# CPSC 501 - Assignment 4, Optimization

Alex Stevenson - 30073617

GitHub Repository: https://github.com/alexs2112/CPSC501-Convolution

**Supporting Material**:

- Other versions of the program code are stored in the baseline branch and the fft branch of the repository.
- The electronic copy of the source code is provided in the `source.zip` file.
- Version control log reports are located in `git-log.txt` and `git-log-full.txt`
- Each section of this report will link to a corresponding commit to see the full code changes as part of that change. Only small code snippets of each change are pasted into this report.
- Profiler reports are located in the `profiling.zip` file.
- Unit tests are stored in the `test.cpp` file in the repository. This can be compiled using `make test` and run as `./test`. Other regression tests were done through manually using the `convolve` executable compiled from this source code.

## Baseline Program

- Initial version where convolution is implemented directly on the time domain in a linear matter.
- The code for this version is stored in the baseline branch of the repository.

**Run Time Performance**:

Length of `FluteDry.wav` : 60 seconds

Length of `tah_mahal.wav` : 3 seconds

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    21m49.301s
user    20m56.925s
sys     0m0.710s
>>> gprof convolve profiling/linear-flute.out
```

Length of `GuitarDry.wav` : 30 seconds

Length of `large_brite_hall.wav` : 2 seconds

- There are 2 channels for this wav file, however the baseline program still works by treating it as a single channel.

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    9m4.331s
user    9m0.709s
sys     0m0.154s
>>> gprof convolve profiling/linear-guitar.out
```

## Algorithm Based Optimization:

- Utilizing the FFT algorithm to re-implement the convolution using the frequency domain.
- The code for this version before any further optimizations is stored in the fft branch of the repository.
- As you can see with the base run time performance below, the algorithm based optimization increased speed of the program by 145 to 185 times.

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    0m9.000s
user    0m8.394s
sys     0m0.130s
>>> gprof convolve profiling/fft-flute.out
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    0m2.947s
user    0m2.562s
sys     0m0.075s
>>> gprof convolve profiling/fft-guitar.out
```

**Regression Testing**:

- Audio files produced from FFT convolution are the same as the ones produced by linear convolution.

## Manual Code Tuning #1:

- The `complex_multiply` function very consistently takes the most time as a function call, taking 60% of the total processing time of the program. This is twice as much as the next largest function.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 59.93      4.92     4.92                             complex_multiply(double*, double*, double*, int)
 30.21      7.40     2.48        2     1.24     1.24  fft_convolution(float*, int, float*, int, double*, int)
  8.04      8.06     0.66                             zero_padding(float*, int, double*, int)
```

- As this complex multiplication is done linearly on rather large inputs, it takes a long time. This can be multithreaded to do the full complex multiplication in parallel across many threads.
- This code tuning creates a new struct to be used as a single parameter, instead of a list of parameters. It then delegates segments of the input arrays across several threads. Each of those threads process their segments of arrays and enters their result into their segment of the output.

**Commit**: ff275d26132e407ef5ddb82fb4d211b12c2e9d62

**Code Changes**:

```
// Allow for 512 threads. This could realistically be larger as the input size is huge
#define COMPLEX_THREADS     512
// Input struct
struct complex_param {
    double *x;
    double *h;
    double *output;
    int size;
};
// Perform the complex multiplication on a delegated segment of input arrays
void *complex_multiply(void *v) {
    complex_param p = ((complex_param *)v)[0];
    for (int k = 0; k < p.size; k += 2) {
        p.output[k] = p.x[k] * p.h[k] - p.x[k+1] * p.h[k+1];
        p.output[k+1] = p.x[k+1] * p.h[k] + p.x[k] * p.h[k+1];
    }
    return 0;
}
// Break the input arrays into several chunks, give each thread a chunk to process
void multithread_multiply(double *x, double *h, double *output, int size) {
    pthread_t ids[COMPLEX_THREADS];
    int chunk = size / COMPLEX_THREADS;
    int i;
    for (i = 0; i < COMPLEX_THREADS; i++) {
        complex_param p;
        p.x = &x[i * chunk];
        p.h = &h[i * chunk];
        p.output = &output[i * chunk];
        p.size = chunk;
        pthread_create(&ids[i], NULL, complex_multiply, (void *)&p);
    }
    for (i = 0; i < COMPLEX_THREADS; i++) {
        pthread_join(ids[i], NULL);
    }
}
```

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    0m6.662s
user    0m5.949s
sys     0m0.192s
>>> gprof convolve profiling/flute-manual-1.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 94.22      5.38     5.38        3     1.79     1.79  four1(double*, int, int)
  1.75      5.48     0.10        2     0.05     0.05  zero_padding(float*, int, double*, int)
  1.75      5.58     0.10                             complex_multiply(void*)
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    0m2.812s
user    0m2.397s
sys     0m0.134s
>>> gprof convolve profiling/guitar-manual-1.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 93.30      2.09     2.09        3     0.70     0.70  four1(double*, int, int)
  3.12      2.16     0.07                             complex_multiply(void*)
  2.23      2.21     0.05        2     0.03     0.03  zero_padding(float*, int, double*, int)
```

- As you can see with the above profiling results, the new `complex_multiply` function takes a fraction of the time that it used to. 1.75-3% of the total program runtime instead of the previous result of 60%

**Regression Testing**:

- This change added an additional test for complex multiplication.
- This test also broke previous unit tests as multiplication does not happen if the number of threads is greater than the size of the input arrays. This has since been fixed.
- Output files from manual regression tests on convolution are the same as before the change.

## Manual Code Tuning #2:

- Zero padding and converting the input samples to complex arrays have been taking a relatively long time after previous optimizations. Some profiling taken from the result of the previous optimization:

```
>>> gprof convolve profiling/flute-manual-1.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 94.22      5.38     5.38        3     1.79     1.79  four1(double*, int, int)
  1.75      5.48     0.10        2     0.05     0.05  zero_padding(float*, int, double*, int)

>>> gprof convolve profiling/guitar-manual-1.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 93.30      2.09     2.09        3     0.70     0.70  four1(double*, int, int)
  3.12      2.16     0.07                             complex_multiply(void*)
  2.23      2.21     0.05        2     0.03     0.03  zero_padding(float*, int, double*, int)
```

- This is because the `zero_padding` function iterates over the `output` array twice, first it zeroes the entire array, and then it copies values from the input array to it.
- This can be optimized by only going over the array once by zeroing the imaginary part of the array at the same time as copying values from the input array.
    - *Note*: If the output array size is greater than twice the length of the input array then the remainder will still need to be zeroed. This is easy to accomplish by starting another `for` loop at the index of `2 * input_length + 1`.

**Commit**: 95c75c13fa76d35a41940a4d9794334dfae4f208

**Code Changes**:

```
void zero_padding(float *signal, int input_size, double *output, int output_size) {
    int i;
    for (i = 0; i < input_size; i++) {
        output[i*2] = (double)signal[i];
        output[i*2 + 1] = 0.0;
    }
    for (i = input_size * 2 + 1; i < output_size; i++) {
        output[i] = 0.0;
    }
}
```

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    0m6.031s
user    0m5.411s
sys     0m0.215s
>>> gprof convolve profiling/flute-manual-2.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 92.47      4.79     4.79        3     1.60     1.60  four1(double*, int, int)
  2.70      4.93     0.14                             complex_multiply(void*)
  2.32      5.05     0.12        2     0.06     0.06  zero_padding(float*, int, double*, int)
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    0m2.611s
user    0m2.244s
sys     0m0.139s
>>> gprof convolve profiling/guitar-manual-2.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 90.82      1.88     1.88        3     0.63     0.63  four1(double*, int, int)
  4.83      1.98     0.10                             complex_multiply(void*)
  1.93      2.02     0.04        2     0.02     0.02  zero_padding(float*, int, double*, int)
```

- The time it takes for the program to complete has dropped by a reasonable amount.
- The relative time it takes for `zero_padding` to finish compared to the other functions of the program has also been dropped by a reasonable amount.

**Regression Testing**:

- A minor bug with the `for` loop values was caught by the existing unit tests.
- Manual convolution testing is successful and has expected results.

## Manual Code Tuning #3:

- The `four1` algorithm as given to us uses doubles as its data type of choice. We don't need that level of precision for these simple convolutions and can change them all to floats.
- As floats are half the size of doubles, this will drastically speed up operations that involve the various double arrays that are prevalent in the code.

**Commit**: fdd35509d66e612d2a9cdd8c27d5902c155f592a

**Code Changes**:

- The main code change is present in the `four1` method, although there are many other places where doubles are changed to floats for performance.
- Consult the github commit to see the full list of changes.

```
void four1(float *data, int nn, int isign) {
    unsigned long n, mmax, m, j, istep, i;
    float wtemp, wr, wpr, wpi, wi, theta;
    float tempr, tempi;
    ...
}
```

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    0m5.100s
user    0m4.475s
sys     0m0.150s
>>> gprof convolve profiling/flute-manual-3.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 92.60      3.88     3.88        3     1.29     1.29  four1(float*, int, int)
  1.91      3.96     0.08                             complex_multiply(void*)
  1.67      4.03     0.07        2     0.04     0.04  zero_padding(float*, int, float*, int)
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    0m2.133s
user    0m1.700s
sys     0m0.110s
>>> gprof convolve profiling/guitar-manual-3.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 88.24      1.35     1.35        3     0.45     0.45  four1(float*, int, int)
  5.88      1.44     0.09                             complex_multiply(void*)
  1.96      1.47     0.03        2     0.01     0.01  zero_padding(float*, int, float*, int)
```

- The number of seconds per call of the `four1` function has been drastically reduced. For the `FluteDry` convolution, this has been reduced from 1.6 seconds per call to 1.29 seconds per call. For the `GuitarDry` convolution, this has been reduced from 0.63 seconds per call to 0.45 seconds per call.
- Changing doubles to floats has decreased the total run speed by 1/6th. That is almost a 20% increase in speed.
- This is pretty clearly a huge performance boost

**Regression Testing**:

- There was mild concern that changing from doubles to floats would cause incorrect outputs as there is a loss of 4 bytes of precision.
- This is not the case, all of the unit tests pass and manually running the convolution code using the new `four1` function produces the same result.

## Manual Code Tuning #4:

- The `zero_padding` function performs repeated multiplications by 2 to get indices of the output array.
- Strength Reduction can be applied to instead add by a fixed value every iteration instead of multiplying.

**Commit**: ed721fdee8e7511158f8b1c2820ba7afa9458e8d

**Code Changes**:

```
void zero_padding(float *signal, int input_size, float *output, int output_size) {
    int i, j;
    for (i = 0, j = 0; i < input_size; i++, j += 2) {
        output[j] = signal[i];
        output[j + 1] = 0.0;
    }
    ...
}
```

- Previously, the index applied to output was `2 * i`
- Strength Reduction is applied to instead increment a second value of `j` by 2 every loop iteration. This can be directly applied as the index of output instead of needing to perform a multiplication.

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    0m4.874s
user    0m4.345s
sys     0m0.121s
>>> gprof convolve profiling/flute-manual-4.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 94.10      3.83     3.83        3     1.28     1.28  four1(float*, int, int)
  2.70      3.94     0.11                             complex_multiply(void*)
  1.47      4.00     0.06        2     0.03     0.03  zero_padding(float*, int, float*, int)
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    0m2.059s
user    0m1.695s
sys     0m0.094s
>>> gprof convolve profiling/guitar-manual-4.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 89.61      1.38     1.38        3     0.46     0.46  four1(float*, int, int)
  5.19      1.46     0.08                             complex_multiply(void*)
  1.30      1.48     0.02        2     0.01     0.01  zero_padding(float*, int, float*, int)
```

- Note that the amount of time required by the `zero_padding` function is reduced in both cases.

**Regression Testing**:

- Previous automated unit tests continue to pass.
- Manual running of the convolution executable works as expected.

## Manual Code Tuning #5:

- The `complex_multiply` function continues to be the second slowest function during execution of the code.
- The performance of this function can be improved by Partially Unrolling the code 3 times

**Commit**: 05695544be809617d20fedbb02247b5f1d7caa86

**Code Changes**:

```
void *complex_multiply(void *v) {
    // Perform complex multiplication
    complex_param p = ((complex_param *)v)[0];
    int k;
    for (k = 0; k < p.size; k += 6) {
        // Re Y[k] = Re X[k] Re H[k] - Im X[k] Im H[k]
        // Im Y[k] = Im X[k] Re H[k] + Re X[k] Im H[k]
        p.output[k] = p.x[k] * p.h[k] - p.x[k+1] * p.h[k+1];
        p.output[k+1] = p.x[k+1] * p.h[k] + p.x[k] * p.h[k+1];
        p.output[k+2] = p.x[k+2] * p.h[k+2] - p.x[k+3] * p.h[k+3];
        p.output[k+3] = p.x[k+3] * p.h[k+2] + p.x[k+2] * p.h[k+3];
        p.output[k+4] = p.x[k+4] * p.h[k+4] - p.x[k+5] * p.h[k+5];
        p.output[k+5] = p.x[k+5] * p.h[k+4] + p.x[k+4] * p.h[k+5];
    }
    if (k == p.size - 4) {
        p.output[k] = p.x[k] * p.h[k] - p.x[k+1] * p.h[k+1];
        p.output[k+1] = p.x[k+1] * p.h[k] + p.x[k] * p.h[k+1];
        p.output[k+2] = p.x[k+2] * p.h[k+2] - p.x[k+3] * p.h[k+3];
        p.output[k+3] = p.x[k+3] * p.h[k+2] + p.x[k+2] * p.h[k+3];
    }
    if (k == p.size - 2) {
        p.output[k] = p.x[k] * p.h[k] - p.x[k+1] * p.h[k+1];
        p.output[k+1] = p.x[k+1] * p.h[k] + p.x[k] * p.h[k+1];
    }
    return 0;
}
```

- The resulting code segment is a lot uglier than before this change, however each iteration of the loop now handles 3 complex values of the input ( `k += 6` ) rather than only a single complex value ( `k += 2` ).

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real    0m4.789s
user    0m4.196s
sys     0m0.148s
>>> gprof convolve profiling/flute-manual-5.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 93.00      3.72     3.72        3     1.24     1.24  four1(float*, int, int)
  3.00      3.84     0.12                             complex_multiply(void*)
  1.75      3.91     0.07        2     0.04     0.04  zero_padding(float*, int, float*, int)
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real    0m2.006s
user    0m1.655s
sys     0m0.087s
>>> gprof convolve profiling/guitar-manual-5.out
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 89.26      1.33     1.33        3     0.44     0.44  four1(float*, int, int)
  4.03      1.39     0.06                             complex_multiply(void*)
  2.01      1.42     0.03  1871492     0.00     0.00  fwriteShortLSB(short, _IO_FILE*)
  2.01      1.45     0.03        2     0.01     0.01  zero_padding(float*, int, float*, int)
```

- This optimization is fairly negligible and as a result it adds a minimal performance upgrade to the program.
- For the second test case of convolving `GuitarDry.wav`, for the first time the `zero_padding` function is no longer in the top 3 functions that take the most time during program execution.

**Regression Testing**:

- Previous unit tests pass and manually running the convolution code outputs correct wav files.
- The old test for the `complex_multiply` function did not actually test a case where the input is not divisible by 6. A new test case has been constructed to test an input value of a length of 8.

## Compiler-Level Optimization:

- Previously, the `convolve` executable was compiled with the `g` debug flag and the `p` profiling flag. Both of these flags slow down performance by including debug and profiling information when the executable runs.
- Adding the `O3` flag to allow the gcc compiler to optimize the executable the maximum allowed amount will also speed up execution time.

**Commit**: 910b2dd8769658b51c23c875fa4a104d3130fe42

**Code Changes**:

```
convolve: convolve.cpp
	g++ -O3 -o convolve convolve.cpp ...

test: test.cpp
	g++ -g -O3 -o test test.cpp ...
```

**Run Time Performance**:

```
>>> time ./convolve input/FluteDry.wav ir/taj_mahal.wav output/out.wav
real	0m2.766s
user	0m2.149s
sys	0m0.154s
```

```
>>> time ./convolve input/GuitarDry.wav ir/large_brite_hall.wav output/out.wav
real	0m1.205s
user	0m0.823s
sys	0m0.106s
```

- Pretty large increases to speed, both convolution tests are running nearly twice as fast as they were after the previous manual optimization.

**Regression Testing**:

- Note that the `-O3` flag was added to the makefile for the unit testing suite. This is to ensure that the functions involved with the convolution are not broken by the compiler reorganizing code.
- Unit Tests continue to pass after having the dependent FFT and Linear Convolution functions optimized.
- Manual regression testing on the full convolution continues to produce the expected output.