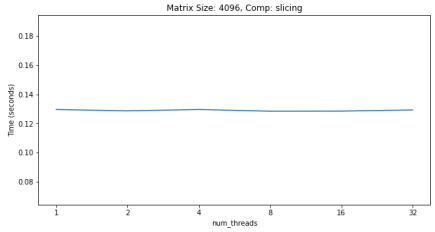




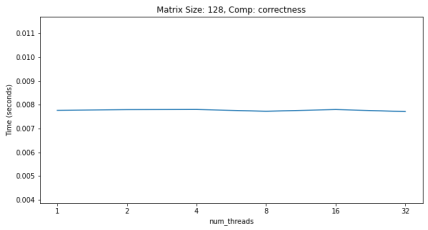


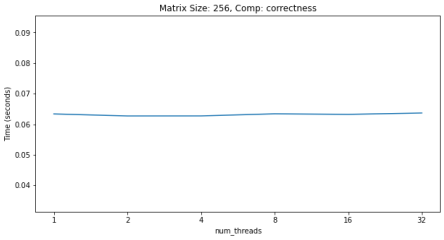


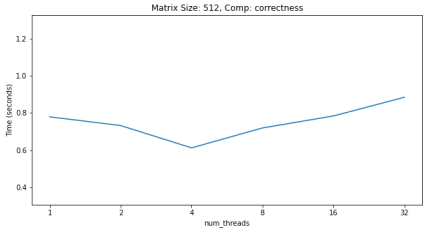


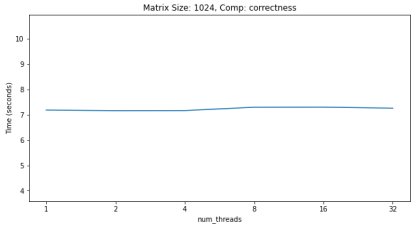


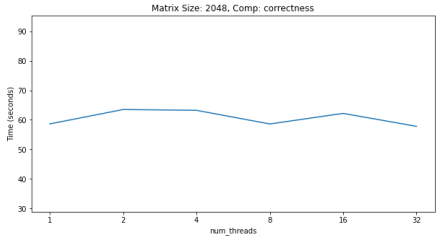
Correctness

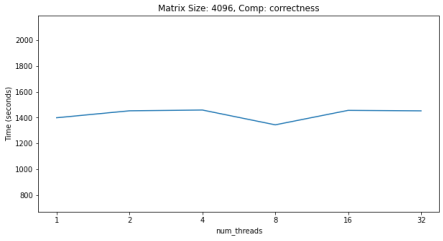




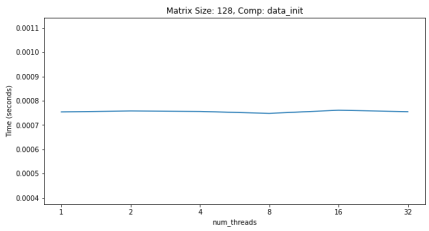


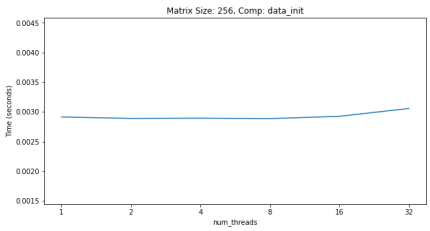


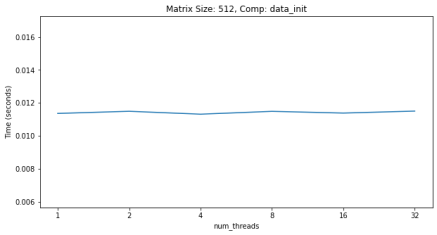


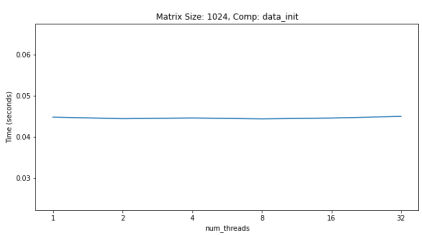


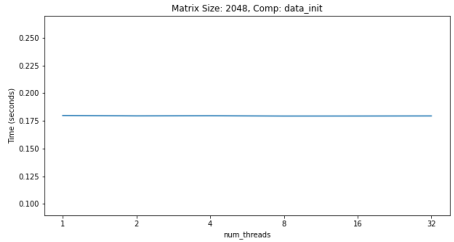
Data Init

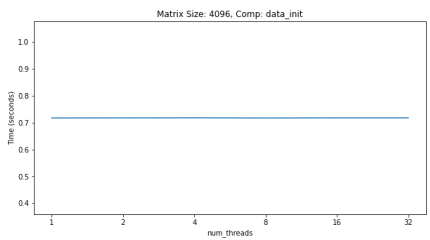










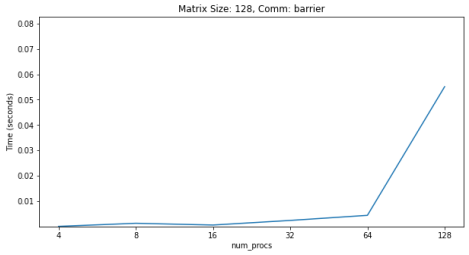


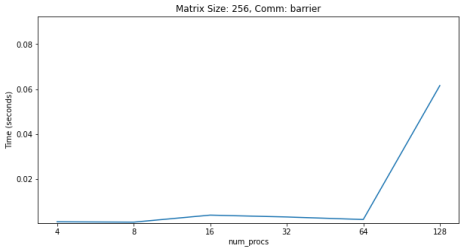
Looking at computation time, all comp_small and comp_large time that aren't 'multi' has a flat line across thread sizes with increasing time across higher matrix sizes, which fits the overall trend we've seen so far. This means that the operations done on the cpu isn't affected by the increase in the number of cuda threads running, and larger matrix sizes take longer time for each computation. Our total computation time decreased with increasing thread size across the board, but we can see that the lower matrix sizes created some deviation in the time. Given how incredibly fast our CUDA computations run (it's taking less than a millisecond to finish our task with a matrix dimension of 128x128), it is feasible that the higher thread count might not necessarily run faster than the lower thread count.

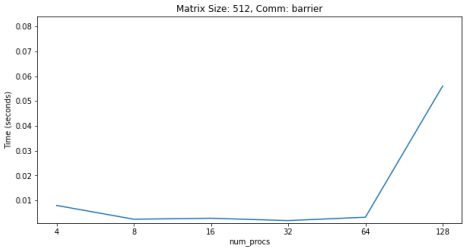
CPU (MPI)

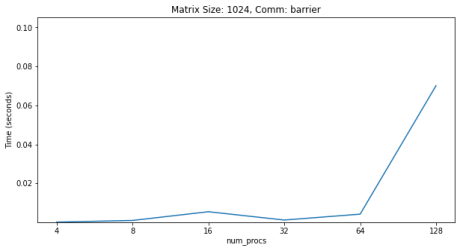
Naive

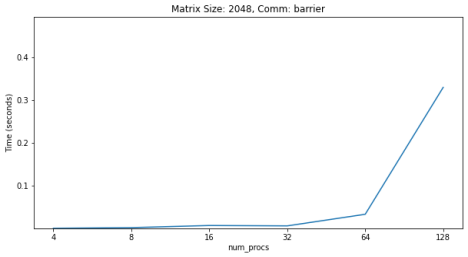
Barrier

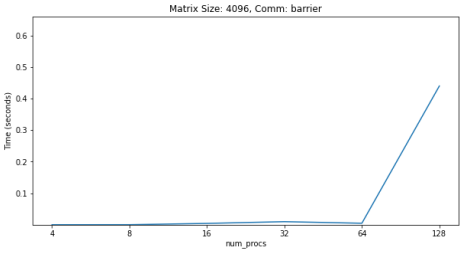




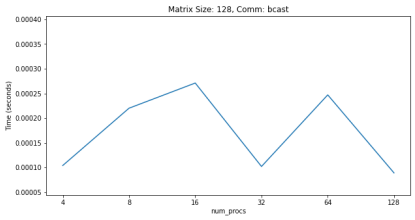


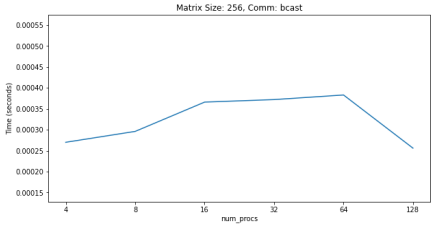


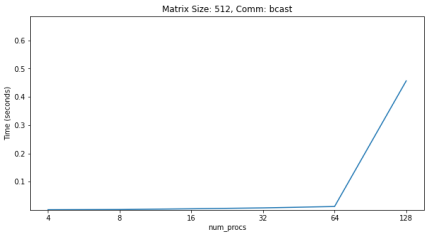


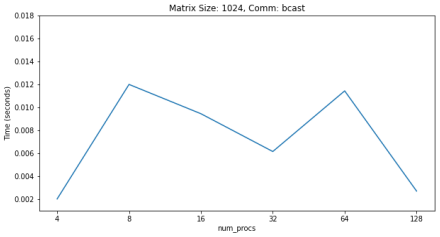


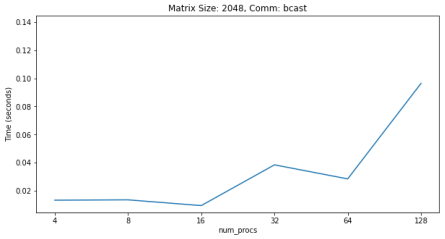
Broadcast

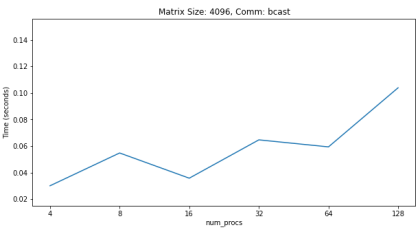




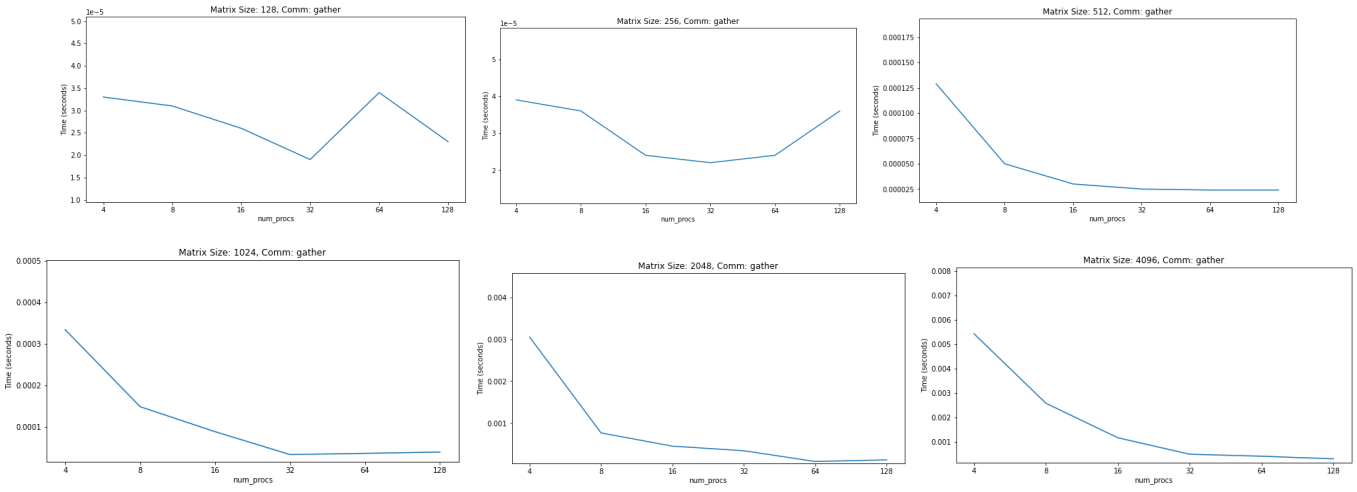




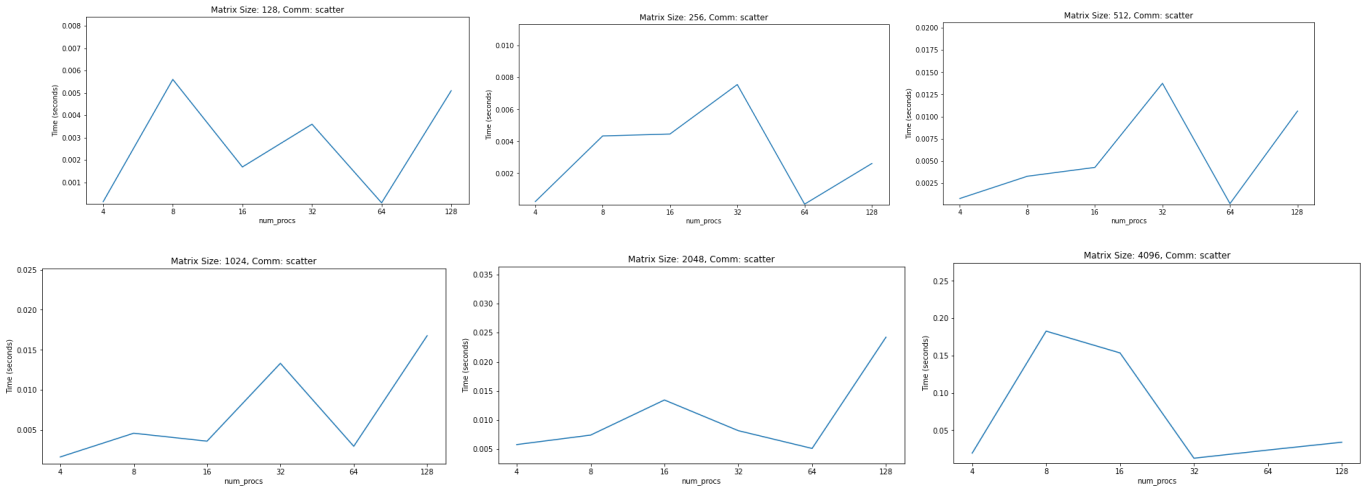




Gather

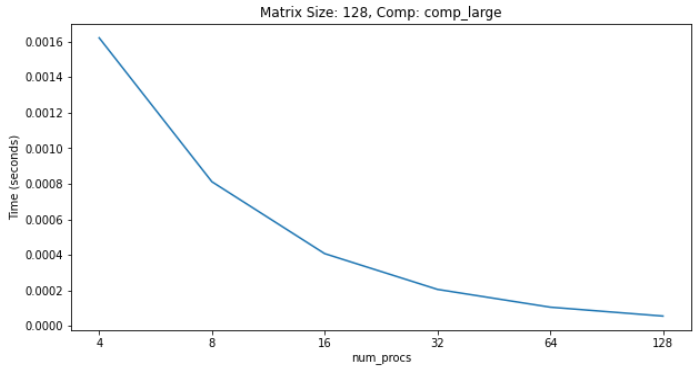


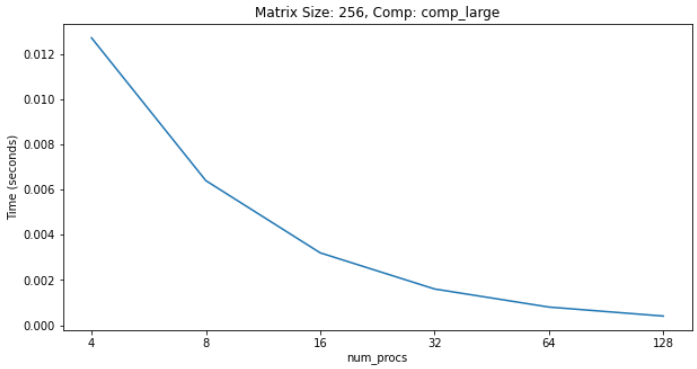
Scatter

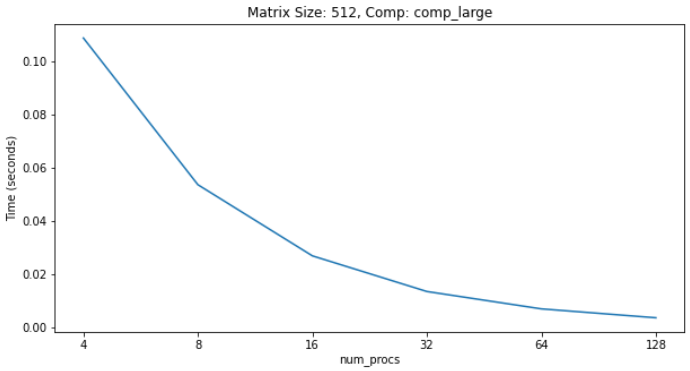


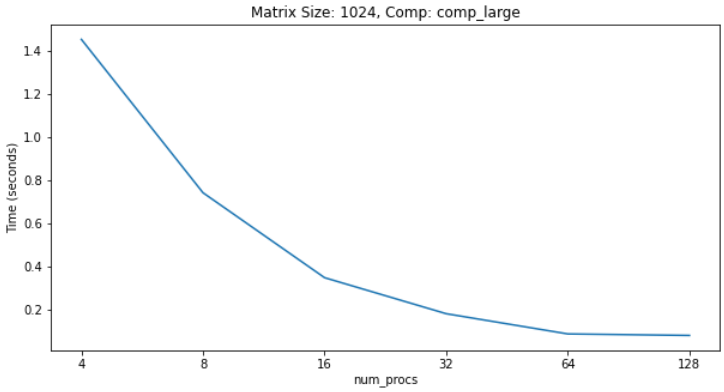
Unlike GPU communication, our MPI communication time had a much higher level of variance across process sizes. Different comm times were recorded over thread sizes, though there's not truly a definite trend to be described. When observing the comm times, we can see that for each matrix size, as we increase the number of worker processes, the time for communication also increases. However there is an edge case when dealing with a 4096 size matrix, where some smaller tasks have a higher time because of the amount of times data has to be called and queued. There is a significant jump in barrier time at process size of 128, and this anomalous effect has been replicated, so it is not due to some confounding factor. This sudden jump also happens in computation time for data_init and correctness check.

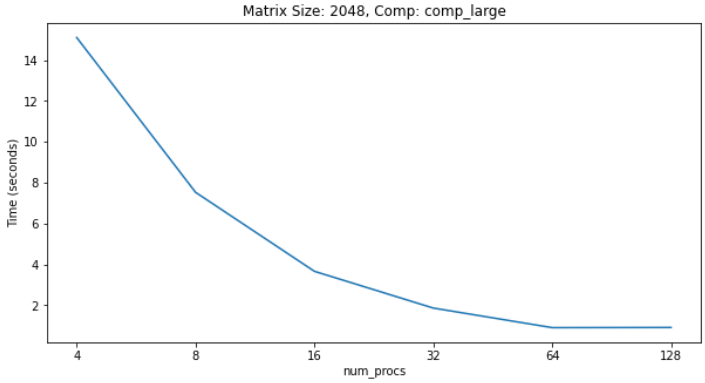
MPI Naive Time

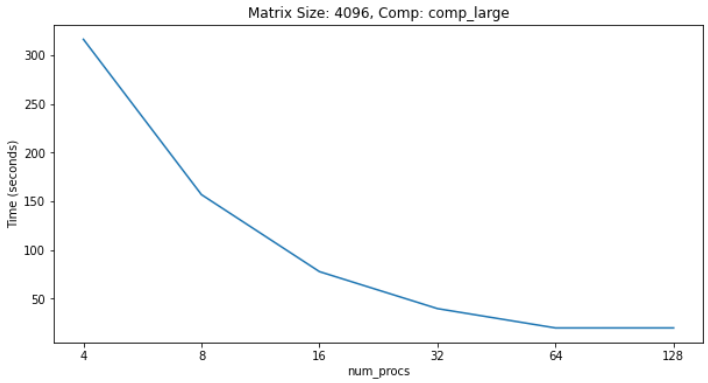




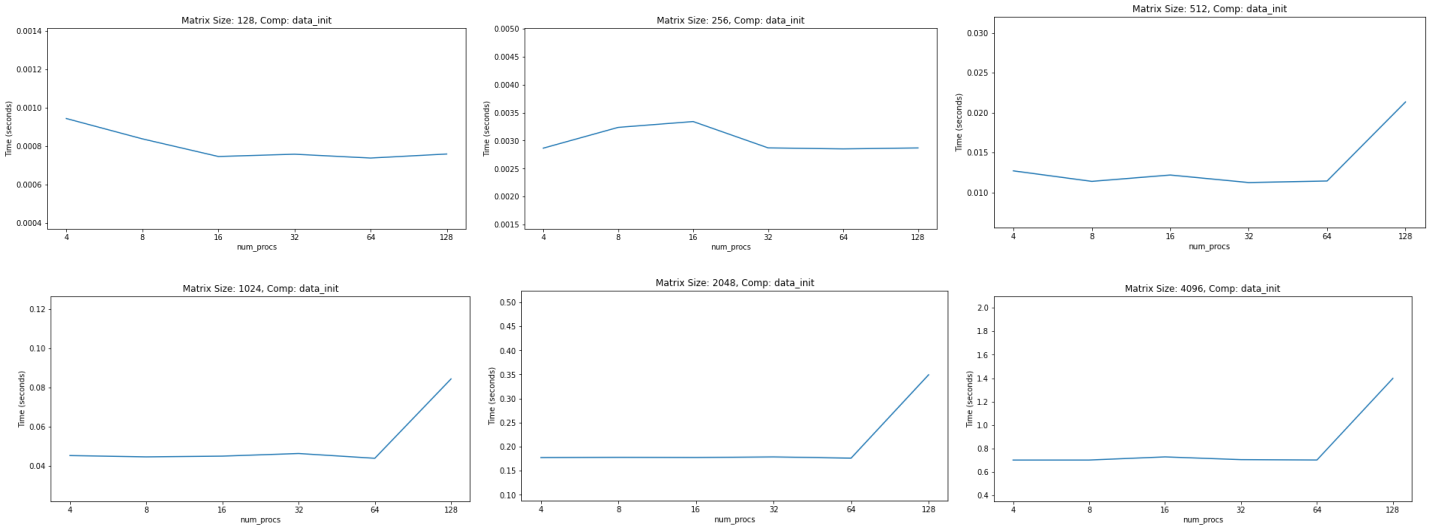




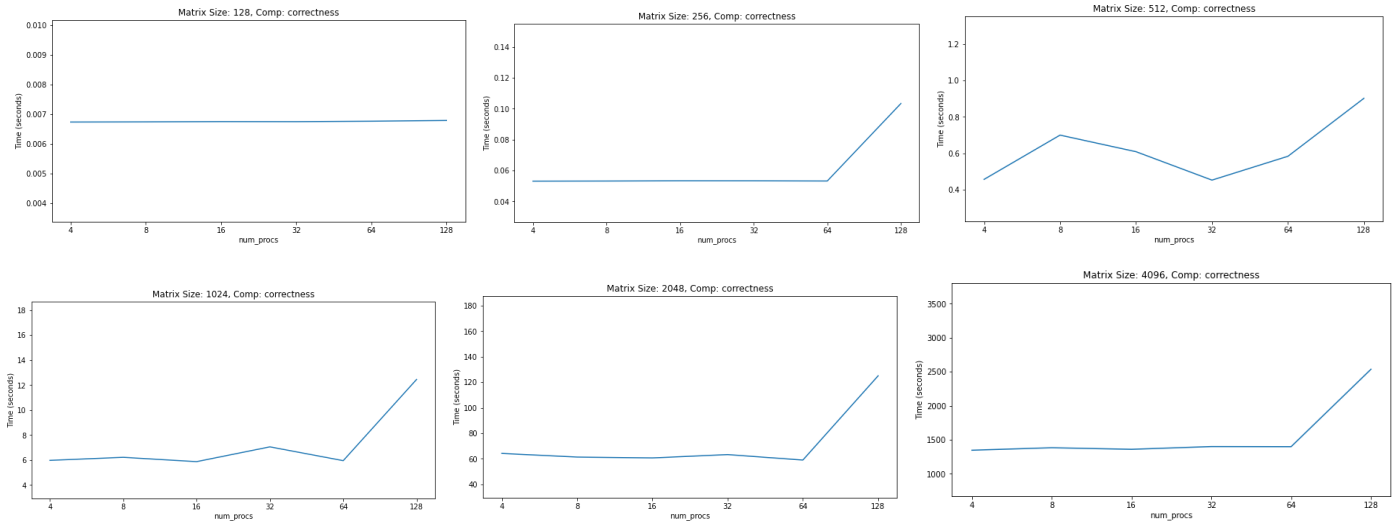




Data Init



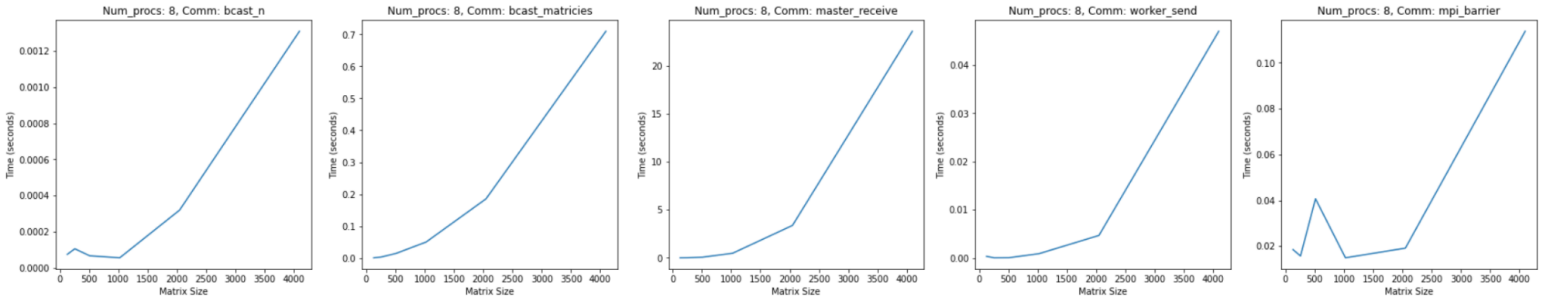
Correctness



In this situation, comp_large indicated the time for our MPI Naive implementation to run matrix multiplication. Even with the sudden uptick in comm and comp time, total runtime performance steadily decreased with a higher process count. Data_init and correctness, which were run on a single rank (rank 0) showed a constant line for every process count except for 128 which increased a significant amount. Our parallelized algorithm ran faster than pure sequential, but it is still not as performant as running on CUDA.

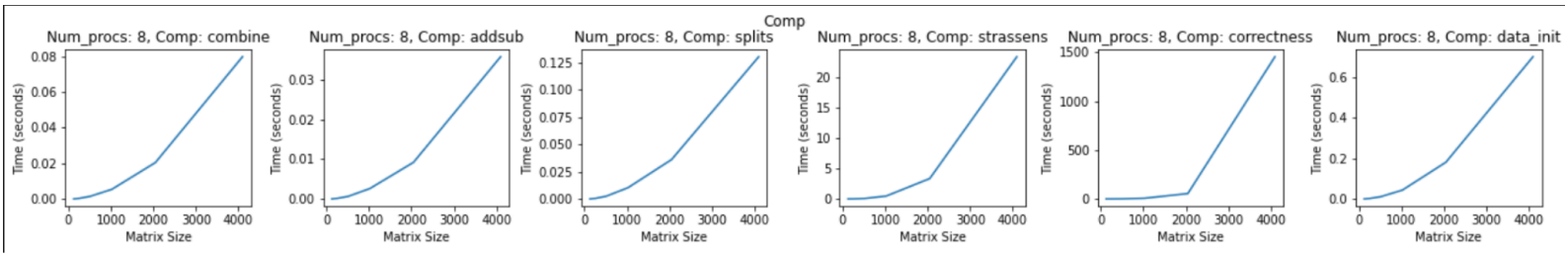
Strassen

Comm



For MPI Strassen implementation, our communication time had a much clearer trend for increasing process sizes. In fact, all measured communication times increased with higher matrix size. During our research, we had trouble finding ways to increase utilization of processes, and so could only perform data collection on a process size of 8. Even so, when compared to the performance of MPI Naive, it is comparable to the 128 process runtime.

Comp



And with computation time, given only one process size, we can see a steady increase in all computation parts, not just the total runtime. As matrix size increases, the time for each step of strassen's (initialization, splitting the matrix, adding and subtracting matrices, and combining each four quadrants matrices into one matrix, and correctness check) all increases as well.

Conclusion

The purpose of this project was to find the most optimal matrix multiple between two Algorithms and the differences in implementation(MPI,CUDA, and Single Core as a control). The project was designed by having a Naive and Strassen implementation on each of the different systems(i.e CUDA, MPI, Single Core). This design was made in choice so that we could compare each system with a control group. Observing this data, we can conclude that the cuda implementation is fastest, then followed by MPI, and lastly our control, linear single core processes. The first main indicator of this is when looking at the smallest sized matrix with the lowest amount of threads/processes, CUDA beats out MPI by almost a factor of three times. This trend follows similarly(in the manner that there is a several factor difference between the two) as the problem size and number of threads/processes grow. Now when comparing the different algorithms, Strassen only beats out Naive when the problem statement is larger in size. When observing the MPI implementations, we see that the difference between the two grows in size especially when the problem size exceeds a 1024x1024 matrix. This can be attributed to the fact that the problem size has crossed over the threshold of being in the millions. In conclusion, the use cases on which matrix multiply to use can be broken down into: the presence of GPU or MPI, and the problem size set.