CPSC 501 A4 Report
Christian Dudar - 10135508

Note: No bonuses have been attempted

Note:
All times are taken using the linux 'time' command
Ex:

```
$  time ./fftConvolve Tabla.wav bigHall.wav fftTablaHall.wav
real    0m0.083s
user   0m0.067s
sys    0m0.002s
```

Times stated in the tables are always the '**real**' time value outputted (not user/sys).

Time-domain convolution to frequency convolution

The time-domain convolution algorithm is a $O(n^2)$ algorithm, while the Fast-fourier convolution is a O(n) algorithm. The following chart shows the differences in time taken to convolve the Tabla.wav file with the bigHall.wav impulse response file.

Audio: Tabla.wav
Impulse: bigHall.wav

Time-domain (s)	Frequency domain (s)	Reduction (%)
641 (10 min 41 sec)	1.12s	99.83 %

Optimizations 1-5

The following 5 optimizations will all involve running the harp.wav audio with bigHall.wav as the impulse response.

Optimization 1

Code Jamming: Placed the zero-padding loops into the same loop as their parameters are identical.

Before:	**After:**

for (int i = 0; i < actualSize; i++){
  inputDataPadded[i] = 0;
}
 	
for(int i = 0; i < actualSize; i++){
  impulseDataPadded[i] = 0;
}
	for (int i = 0; i < actualSize; i++){
	  inputDataPadded[i] = 0;
	  impulseDataPadded[i] = 0;
	}

Audio: harp.wav
Impulse: bigHall.wav

FFT base program (s)	Optimization 1 (s)	Reduction
14.054	13.582	3.35%

Optimization 2

Partial Unrolling: Partial unrolling of the input and impulse zero-padded data arrays. Since the arrays are always a power of 2 no check is needed for the last element.

Before:	**After:**

for (int i = 0; i < actualSize; i++){
  inputDataPadded[i] = 0;
  impulseDataPadded[i] = 0;
  }
	for (int i = 0; i < actualSize; i = i + 2){
	  inputDataPadded[i] = 0;
	  inputDataPadded[i + 1] = 0;
	  impulseDataPadded[i] = 0;
	  impulseDataPadded[i + 1] = 0;
	  }

Audio: harp.wav
Impulse: bigHall.wav

Optimization 1 (s)	Optimization 2 (s)	Reduction
13.582	13.347s	1.73%

Optimization 3

Minimizing work inside loops: This technique takes any calculations used in loops that result in constants and calculates them before the loop, so that the same value does not need to be calculated each time.

Before:

```
for(int i = 0; i < Subchunk2Size / (bitsPerSample / 8); i++){

   input.read((char*)&sample, 2);

   double converted = (double) sample / (double)INT16_MAX;

         if (converted < -1.0){
         converted = -1.0;
          }

         dataArray[i] = converted;
}
```

After:

```
int numSamples = Subchunk2Size / (bitsPerSample / 8);

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

   input.read((char*)&sample, 2);

   double converted = (double) sample / (double)INT16_MAX;

         if(converted < -1.0){
         converted = -1.0;
         }

         dataArray[i] = converted;

}
```

Audio: harp.wav
Impulse: bigHall.wav

Optimization 2 (s)	Optimization 3 (s)	Reduction
13.347	13.162s	1.38%

Optimization 4

Reduce Array References: The complex number multiplication loop involves accessing identical array references more than once. We can reduce array references (by just finding them once per loop and assigning them to variables) to save CPU time.

Before:

```
for(int i = 0; i < actualSize; i = i + 2){
outputData[i] = (inputDataPadded[i] * impulseDataPadded[i]) - (inputDataPadded[i + 1] * impulseDataPadded[i + 1]);

outputData[i + 1] = (inputDataPadded[i + 1] * impulseDataPadded[i]) + (inputDataPadded[i] * impulseDataPadded[i + 1]);
}
```

After:

```
for(int i = 0; i < actualSize; i = i + 2){
   double realInput = inputDataPadded[i]
   double realImpulse = impulseDataPadded[i]
   double imaginaryInput = inputDataPadded[i + 1]
   double imaginaryImpulse = impulseDataPadded[i + 1];

outputData[i] = (realInput * realImpulse) - (imaginaryInput * imaginaryImpulse);
outputData[i + 1] = (imaginaryInput * realImpulse) + (realInput * imaginaryImpulse);
}
```

Optimization 3 (s)	Optimization 4 (s)	Reduction
13.162	12.92	1.8%

Optimization 5

Strength Reduction:

Many lines in the code involve multiplying by 2. We can reduce the cost of this by performing bitshifts rather than multiplication.

while(longestInput > powerOf2){
powerOf2 *= 2;
}
 	
for(int i = 0; i < N; i++){
inputDataPadded[i * 2] = x[i];
}
	while(longestInput > powerOf2){
	powerOf2<<1;
	}
	
	for(int i = 0; i < N; i++){
	inputDataPadded[i<<1] = x[i];
	}

Note: This change was applied to more places in the code than is shown above.

Optimization 4 (s)	Optimization 5 (s)	Reduction
12.92	12.57	2.7%

Optimization 6 (Compiler level optimization)

Compiling the fftConvolve file with -o3 flags enables compiler-level optimization

```
$  g++ -O3 fftConvolve5.cpp -o fftCompilerOptimized
$ time ./fftCompilerOptimized  harp.wav bigHall.wav harpHallcompiler.wav
```

Optimization 5 (s)	CompilerOptimized (s)	Reduction
12.57	7.82	37.8%