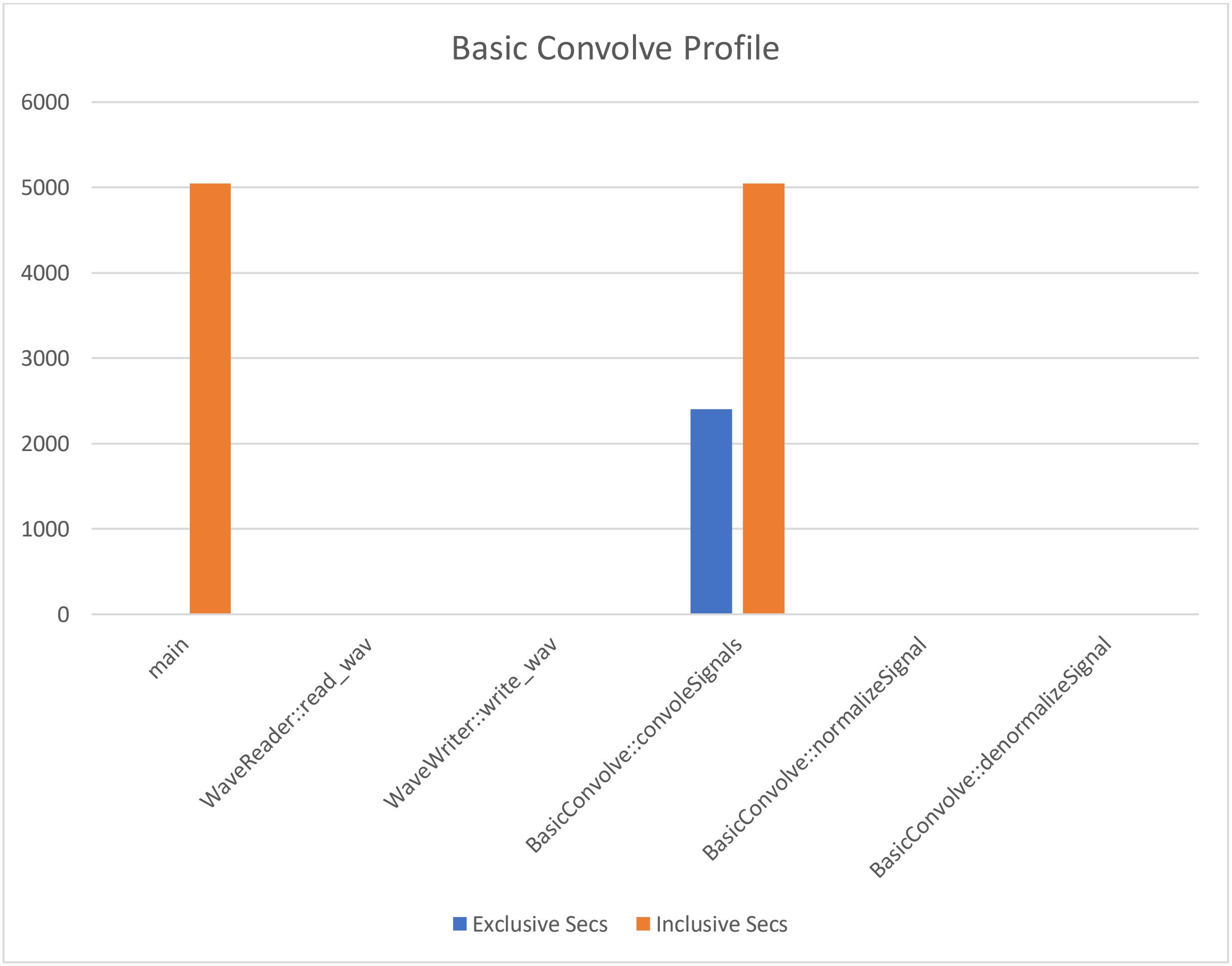
CPSC501 Assignment 4 Report

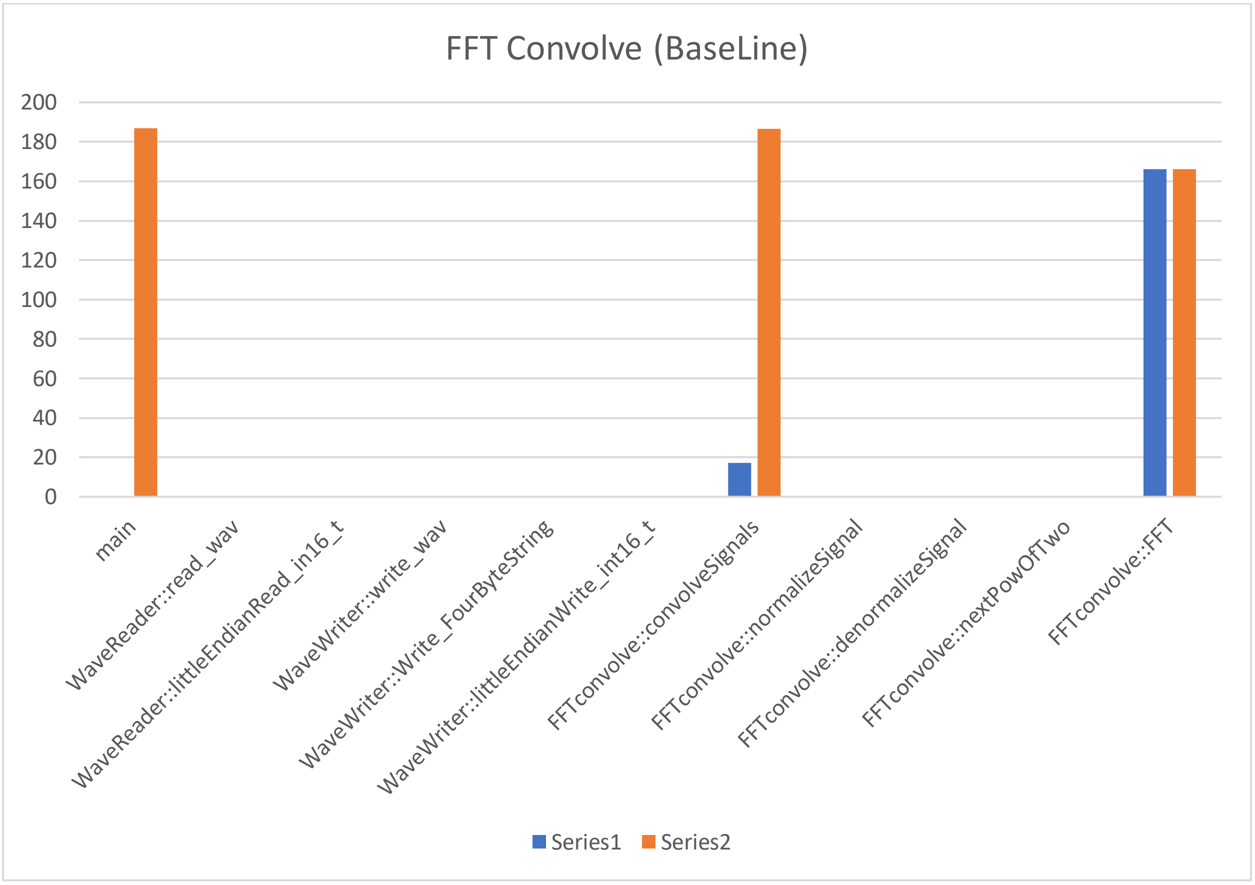
Brett Gattinger (30009390)

Introduction:

Immediately below is the profile of the Basic convolve program. Each function in the program is profiled by exclusive and inclusive seconds. The Exclusive seconds for each function are the time spent during the execution of that function, but not of any of its callees. The Inclusive seconds for each function are the time spent during the execution of that function, including any other functions called from it (and their callees, and so on). For example, the entire runtime of the program is capture in the inclusive measurement of the main function, as this time includes the time spent in the main function plus the time spent in all other functions it calls. We can see here that the basic convolve program took around 5000 seconds (or about 83 minutes) with most of that time being spent in the convolveSignals function. This makes sense since, as the function name suggests, this is where the expensive signal convolution takes place. If you're wondering where the other seconds are (as the other functions visible here clearly can not possibly account for the remaining time), these are spent in a myriad of background functions called from the standard c/c++ library as part of the programs functioning. For simplicity we will ignore those functions and focus only on the ones I have written and optimized for this assignment.



In the next chart we see the profile of the FFT convolution program. Again, most of the time is spent in the function responsible for convolving the input signal and impulse response, however, unlike the basic convolution program which took over an hour to run the FFT convolution program took only 180 seconds (or 3 minutes). This showcases the power of Algorithm-based optimization as this is more than a 25-fold increase in speed. But can we do better? Probably not but let's try anyway.



***Important note: Each of the following optimizations are successively applied to the FFT convolution program and implemented one on top of the other. That is, optimization 2 is applied to the program with optimization 1 already applied, optimization 3 is applied to the program with optimization 1 and 2 already applied and so on. I realize at the time of writing this report that the assignment may have called for analysis of these optimizations to be performed in isolation, but it is too late to change it now.

Optimization 1:

Optimization 1 involves the precomputing and caching of calculations repeatedly used by the FFT algorithm to transform time-domain signals into their frequency-domain counterparts.

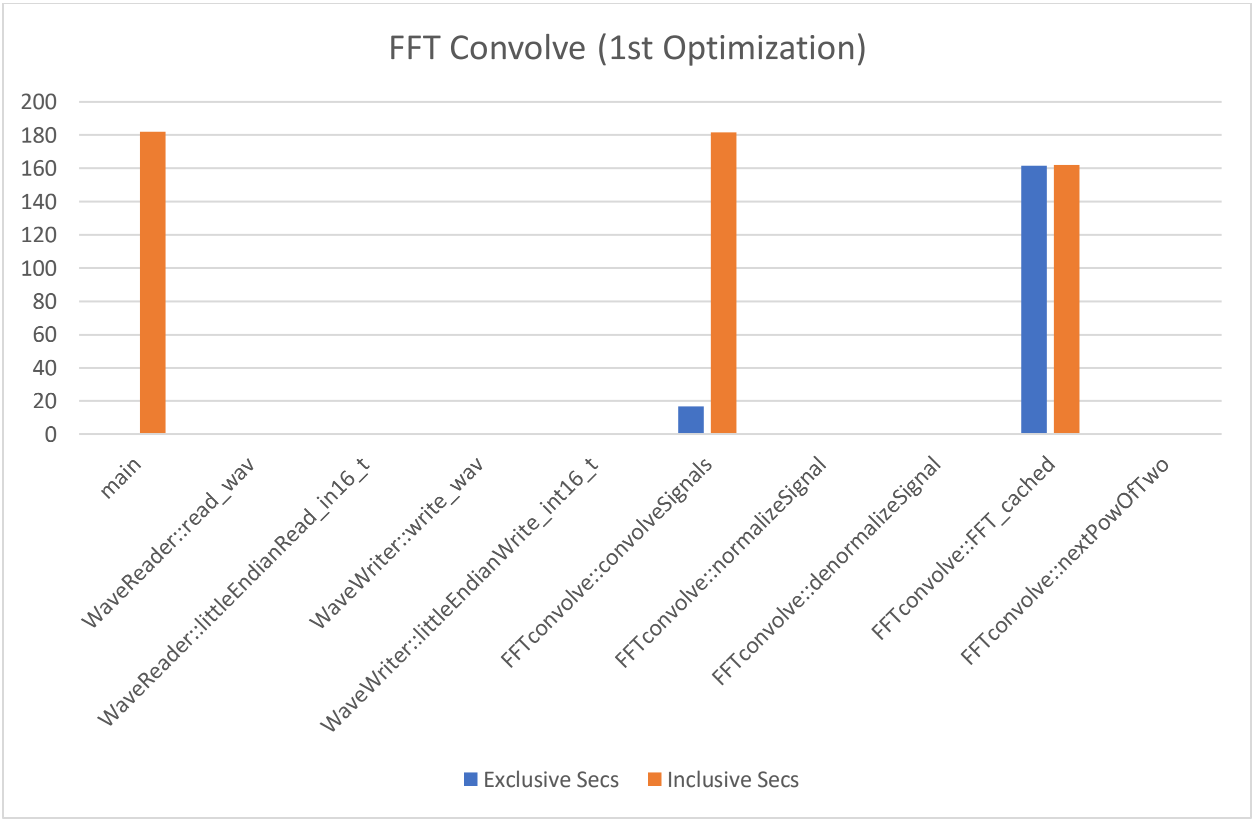
```
    mmax = 2;
    while (n > mmax) {
    istep = mmax << 1;
    theta = isign * (6.28318530717959 / mmax);
    wtemp = sin(0.5 * theta);
    wpr = -2.0 * wtemp * wtemp;
    wpi = sin(theta);
    wr = 1.0;
    wi = 0.0;
```

above is the block of code inside the FFT algorithm which contains the calculations in question, specially wtemp, wpr, and wpi. Because I was using the overlap-add method for my FFT convolution program these series of values are being recalculated again and again each time I fed the FFT algorithm the next segment of the input signal. That is, because these calculations were based on the length of the segments being fed in and the length of the segments never changed, the FFT algorithm was calculating the same sets of values for these variables each time it was called to transform an input signal segment. To optimize this, I created this function here:

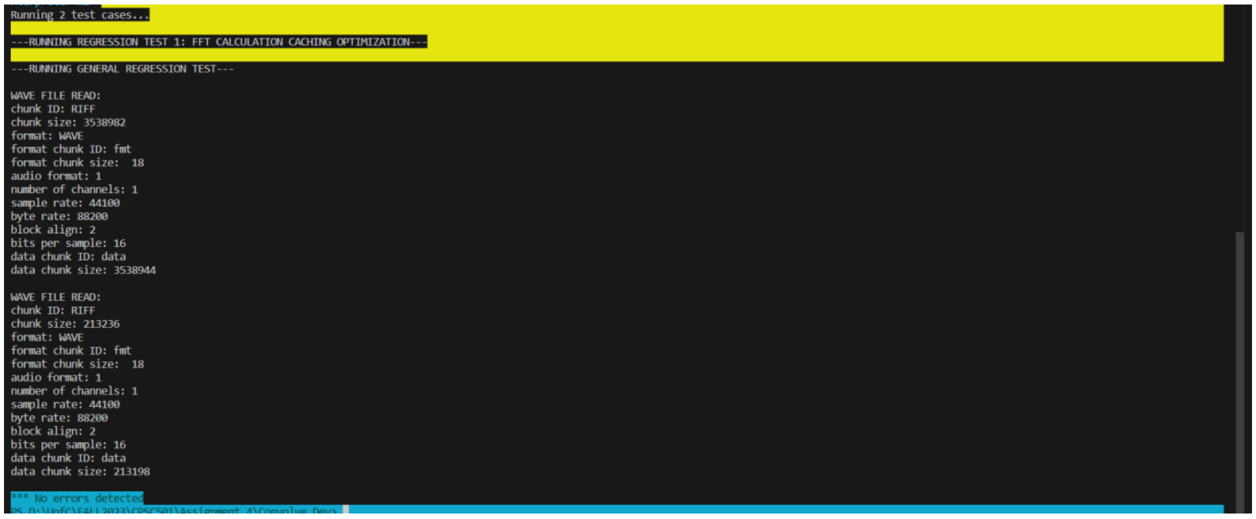
```
void FFTconvolve::initialize_cachedFFTSegmentCalculations(size_t
expectedConvolvedArraySegmentSize) {
    unsigned long n, mmax, istep;
    double theta, wtemp, wpr, wpi;
    n = (nextPowOfTwo(expectedConvolvedArraySegmentSize) << 1);
    mmax = 2;
    while (mmax < n) {
        istep = mmax << 1;
        theta = (1) * (6.28318530717959 / mmax);
        wtemp = sin(0.5 * theta);
        wpr = -2.0 * wtemp * wtemp;
        wpi = sin(theta);
        cachedFFTSegmentCalculations.push_back(
            std::tuple<double, double, double>(wtemp, wpr, wpi)
        );

        mmax = istep;
    }
}
```

which is called once and computes and caches all the calculations necessary for the FFT algorithm to operate on any one segment fed in. I then modified the original block of code in the FFT algorithm to simply retrieve these values when needed:

```
    mmax = 2;
    size_t cacheIndex = 0;
    while (n > mmax) {
        istep = mmax << 1;

        if (isign == 1) {
            wtemp = std::get<0>(cachedFFTSegmentCalculations[cacheIndex]);
            wpr = std::get<1>(cachedFFTSegmentCalculations[cacheIndex]);
            wpi = std::get<2>(cachedFFTSegmentCalculations[cacheIndex]);
        } else if (isign == -1) {
            wtemp = std::get<0>(cachedIFFTSegmentCalculations[cacheIndex]);
            wpr = std::get<1>(cachedIFFTSegmentCalculations[cacheIndex]);
            wpi = std::get<2>(cachedIFFTSegmentCalculations[cacheIndex]);
        }
```
Here are the results of the optimization:



The difference is small, but if you compare these results to the baseline, we can see 4 second decrease in overall program execution time with that decrease being traced back to a 4-5 second decrease in execution time of the FFT algorithm (here renamed to FFT_cached). The regression tests for this optimization can be found in UnitAndRegressionTest.cpp of the final version of my program. Specifically, they are tests “fifth_test” and the “fileComparisonTest”. The first test calls the unoptimized and optimized version of the FFT algorithm on copies of the same array of doubles and checks to see if they return the same transformation. The second test simply runs the entire FFT convolution program with the optimization on the standardized input signal and impulse response and compares it’s out.wav to the comp.wav file generated by the baseline FFT convolution program to see if they are the same. Both regression tests were passed:



```
Running 2 test cases...

---RUNNING REGRESSION TEST 1: FFT CALCULATION CACHING OPTIMIZATION---

---RUNNING GENERAL REGRESSION TEST---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 3538982
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 3538944

WAVE FILE READ:
chunk ID: RIFF
chunk size: 213236
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 213198

*** No errors detected
PS D:\UofC\FALL2023\CPSC501\Assignment 4\Convolve Dev>
```

Optimization 2:

Optimization 2 is the partial jamming of two pairs of consecutive for-loops in the FFT convolution program responsible for initializing and loading the arrays that the FFT algorithm operates on with the input signal and impulse response data. These two pairs of for loops are shown below:

```
//apply FFT to impulse response to get frequency response
    size_t FR_FFT_arrayOpSize = 2 * nextPowOfTwo(expectedConvolvedArraySegmentSize);
    double *frequencyResponse = new double[FR_FFT_arrayOpSize]();
    for (size_t i = 0; i < FR_FFT_arrayOpSize; i++) {
        frequencyResponse[i] = 0;
    }
    for (size_t i = 0, j = 0; j < normalizedImpulseResponse.size(); i+=2, j++) {
        frequencyResponse[i] = normalizedImpulseResponse[j];
    }
```

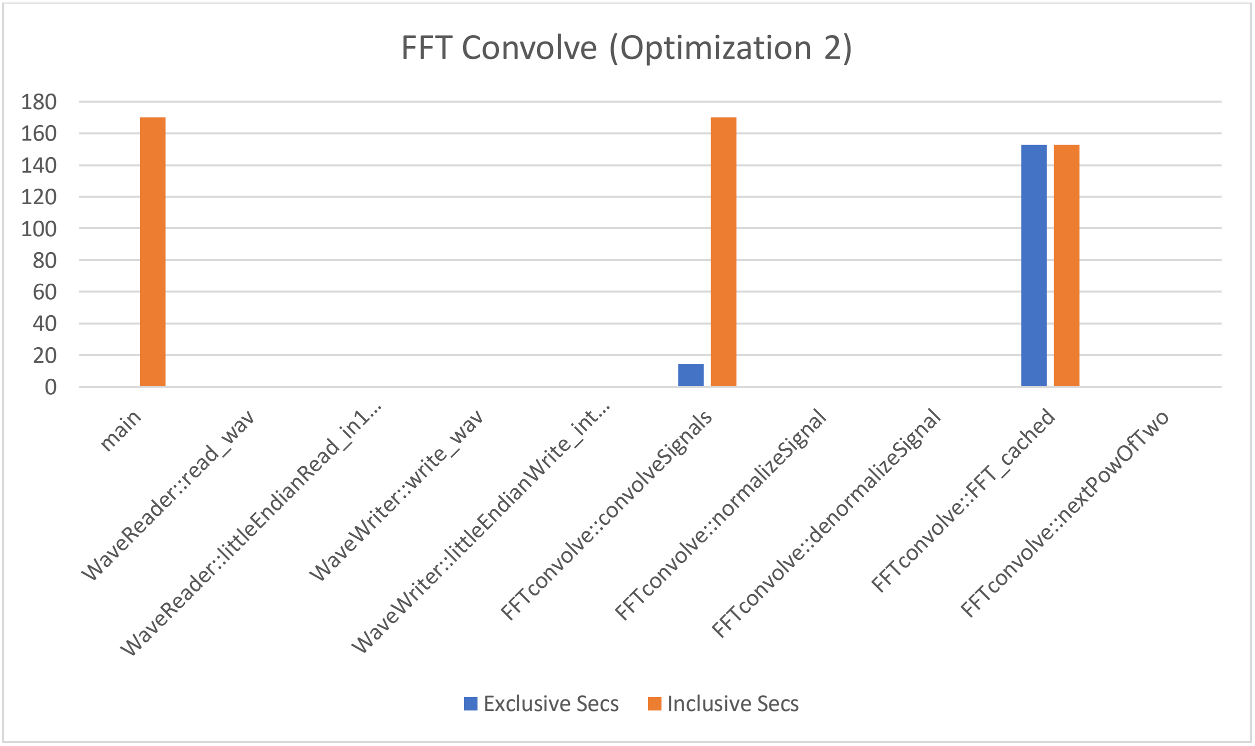
```
//load in next segment of input signal
        size_t ISS_FFT_arrayOpSize = 2 * nextPowOfTwo(expectedConvolvedArraySegmentSize);
        double *frequencySpectrumSegment = new double[ISS_FFT_arrayOpSize]();
        for (size_t j = 0; j < ISS_FFT_arrayOpSize; j++) {
            frequencySpectrumSegment[j] = 0;
        }
        for (size_t j = 0, k = 0; k < SEGMENT_SIZE; j+=2, k++) {
            frequencySpectrumSegment[j] = *segmentCursor;
            std::advance(segmentCursor, 1);
        }
```

The first pair loads the impulse response, the second pair is located inside the main for-loop of the FFTconvolve::convolveSignals function (you can see both these in the initial un-optimized version of my program). The first loop in both pairs 0-initializes the arrays, the second loop loads the signal data into every 2nd cell of the arrays starting at index 0. The inefficiency lies in the fact that between each pair of for-loops the first N cells of the arrays (with N being the size of the data being loaded) are visited twice. To optimize this inefficiency out I partially jammed the latter loop in both pairs into the former loop so that the first 2N cells are loaded with the signal data (alternating between 0-valued cells) and the remaining cells 0-initialized. This is what the first for-loop pair looks like after the optimization:

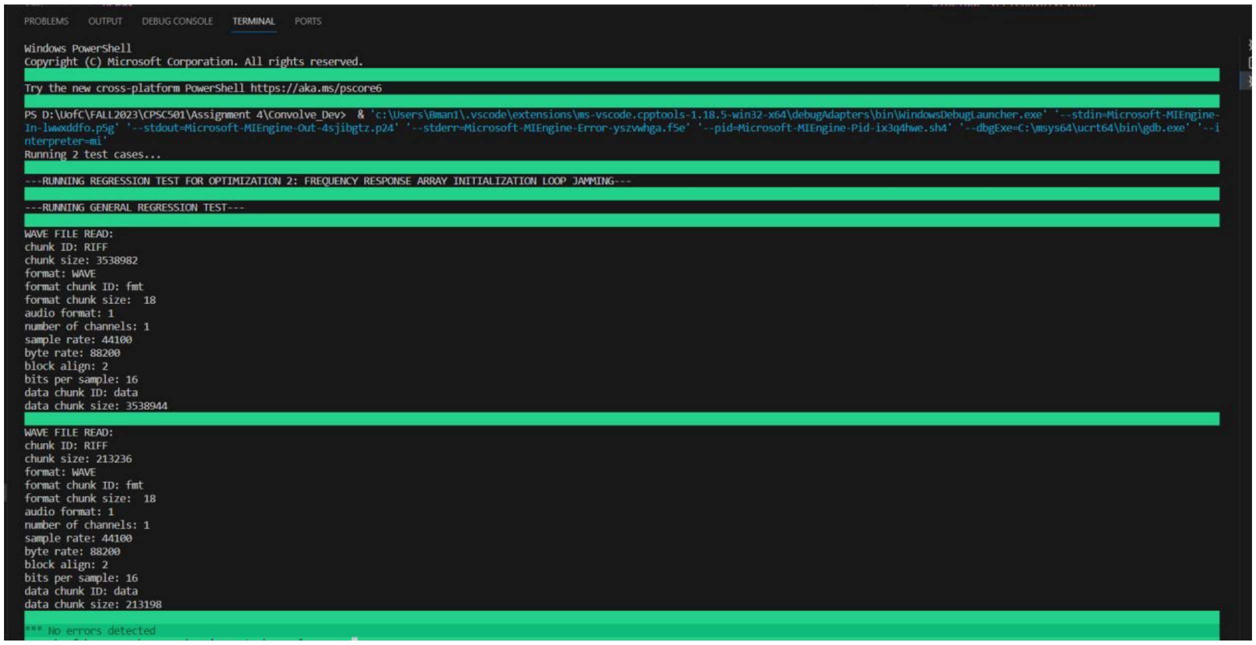
```
size_t ISS_FFT_arrayOpSize = 2 * nextPowOfTwo(expectedConvolvedArraySegmentSize);
        double *frequencySpectrumSegment = new double[ISS_FFT_arrayOpSize]();
        endOfLoadedDataIndex = 0;
        for (size_t j = 0, k = 0; k < SEGMENT_SIZE; j+=2, k++) {
            frequencySpectrumSegment[j] = *segmentCursor;
            frequencySpectrumSegment[j+1] = 0;
            endOfLoadedDataIndex = j+1;
            std::advance(segmentCursor, 1);
        }
        for (size_t j = endOfLoadedDataIndex; j < ISS_FFT_arrayOpSize; j++) {
            frequencySpectrumSegment[j] = 0;
        }
```

Two loops are still involved (hence why I call the jamming partial) but the total number of iterations is reduced from ArrayCapacity+SignalSize to just ArrayCapacity (note that the ArrayCapacity in my program is always > 2 * signal size, this is just a requirement of the FFT algorithm that works on these arrays).

Here is the result of the 2nd optimization:



Comparing these results to the previous optimization we can see a 12 second decrease in total program runtime with most of that time decrease coming from a reduction in the execution time of the FFTconvolve::convolveSignals function where the 2nd optimization took place, not bad. The regression tests for this optimization can be found in UnitAndRegressionTest.cpp of the final version of my program, the first of these is “sixth_test” which loads one array in the same manner as the unjammed loops before the optimization and then loads another in the partially jammed manner after the optimization and then checks to see that the arrays are indeed loaded with the same values in the same cells. This was an initial test just to see if my logic in implementing the optimization was sound. The second regression test is the same one I used to test the first optimization, “fileComparisonTest”. Both of these passed:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS D:\UofC\FALL2023\CPSC501\Assignment 4\Convolve_Dev>  & 'c:\Users\Bman1\.vscode\extensions\ms-vscode.cpptools-1.18.5-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-lwwxddfo.p5g' '--stdout=Microsoft-MIEngine-Out-4sjibgtz.p24' '--stderr=Microsoft-MIEngine-Error-yszwhga.f5e' '--pid=Microsoft-MIEngine-Pid-ix3q4hwe.sh4' '--dbgExe=C:\msys64\ucrt64\bin\gdb.exe' '--interpreter=mi'
Running 2 test cases...

---RUNNING REGRESSION TEST FOR OPTIMIZATION 2: FREQUENCY RESPONSE ARRAY INITIALIZATION LOOP JAMMING---

---RUNNING GENERAL REGRESSION TEST---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 3538982
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 3538944

WAVE FILE READ:
chunk ID: RIFF
chunk size: 213236
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 213198

*** No errors detected
```

Optimization 3:

Optimization 3 was simply the unrolling of the for-loops used in my programs data normalization and denormalization functions:

```
std::vector<double> FFTconvolve::normalizeSignal(std::vector<int16_t> & inSignal) {
    std::vector<double> normalizedSignal(inSignal.size());
    normalizedSignal.reserve(inSignal.size());
    for (int i = 0; i < inSignal.size(); i++) {
        double sample = inSignal[i];
        normalizedSignal[i] = ( double(2) * ( (sample - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
    }
    return normalizedSignal;
}
```

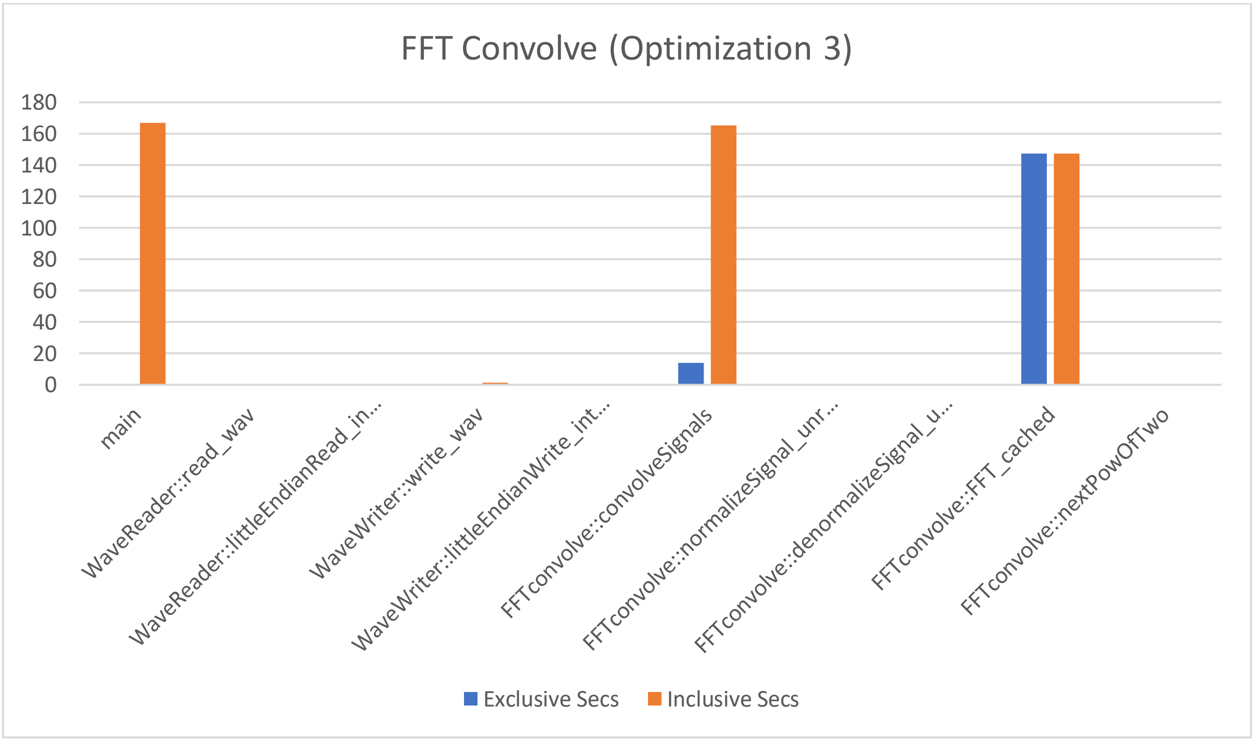
```
std::vector<int16_t> FFTconvolve::denormalizeSignal(std::vector<double> &
inSignal) {
    std::vector<int16_t> denormalizedSignal(inSignal.size());
    denormalizedSignal.reserve(inSignal.size());
    for (int i = 0; i < inSignal.size(); i++) {
        double normalizedSample = inSignal[i];
        denormalizedSignal[i] = int16_t( (normalizedSample - double(1)) *
(((double)SAMPLE_MAX - (double)SAMPLE_MIN) / double(-1-1)) ) +
(double)SAMPLE_MIN;
    }
    return denormalizedSignal;
}
```

Pretty plain and simple, I just unrolled each of these for-loops 3 times, Only the first unrolling is shown here as it naturally extends the code quite a bit:

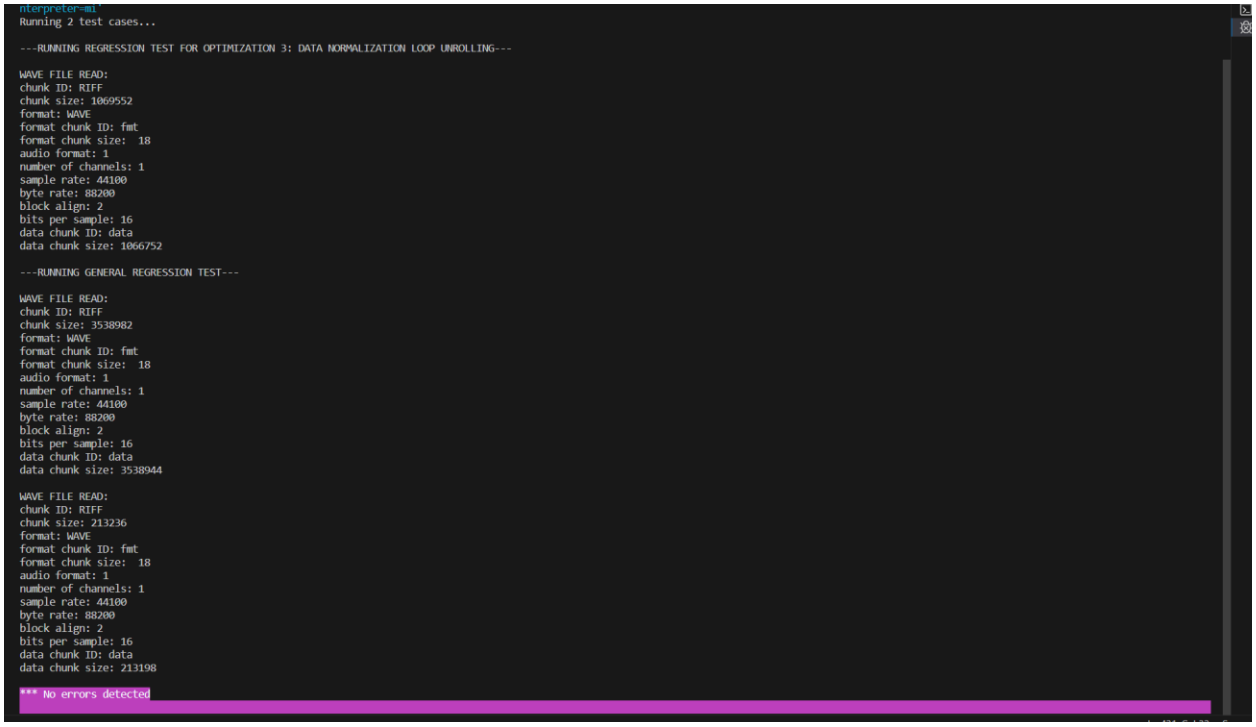
```
std::vector<double> FFTconvolve::normalizeSignal_unrolled(std::vector<int16_t> & inSignal) {
    size_t inSignalSize = inSignal.size();
    std::vector<double> normalizedSignal(inSignalSize);
    normalizedSignal.reserve(inSignalSize);
    size_t i;
    for (i = 0; i < inSignalSize-3; i+=4) {
        double sample1 = inSignal[i];
        double sample2 = inSignal[i+1];
        double sample3 = inSignal[i+2];
        double sample4 = inSignal[i+3];
        normalizedSignal[i] = ( double(2) * ( (sample1 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) -
double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[i+1] = ( double(2) * ( (sample2 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) -
double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[i+2] = ( double(2) * ( (sample3 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) -
double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[i+3] = ( double(2) * ( (sample4 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) -
double(SAMPLE_MIN)) ) ) - double(1);
    }

    if (i == inSignalSize - 3) {
        double sampleB = inSignal[inSignalSize - 3];
        double sampleC = inSignal[inSignalSize - 2];
        double sampleD = inSignal[inSignalSize - 1];
        normalizedSignal[inSignalSize-3] = ( double(2) * ( (sampleB - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[inSignalSize-2] = ( double(2) * ( (sampleC - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
```

```
        normalizedSignal[inSignalSize-1] = ( double(2) * ( (sampleD - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
    } else if (i == inSignalSize - 2) {
        double sampleC = inSignal[inSignalSize - 2];
        double sampleD = inSignal[inSignalSize - 1];
        normalizedSignal[inSignalSize-2] = ( double(2) * ( (sampleC - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[inSignalSize-1] = ( double(2) * ( (sampleD - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
    } else if (i == inSignalSize - 1) {
        double sampleD = inSignal[inSignalSize - 1];
        normalizedSignal[inSignalSize-1] = ( double(2) * ( (sampleD - double(SAMPLE_MIN)) /
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) ) ) - double(1);
    }
    return normalizedSignal;
}
```

Here are the results of that optimization:



Only a 4 second decrease in overall program runtime relative to the previous 2 optimizations combined with that decrease being in the FFTconvolve::convolveSignals function where the data normalization and denormalization are called. Not great, not terrible. The regression tests for this optimization can be found in UnitAndRegressionTest.cpp of the final version of my program, “seventh_test” and the “fileComparisonTest” again. “seventh_test” simply calls the rolled and unrolled versions of the data normalization functions on the same signals and compares their output to see if they are equivalent. Both regression tests were passed:



```
nterpreter=mi'
Running 2 test cases...

---RUNNING REGRESSION TEST FOR OPTIMIZATION 3: DATA NORMALIZATION LOOP UNROLLING---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 1069552
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 1066752

---RUNNING GENERAL REGRESSION TEST---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 3538982
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 3538944

WAVE FILE READ:
chunk ID: RIFF
chunk size: 213236
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 213198

*** No errors detected
```

Optimization 4:

Optimization 4 involved 2 manual code tunings, elimination of common subexpressions and elimination of runtime datatype conversions. These code tunings were motivated by the previous 3rd optimization (see the unrolled loop above) which introduced several lines of common subexpressions and datatype conversions for example:

```
    for (i = 0; i < inSignalSize-3; i+=4) {
        double sample1 = inSignal[i];
        double sample2 = inSignal[i+1];
        double sample3 = inSignal[i+2];
        double sample4 = inSignal[i+3];
        normalizedSignal[i] = ( double(2) * ( (sample1 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) - 
double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[i+1] = ( double(2) * ( (sample2 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) - 
double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[i+2] = ( double(2) * ( (sample3 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) - 
double(SAMPLE_MIN)) ) ) - double(1);
        normalizedSignal[i+3] = ( double(2) * ( (sample4 - double(SAMPLE_MIN)) / (double(SAMPLE_MAX) - 
double(SAMPLE_MIN)) ) ) - double(1);
    }
```

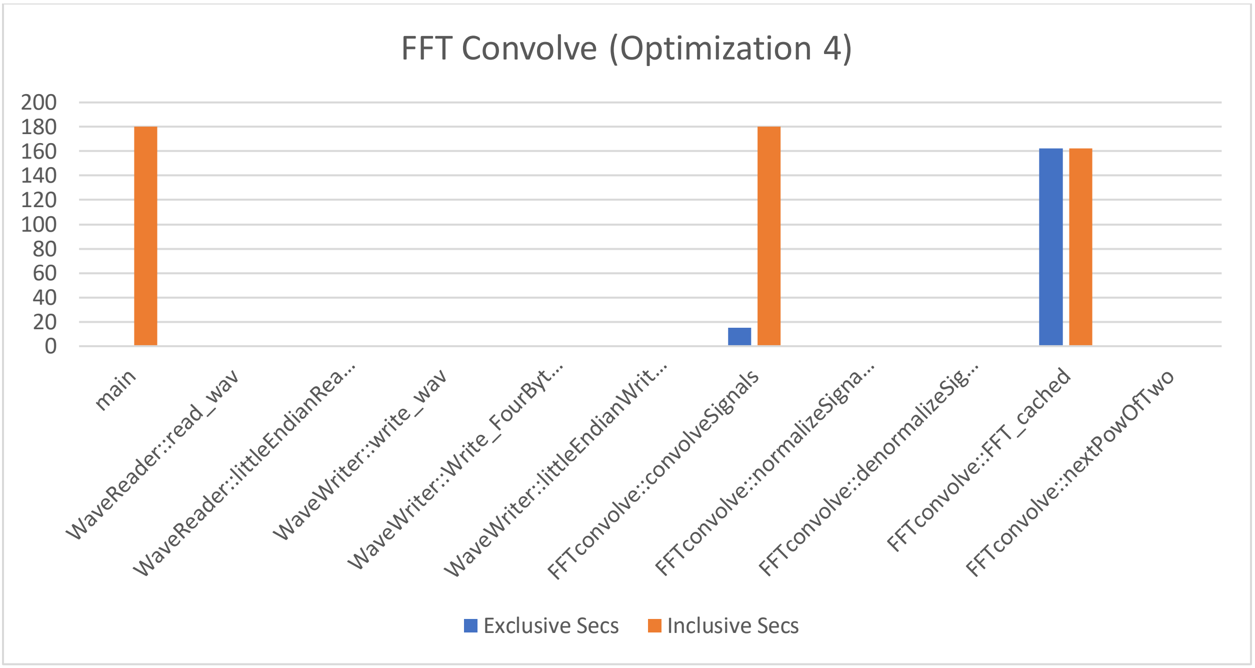
In just the first part of the unrolling's we see the expression:

```
(double(SAMPLE_MAX) - double(SAMPLE_MIN)) )
```

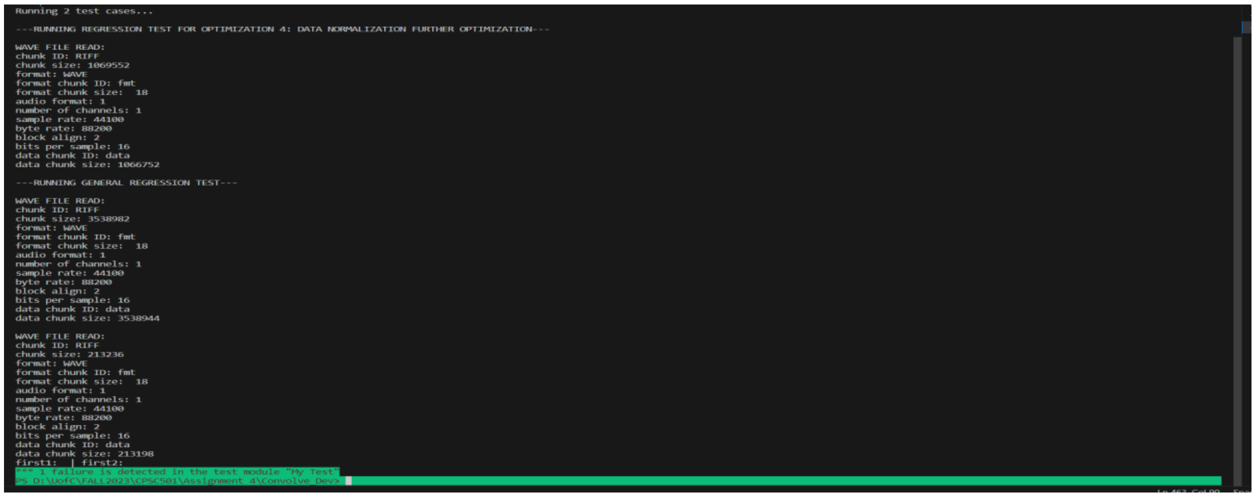
Repeated several times not to mention all those double() data type conversions. Optimization 4 eliminates these by declaring several new double constants in FFTconvolve.hpp and now the unrolled data normalization functions look like this:

```
std::vector<double> FFTconvolve::normalizeSignal_unrolled_optimized(std::vector<int16_t> & inSignal) {
    size_t inSignalSize = inSignal.size();
    std::vector<double> normalizedSignal(inSignalSize);
    normalizedSignal.reserve(inSignalSize);
    size_t i;
    for (i = 0; i < inSignalSize-3; i+=4) {
        normalizedSignal[i] = ( NORM_TWO * ( (inSignal[i] - SAMPLE_MIN_D) / NORM_DENOM ) ) - NORM_ONE;
        normalizedSignal[i+1] = ( NORM_TWO * ( (inSignal[i+1] - SAMPLE_MIN_D) / NORM_DENOM ) ) - NORM_ONE;
        normalizedSignal[i+2] = ( NORM_TWO * ( (inSignal[i+2] - SAMPLE_MIN_D) / NORM_DENOM ) ) - NORM_ONE;
        normalizedSignal[i+3] = ( NORM_TWO * ( (inSignal[i+3] - SAMPLE_MIN_D) / NORM_DENOM ) ) - NORM_ONE;
    }
    if (i == inSignalSize - 3) {
        normalizedSignal[inSignalSize-3] = ( NORM_TWO * ( (inSignal[inSignalSize - 3] - SAMPLE_MIN_D) /
NORM_DENOM ) ) - NORM_ONE;
        normalizedSignal[inSignalSize-2] = ( NORM_TWO * ( (inSignal[inSignalSize - 2] - SAMPLE_MIN_D) /
NORM_DENOM ) ) - NORM_ONE;
        normalizedSignal[inSignalSize-1] = ( NORM_TWO * ( (inSignal[inSignalSize - 1] - SAMPLE_MIN_D) /
NORM_DENOM ) ) - NORM_ONE;
    } else if (i == inSignalSize - 2) {
        normalizedSignal[inSignalSize-2] = ( NORM_TWO * ( (inSignal[inSignalSize - 2] - SAMPLE_MIN_D) /
NORM_DENOM ) ) - NORM_ONE;
        normalizedSignal[inSignalSize-1] = ( NORM_TWO * ( (inSignal[inSignalSize - 1] - SAMPLE_MIN_D) /
NORM_DENOM ) ) - NORM_ONE;
    } else if (i == inSignalSize - 1) {
        normalizedSignal[inSignalSize-1] = ( NORM_TWO * ( (inSignal[inSignalSize - 1] - SAMPLE_MIN_D) /
NORM_DENOM ) ) - NORM_ONE;
    }
    return normalizedSignal;
}
```

Unfortunately, the results of this optimization are counter to our intentions:



After applying this optimization to our program on top of the previous optimizations we actually gain 14 seconds in total program execution time. It can be more easily seen in "FFTConvovleProfileCapture_Opt4_DataNormalizationFurtherOptimization.csv" than in the graph here, but this increase in time seems to stem from the effect of the optimization on the FFTconvolve::denormalizeSignal() function which goes from an Exclusive runtime of 0.0169538 seconds to 0.0311141 (so its runtime essentially triples!). This might have something to do with the data conversion from doubles to int16_t's as that is what the function deals in. This might also explain why this optimization passes the regression test "eighth_test" but fails the "fileComparisonTest". To explain, the denormaliztion function takes in a vector of doubles and handles data only in doubles except when it needs to return a vector of int16_t's, this data conversion from double to integer results in a rounding down of data values, as such the integers the function returns are now 1 less than they used to be (regression test "eighth_test" accounts for this but the other regression test does not):


```
Running 2 test cases...

---RUNNING REGRESSION TEST FOR OPTIMIZATION 4: DATA NORMALIZATION FURTHER OPTIMIZATION---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 1069552
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 1066752

---RUNNING GENERAL REGRESSION TEST---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 3538982
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 3538944

WAVE FILE READ:
chunk ID: RIFF
chunk size: 213236
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 213198
first1:  | first2:
*** 1 failure is detected in the test module "My Test"
PS D:\UofC\FALL2023\CPSC501\Assignment 4\Convolve Dev>
```

Optimization 5:

Optimization 5 deals with the FFTconvolve::nextPowOfTwo function which takes in a number and returns the next power of two greater than that number.

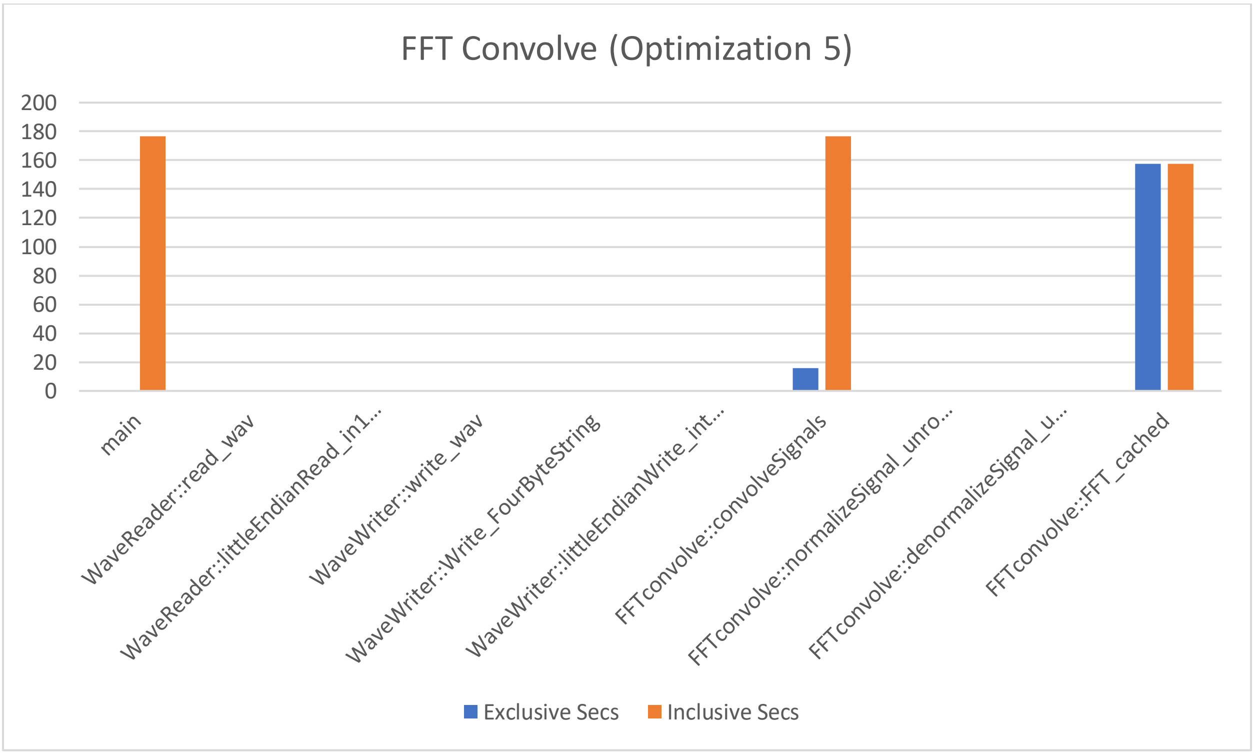
```
size_t FFTconvolve::nextPowOfTwo(size_t x) {
    return pow(2, ceil(log((x))/log(2)));
}
```

This function is used for calculating the required array size for arrays operated on by the FFT algorithm. Optimization 5 does two things: it makes this function inline and reduces the number of unnecessary calls made to it in the FFTconvolve::convolveSignals function. Before the optimization FFTconvolve::convolveSignals was calling the FFTconvolve::nextPowOfTwo function every time it called the FFT algorthim:

```
FFT_cached(frequencyResponse-1, nextPowOfTwo(expectedConvolvedArraySegmentSize),
1);
```

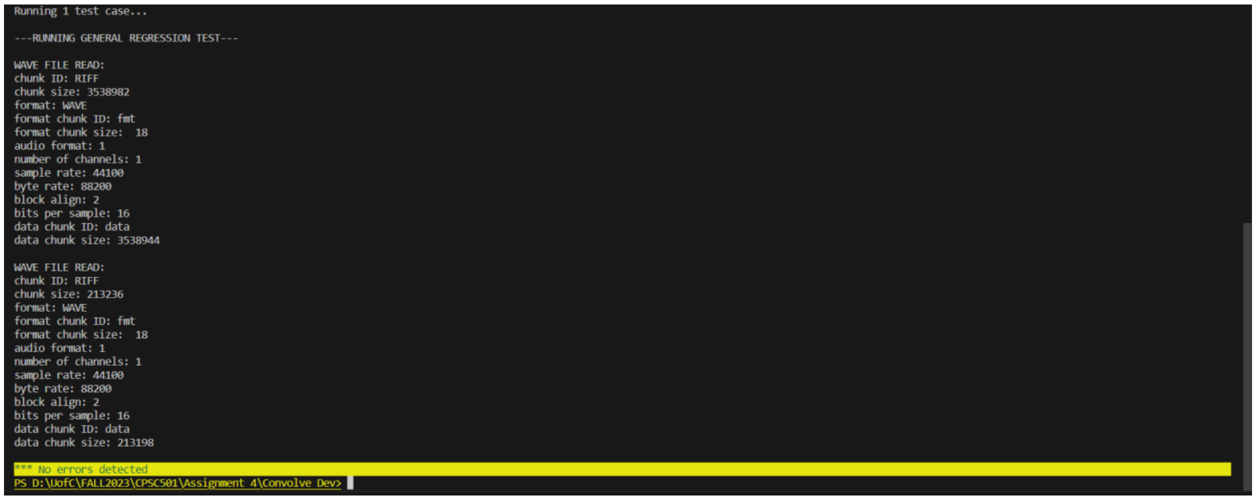
Basically, this line of code was being executed when the impulse response was being transformed and when each segment of the input signal was being transformed (which might happen thousands of time!)

Optimization 5 then is a combination of not only making the FFTconvolve::nextPowOfTwo function inline but more importantly reducing calls to it in order to minimize work inside the main for-loop of the FFTconvolve::convolveSignals function. Here is the result:



Now since FFTconvolve::nextPowOfTwo is inline it doesn't show up in the profiler but it did have a positive effect. Total overallprogram runtime was reduced by 4 seconds with those 4 seconds being shaved off in the FFTconvolve::convolveSignals function.

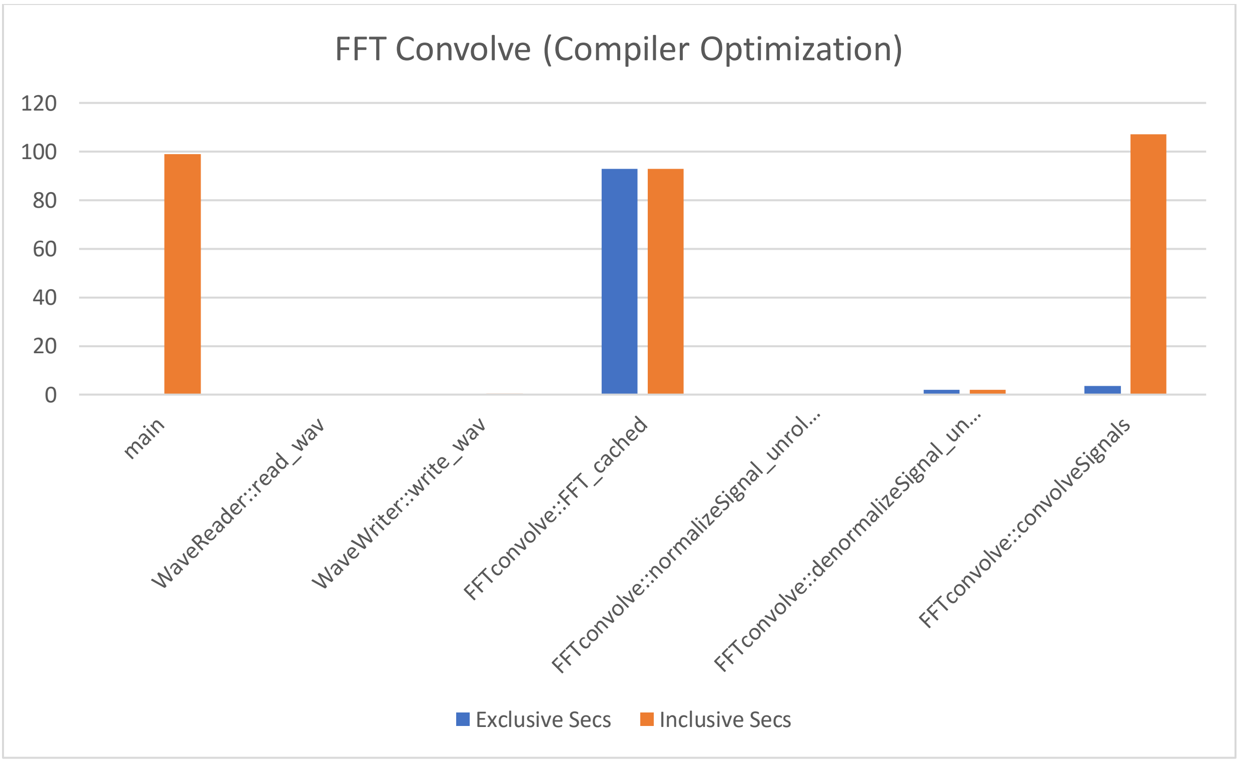
Since this was a small optimization the only regression testing used was “fileComparisonTest” which it passed



```
Running 1 test case...

---RUNNING GENERAL REGRESSION TEST---

WAVE FILE READ:
chunk ID: RIFF
chunk size: 3538982
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 3538944

WAVE FILE READ:
chunk ID: RIFF
chunk size: 213236
format: WAVE
format chunk ID: fmt
format chunk size:  18
audio format: 1
number of channels: 1
sample rate: 44100
byte rate: 88200
block align: 2
bits per sample: 16
data chunk ID: data
data chunk size: 213198

*** No errors detected
PS D:\UofC\FALL2023\CPSC501\Assignment 4\Convolve Dev>
```

Compiler Optimization:

The compiler optimization was executed on the final version of the FFT convolution program at level O2. Here are the results:



As we can see from the graph this is a huge improvement with the total program runtime coming in at under 100 seconds!