### Getting Started

1) Configurations

- Open up Main.cpp. At the top of the file, there should be two #define directives, FFT and TESTS.

- To enable tests BEFORE executing the program, uncomment #define TESTS.

- Note that tests will delay the execution of the program by at least a minute

- To use the algorithm based optimization, FFT Convolution, uncomment #define FFT

2) Compilation

g++ \*.cpp –o a4 (or g++-4 \*.cpp –o a4 for 64-bit Windows Cygwin users)

3) Execution

./a4 (or a4 for Windows users)

### What Works

* Baseline Program (on .wav files)
* Algorithm-Based Optimization version of Baseline (on .wav files)
* Bonus #1: Stereo detection on Impulse Response (on .wav files)

### What Doesn’t Work

* Bonus #2: Handle aiff and snd file formats
* Using a snd file as any input or output file will produce a file that CAN be opened, but does not sound correct.
* Only a wave input / snd impulse / wav output, sounds close, but still not correct.
* I narrowed the reason down to reading the SND file data. Everything else works because I tried creating an SND file with WAVE data and it works. Therefore I’m reading the SND file data incorrectly, but I’m not sure why what I have is wrong.
* Using an aiff file as any input or output file will either produce (1) a file that CANNOT be opened (only tested with Windows Media Player) or (2) an openable file that has incorrect sounding output.
* I didn’t have much time to test this further, so I’m not sure why it doesn’t work.
* It may have to do with the fact that I just made a barebones AIFF file (all chunks except Common and Sound Data are optional, see References [5]). I tested opening the sound files with Windows Media Player, and it may have relied on these extra chunks.
* NOTE: I used sox to convert the dry recording and impulse responses from wave to snd and aiff. It is possible that I used sox incorrectly (maybe I didn’t set endian, b, c, or r flags correctly, or maybe more flags needed to be set) and that may have contributed to my strange outputs.

### Testing

* Unit tests are provided. four1 FFT and IFFT, and convolve/fftConvolve wrappers that take SoundFile\* are more difficult to test, and they are not tested directly.
* However, these convolve/fftConvolve wrappers are wrapping convolve/fftConvolve methods that take double\* signals. These lower level methods *are* tested.
* Four1 is indirectly tested through the regression test.
* The main regression test is found in RegressionTest.cpp and compares the output of the baseline program, with the algorithm-based optimization version of the baseline program (i.e. using FFT convolution).
* All tests were run after every optimization.

**Testing Constants and Inputs**

* Dry Recording: **testCase1.wav** – Mono 16-bit Sample Size
* Impulse Response: **Parking Garage.wav** – Mono 16-bit Sample Size

### Profiling

**Tool:** Microsoft Visual Studio Profiling Tools (see References [0])

**Measurement:** *Elapsed Inclusive* (for the whole program, see References [1]). This includes function stack set up and tear downs and system calls. Tests are turned off before profiling so tests are not included.

**Recording**: Each profiling session was run 10 times and the best time was recorded in this report.

### Optimizations Performed

**Template**

**Code Before** **Code After**

<CODE BEFORE> <CODE AFTER>

**Steps**

1. The first step is always committing to version control, so they are not listed below.
2. The last step is always running tests, and it they pass, committing to version control, and profiling.

**Total Time Before:**

**Total Time After:**

These times are measured as *Elapsed Inclusive* in the scope of the entire program (i.e. main()). This means that the total time that is spent executing the *program* from start to finish is measured. This includes function stack set up and tear downs, as well as system calls. Tests are turned off before profiling so tests are not measured.

**Speed Gain:**

**Results So Far:**

This shows the trending total execution time of the program thus far. Ideally, as this table grows, the execution time as it grows.

\*\* NOTE \*\*

The total time before any optimizations (i.e. the total time of the baseline program) is **359525.44 ms**

**Algorithm Based Optimization – FFT Convolution**

**Code Before** **Code After**

int main(int argc, char\* argv[]) {

// dry is input signal, ir is impulse

Convolver::fftConvolve(dryNormalized,

dry->getDataSize(),

irNormalized,

ir->getDataSize(),

result, P);

. . .

}

}

int main(int argc, char\* argv[]) {

// dry is input signal, ir is impulse

Convolver::convolve(dryNormalized,

dry->getDataSize(),

irNormalized,

ir->getDataSize(),

result, P);

. . .

}

}

\* See Convolver.cpp for convolve() \* See Convolver.cpp for fftConvolve()

**Steps**

1. Make fftConvolve() for the optimized algorithm with the same arguments/signatures as convolve()
2. Convert input signals to frequency domain before applying convolution
3. Convert results back to time domain

Step 2 is where the magic happens. Only one linear loop, rather than a nested loop, through the signal for complex multiplication is required. O(N) is better than O(N\*M).

**Total Time Before:** 359,525.44 ms

**Total Time After:** 9,184.23 ms

**Speed Gain:** 350,341.21 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |

**Code Tuning 1 – Eliminate Common Subexpressions**

**Code Before** **Code After**

void Convolver::zeroPadAndTimeToFreqDomain(…) {

int i = 0;

int i2;

for (; i < timeDomainLen; i++ ) {

i2 = i\*2;

outputFreqDomain[i2] = timeDomain[i];

outputFreqDomain[i2+1] = 0;

}

for (; i < structuredSize; i++ ) {

i2 = i\*2;

outputFreqDomain[i2] = 0;

outputFreqDomain[i2+1] = 0;

}

}

void Convolver::fftConvolve(…) {

int structuredSize = X;

int structuredSize2 = structuredSize\*2;

//Replace all structuredSize\*2 with structuredSize2

}

int main(int argc, char\* argv[]) {

int halfM = ir->getDataSize()/2;

// Replace all ir->getDataSize()/2 with halfM

}

void Convolver::zeroPadAndTimeToFreqDomain(…)

{

int i = 0;

for (; i < timeDomainLen; i++ ) {

outputFreqDomain[i\*2] = timeDomain[i];

outputFreqDomain[i\*2+1] = 0;

}

for (; i < structuredSize; i++ ) {

outputFreqDomain[i\*2] = 0;

outputFreqDomain[i\*2+1] = 0;

}

}

void Convolver::fftConvolve(…) {

int structuredSize = X; // X is some value

// Many occurences of structuredSize\*2

}

int main(int argc, char\* argv[]) {

// Many occurences of ir->getDataSize()/2

}

**Steps**

**Steps**

1. Find all values that are being repeatedly calculated (in this case, the loop counter i, the size of impulse response, and the structuredSize value is always being doubled or halved.
2. Calculate these values once, store them, and just use that pre-calculated value instead of recalculating.

**Total Time Before:** 9,184.23 ms

**Total Time After:** 8,792.83 ms

**Speed Gain:** 391.4 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |

**Code Tuning 2 – Strength Reduction**

**Code Before** **Code After**

// Replace all occurrences of a multiplication or a division by 2 with bit shifting (i.e. use << to multiply or >> to divide)

// Many occurrences of a multiplication or a division by 2

**Steps**

1. Find all occurrences where there is a multiplication or a division by N where N is a power of 2.
2. Replace multiplication by N with shifting left floor(sqrt(N)) times
3. Replace division by N with shifting right floor(sqrt(N)) times

**Total Time Before:** 8,792.83 ms

**Total Time After:** 8,553.74 ms

**Speed Gain:**  239.09 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |
| Strength Reduction | 8553.74 |

**Code Tuning 3 – Minimize References** (Dereferencing a pointer or indexing an array)

**Code Before** **Code After**

int main(int argc, char\* argv[]) {

// dry is an object containing the dry recording

// ir is an object containing the impulse response

// Many occurences of:

dry->getData();

dry->getDataSize();

ir->getData();

ir->getDataSize();

. . .

}

int main(int argc, char\* argv[]) {

// dry is an object containing the dry recording

// ir is an object containing the impulse response

short\* dryData = dry->getData();

short\* irData = ir->getData();

int dryDataSize = dry->getDataSize();

int irDataSize = ir->getDataSize();

// Use above cached values rather than dereferencing dry and ir every time

}

**Steps**

**Steps**

1. Make local variables containing the values that are often obtained through dereferencing some pointer
2. Use these local variables rather than dereferencing
3. There are no assignments through dereferencing in this case, so we don’t have to worry about that

**Total Time Before:** 8,553.74 ms

**Total Time After:** 8,277.55 ms

**Speed Gain:** 276.19 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |
| Strength Reduction | 8553.74 |
| Minimize References | 8277.55 |

**Code Tuning 4 – Substitute Function with Macro** (Not one of the Code Tuning techniques in lecture)

**Code Before**

double Convolver::normalize(double in, double fromMin, double fromMax, double toMin, double toMax) {

return ( (toMax-toMin) \* (in-fromMin) / (fromMax-fromMin) ) + toMin;

}

// Code calling Convolver::normalize()

**Code After**

#define NORMALIZE(x,fromMin,fromMax,toMin,toMax) (((toMax)-(toMin))\*((x)-(fromMin)) / ((fromMax)-(fromMin))) + (toMin)

// Code calls uses NORMALIZE macro instead of Convolver::normalize()

**Steps**

1. Write macro for the return value mathematic calculation
2. Make normalize() call the macro and return that instead. Test.
3. If working, remove normalize() and replace all calls to it with the macro.

**Total Time Before:** 8,277.55 ms

**Total Time After:** 8093.92ms

**Speed Gain:** 183.63 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |
| Strength Reduction | 8553.74 |
| Minimize References | 8277.55 |
| Substitute Function with Macro | 8093.92 |

**Code Tuning 5 – Jamming**

**Code Before** **Code After**

for (int i = 0; i < structuredSize2; i+=2) {

if (R[i] < min)

min = R[i];

if (R[i] > max)

max = R[i];

}

// use min and max

int min = \*min\_element(R, R+structuredSize2);

int max = \*max\_element(R, R+structuredSize2);

// use min and max

**Steps**

1. Find occurrences of two similar, but separate/independent loops. We can merge them together to reduce loop overhead (loop guard, variable increment, jmp assembly instruction that goes back up to top of the loop, etc) as well as only do one pass, rather than two.
2. Find the range of overlapping loop counters, make a new loop for that, and do work of both previous loops in there.
3. Since there are no iterations special or unique to the two original loops, we can completely merge (jam/fuse) both loops into one.

**Total Time Before:** 8093.92 ms

**Total Time After:**  7,866.74 ms

**Speed Gain:** 227.18 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |
| Strength Reduction | 8553.74 |
| Minimize References | 8277.55 |
| Substitute Function with Macro | 8093.92 |
| Jamming | 7866.74 |

**Code Tuning 6 – Partial Unrolling**

**Code Before** **Code After**

// Make each iteration do the work of two iterations

int i;

for (i = 0; i < len-1; i+=2) {

signal[i] = (double) data[i] / min;

signal[i+1] = (double) data[i] / min;

}

if (i == len-1)

signal[i] = (double) data[i] / min;

// All code that loops over an array one by one, ie:

for (int i = 0; i < len; i++) {

signal[i] = (double) data[i] / min;

}

**Steps**

1. Find all loops that traverse an array and performs some reading or manipulation on each index.
2. Double the loop body by doing the work of two iterations in one iteration. Generally this means doing the work of iteration i and also i+1.
3. Increment the loop counter by two rather than one. This effectively results in half the number of iterations, for each loop you can apply this to.
4. Be sure to handle both odd and even loop guards

**Total Time Before:** 7866.74 ms

**Total Time After:**  7692.11 ms

**Speed Gain:** 174.63 ms

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |
| Strength Reduction | 8553.74 |
| Minimize References | 8277.55 |
| Substitute Function with Macro | 8093.92 |
| Jamming | 7866.74 |
| Partial Unrolling | 7692.11 |

NOTE: This technique should generate much better results than shown here, The reason it doesn’t is because my loops were only run one or two times, so optimizing them doesn’t have much of an effect. It may also have helped to do 3 or 4 iterations of work within 1 loop body.

**Compiler Optimization (-O3)**

**Steps**

1. Compile with –O3 flag

**Total Time Before:** 7692.11 ms

**Total Time After:**  5124.24 ms

**Speed Gain:**

**Results So Far:**

|  |  |
| --- | --- |
| **Optimization Technique** | **Total Execution Time (ms)** |
| None | 359525.44 |
| Algorithm-Based Optimization - FFT Convolution | 9184.23 |
| Eliminate Common Subexpressions | 8792.83 |
| Strength Reduction | 8553.74 |
| Minimize References | 8277.55 |
| Substitute Function with Macro | 8093.92 |
| Jamming | 7866.74 |
| Partial Unrolling | 7692.11 |
| Compiler Optimization (-O3) |  |

### References

Microsoft Visual Studio Profiling Tools: [0] <http://msdn.microsoft.com/en-us/library/z9z62c29.aspx>

Documentation: [1] <http://msdn.microsoft.com/en-us/library/dd264994.aspx>

Snd File Format:

[2] <http://sox.sourceforge.net/AudioFormats-11.html>

[3] <http://en.wikipedia.org/wiki/Au_file_format>

[4] <http://www-mmsp.ece.mcgill.ca/documents/audioformats/AU/AU.html>

Aiff File Format:

[5] <http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/AIFF/Docs/AIFF-1.3.pdf>

[6] <http://muratnkonar.com/aiff/index.html>

Leonard Manzara & Abbas Sarraf : Amazing and helpful professor and TA.