CPSC 501 - Assignment #4 Optimization Report
Samuel Niu - 10047006

****NOTE: All tests for this report were done using putty to connect to the school linux servers so there may be a slight increase in times. Multiple tests were run to determine an average (average of 5)****

Baseline convolution vs FFT Convolution

>The following test was done with the following files:

>Tabla.wav (23 seconds, Size = 2,094,028 bytes)

> BHIR.wav (3 seconds, 213,244 bytes)

Baseline	FFT	Improvement
21 minutes 41.189 seconds Total of 1301.189 seconds	2.032 seconds	640.348917 times faster

**As the following codes will be tested only on FFT, longer audio files will be used

>Harp.wav (3 minutes 2 seconds, 16,142,446 bytes)

>AP.wav (5 seconds, 455,038 bytes)

** Sometimes there will be more examples of the code tuning done that is not shown but will be portrayed in the final result

<u>Code Tune #1: Strength Reduction</u>

There are many instances where we multiply/divide a number repeatedly by 2. In these scenarios, we can save computation time by doing a logical shift left (or right) instead of multiplication or division.

Before	**After**

while(numSquared < maxLength){
numSquared *= 2;
}
	while(numSquared < maxLength){
	numSquared = numSquared << 1;
	}

// multiply n by itself to get nn
numSquared *= 2;
	// multiply n by itself to get nn
	numSquared = numSquared << 1;

for (i = 0; i < numInput; i++){
inputPadded[i*2] = signalInput[i];
}
	for (i = 0; i < numInput; i++){
	inputPadded[i<<1] = signalInput[i];
	}

for (i = 0; i < numImpulse; i++){
impulsePadded[i*2] =
ImpulseResponse[i];
}
	for (i = 0; i < numImpulse; i++){
	impulsePadded[i<<1] =
	ImpulseResponse[i];
	}

Timing (Original FFT vs Post-Tune #1)

FFT	FFT + Tune #1	Improvement
24.406 seconds	23.424 seconds	4.02%

Code Tune #2: Jamming (Fusion)

Several loops were created with the same starting point and length. These can be merged into a single loop in order to avoid the overhead time of searching through the loop condition

Before	**After**

// Zero padding new arrays inputPadded
for(i=0; i < numSquared * 2;i++){
inputPadded[i] = 0;
}
 	
// Zero padding impulsePadded
for(i=0; i < numSquared * 2; i++){
impulsePadded[i] = 0;
}
	// Zero padding new arrays inputPadded and impulsePadded
	for(i=0; i < numSquared * 2;i++){
	inputPadded[i] = 0;
	impulsePadded[i] = 0;
	}

Timing (Post-Tune #1 vs Post-Tune #2)

FFT + Tune #1	FFT + Tune #2	Improvement
23.424 seconds	22.829 seconds	2.61%

Code Tune #3: Partial Unrolling of Loops

In the convolve function, there is a loop that has a guaranteed length to the power of 2 (outside of the case where length = 1). This means we can double the number of elements that gets added per cycle and reduce the number of total cycles in half

Before	**After**

for(i=0; i < numSquared;i++){
inputPadded[i] = 0;
impulsePadded[i] = 0;
}
	for(i=0; i < numSquared / 2;i++){
	inputPadded[i] = 0;
	inputPadded[i+1] = 0;
	impulsePadded[i] = 0;
	impulsePadded[i+1] = 0;
	}

Timing (Post-Tune #2 vs Post-Tune #3)

FFT + Tune #2	FFT + Tune #3	Improvement
22.829 seconds	21.675 seconds	5.32%

<u>Code Tune #4: Minimize work done in loops</u>

There are many loops where an operation gets done in the condition check. These can be removed from the loop so it doesn't have to get recalculated every single time.

Before	**After**

for(i=0; i < **numSquared / 2**;i++){
inputPadded[i] = 0;
inputPadded[i+1] = 0;
impulsePadded[i] = 0;
impulsePadded[i+1] = 0;
}
	numSquared = numSquared << 1;
	Int nHolder = numSquared;
	
	for(i=0; i < **nHolder**;i++){
	inputPadded[i] = 0;
	impulsePadded[i] = 0;
	}

for(i=0; i < **numSquared * 2**; i+=2){
// Real = (R * R) - (I * I)
oData[i] = (inputPadded[i] * impulsePadded[i]) - (inputPadded[i+1] * impulsePadded[i+1]);
	numSquared = numSquared << 1;
	Int nHolder = numSquared;
	
	or(i=0; i < **numSquared**; i+=2){
	// Real = (R * R) - (I * I)

}	oData[i] = (inputPadded[i] * impulsePadded[i]) - (inputPadded[i+1] * impulsePadded[i+1]);
	
	}

four1(oData-1, **numSquared**, -1);	Int nHolder = numSquared/2; ; four1(oData-1, **nHolder**, -1);

Timing (Post-Tune #3 vs Post-Tune #4)

FFT + Tune #3	FFT + Tune #4	Improvement
21.675 seconds	21.465 seconds	0.98%

<u>Code Tune #5: Reduce Array References</u>

At the complex multiplication portion of FFTConvolution references, there were multiple calls to the same array element in the same phase of the loop. By declaring the array values at the beginning of the loop, we only need to reference each array element once instead of 2+ times

Before	**After**

// Complex multiplcation
for(i=0; i < numSquared * 2; i+=2){
// Real = (R * R) - (I * I)
oData[i] = (inputPadded[i] * impulsePadded[i]) - (inputPadded[i+1] * impulsePadded[i+1]);
 	
// Imaginary = (R * I) + (R * I)
oData[i+1] = (inputPadded[i] * impulsePadded[i+1]) + (inputPadded[i+1] * impulsePadded[i]);
}
	// Complex multiplcation
	double R1,R2,I1,I2;
	for(i=0; i < numSquared; i+=2){
	R1 = inputPadded[i];
	R2 = impulsePadded[i];
	I1 = inputPadded[i+1];
	I2 = impulsePadded[i+1];
	
	// Real = (R * R) - (I * I)
	oData[i] = (R1 * R2) - (I1 * I2);
	
	// Imaginary = (R * I) + (R * I)
	oData[i+1] = (R1 * I2) + (R2 * I1);
	}

Timing (Post-Tune #2 vs Post-Tune #3)

FFT + Tune #4	FFT + Tune #5	Improvement
21.465 seconds	21.100 seconds	1.73%

Compiler Level Optimizations

FFT + Tune #5	FFT + Tune #5 + O3 Optimization	Improvement
21.100 seconds	14.063 seconds	50.039%