**Assignment 4 Report**

**Konrad Wisniewski**

**Algorithm Optimization**

Direct time domain convolution is extremely slow. The algorithm clocks in at O(n\*m), where n is the size of the input sample and m is the size of the impulse response.

To fix this, I implemented a version of FFT convolution to make the program run at O(nlogn).

With just a change in the algorithm, here are the results:

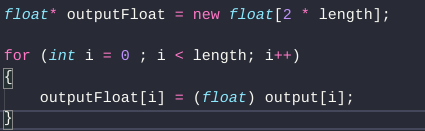
|  |  |  |  |
| --- | --- | --- | --- |
| **Program** | **Time (seconds)** | **Time spent in Colvolve Function (seconds)** | **Difference from baseline (seconds)** |
| Baseline | 413.08 | 413.06 | 0 |
| FFT | 4.16 | 4.05 | -409.01 |

**Manual Code Tuning**

Data for the speed tests can be found below. Profiler results can be found in “./Profiler Data”. Each version of the code can be found in the zip folder.

1. Getting rid of data type conversions

My code had a sloppy conversion of a double array to a float array. This is actually completely unnecessary. I decided to restructure a bit to remove this code. The code is only called once in the ComplexMultiplication function, so I did not expect a large difference in performance.

The actual time spent in the method was halved from 0.04 to 0.02. In addition the code went from taking 4.16 to 3.89 seconds. It seems like variations in background process changed the result. 

1. Unroll Loop in four1

I partially unrolled the loop at the start of four1. The increase in speed was really surprising, over half a second.

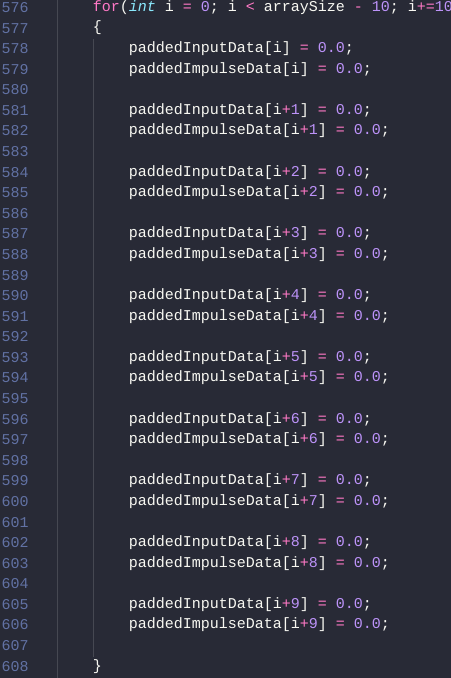
1. Partially Unroll Post Processing loop

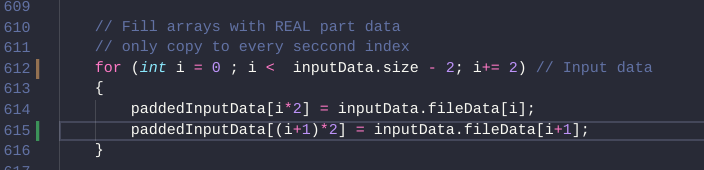
I went a little crazy and partially unrolled this loop by 20. Minor speed gain.



1. Partially Unrolled Padding loop

The loop to pad the impulse buffers with data was altered and showed a good amount of speedup. I had already been smart enough to fill both of the arrays in the same loop.



1. Partially Unrolled data loop

This is another unroll. I am adding the real data to the input impulse buffer.

|  |  |  |
| --- | --- | --- |
| **Manual Code Tune Method** | **Total Time** | **Difference from last implementation** |
| Getting rid of unnecessary Conversions | 3.89 | 0.27 |
| Unrolling Loop four1 | 3.82 | 0.07 |
| Partially Unroll Post Processing loop | 3.62 | 0.10 |
| Partially Unrolled Padding loop | 3.60 | 0.02 |
| Partially Unrolled data loop | 3.54 | 0.6 |

**Compiler-Level Optimization**

Compiler optimization was as simple as adding the -O3 option when compiling.

It seemed to have a huge effect on the speed of the code, almost doubling the speed.

|  |  |  |
| --- | --- | --- |
| **Manual Code Tune Method** | **Total Time** | **Difference from last implementation** |
| Compiler Optimization | 1.93 | 1.61 |

**Regression Tests**

I did the regressions test by converting the wav files to hex and then comparing those hex files. If the algorithm worked, the differences should be extremely minor.

Example from regressionTest1.txt:

Resulted in:

25900c25900

< 000652b0: 221c c3d2 eb1b 6bd2 e11c 78d1 c11d bed0 ".....k...x.....

---

> 000652b0: 221c c3d2 eb1b 6ad2 e11c 78d1 c11d bed0 ".....j...x.....

33639c33639

< 00083660: 74bf 4fd6 35be 11d5 03be 48d5 6fbd bfd2 t.O.5.....H.o...

---

> 00083660: 74bf 4fd6 35be 10d5 03be 48d5 6fbd bfd2 t.O.5.....H.o…

This is just a minor deviation from the first file due to the rounding of floats. This seems like a decent way to test but the output is variable and a little unreliable so I added some extra tests.

These other tests included comparing file sizes, sound, and music length. All the regression tests passed.

If the “/regressionTest#” file is empty, there was absolutely no difference between it and the last iteration.

All of the results can be found in the “../Regression Tests” folder of the project.