CPSC 501 Assignment 4 Report

Brandon Brien  
1007988

**Baseline Program**:

The baseline program is written in C and uses the gcc compiler. Measurements were taken at various points, including an overall measurement. The program was run five times to get an average for the each interval. The intervals include: reading the dry sound file, reading the impulse response, doing convolution in the time domain, writing the wave file, and overall time to finish the program from start to finish. The times in the columns will not added up to the overall time because things like malloc, and freeing memory are not included. All times are given in seconds and the results were as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| Reading Dry | 0.1260 | 0.1270 | 0.1360 | 0.1240 | 0.1300 | 0.1286 |
| Reading IR | 0.0070 | 0.0070 | 0.0080 | 0.0070 | 0.0080 | 0.0074 |
| Convolution | 997.1290 | 990.9510 | 997.9550 | 995.2840 | 994.7640 | 995.200 |
| Writing | 0.1300 | 0.1210 | 0.1220 | 0.1280 | 0.1270 | 0.1256 |
| Overall | 997.2630 | 991.0850 | 998.2220 | 996.8650 | 995.2380 | 995.7346 |

**Algorithm Based Optimization:**

A basic version of the FFT convolution was used for. The FFT algorithm is as described in the handouts given by Professor Manzara. The time measurement for convolution includes all the preprocessing that’s needed to convolve in the frequency domain (FFT on dry sound, impulse response, and IFFT on the resulting data). The results were as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| Reading Dry | 0.1310 | 0.1370 | 0.1350 | 0.1290 | 0.1310 | 0.1326 |
| Reading IR | 0.0090 | 0.0080 | 0.0090 | 0.0090 | 0.0080 | 0.0086 |
| Convolution | 3.4730 | 3.1970 | 3.1840 | 3.2360 | 3.2170 | 3.2614 |
| Writing | 0.1320 | 0.1150 | 0.1210 | 0.1270 | 0.1170 | 0.1224 |
| Overall | 3.8590 | 3.5710 | 3.5680 | 3.5960 | 3.5850 | 3.6358 |

Comparing the **averages** to the baseline program we get:

|  |  |  |
| --- | --- | --- |
|  | Baseline | Algorithm Optimization |
| Reading Dry | 0.1286 | 0.1326 |
| Reading IR | 0.0074 | 0.0086 |
| Convolution | 995.200 | 3.2614 |
| Writing | 0.1256 | 0.1224 |
| Overall | 995.7346 | 3.6358 |

It’s evident that the convolution is significantly faster when converting the data to frequency domain. The convolution speed is around 305 times faster in the frequency domain.

**Code Tuning #1:**

In order to use the four1 FFT function the size of the array needs to be a power of two. So the next power of two after the size of the original data is needed. Original, to find the number a loop was used that multiplied by two until it was bigger than the original size. The code has been tuned to use bit shifting instead. This particular code section runs very quickly so it was looped to 10,000,000 times and each version was tested five times. The results are as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| For-loop | 8.648 | 8.631 | 9.248 | 8.762 | 9.077 | 8.859 |
| Bit shifting | 1.290 | 1.438 | 1.365 | 1.302 | 1.203 | 1.320 |

The bit shifting method is 6.7 times faster than the for-loop.

//New Code

int sizeOfDubDry;

int n;

n= dryDataSize;

n--;

n = n | n>>1;

n = n | n>>2;

n = n | n>>4;

n = n | n>>8;

n = n | n>>16;

n++;

sizeOfDubDry = n\*2;

//Old Code

double nextPowTwoDry;

int sizeOfDubDry;

nextPowTwoDry = 2;

while(nextPowTwoDry < dryDataSize)

{

nextPowTwoDry = nextPowTwoDry\*2;

}

sizeOfDubDry = nextPowTwoDry\*2;

**Code Tuning #2:**

Instead of using divide, bit shifting is used instead. The code was iterated over many times in order to get measurable times. The results of the tuning is:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| Division | 5.673 | 5.670 | 5.679 | 5.638 | 5.681 | 5.6682 |
| Bit Shifting | 5.594 | 5.596 | 5.598 | 5.631 | 5.608 | 5.6054 |

The bit shifting average seems to be slightly lower than the division time.

//New

sizeOfResult = sizeOfResult>>1;

sizeOfResult--;`

//Old Code

sizeOfResult = (sizeOfResult/2) -1;

**Code Tuning #3:**

This code tuning reduces the amount of array access. It now sets a temp variable to the array index so that we only need to access it once, but use it multiple times.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| Multiple | 0.204 | 0.219 | 0.286 | 0.240 | 0.220 | 0.2338 |
| Single | 0.229 | 0.191 | 0.218 | 0.194 | 0.201 | 0.2066 |

The single access method is 1.13 times faster, or around 13%.

//New Code

for(int i=0; i<sampleCount; ++i)

{

temp = resultData[i];

if(temp > 1)

temp = 1;

else if(temp < -1)

temp = -1;

temp \*= MAX\_VAL;

//write to file

short sample = (short) temp;

fwrite(&sample, 1, bytesPerSample, out);

}

//Old Code

for(int i=0; i<sampleCount; ++i)

{

//scale

if(resultData[i] > 1)

resultData[i] = 1;

else if(resultData[i] < -1)

resultData[i] = -1;

resultData[i] \*= MAX\_VAL;

//write to file

short sample = (short) temp;

fwrite(&sample, 1, bytesPerSample, out);

}

**Code Tuning #4:**

This code tuning reduces the times the write function loops. Each iteration does mutliple steps at once. Thus, reducing the amount of overhead but making the code much less reable.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| 1 per Loop | 0.229 | 0.191 | 0.218 | 0.194 | 0.201 | 0.2066 |
| 10 per Loop | 0.188 | 0.191 | 0.190 | 0.185 | 0.199 | 0.1906 |

When the loop does the work of 10 iterations in 1 iteration there appears to be an 8.3% inrease in speed.

//New Code

for(int i=0; i<sampleCount; ++i)

{

//scale

if(resultData[i] > 1)

resultData[i] = 1;

else if(resultData[i] < -1)

resultData[i] = -1;

resultData[i] \*= MAX\_VAL;

//write to file

short sample = (short) temp;

fwrite(&sample, 1, bytesPerSample, out);

…

i++;

temp = resultData[i];

if(temp > 1)

temp = 1;

else if(temp < -1)

temp = -1;

temp \*= MAX\_VAL;

//write to file

short sample = (short) temp;

fwrite(&sample, 1, bytesPerSample, out);

//8 more iterations here

…

}

//Old Code

for(int i=0; i<sampleCount; ++i)

{

temp = resultData[i];

if(temp > 1)

temp = 1;

else if(temp < -1)

temp = -1;

temp \*= MAX\_VAL;

//write to file

short sample = (short) temp;

fwrite(&sample, 1, bytesPerSample, out);

}

**Compiler Optimization:**

The gcc flag –O3 was used as the compiler optimization and the overall run time of the program was measured. The results are as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Average |
| No opt. | 3.426 | 3.395 | 3.374 | 3.303 | 3.375 | 3.3746 |
| -O3 | 2.090 | 2.181 | 2.028 | 2.025 | 2.162 | 2.0972 |

The compiler optimization made the program run 1.6 times as fast, or about 60% faster.

**Conclusion:**

After all the optimizations have been made, there final version can run in 2.0972 seconds, whereas the base line program was taking upwards of 990 seconds. This a drastic improvement, as now this program could be used in a practical sense if someone actually wanted to convolve their dry audio with an impulse response, especially if the clip is longer than 40 seconds. The function named compareOutputs takes two file names and checks that they contain (roughly) the same data. This was used after each code tuning to make sure that the audio files were still the same.