Input Files

Files used for testing: input “inputGuitar.wav” and impulse response “impulseTajMahal.wav”. Since the baseline program is linear and time-invariant, only one set of input and impulse response files were used for formal testing.

Optimizations and Tunings

Overview

1. Changed from input side convolution algorithm to fast Fourier transform
2. Precompute results of expensive operations in four1
   2a. Exploit algebraic identities to remove an argument and an operation
3. Strength reduction for finding the next power of two
4. Partial unrolling for second half of four1
5. Minimize array references when performing complex multiplication
6. Caching a value to reread rather than recompute
7. Jamming loops used for scaling and jamming the output signal
   7a. Strength reduction in finding the maximum absolute value in an array
   7b. Reducing array minimizing array references by caching values
8. Compiler optimization -O3
9. Compiler optimization -Ofast

1 – Algorithm-Based Optimization

The algorithm for convolution was changed from input side convolution to fast Fourier transform which is significantly faster. This provides the most powerful optimization by far.

<table>
<tr><th>Code Before</th><th>Code After</th></tr>
<tr><td colspan="2">convolve</td></tr>
<tr><td>

```
int n, m;
y = (double *) malloc (sizeof(double) * P);

if (P != (N + M - 1)) {
    cout << "Invalid output signal size. Aborting
Convolution." << endl;
    return y;
}

for (n = 0; n < P; n++)
    y[n] = 0.0;

for (n = 0; n < N; n++)
    for (m = 0; m < M; m++)
        y[n + m] += x[n] * h[m];

return y;
```

</td><td>

```
y = (double *) malloc (sizeof(double) * P);

int arrayLength = pow(2, ceil(log(max(N, M))/log(2)))
* 2;

auto *input = new double[arrayLength] {};
auto *impulse = new double[arrayLength] {};
auto *output = new double[arrayLength] {};

for (int i = 0; i < N; i++)
    input[i*2] = x[i];
for (int i = 0; i < M; i++)
    impulse[i*2] = h[i];

four1(input - 1, arrayLength / 2, 1);
four1(impulse - 1, arrayLength / 2, 1);

for (int i = 0; i < arrayLength; i+=2) {
    output[i] = input[i] * impulse[i] - input[i+1] *
impulse[i+1];
    output[i+1] = input[i+1] * impulse[i] + input[i]
* impulse[i+1];
}

four1(output - 1, arrayLength / 2, -1);

for (int i = 0; i < P*2; i+=2)
    y[i/2] = output[i];

return y;
```

</td></tr>
<tr><td colspan="2">four1</td></tr>
<tr><td></td><td>

```
unsigned long n, mmax, m, j, istep, i;
double wtemp, wr, wpr, wpi, wi, theta;
double tempr, tempi;

n = nn << 1;
j = 1;
```

</td></tr>
</table>

	for (i = 1; i < n; i += 2) {
	  if (j > i) {
	  SWAP(data[j], data[i]);
	  SWAP(data[j+1], data[i+1]);
	  }
	  m = nn;
	  while (m >= 2 && j > m) {
	  j -= m;
	  m >>= 1;
	  }
	  j += m;
	}
	
	mmax = 2;
	while (n > mmax) {
	  istep = mmax << 1;
	  theta = isign * (TWO_PI / mmax);
	  wtemp = sin(0.5 * theta);
	  wpr = -2.0 * wtemp * wtemp;
	  wpi = sin(theta);
	  wr = 1.0;
	  wi = 0.0;
	  for (m = 1; m < mmax; m += 2) {
	  for (i = m; i <= n; i += istep) {
	  j = i + mmax;
	  tempr = wr * data[j] - wi * data[j+1];
	  tempi = wr * data[j+1] + wi * data[j];
	  data[j] = data[i] - tempr;
	  data[j+1] = data[i+1] - tempi;
	  data[i] += tempr;
	  data[i+1] += tempi;
	  }
	  wr = (wtemp = wr) * wpr - wi * wpi + wr;
	  wi = wi * wpr + wtemp * wpi + wi;
	  }
	  mmax = istep;
	}

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	422.502	1.956	21600.307%

2 – Precompute Results

Since most of the time (94.20%) of the time is spent in the four1 function, this is an important bottleneck to optimize. I noticed that although the function is very well optimized, it recomputes sin values that are not necessary after the first call. I cached memory on the first time four1 is called so these values can be cached for later use. Caching did not provide as much of an improvement as expected.

Code Before	Code After
cache	

	double* cache = (double *) malloc (1024);
	char cacheType = 0;
	
	...
	
	free(cache);

four1	

while (n > mmax) {
  istep = mmax << 1;
  theta = isign * (TWO_PI / mmax);
  wtemp = sin(0.5 * theta);
  wpr = -2.0 * wtemp * wtemp;
  wpi = sin(theta);
  wr = 1.0;
  wi = 0.0;
 	
  ...
}
	while (n > mmax) {
	  istep = mmax << 1;
	
	  if (cacheType == 0) {
	  theta = isign * (TWO_PI / mmax);
	  wtemp = sin(0.5 * theta);
	  wpr = -2.0 * wtemp * wtemp;
	  wpi = sin(theta);
	
	  cache[counter] = wpr;
	  cache[counter + 1] = wpi;
	  }
	  else if(cacheType == 1) {
	  wpr = cache[counter];

	wpi = cache[counter + 1];
	}
	else {
	wpr = cache[counter];
	wpi = cache[counter + 1] * - 1;
	}
	
	...
	
	counter += 2;
	}
	cacheType++;

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	1.758	1.721	102.14%

2a – Exploit Algebraic Identities

The algebraic property found here is equivalent of A = B and A = true then B = true where A = cacheType and B = isign. This eliminated one multiplication operation and parameter in for the four1 function.

Code Before	Code After
four1 calls	

four1(input - 1, arrayLength / 2, 1);
four1(impulse - 1, arrayLength / 2, 1);
 	
...
 	
four1(output - 1, arrayLength / 2, -1);
	four1(input - 1, arrayLength / 2);
	four1(impulse - 1, arrayLength / 2);
	
	...
	
	four1(output - 1, arrayLength / 2);

four1	

if (cacheType == 0) {
theta = isign * (TWO_PI / mmax);
wtemp = sin(0.5 * theta);
wpr = -2.0 * wtemp * wtemp;
wpi = sin(theta);
 	
cache[counter] = wpr;
cache[counter + 1] = wpi;
}
	if (cacheType == 0) {
	theta = TWO_PI / mmax;
	wtemp = sin(0.5 * theta);
	wpr = -2.0 * wtemp * wtemp;
	wpi = sin(theta);
	
	cache[counter] = wpr;
	cache[counter + 1] = wpi;
	}

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	1.721	1.706	100.879%

3 – Strength Reduction

This code snippet finds the next power of two. The first function uses a pow, ceil, log, and max, which are relatively expensive operations. The code that replaces this finds the next power of two for unsigned ints using bitwise operations. Although the relative efficiency is vastly improved, the improvement is not very significant in terms of the overall program.

Code Before	Code After
convolve	

int arrayLength = pow(2, ceil(log(max(N, M))/log(2))) * 2;	unsigned int arrLen = max(N, M) - 1;
	arrLen |= arrLen>>1;
	arrLen |= arrLen>>2;
	arrLen |= arrLen>>4;
	arrLen |= arrLen>>8;
	arrLen |= arrLen>>16;
	arrLen = (arrLen + 1) << 1;

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	1.66e-05	4.33e-07	3833.718%

4 – Partial Unrolling

As mentioned earlier, since the four1 function is the bottleneck, I was able to find another optimization I could implement. The inner loop for the second half of the function was unrolled. Similar to optimization 2, these results were underwhelming. Note: I tried unstitching this loop later on, but it surprisingly hindered the performance.

Code Before	Code After
four1	

Code Before:

```
for (m = 1; m < mmax; m += 2) {
    for (i = m; i <= n; i += istep) {
        j = i + mmax;
        tempr = wr * data[j] - wi * data[j+1];
        tempi = wr * data[j+1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
        data[i] += tempr;
        data[i+1] += tempi;
    }
    wr = (wtemp = wr) * wpr - wi * wpi + wr;
    wi = wi * wpr + wtemp * wpi + wi;
}
```

Code After:

```
for (m = 1; m < mmax - 1; m += 2) {
    for (i = m; i <= n; i += istep) {
        j = i + mmax;
        tempr = wr * data[j] - wi * data[j+1];
        tempi = wr * data[j+1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
        data[i] += tempr;
        data[i+1] += tempi;
    }
    wr = (wtemp = wr) * wpr - wi * wpi + wr;
    wi = wi * wpr + wtemp * wpi + wi;
    for (i = (m+=2); i <= n; i += istep) {
        j = i + mmax;
        tempr = wr * data[j] - wi * data[j+1];
        tempi = wr * data[j+1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
        data[i] += tempr;
        data[i+1] += tempi;
    }
    wr = (wtemp = wr) * wpr - wi * wpi + wr;
    wi = wi * wpr + wtemp * wpi + wi;
}
if (m == mmax - 1) {
    for (i = m; i <= n; i += istep) {
        j = i + mmax;
        tempr = wr * data[j] - wi * data[j+1];
        tempi = wr * data[j+1] + wi * data[j];
        data[j] = data[i] - tempr;
        data[j+1] = data[i+1] - tempi;
        data[i] += tempr;
        data[i+1] += tempi;
    }
}
```

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	1.776	1.767	100.509%

5 – Minimize Array References

Array accesses take significantly more time to access than variables in most cases. Initially, this optimization actually slowed down the results. I later found out that declaring the variables inReal, inComp, irReal, and IrComp outside the loop reduces variable essentializations which sped up the program.

Code Before	Code After
convolve	

Code Before:

```
for (int i = 0; i < arrLen; i+=2) {
    output[i] = input[i] * impulse[i] - input[i+1] *
impulse[i+1];
    output[i+1] = input[i+1] * impulse[i] + input[i]
* impulse[i+1];
}
```

Code After:

```
double inReal;
double inComp;
double irReal;
double irComp;
for (int i = 0; i < arrLen; i+=2) {
    inReal = input[i];
    inComp = input[i+1];
    irReal = impulse[i];
    irComp = impulse[i+1];
    output[i] = inReal * irReal - inComp * irComp;
    output[i+1] = inComp * irReal + inReal * irComp;
}
```

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	0.013	0.011	118.182%

6 – Caching

I was calculating half the size of arrLen for every four1 function call. This was improved by caching this result. Since the end of optimization 3, the value of arrLen is doubled, the value could be cached here, which means that halfArrLen doesn't need to be directly computed.

Code Before	Code After
convolve	

`unsigned int arrLen = max(N, M) - 1;`
`arrLen |= arrLen>>1;`
`arrLen |= arrLen>>2;`
`arrLen |= arrLen>>4;`
`arrLen |= arrLen>>8;`
`arrLen |= arrLen>>16;`
`arrLen = (arrLen + 1) << 1;`
	`unsigned int arrLen = max(N, M) - 1;`
	`arrLen |= arrLen>>1;`
	`arrLen |= arrLen>>2;`
	`arrLen |= arrLen>>4;`
	`arrLen |= arrLen>>8;`
	`arrLen |= arrLen>>16;`
	`int halfArrLen = ++arrLen;`
	`arrLen <<= 1;`

four1 function calls	

`four1(input - 1, arrLen / 2);`
`four1(impulse - 1, arrLen / 2);`
...
`four1(output - 1, arrLen / 2);`
	`four1(input - 1, halfArrLen);`
	`four1(impulse - 1, halfArrLen);`
	...
	`four1(output - 1, halfArrLen);`

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	1.978	1.894	104.435%

7 – Jamming

I found two loops that operate on the output signal array by scaling and filtering it. I combined these loops into one to reduce the amount of loop overhead required. This provided a higher relative efficiency value than expected as I was only able to reduce three loops into two.

Code Before	Code After
main	

`double maxAbsOutput = 0;`
`for (int i = 0; i < outputSize; i ++)`
`  if (abs(output[i]) > maxAbsOutput)`
`  maxAbsOutput = abs(output[i]);`
 	
`for (int i = 0; i < outputSize; i ++)`
`  output[i] /= maxAbsOutput;`	

convolve	

`for (int i = 0; i < P*2; i+=2)`
`  y[i/2] = output[i];`
	`double maxAbsOutput = 0;`
	`for (int i = 0; i < P*2; i ++)`
	`  if (abs(output[i]) > maxAbsOutput)`
	`  maxAbsOutput = abs(output[i]);`
	
	`for (int i = 0; i < P*2; i+=2)`
	`  y[i/2] = output[i] / maxAbsOutput;`

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	0.097	0.026	373.077%

7a – Strength Reduction

I noticed that the abs function was called 385,584 times for each time the program runs. I found an optimization for the algorithm that would reduce the amount of calls to abs to only 1. Although this requires more variables, and conditionals, they are more efficient.

Code Before	Code After
convolve	

`double maxAbsOutput = 0;`
`for (int i = 0; i < P*2; i ++)`
`  if (abs(output[i]) > maxAbsOutput)`
`  maxAbsOutput = abs(output[i]);`
 	
`for (int i = 0; i < P*2; i+=2)`
`  y[i/2] = output[i] / maxAbsOutput;`
	`double maxOutput = 0;`
	`double minOutput = 0;`
	`for (int i = 0; i < P*2; i+=2) {`
	`  if (output[i] < minOutput)`
	`  minOutput = output[i];`
	`  else if (output[i] > maxOutput)`
	`  maxOutput = output[i];`
	`}`
	`double maxAbsOutput = max(maxOutput, abs(minOutput));`
	`/* Discard any imaginary components */`
	`for (int i = 0; i < P*2; i+=2)`
	`  y[i/2] = output[i] / maxAbsOutput;`

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	0.026	0.013	200.000%

7b – Minimize Array References

When the array is being processed, output[i] is accessed 4 times. As mentioned in optimization 5, array accesses take significantly more time to access than variables in most cases. I minimized accesses from 4 down to 1 which further improved this loop.

Code Before	Code After

`double maxOutput = 0;`
`double minOutput = 0;`
`for (int i = 0; i < P*2; i+=2) {`
`  if (output[i] < minOutput)`
`  minOutput = output[i];`
`  else if (output[i] > maxOutput)`
`  maxOutput = output[i];`
`}`
	`double maxOutput = 0;`
	`double minOutput = 0;`
	`double currOutput;`
	`for (int i = 0; i < P*2; i+=2) {`
	`  currOutput = output[i];`
	`  if (currOutput < minOutput)`
	`  minOutput = currOutput;`
	`  else if (currOutput > maxOutput)`
	`  maxOutput = currOutput;`
	`}`

	Before Optimization	After Optimization	Relative Efficiency
Timing (seconds)	0.013	0.012	108.333%

8 – Compiler Optimization -O3

I began optimizing with the compiler with the most powerful gcc-based optimization I was aware of, -O3. Since my regression tests passed, I went to find any more rigorous optimizations I can perform even though this already provided a major improvement.

9 – Compiler Optimization -Ofast

I learned that the flag -Ofast performs all the optimizations of level -O3 as well as other non-standard-compliant optimizations such as -ffast-math. This surprisingly passed by regression tests and provided a slight improvement over -O3. I was unfortunately not able to find any other optimizations that would improve my performance further using gcc.

Profiling Results

Methodology

A version of the code without the timings was used for profiling. The reason for this is because the number of timings changes per version. Doing this allows for profiling results to be more accurate and consistent.

Profiling was accomplished by using using GPROF. The output files shown below are the **sum of 10 tests**. This allows for more accurate and consistent timings as some of these timings are very marginal optimizations. The method used for doing this is outlined at https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_6.html

Summary of Results

	Average Time	Efficiency to Prior	Efficiency to Initial
0 – Baseline Program	719.761 seconds	N/A	N/A
1 – Algorithm Based Optimization	2.164 seconds	33260.675%	33260.67%
2 – Precompute Results	2.104 seconds	102.851%	34209.17%
2a – Exploit Algebraic Identities	2.103 seconds	100.047%	34225.44%
3 – Strength Reduction	2.079 seconds	101.154%	34620.54%
4 – Partial Unrolling	2.046 seconds	101.612%	35178.93%
5 – Minimize Array References	2.043 seconds	100.146%	35230.59%
6 – Caching	2.023 seconds	100.988%	35578.89%
7 – Jamming	2.017 seconds	100.297%	35684.73%
7a – Strength Reduction	1.995 seconds	101.102%	36078.25%
7b – Minimize Array References	1.985 seconds	100.503%	36260.00%
8 – Compiler Optimization -O3	0.980 seconds	202.551%	73445.00%
9 – Compiler Optimization -Ofast	0.970 seconds	101.031%	74202.16%

Final program runs **742.022 times faster** than baseline!