**Three Approaches to Implementing the Two-Dimensional Fourier Transform using Thread, MPI, and Cuda**

Wootae Song: [wootae@gatech.edu](mailto:ali97@gatech.edu)

Alberto Li: [ali97@gatech.edu](mailto:ali97@gatech.edu)

Edward Zhou: [edz@gatech.edu](mailto:edz@gatech.edu)

Benjamin Thornbloom: [bthornbloom3@gatech.edu](mailto:bthornbloom3@gatech.edu)

Due: Dec 13, 2018

**Abstract**

The goal of the final project was to implement the two-dimensional fourier transform using three different methods: std::Thread library, MPI, and Cuda. We have also implemented the inverse discrete fourier transform to validate our results. Each method of implementation was uniquely written for optimized runtime based on the computing model. We compared the performance of each method against each other. This information is shown in the results section.

STD::Thread

The first implementation of the two-dimensional fourier transform was a multi-threaded approach using the std::thread library and running on a machine with eight cores. When the source code is compiled, it will produce an executable of the name “p31”. This method was implemented using an approach as suggested by the project description.

MPI

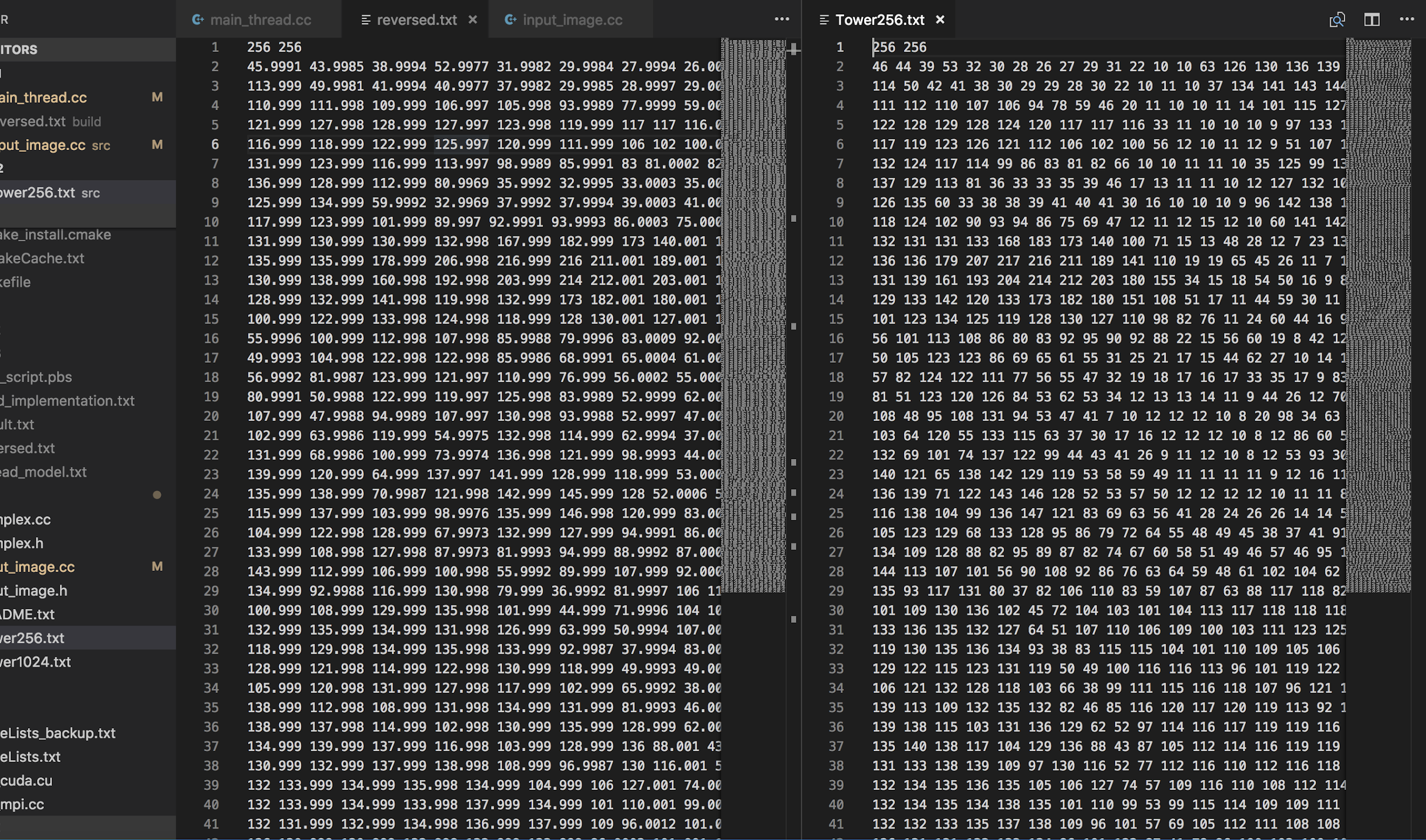
The second implementation of the two-dimensional fourier transform was an MPI approach running on a coc-ice MPI job using a pbs script with 8 MPI ranks. When the source code is compiled, it will produce an executable of the name “p32”. This method was implemented using the Danielson-Lanczos algorithm, an approach as suggested by the project description.

CUDA

The third implementation of the two-dimensional fourier transform was a GPU approach using CUDA and running on a coc-ice job queue. When the source code is compiled, it will produce an executable of the name “p33”. This method was implemented using an approach.

**Inverse Discrete Fourier Transform (Reverse DFT)**

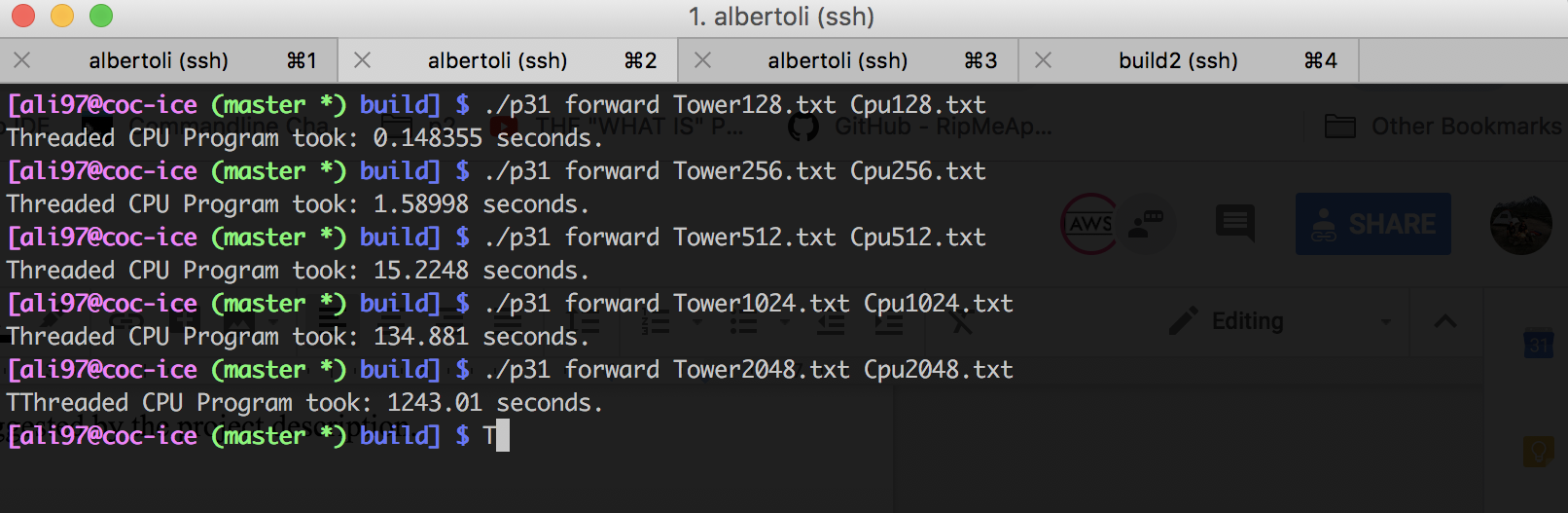
For the Thread computing model, we have implemented the inverse discrete fourier transform. Adding this functionality has allowed us to validate the accuracy of our forward DFT since IDFT(DFT(Image))) will reproduce the values of the original image. Below is a screenshot of the first few entries of the IDFT on Tower256.txt.



**Figure 1.** First few entries of the IDFT when running ./p31 reverse Tower256.txt reversed.txt

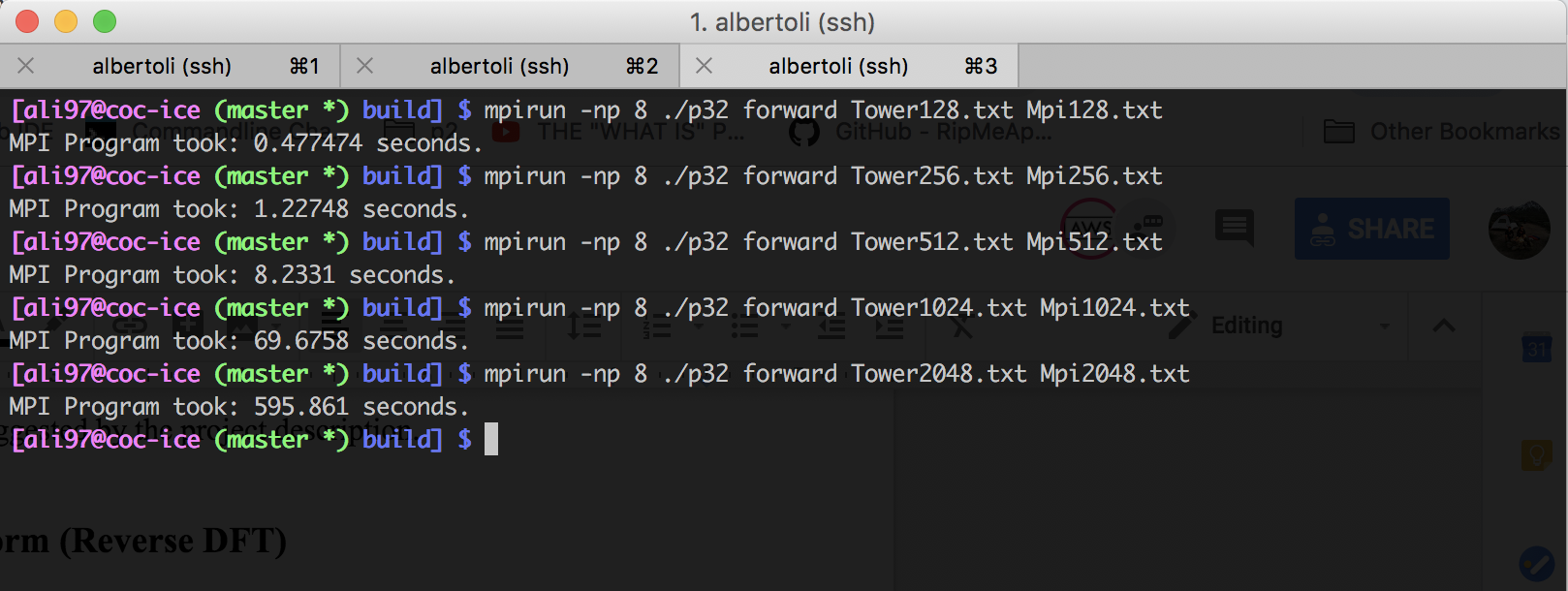
**Results**

The std::Thread, MPI, and CUDA computing methods were tested with text files of the following sizes: 128x128, 256x256, 512x512, 1024x1024, and 2048x2048. The execution times for each of the methods were timed. std::Thread and MPI were timed using std::chrono’s high resolution clock at a microsecond granularity and CUDA was timed by submitting pbs scripts and looking at the walltime due to nvcc’s bug with chrono.

std::Thread:

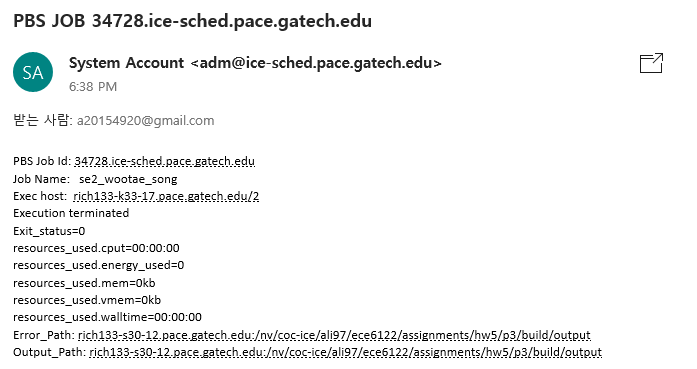
**Figure 2.** std::Thread’s timing for all the sizes

MPI:

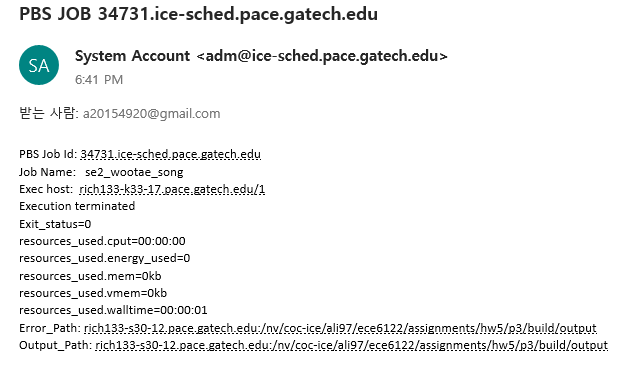


**Figure 3.** MPI’s timing for all the sizes

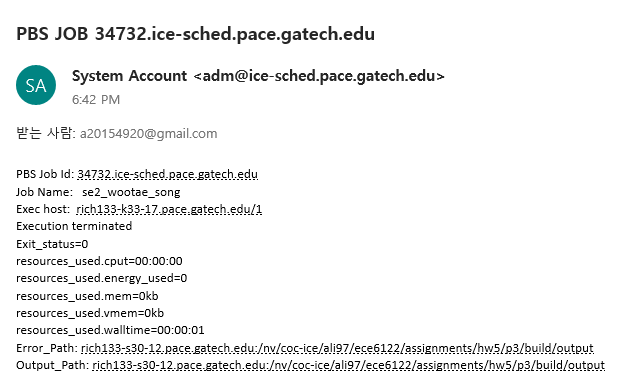
CUDA:



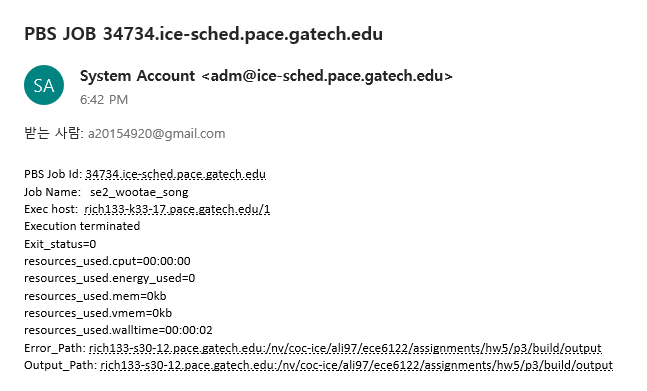
**Figure 4.** CUDA’s timing for 128x128



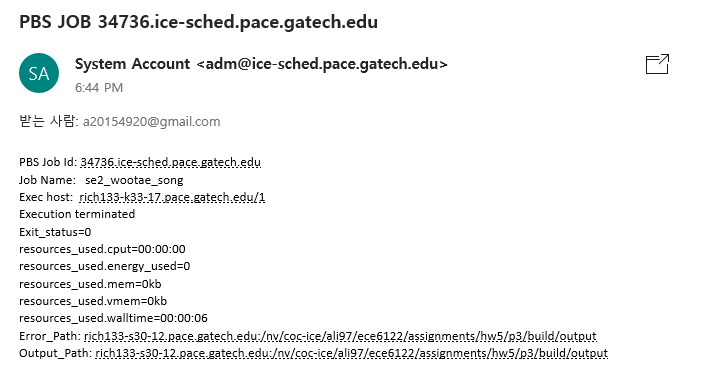
**Figure 5.** CUDA’s timing for 256x256



**Figure 6.** CUDA’s timing for 512x512

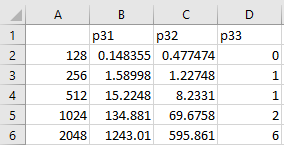


**Figure 7.** CUDA’s timing for 1024x1024



**Figure 8.** CUDA’s timing for 2048x2048

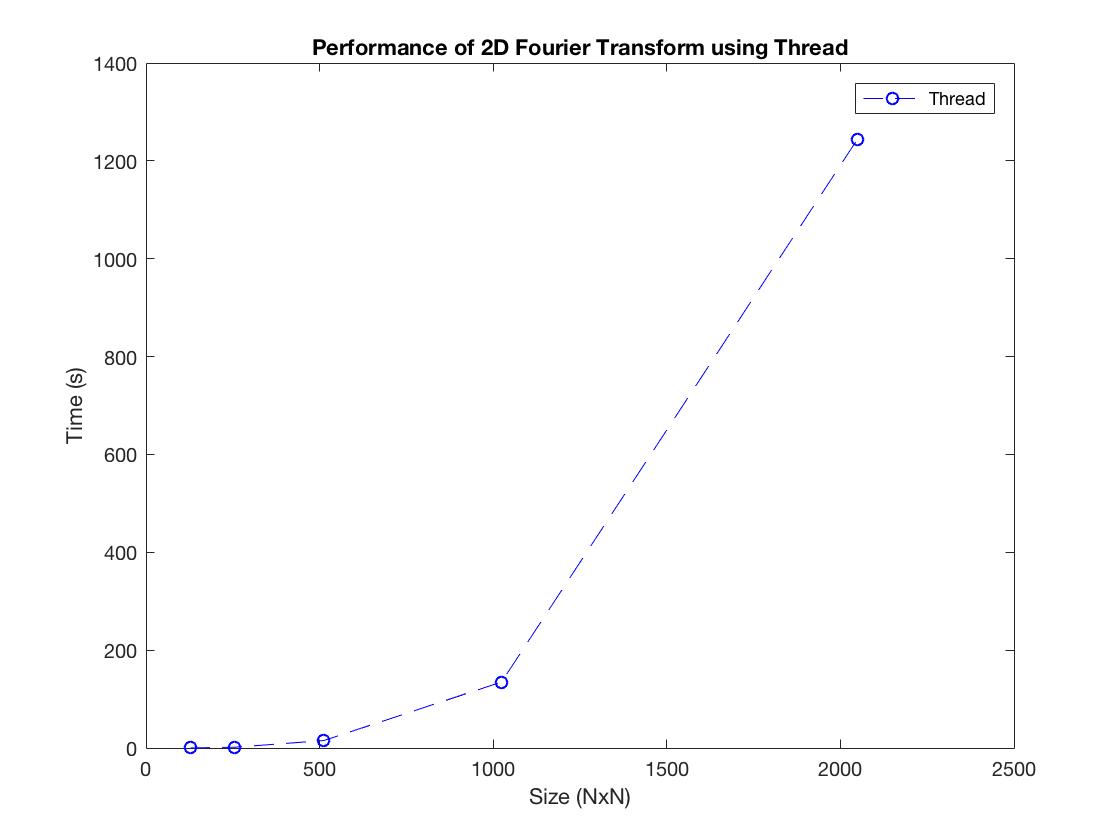
The results of each DFT computing method were compared against the results of the corresponding sizes (128x128, 256x256, ...,2048x2048) and the results of each matched. The table below summarizes the results:



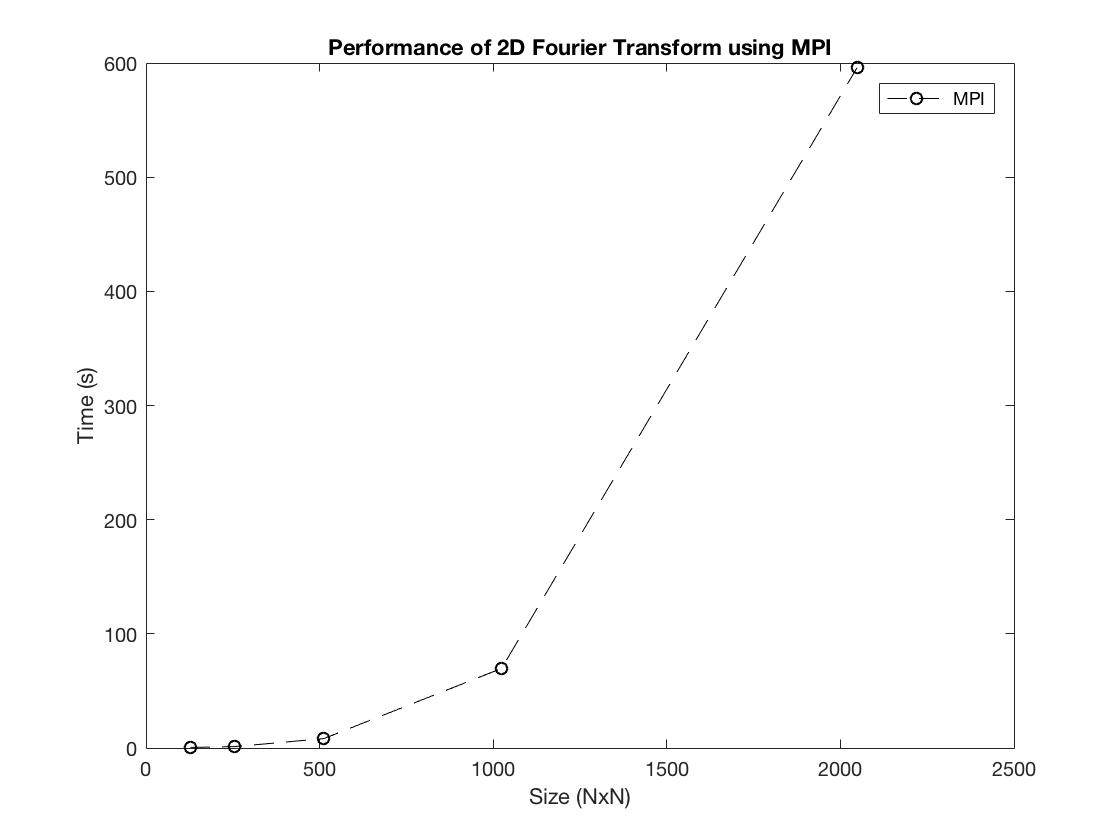
**Figure 9.** Results matrix in seconds.

As expected, CUDA took significantly less time than the other methods and std::Thread took the most time among all the methods. For the 2048x2048 DFT, CUDA only took 6 seconds to run while std::Thread took around 21 minutes which was 207 times longer. Additionally for the 2048x2048 DFT, CUDA only took 6 seconds to run while MPI took around 10 minutes which was 99 times longer. Lastly for the 2048x2048 DFT, MPI only took 10 minutes to run while CUDA took around 21 minutes which was 2 times longer. As N increases, the difference in execution time of each of the computing methods will increase drastically as std::Thread is O(), MPI is O(), and CUDA is O(1).

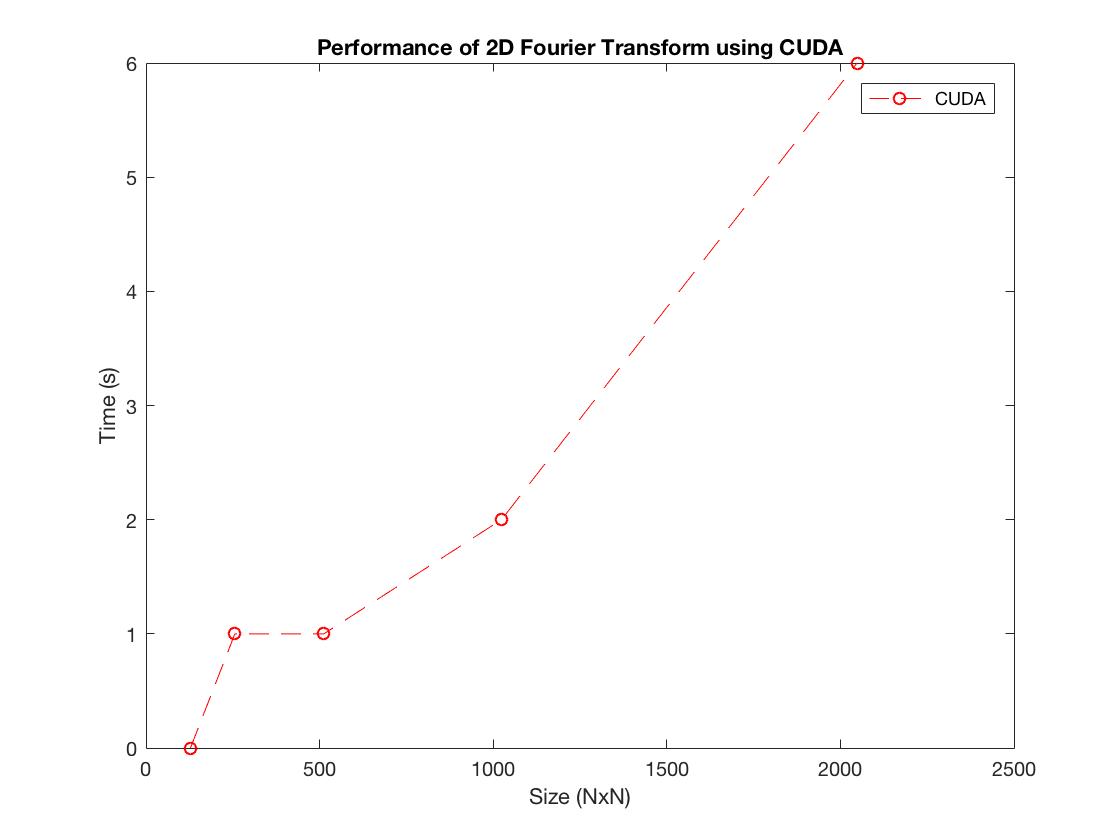
These performance comparisons are visualized separately (Figures. 10, 11, 12) and together (Figure. 13). By comparing the runtime results of the three different DFT implementations, the following graphs below were produced.



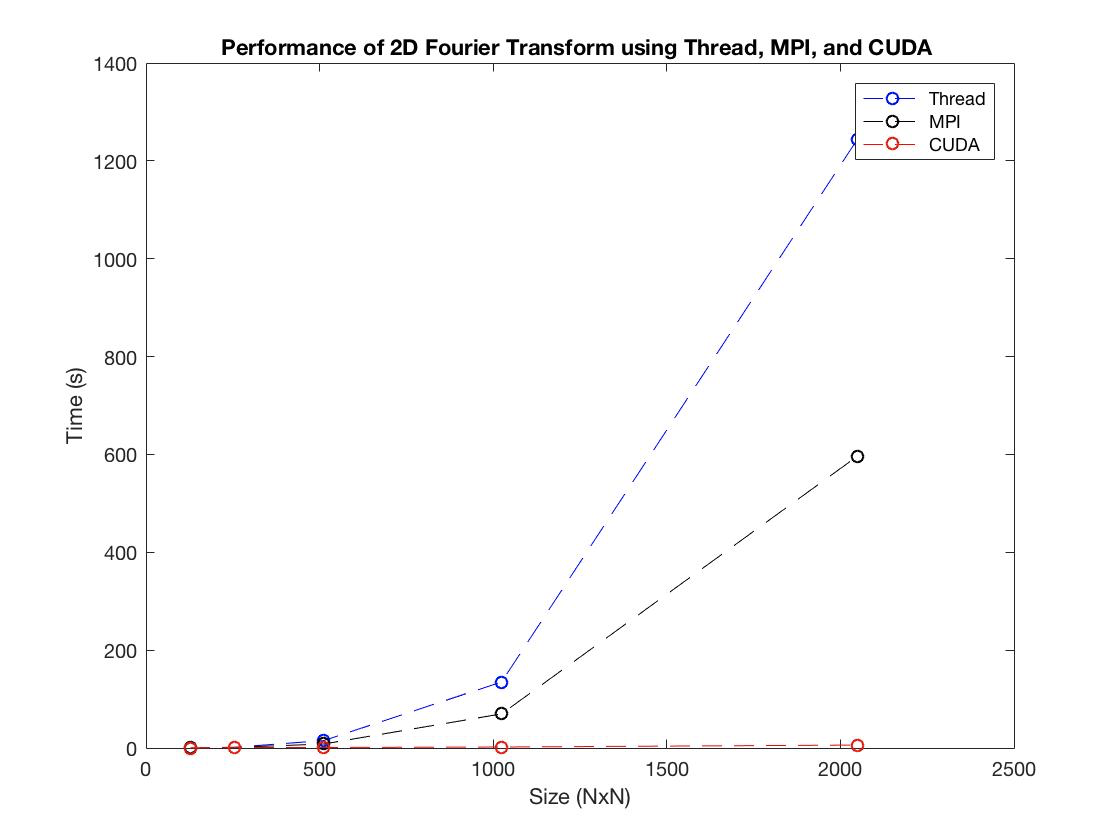
**Figure 10.** Time (s) vs Size (NxN) for std::Thread



**Figure 11.** Time (s) vs Size (NxN) for MPI



**Figure 12.** Time (s) vs Size (NxN) for CUDA



**Figure 13.** Time (s) vs Size (NxN) for std::Thread, MPI, and CUDA

In conclusion, the results of our three unique implementations based on the programming models: std::Thread, MPI, and CUDA suggest that GPU computation was the most efficient, MPI was the second most efficient, and std::Thread was the least efficient.