Prog2 Writeup
Steven Kravitz

During the prog2 assignment, I made 4 versions of the fast fourier transformation in order to optimize it. They were: a version with reduced code motion, a version with reduced memory referencing, a version with 2x1 loop unrolling, and a version with 2x2 loop unrolling. I also modified the Shell script to print out headers for each of the versions. Lastly, I made a makefile that compiles the code in DEBUG mode and with N set to 3 to allow for easy verification of the validity of the results. To see these results just enter 'make' and './prog2' into the console.

The first optimization was in the function 'fft_codeMotion' and involved reducing code motion. Firstly, I precalculated the values of 2 * PI in a variable called 'twoPI' and I precalculated sqrt(width*height) in a variable called 'hypotenuse'. Next I simplified the function by precalculating the values inside of the trigonometric functions in variables called 'xValue' and 'yValue.' These variables are iterated with 'xIterator' and 'yIterator' to ensure that the value from the trig functions is always identical to the original function. The final optimization in this function was pulling the calculation amplitudeOut out of the two innermost for loops because this value only needs to be calculated with the final values of realOut and imagOut. This optimization had a massive improvement on performance compared to the original.

The next optimization was called 'fft_memRef' and improved off of 'fft_codeMotion' by reducing memory references. The first step was by saving inputData[ySpace][xSpace] in a variable called 'inputDataVar' so that this array did not need to be called from memory every time this value was needed. The second optimization was in using accumulators called 'realOutSum' and 'imagOutSum' to store the values of realOut and imagOut respectively. This reduces the number of times that the realOut and imagOut arrays need to be accessed. This optimization had a minor improvement on performance due to the fact that the performance was already really good.

The next optimization was called 'fft_2x1' and improved off of 'fft_memRef' by applying 2x1 unrolling to the function. In this function, the innermost loop is modified so that it increments twice for each iteration. As such, inside the loop, the calculations are done for both xValue and xValue+1. After this for loop, there is another loop that calculates the remaining one or two iterations (depending on the parity of the length of N) that were not completed. This optimization had a negligible impact on performance because the function added overhead with the second for loop and also still has sequential dependency.

The final optimization was called 'fft_2x2' and improved off 'fft_2x1' by modifying the unrolling to be 2x2 as opposed to 2x1. The main difference is that this function used two separate accumulators for both realOut and imagOut instead of just one. This optimization also had a negligible impact on performance because the function still has sequential dependency and added overhead from loop unrolling.

Overall, this project demonstrated the impact of performance optimizations and why it is so important to write efficient code.