CPSC 501 Assignment 4 Report

Tyrone Lagore (10151950)

Due Friday December 9, 4:00PM

Code Optimization

Optimization 1:

Unrolling:

Unrolled the loop in convolve to do 3 iterations per loop.

Code before:

```
for(wav_index = 0; wav_index < w_size; wav_index++){
      for(ir_index = 0; ir_index < ir_size; ir_index++){
       output[wav_index + ir_index] += wav_data[wav_index] * ir_data[ir_index];
      }
```

Code after:

```
for(ir_index = 0; ir_index < ir_size - 2; ir_index+=3){
       output[wav_index + ir_index] += wav_data[wav_index] * ir_data[ir_index];
       output[wav_index + (ir_index + 1)] += wav_data[wav_index] * ir_data[ir_index + 1];
       output[wav_index + (ir_index + 2)] += wav_data[wav_index] * ir_data[ir_index + 2];
      }

      if(ir_index == (ir_size - 2)){
       output[wav_index + (ir_index - 2)] += wav_data[wav_index] * ir_data[ir_index - 2];
       output[wav_index + (ir_index - 1)] += wav_data[wav_index] * ir_data[ir_index - 1];
      }

      if(ir_index == (ir_size - 1)){
       output[wav_index + (ir_index - 1)] += wav_data[wav_index] * ir_data[ir_index - 1];
      }
```

Optimization 2:

Sentinel Values, Busy Loop:

As the impulse response is almost always smaller, I placed the wav input file loop on the inside of the convolve loop.

Code before:

```
for(wav_index = 0; wav_index < w_size; wav_index++){
   for(ir_index = 0; ir_index < ir_size - 2; ir_index+=3){
       output[wav_index + ir_index] += wav_data[wav_index] * ir_data[ir_index];
       output[wav_index + (ir_index + 1)] += wav_data[wav_index] * ir_data[ir_index + 1];
       output[wav_index + (ir_index + 2)] += wav_data[wav_index] * ir_data[ir_index + 2];
      }
      if(ir_index == (ir_size - 2)){
       output[wav_index + (ir_index - 2)] += wav_data[wav_index] * ir_data[ir_index - 2];
       output[wav_index + (ir_index - 1)] += wav_data[wav_index] * ir_data[ir_index - 1];
      }
      if(ir_index == (ir_size - 1)){
       output[wav_index + (ir_index - 1)] += wav_data[wav_index] * ir_data[ir_index - 1];
  }
```

Code After:

```
 for(ir_index = 0; ir_index < ir_size; ir_index++){
    for(wav_index = 0; wav_index < w_size; wav_index+=3)
       {
```

```
            output[wav_index + ir_index] += wav_data[wav_index] * ir_data[ir_index];
            output[wav_index + 1 + ir_index] += wav_data[wav_index + 1] * ir_data[ir_index];
            output[wav_index + 2 + ir_index] += wav_data[wav_index + 2] * ir_data[ir_index];
         }
        if(wav_index == (wav_size - 2)){
         output[wav_index + ir_index - 2] += wav_data[wav_index - 2] * ir_data[ir_index];
         output[wav_index + ir_index - 1] += wav_data[wav_index - 1] * ir_data[ir_index];
        }
        if(wav_index == (wav_size - 1)){
         output[wav_index + ir_index - 1] += wav_data[wav_index - 1] * ir_data[ir_index];
        }
```

Optimization 3:

Inline functions (C Macros):

I was frequently needing to obtain the nearest power of 2 of a number. So I rewrote it as a function:

Code before:

```
size = (1 << ((int)log2(length) + 1));
```

Code after:

```
#define NEAREST_POW2(a) (1 << ((int)log2(a) + 1))
```

Now called like:

```
size = NEAREST_POW2(length);
```

Optimization 4:

Minimizing array references:

I noticed in the inner loop of convolve, the ir value at ir_data[ir_index] was static, but referenced 5 times.

Code before:

```
  for(ir_index = 0; ir_index < ir_size; ir_index++){
        for(wav_index = 0; wav_index < w_size; wav_index+=3)
         {
            output[wav_index + ir_index] += wav_data[wav_index] * ir_data[ir_index];
            output[wav_index + 1 + ir_index] += wav_data[wav_index + 1] * ir_data[ir_index];
            output[wav_index + 2 + ir_index] += wav_data[wav_index + 2] * ir_data[ir_index];
         }

        if(wav_index == (wav_size - 2)){
         output[wav_index + ir_index - 2] += wav_data[wav_index - 2] * ir_data[ir_index];
         output[wav_index + ir_index - 1] += wav_data[wav_index - 1] * ir_data[ir_index];
        }

        if(wav_index == (wav_size - 1)){
         output[wav_index + ir_index - 1] += wav_data[wav_index - 1] * ir_data[ir_index];
        }
```

Code after:

```
float ir_val;
for(ir_index = 0; ir_index < ir_size; ir_index++){
      ir_val = ir_data[ir_index];
      for(wav_index = 0; wav_index < w_size; wav_index+=3)
       {
          output[wav_index + ir_index] += wav_data[wav_index] * ir_val;
          output[wav_index + 1 + ir_index] += wav_data[wav_index + 1] * ir_val;
```

```
        output[wav_index + 2 + ir_index] += wav_data[wav_index + 2] * ir_val;
       }

      if(wav_index == w_size - 2)){
       output[wav_index + ir_index - 2] += wav_data[wav_index - 2] * ir_val;
       output[wav_index + ir_index - 1] += wav_data[wav_index - 1] * ir_val;
      }

      if(wav_index == (w_size - 1)){
       output[wav_index + ir_index - 1] += wav_data[wav_index - 1] * ir_val;
      }
```

Optimization 5:

Data transformation:

In the function floatArrayToShort, a multiplier is handed in. This multiplier is a float by the function definition, but will never be larger than a maximum sized short (32767). The value is also stored in a short.

Code before:

```
short* floatArrToShort(float* arr, unsigned int *out_bytes, unsigned int size, float
multiplier){
  int i;
  short *output;
  //ensure we are short aligned. Also need half the number of bytes for short
  (*out_bytes) = (size + size % BYTES_SHORT) / 2;
  output = malloc(*out_bytes);
  if(output != NULL){
    for (i = 0; i < (size / BYTES_FLOAT); i++){
      output[i] = (short)(arr[i] * multiplier);
    }
  }else{
    (*out_bytes) = 0;
  }
  return output;
}
```

Code after:

```
short* floatArrToShort(float* arr, unsigned int *out_bytes, unsigned int size, float
multiplier){
  int i;
  short *output;
  //ensure we are short aligned. Also need half the number of bytes for short
  (*out_bytes) = (size + size % BYTES_SHORT) / 2;
  output = malloc(*out_bytes);
  short intMultiplier = (short)multiplier;
  if(output != NULL){
    for (i = 0; i < (size / BYTES_FLOAT); i++){
      output[i] = (short)(arr[i] * intMultiplier);
    }
  }else{
    (*out_bytes) = 0;
  return output;
}
```

Optimization Test results:
Time domain optimizations:

Times improved steadily over the course of the optimizations.

Original Time -O1 Time domain

Time %	Cumulative S	Self seconds	function
100.13	498.26	498.26	convolve
0	498.27	0.01	normalizeArray
0	498.28	0.01	saveOutput
0	498.28	0	write_little_endian
0	498.28	0	getHeaderInfo
0	498.28	0	getWavData
0	498.28	0	shortArrToFloat
0	498.28	0	cleanup
0	498.28	0	floatArrToShort
0	498.28	0	getMaxElementFloat

First optimization (with -O1)

Time %	Cumulative S	Self seconds	function
100.11	456.78	456.78	convolve
0	456.79	0.01	write_little_endian
0	456.8	0.01	shortArrToFloat
0	456.81	0.01	normalizeArray
0	456.82	0.01	saveOutput
0	456.82	0	getHeaderInfo
0	456.82	0	getWavData
0	456.82	0	cleanup
0	456.82	0	floatArrToShort
0	456.82	0	getMaxElementFloat

Second optimization (with -O1) Time domain

Time %	Cumulative S	Self seconds	function
99.75	438.8	438.8	convolve
0	438.82	0.02	write_little_endian
0	438.83	0.01	floatArrToShort
0	438.84	0.01	getMaxElementFloat
0	438.84	0	getHeaderInfo
0	438.84	0	getWavData
0	438.84	0	shortArrToFloat
0	438.84	0	cleanup
0	438.84	0	normalizeArray
0	438.84	0	saveOutput

Fourth optimization (with -O1) Time domain

Time %	Cumulative S	Self seconds	function
100.11	402.87	402.87	convolve
0	402.88	0.01	floatArrToShort
0	402.89	0.01	getMaxElementFloat
0	402.9	0.01	saveOutput
0	402.9	0	write_little_endian
0	402.9	0	getHeaderInfo
0	402.9	0	getWavData
0	402.9	0	shortArrToFloat
0	402.9	0	cleanup
0	402.9	0	normalizeArray

Fifth optimization (with -O1) Time domain

Time %	Cumulative S	Self seconds	function
100.11	395.99	395.99	convolve
0	396	0.01	floatArrToShort
0	396.01	0.01	getMaxElementFloat
0	396.02	0.01	saveOutput
0	396.03	0	write_little_endian
0	396.04	0	getHeaderInfo
0	396.05	0	getWavData
0	396.06	0	shortArrToFloat
0	396.07	0	cleanup
0	396.08	0	normalizeArray

Frequency domain optimization

No noticeable change, but frequency-domain was already quite quick.

Original Frequency (-O1)

Time %	Cumulative S	Self seconds	function
95.38	2.02	2.02	fft
0.94	2.04	0.02	shortArrToDouble
0.47	2.05	0.01	getMaxElementDouble
0.47	2.06	0.01	getMinElement
0.47	2.07	0.01	multiplyComplex
0.47	2.08	0.01	normalizeArray
0.47	2.09	0.01	postprocessData
0.47	2.1	0.01	preprocessData
0.47	2.11	0.01	saveOutput
0.47	2.12	0.01	write_little_endian

Third optimization

Time %	Cumulative S	Self seconds	function
95.38	2.02	2.02	fft
0.94	2.04	0.02	shortArrToDouble
0.47	2.05	0.01	getMaxElementDouble
0.47	2.06	0.01	getMinElement
0.47	2.07	0.01	multiplyComplex
0.47	2.08	0.01	normalizeArray
0.47	2.09	0.01	postprocessData
0.47	2.1	0.01	preprocessData
0.47	2.11	0.01	saveOutput
0.47	2.12	0.01	write_little_endian

GPROF compiler level optimizations in separate document “compiler_level_optimizations.pdf”

Refactoring Catalog

Refactor 1:

What needed to be improved? (Bad code smell)

The bad code smell here was long method. Main was doing everything, including writing everything to file.

Mechanics

- Duplicated code was moved into a new properly named method (saveOutput)
- Extraneous references were resolved
- References from old method calls were updated to call the new function
- Compiled and tested

Illustration

```
int main(int argc, char **argv){

       ...
max = getMaxElementFloat(foutput, fout_bytes / 4);
  if (max > 1){
    if(_Debug)
      printf("Max element in output array before conversion was %f, normalizing..\n", max);

    normalizeArray(foutput, fout_bytes / 4, max);
  }

  output = floatArrToShort(foutput, &out_bytes, fout_bytes, SHORT_MULTIPLIER);
  if(_Debug == TRUE)
    printf("Float output bytes: %u\nShort output bytes: %u\nExpected output bytes: %u\n",
          fout_bytes, out_bytes, fout_bytes / 2);

  fp = fopen(out_file_str ,"w+");

  fwrite("RIFF", 4, BYTE, fp);
  iBuffer = out_bytes - 36;
  write_little_endian(iBuffer, BYTES_INT, fp);
  //fwrite(&iBuffer, BYTES_INT, BYTE, fp);

  fwrite("WAVE", 4, BYTE, fp);
  fwrite("fmt ", 4, BYTE, fp);

  //data chunk is 16 bytes long
  iBuffer = 16;
      … //continues on
}
```

Changed to

```
void saveOutput(char *out_file_str, float *foutput, unsigned int fout_bytes,
             struct WavHeader wav_header){
  unsigned int out_bytes;
  short *output;
  FILE *fp;
  int iBuffer;
  int i;
  float max;

  max = getMaxElementFloat(foutput, fout_bytes / 4);
  if (max > 1){
    if(_Debug)
      printf("Max element in output array before conversion was %f, normalizing..\n", max);

    normalizeArray(foutput, fout_bytes / 4, max);
  }

  output = floatArrToShort(foutput, &out_bytes, fout_bytes, SHORT_MULTIPLIER);
  if(_Debug == TRUE)
    printf("Float output bytes: %u\nShort output bytes: %u\nExpected output bytes: %u\n",
          fout_bytes, out_bytes, fout_bytes / 2);

  fp = fopen(out_file_str ,"w+");

  fwrite("RIFF", 4, BYTE, fp);
  iBuffer = out_bytes - 36;
  write_little_endian(iBuffer, BYTES_INT, fp);
  //fwrite(&iBuffer, BYTES_INT, BYTE, fp);

  fwrite("WAVE", 4, BYTE, fp);
  fwrite("fmt ", 4, BYTE, fp);

  //data chunk is 16 bytes long
  iBuffer = 16;
  fwrite(&iBuffer, BYTES_INT, BYTE, fp);
  //same format as header
  write_little_endian(wav_header.format_type, sizeof(wav_header.format_type), fp);
  write_little_endian(wav_header.num_channels, sizeof(wav_header.num_channels), fp);
  write_little_endian(wav_header.sample_rate, sizeof(wav_header.sample_rate), fp);
  write_little_endian(wav_header.byte_rate, sizeof(wav_header.byte_rate), fp);
  write_little_endian(wav_header.block_alignment, sizeof(wav_header.block_alignment),
fp);
  write_little_endian(wav_header.bits_per_sample, sizeof(wav_header.bits_per_sample),
fp);

  fwrite("data", 4, BYTE, fp);
  write_little_endian(out_bytes, BYTES_INT, fp);
```

```
  if(_Debug == TRUE)
    printf("Header data printed to %s\n", out_file_str);

  printf("outbytes before write %u\n", out_bytes);

  for(i = 0; i < out_bytes / 2; i++)
    write_little_endian((unsigned int)(output[i]), sizeof(wav_header.block_alignment),
fp);

  if(_Debug == TRUE)
    printf("Convoluted sample data written to file.\n");

  free(output);
  fclose(fp);
}
```

How was the code tested?

Several iterations of the program were run to ensure that they were still writing to output properly.

Why is the code better structured after the refactoring? Does the result of the refactoring suggest or enable further refactoring?

The function offers portability and makes the code more readable. The result does suggest more refactoring in the future as saveOuptut may belong more with a wav file library in the future. It is a little too hard coded to this exact implementation. It would be nice if the function was more generic and could be called from more contexts.

Notes:

I only implemented single channel convolution.

There were more refactorings, but I ran short on time and was not able to fully document them. Among them, I relocated code to new libraries under "../shared" that both versions of the convolution reference. Additionally I created a new file called utils, moved a few of the utility functions to this file and referenced them by including the c files.

It should also be noted that the **frequency convolution version of the program does not produce proper output (produces a wav file of purely static sound).** However, as the program is executing all the code – I have assumed the times represented by the frequency version of the program are representative of it's performance speed were it producing output properly and have timed the program as such.

The time domain program produces proper results.