Tristan Elma
Student ID: 1093937234
EE 451 Project Report

Group members: Tristan Elma (independent)
Professor Xuehai Qian

Artificial Reverberation through Parallelized Direct Convolution

Abstract: I analyze the parallelization of direct convolution algorithms which produce artificial reverberation effects on inputs of audio data, as determined by impulse responses.

Introduction / Background

Reverberation, also known as *reverb*, is a sound effect used in music and sound engineering which simulates the way sounds reflect within rooms to create echoes. Sounds reflect off of walls, floors, and other objects in complex ways, and the reflections interact with each other as well as the original source of the sound. What reaches a listener's ear may be the sum of different sources, including the original sound and some delayed components of the sound. In digital signal processing, it is possible to simulate this effect using filters.

The *Tapped Delay Line filter*, when implemented with no feedback loops, is a type of Finite Impulse Response (FIR) filter, in which the output signal will eventually settle to 0 in a finite amount of time, given a finite-length input. The Tapped Delay Line filter is suitable for creating artificial reverberation.

Tapped Delay Line filters (see Figure 1) work by extracting input samples at various delays (taps), modifying the amplitudes of each tap by multiplying them with weights as described by the *impulse response* (also known as the *transfer function*), and summing them together to generate the output. The presence of the delayed samples in the final output simulates sound reflections in a physical space, producing the effect of reverberation. Feeding the input through the delay line is equivalent to the convolution of the input with the impulse response. For this reason, this method of reverberation is known as *convolution reverb*.

In convolution reverb, the impulse response is the sequence of weights which simulates the acoustic characteristics of physical spaces. In practice, impulse responses can be directly recorded by playing extremely short sounds in reverberant spaces and recording the echoes. With the impulse responses, the reverberation effects of the spaces can be artificially applied to any input sound through the convolution of the input with the impulse response.

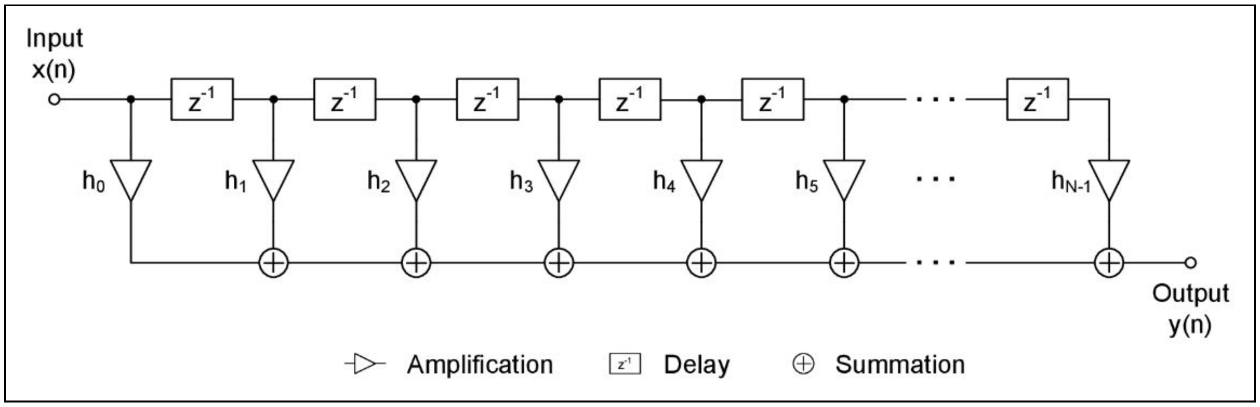


Fig. 1: Tapped Delay Line filter. x(n) represents the input data, and h(n) represents the impulse response data [2].

Existing Algorithms and Previous Work

Digital signal processing encompasses many theories and techniques that can be applied to signal-based applications such as reverb. Because of this, there are infinitely many ways to produce reverb, with each one navigating the tradeoff of quality and performance.

One popular reverberation algorithm is known as *Schroeder reverb*, which uses different filters called *comb filters* and *all-pass filters* instead of Tapped Delay Line filters. These filters provide better-sounding reverberation effects. There is a famous open-source implementation of artificial reverberation based on Schroeder reverb called *Freeverb*, which has very high sound quality. However, Shroeder reverb variants are more complex, and they are not as easily parallelizable as the direct convolution / Tapped Delay Line method.

There is also a technique which achieves discrete convolution (and hence the tapped delay line effect) using circular convolution [2]. The Circular Convolution Property states that circular convolution in the time domain equals the pairwise product of spectral coefficients in the frequency domain. Along with the Convolution Theorem, fourier transforms can be used to convert the input and impulse response function into the frequency domain, take the pairwise products, and convert them back into the time domain to obtain the output. This method is significantly faster, and in practice, it is used almost exclusively over direct convolution. However, for this project, I've implemented **direct convolution** (keeping everything in the time domain). It's quite simple, and far from the best method available, but I chose it because it is so computationally expensive that it's typically infeasible in sequential implementations, but its performance can be improved through parallelization. I hoped that by parallelizing it, I could make direct convolution a viable option for some input cases.

Problem Description and Implementations

Terminology

p := Number of available processors
x := The input, which is a time series of audio samples
M := Number of samples in the input x
h := The sequence of coefficients or weights associated with each tap, also known as the impulse response (this is supplied by another audio file)
N := Number of taps (the number of samples in the impulse response audio file)
y := The output, a time series of audio samples
Note: the number of output samples is M+N-1.

Direct, Discrete Convolution

To achieve artificial reverberation, we must convolve the input x(n) consisting of M samples and the transfer function h(n) (also known as the impulse response) consisting of N taps (or samples) to produce the output y(n). The general implementation of this is as follows: In each iteration *k*, input sample x(*k*) enters the Tapped Delay Line and is paired with tap h(0), and samples already in the Tapped Delay Line propagate one step to be paired with the next tap. Then, output sample y(k) is calculated. The value y(k) is the sum of N products, namely the product of each tap coefficient and the input sample with which it is currently paired. Specifically, we see that

$$y(k) \;=\; \sum_{j=0}^{N-1} h(j) * x(k-j)\ .$$

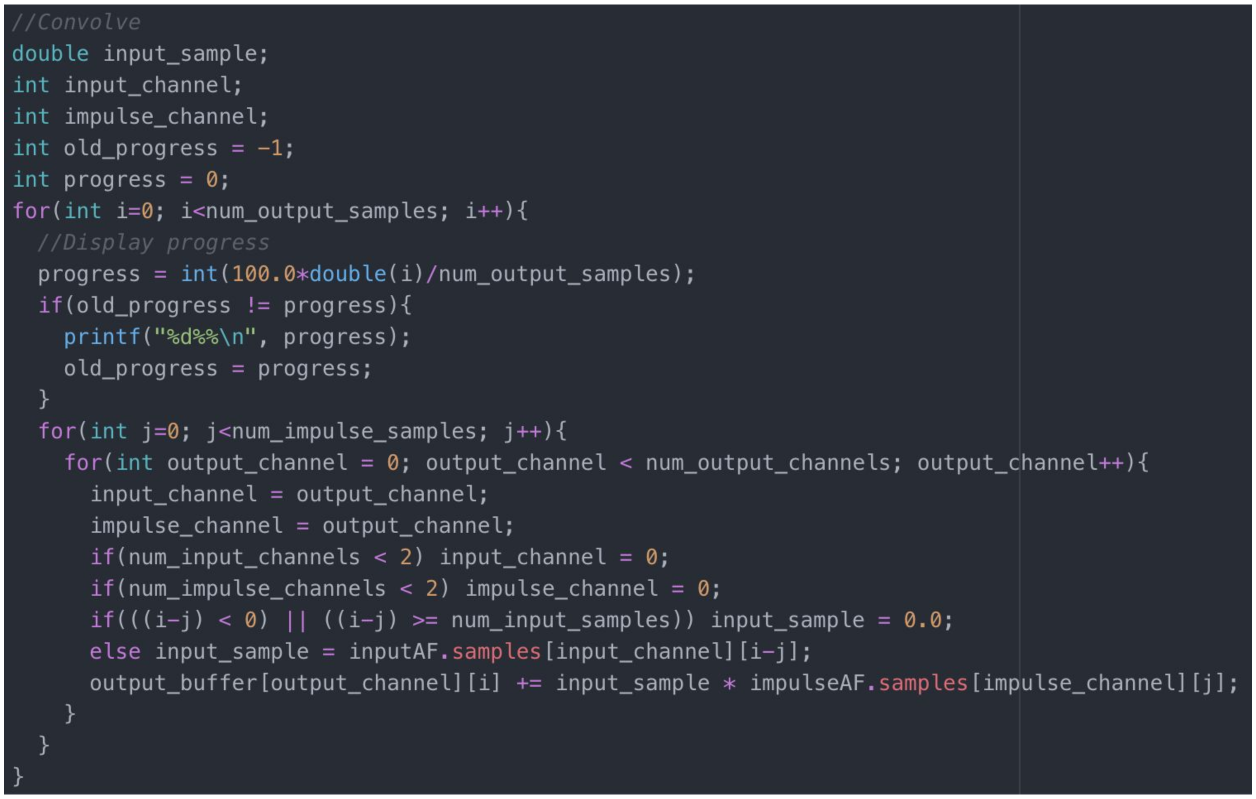
However, since there are M input samples, and the length of the Tapped Delay Line is N, the last input sample will finish propagating at iteration M+N-1, and thus M+N-1 output samples are produced. Therefore, in the first N iterations, there are taps in the Tapped Delay Line which have not yet received an input sample x(i) because none have propagated to them. In the equation, this arises when j>k. Similarly, in the final N iterations, there are taps which are not paired with input samples because the last sample has begun propagating through and no further samples have entered behind it. In the equation, this arises when (k-j)>(M-1). In these cases, the lack of a sample is represented by a 0. Thus we can define padded input x'(i) := {x(i) if 0<=i<M, 0 otherwise}. With this, we can modify the above expression to account for zero-padding as follows:

$$y(k) \; = \; \sum_{j=0}^{N-1} h(j) * x'(k-j) \, .$$

Algorithms

Since the implementation of the Tapped Delay Line Filter involves iterated summation, and since each output is independent of one another, there is an opportunity to parallelize the computation among separate processing units. For this project, I analyze the performance of three different methods/algorithms that implement direct convolution. The first is a single-processor sequential algorithm, while the second and third are parallel algorithms.

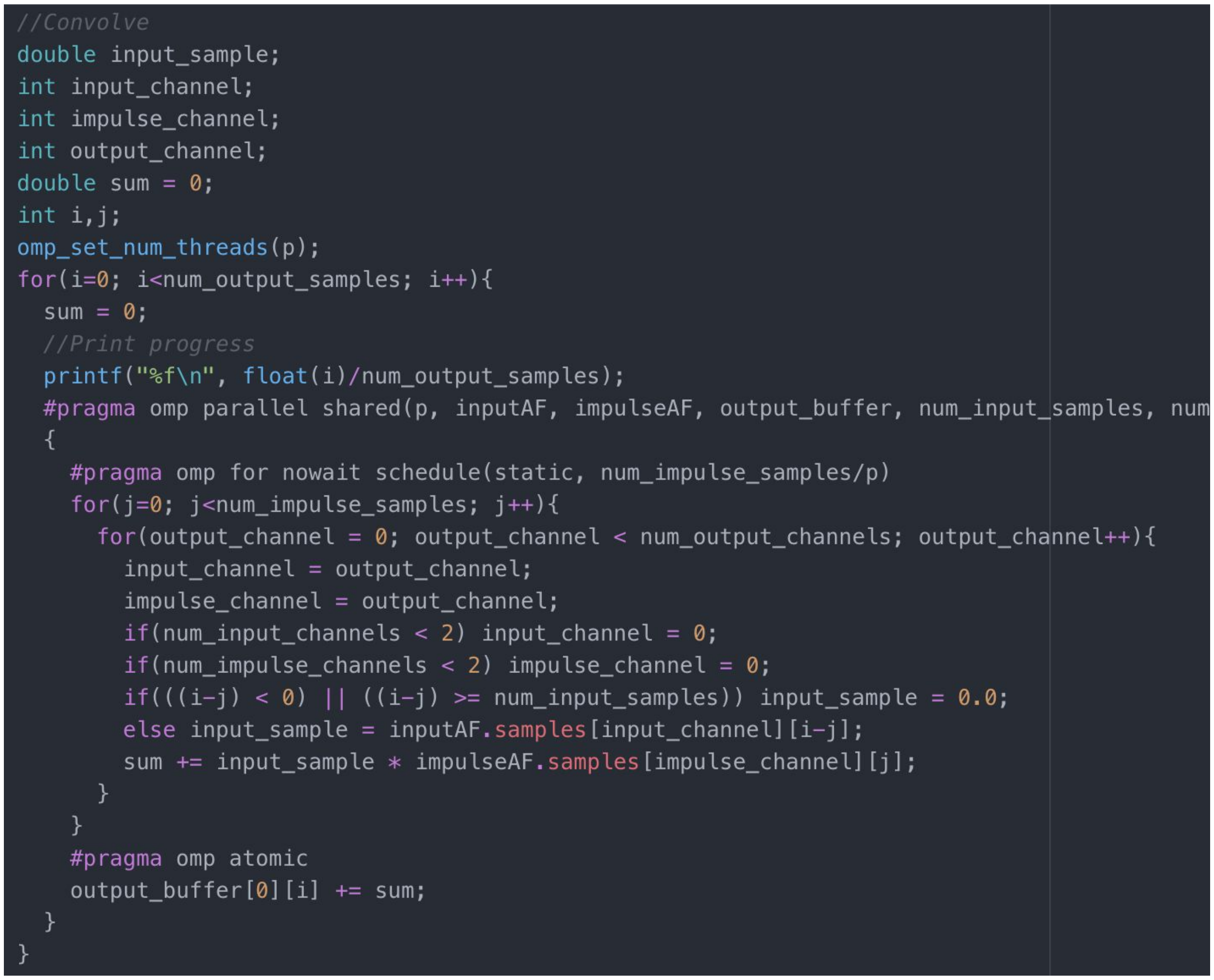
1. Single-processor sequential algorithm
    - Each y(k) of the M+N-1 output samples is calculated in sequential iterations, with each calculation involving N multiply-adds. This gives an overall runtime of O(N*(M+N-1)) = O(NM+N^2). If N>=M, then the runtime is O(N^2). If N<M, the runtime is O(NM). Typically, N<<M, so O(NM) is perhaps more appropriate.
    - Code snippet:



```
//Convolve
double input_sample;
int input_channel;
int impulse_channel;
int old_progress = -1;
int progress = 0;
for(int i=0; i<num_output_samples; i++){
  //Display progress
  progress = int(100.0*double(i)/num_output_samples);
  if(old_progress != progress){
    printf("%d%%\n", progress);
    old_progress = progress;
  }
  for(int j=0; j<num_impulse_samples; j++){
    for(int output_channel = 0; output_channel < num_output_channels; output_channel++){
      input_channel = output_channel;
      impulse_channel = output_channel;
      if(num_input_channels < 2) input_channel = 0;
      if(num_impulse_channels < 2) impulse_channel = 0;
      if(((i-j) < 0) || ((i-j) >= num_input_samples)) input_sample = 0.0;
      else input_sample = inputAF.samples[input_channel][i-j];
      output_buffer[output_channel][i] += input_sample * impulseAF.samples[impulse_channel][j];
    }
  }
}
```

2. Data parallelism of the impulse response with OpenMP
    - This algorithm parallelizes the computation of *each individual output* y(k) among p processors. The *sequence* of outputs y(0)...y(N+M-1) are still computed one after another in the same manner as the sequential algorithm. But in this algorithm, the multiply-adds involved in calculating each *individual* y(k) are executed in parallel.
    - In iteration k, the algorithm computes one output y(k). y(k) is calculated as the sum of the products of the input samples currently in the delay line and the impulse response values currently paired with each input sample. To divide the work among processors, each processor is assigned to a chunk of (N/p) impulse response values and computes a local sum of the (N/p) products of inputs and impulse response values. Then, at the end of each iteration, the local sums are added together to compute the value y(k).
    - Because y(k) is a shared variable, and because all processors compute part of the same y(k) simultaneously, the update to y(k) must be done as an atomic (or sequential) operation to eliminate the possibility of a data race. The local sum variables ensure that only p atomic operations are performed in each iteration, but this still presents a significant source of slowdown because it is incurred in each iteration, of which there are M+N-1.

    - The time complexity of this algorithm is still the same order as the sequential solution O(MN+N^2), but it has the potential to have a maximum speedup of a constant *p*.
    - This algorithm can be seen as implementing an weight-stationary data flow, as each PE receives all the input activations, while the weights (from impulse response data) are partitioned and distributed among the PE's, and the PE's must accumulate their sums before producing each output. Thus this algorithm maximizes the local reuse of weights.
    - Code snippet:

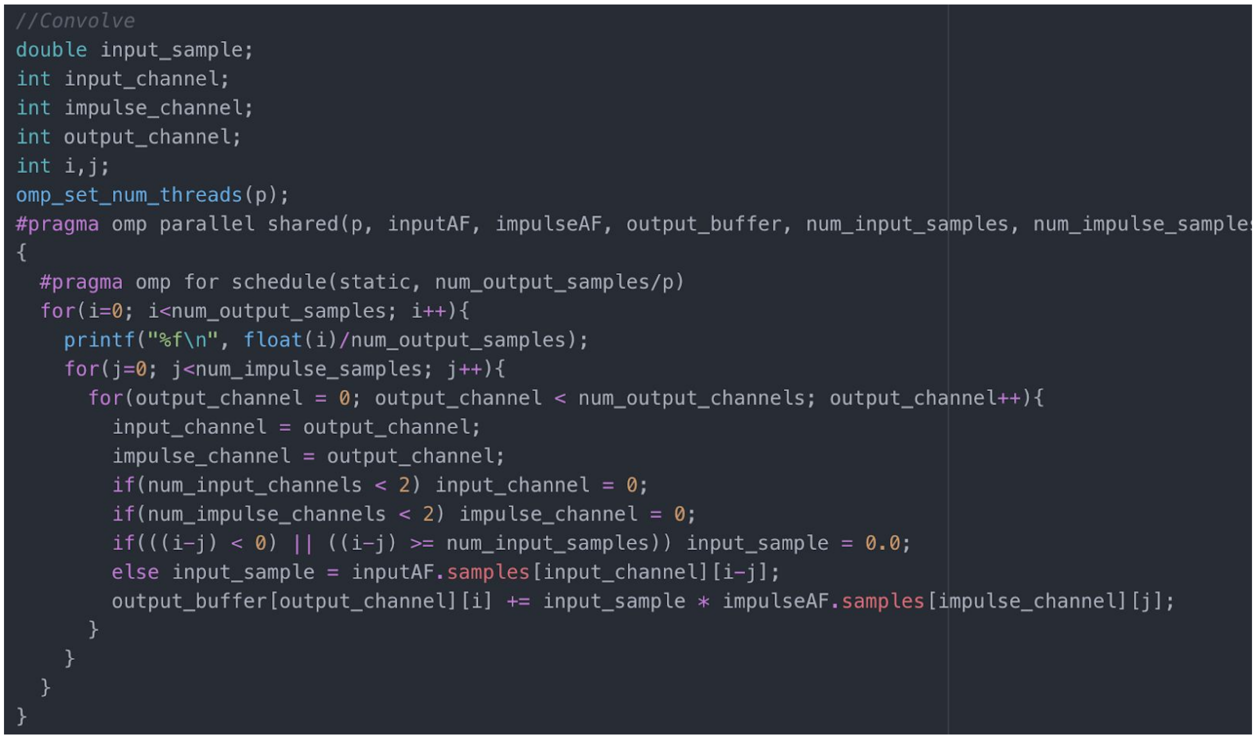


```
//Convolve
double input_sample;
int input_channel;
int impulse_channel;
int output_channel;
double sum = 0;
int i,j;
omp_set_num_threads(p);
for(i=0; i<num_output_samples; i++){
  sum = 0;
  //Print progress
  printf("%f\n", float(i)/num_output_samples);
  #pragma omp parallel shared(p, inputAF, impulseAF, output_buffer, num_input_samples, num
  {
    #pragma omp for nowait schedule(static, num_impulse_samples/p)
    for(j=0; j<num_impulse_samples; j++){
      for(output_channel = 0; output_channel < num_output_channels; output_channel++){
        input_channel = output_channel;
        impulse_channel = output_channel;
        if(num_input_channels < 2) input_channel = 0;
        if(num_impulse_channels < 2) impulse_channel = 0;
        if(((i-j) < 0) || ((i-j) >= num_input_samples)) input_sample = 0.0;
        else input_sample = inputAF.samples[input_channel][i-j];
        sum += input_sample * impulseAF.samples[impulse_channel][j];
      }
    }
    #pragma omp atomic
    output_buffer[0][i] += sum;
  }
}
```

3. Data parallelism of the output with OpenMP
    - This solution uses the same sequential multiply-adds as the sequential solution, but in this case, output samples are computed in parallel, with the outputs partitioned among the processors. This makes different subprocesses responsible for different samples of the output. Specifically, whereas one processor was responsible for computing all outputs in the sequential solution, in this algorithm, subprocess *k*, where 0 <= k < M+N-1, is

used to calculate only one sample of the output, namely y(k). To compute y(k), the subprocess sequentially performs N multiply-adds, the same method as in the sequential solution.

- Unlike the impulse-parallel solution, which partitioned the N taps among the *p* processors, this method partitions the M+N-1 output samples among the *p* processors. Thus, each processor computes (M+N-1)/p samples of the output.
- The time complexity of this algorithm is still the same order as the sequential solution O(MN+N^2), but it has the potential to have a maximum speedup of a constant *p*. While the first parallel solution (impulse parallelism) has speedup based on the N multiply-adds involved in calculating every output, this parallel solution (output parallelism) parallelizes the computation of the sequence of M+N-1 outputs, but calculates each individual output with N sequential multiply-adds. So, depending on the input size and the impulse size, either of these solutions could perform better.
- This algorithm can be seen as implementing an output stationary data flow, as the weights (from impulse response data) are replicated in each PE, the input data goes through all the PE's, and the PE's accumulate local sums corresponding to different outputs. Thus this algorithm maximizes local accumulation, and since the data written by the PE's are at different indices, there is no need to handle a data race.
- Code snippet:



```
//Convolve
double input_sample;
int input_channel;
int impulse_channel;
int output_channel;
int i,j;
omp_set_num_threads(p);
#pragma omp parallel shared(p, inputAF, impulseAF, output_buffer, num_input_samples, num_impulse_sample
{
  #pragma omp for schedule(static, num_output_samples/p)
  for(i=0; i<num_output_samples; i++){
    printf("%f\n", float(i)/num_output_samples);
    for(j=0; j<num_impulse_samples; j++){
      for(output_channel = 0; output_channel < num_output_channels; output_channel++){
        input_channel = output_channel;
        impulse_channel = output_channel;
        if(num_input_channels < 2) input_channel = 0;
        if(num_impulse_channels < 2) impulse_channel = 0;
        if(((i-j) < 0) || ((i-j) >= num_input_samples)) input_sample = 0.0;
        else input_sample = inputAF.samples[input_channel][i-j];
        output_buffer[output_channel][i] += input_sample * impulseAF.samples[impulse_channel][j];
      }
    }
  }
}
```

Note: While I did not test an input-stationary or input-parallel algorithm to fully explore the 1-D convolution design space, I believe that such an algorithm would perform very similarly to the impulse-parallel or weight-stationary algorithm. An input-stationary algorithm would flip the order of the convolution from x(n) * h(n) to h(n) * x(n), which has the same outcome due to the commutativity of convolution. But in the code for an input-stationary data flow, the parallel region and the parallel for-loop would still be declared within the first for-loop as it is in the weight-stationary version, so the parallel overhead (discussed further in the Evaluation section) is similar to the weight-stationary. In this way, I discovered that input-stationary and weight-stationary implementations suffer from similar advantages and flaws in code design, which makes sense since weights and input activations are only nominally different; they are ultimately interchangeable in convolution. On the other hand, output-stationary implementations have the unique advantage of avoiding data races, thus allowing for atomic writes to the output to be omitted.

Evaluation

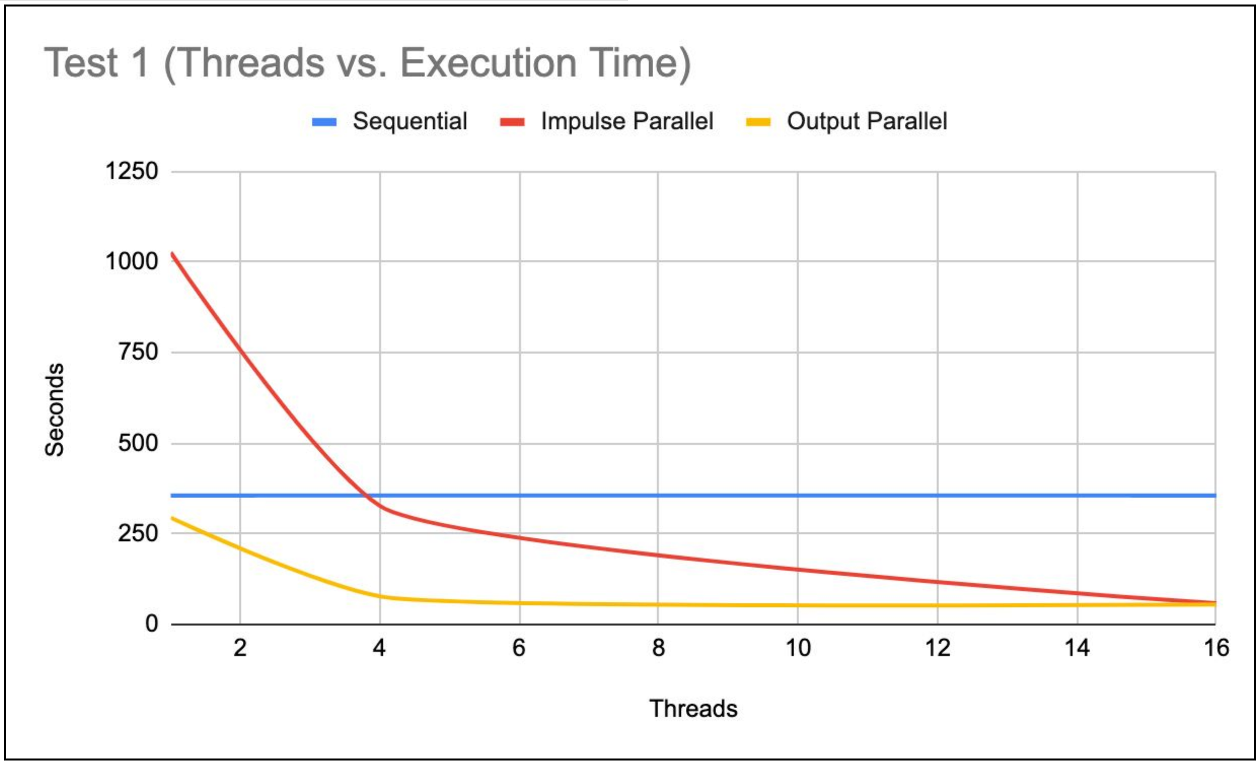
For my dataset, I have recorded my own input audio files and obtained high-quality impulse response files to ensure that the reverberation works as desired. This also allowed me to change the sizes of the input and impulse response files to demonstrate performance in different cases. In addition to varying the input data, I could also affect the performance by altering the number of threads used in execution of the parallel algorithms. Below, I show the performance metrics for a number of different input cases, using 1, 4, and 16 threads in each test:

Test 1: Long input, moderate-length impulse response
Input: “speech_30sec_mono.wav”
Impulse response: “bottle_hall_mono.wav”

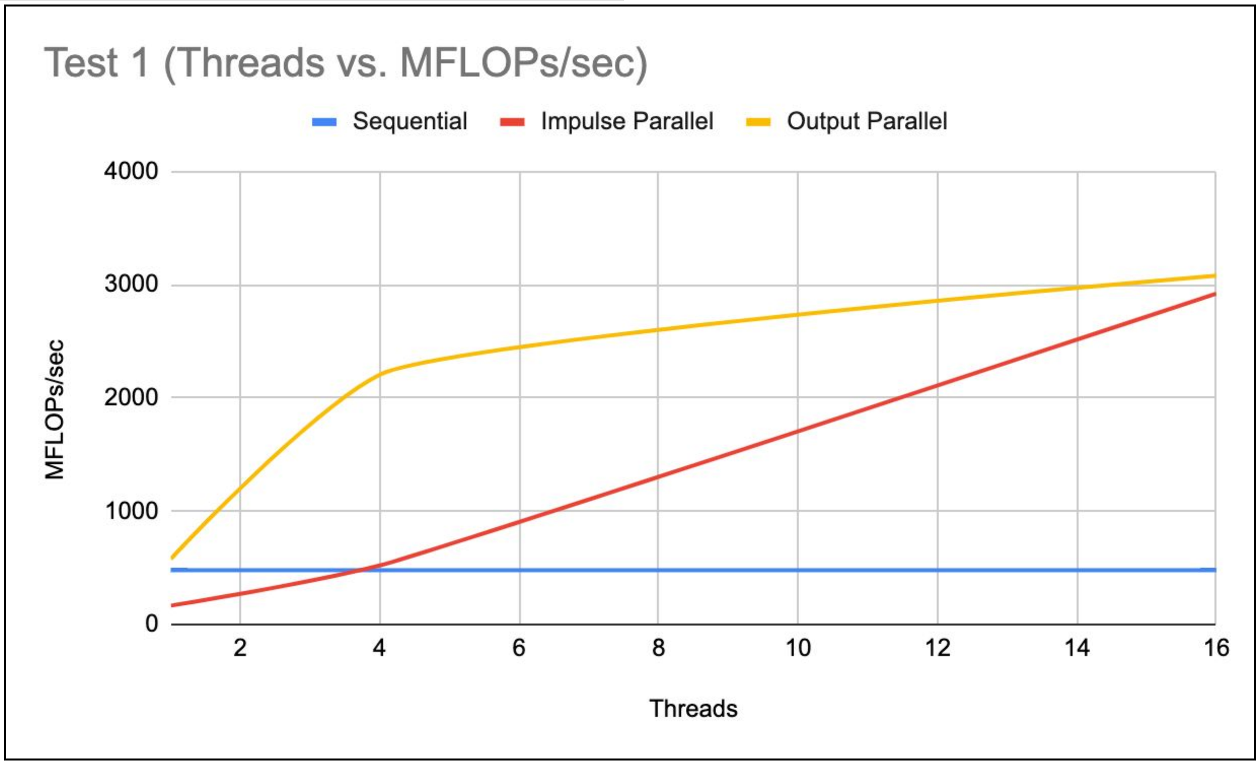
Analysis: This test was designed to be a general stress test to see how the algorithms perform with both a longer input and impulse response. The results turned out as expected in that the parallel algorithms outperformed the sequential algorithms as the number of threads increased. For 4 threads, the output-parallel algorithm even had linear speedup over the sequential solution. For 16 threads, though there was speedup, neither parallel algorithm yielded results close to linear speedup, due to overhead. However, it was interesting to see the differences in parallel overhead between the two parallel algorithms. Particularly, it was clear that the impulse parallel algorithm had much worse parallel overhead, simply by viewing the performance with 1 thread. Even when the impulse parallel algorithm is run on only 1 thread, the parallel region is still declared in each iteration of the outer for-loop, meaning the parallel overhead depends on the size of the output. The output parallel algorithm, however, only incurs the parallel overhead of the parallel region declaration and the parallel for-loop declaration once, when the convolution begins.

Results:

Test 1: Threads vs. Execution Time			
Threads	Sequential	Impulse Parallel	Output Parallel
1	355.89	1026.79	294.38
4	-	327.01	77.28
16	-	58.35	55.36



Test 1: Threads vs. MFLOPs/sec			
Threads	Sequential	Impulse Parallel	Output Parallel
1	479.12	166.06	579.22
4	-	521.41	2206.18
16	-	2921.92	3079.88

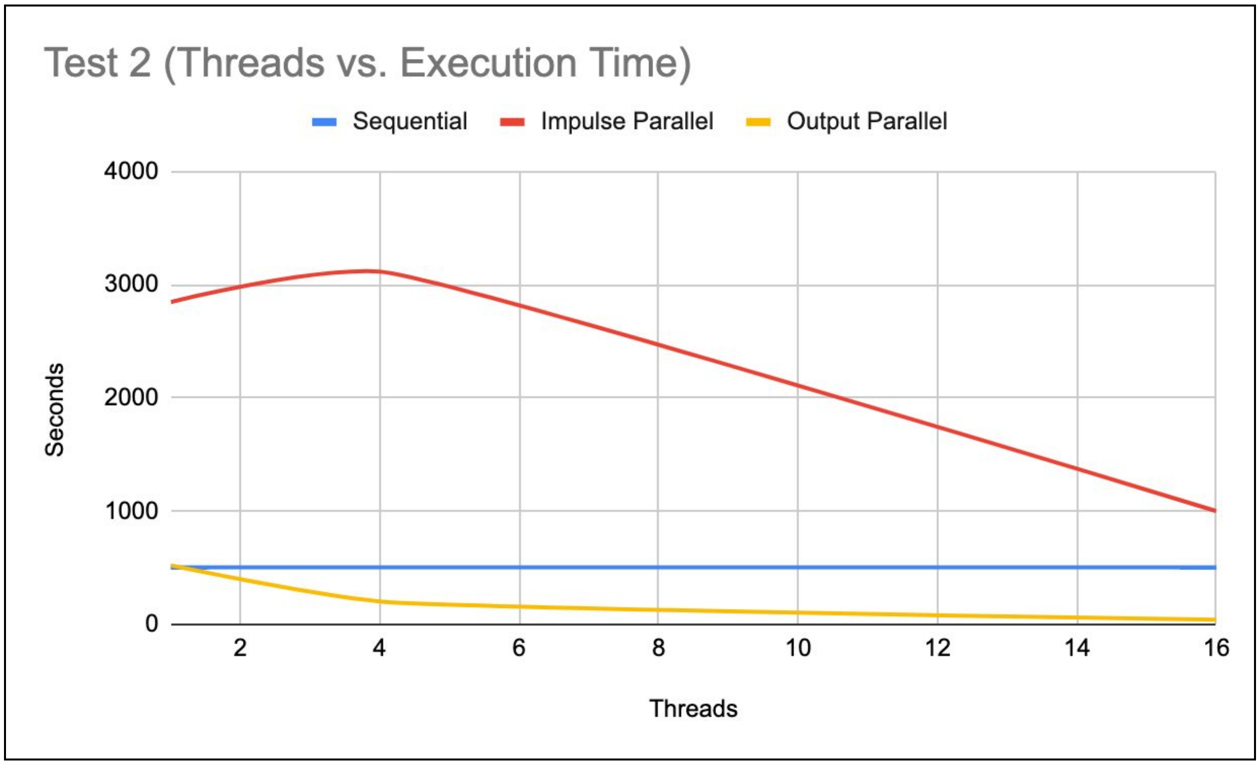


Test 2: Short input, long impulse response
Input: “speech_4sec_mono.wav”
Impulse response: “church_mono.wav”

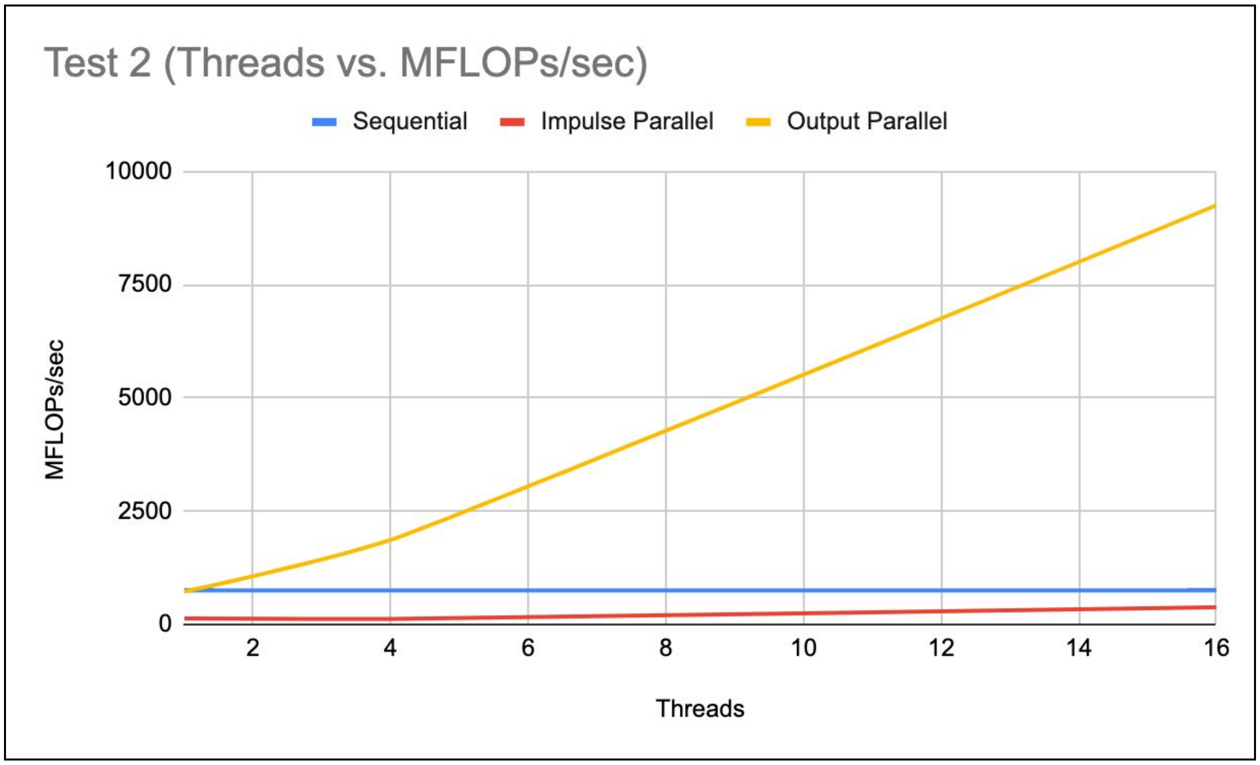
Analysis: This test was designed to provide a significantly longer impulse response and shorter input file. Initially, I expected this test case to demonstrate the strength of the impulse-parallel algorithm over the output-parallel algorithm because the impulse length would be quite large, and the impulse-parallel algorithm would partition this among the different processors. However, the results contrasted my expectations. In fact, the impulse-parallel algorithm ran much more slowly than both of the other algorithms, and failed to execute more quickly than the sequential solution even when run with 16 threads, while the output-parallel algorithm performed well. I ultimately determined that this can be explained by load imbalance. In the impulse-parallel algorithm, each processor computes a partial sum which must be combined to produce a single output sample. Because these processors cannot all write to the same memory location, they must maintain local sums and update the output sequentially through an atomic operation. Because each processor must contribute its partial sum to the total sum before all the processors can move to the next output, this creates a barrier; each processor must finish its local sum before any processor can move on. This means that if one processor takes a very long time to execute, the other processors must wait for that one, creating a load imbalance. And with a very long impulse response, each processor is responsible for much more computation in this impulse-parallel algorithm, so the load imbalance problem is exacerbated by the large size of the impulse response. I verified that this was the problem firstly by observing the execution pattern of the different processors. I noticed that in each iteration, most of the processors would finish executing quickly (a few milliseconds), then one or a few processors would take significantly more time to finish executing (on the order of seconds). To alleviate this, I changed the scheduling method of the parallel for-loop from static (allocating data in equal-sized chunks) to dynamic (data is allocated to processors as they become available). This way, I got a much more smooth and uniform execution pattern among the processors, which eliminated the load imbalance but increased the overhead (as dynamic scheduling is less efficient), so the algorithm still did not perform very well. On the other hand, the output-parallel algorithm does not have a load imbalance problem in the first place because the processors do not have to wait for each other to finish during each iteration, as they act on separate pieces of the output. Thus, it performed well with multithreading.

Results:

Test 2: Threads vs. Execution Time			
Threads	Sequential	Impulse Parallel	Output Parallel
1	504.18	2846.36	522.11
4	-	3115.22	202.75
16	-	1000.33	40.92



Test 2: Threads vs. MFLOPs/sec			
Threads	Sequential	Impulse Parallel	Output Parallel
1	750.89	133.01	725.1
4	-	121.53	1867.21
16	-	378.45	9251.64

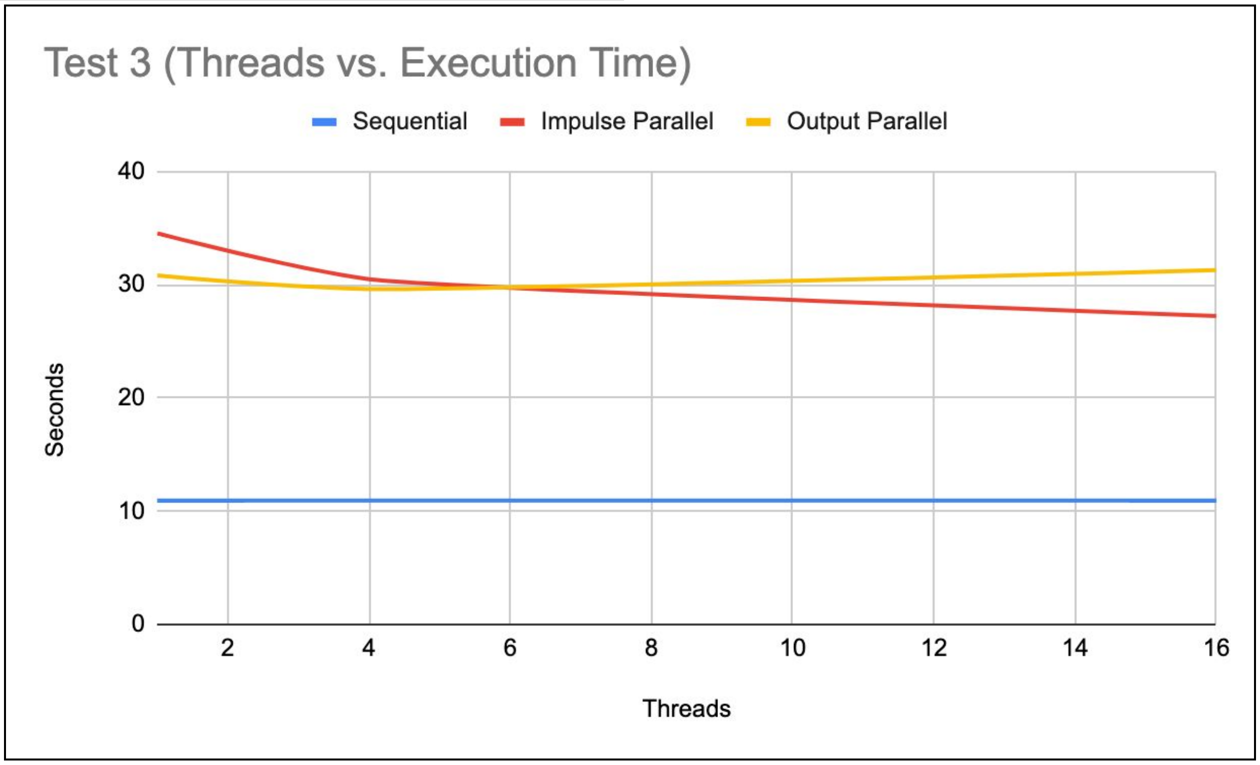


Test 3: Long input, short impulse response
Input: "speech_30sec_mono.wav"
Impulse response: "cabinet_mono.wav"

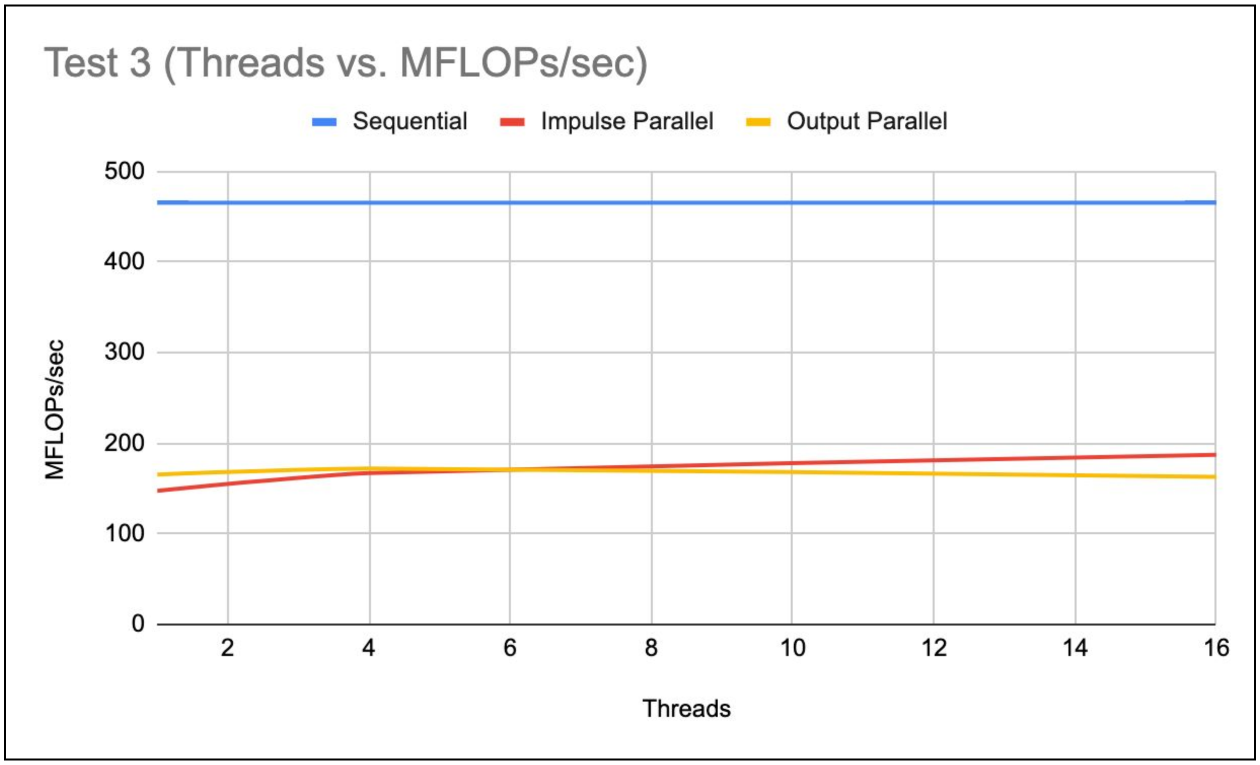
Analysis: This test was designed to provide a significantly longer input and shorter impulse response. I expected this test case to demonstrate the strength of the output-parallel algorithm over the impulse-parallel algorithm because the output length would be quite large, while the impulse response remained short. The output-parallel algorithm could partition the large output among the different processors. Unfortunately, the results were not very interesting because shortening the impulse response made the overall execution so short that the parallel speedup was unable to make up for the parallel overhead. In fact, the sequential algorithm outperformed both parallel algorithms. Typically, impulse response files are much shorter than input files already, so by shortening the impulse response even further, I greatly decreased the length of the convolution, which is proportional to the product of the input length and the impulse response length.

Results:

Test 3: Threads vs. Execution Time			
Threads	Sequential	Impulse Parallel	Output Parallel
1	10.94	34.54	30.81
4	-	30.49	29.61
16	-	27.24	31.29



Test 3: Threads vs. MFLOPs/sec			
Threads	Sequential	Impulse Parallel	Output Parallel
1	465.48	147.56	165.46
4	-	167.16	172.09
16	-	187.11	162.88



Conclusions:
The tests demonstrated that the most significant source of slowdown for the parallel algorithms was parallel overhead. In the case of the impulse-parallel algorithm, which incurred parallel overhead in each iteration of the outer for-loop, the overhead was quite significant, which is proven by looking at single-thread performance across the test cases. The output-parallel implementation, on the other hand, was much more elegant because the overhead was incurred only once, at the beginning of the convolution.

As proven by the second test case, it was also clear that it is crucial to avoid atomic writes and barriers in order to avoid load imbalances. I learned that the output-stationary code structure naturally avoids this.

The best speedup was achieved with large inputs, such as in the first test case. With a large input file and large impulse response file, the amount of computation involved in the convolution was much higher. In the case of the output-parallel algorithm, the parallel overhead was a constant, while the execution time saved in comparison to the sequential algorithm increased with the size of the computation. Thus the proportion of the overhead time to overall time was less when the computation was greater, which has the effect of making the overhead less significant in terms of total execution time, making the overall speedup greater.

While my stretch goal was to make direct convolution feasible for real-world reverb applications through parallelization, I was unable to do so. Even the output-parallel algorithm running with 16 threads was not competitive by today's standards. However, since performance continued to improve as I increased the number of threads, I believe that direct convolution could be improved even further in GPU programming, so perhaps a CUDA implementation would be an interesting future project.

Instructions for Compiling and Running the Code

Note: The main programs I wrote are called "sequential.cpp", "impulse_parallel.cpp", and "output_parallel.cpp". These programs depend on the "AudioFile" C++ library written by Adam Stark, which is used for reading and writing audio files. All needed code is included in the submission under "src/" and "AudioFile.h". Source: https://github.com/adamstark/AudioFile

Single-processor sequential algorithm
Compiling:
```
g++ -std=c++1y -O3 -o <object_filename> sequential.cpp
```

Running:
```
./<object_filename> <input_file> <impulse_response_file> <output_file> <wetness>
```
Note: <wetness> is a decimal between 0 and 1, representing the "wetness" of the reverb, which is the ratio of the volume of the reflected signals to the dry signal within the output.
Example:
```
./run sample_audio_files/speech_4sec_mono.wav
sample_audio_files/bottle_hall_mono.wav sample_audio_files/output_speech.wav .3
```

Data parallelism of the impulse response with OpenMP
Compiling:
```
g++ -std=c++1y -O3 -o <object_filename> impulse_parallel.cpp -fopenmp
```

Running:
```
./<object_filename> <input_file> <impulse_response_file> <output_file>
<wetness> <num_threads>
```

Note: <wetness> is a decimal between 0 and 1, representing the “wetness” of the reverb, which is the ratio of the volume of the reflected signals to the dry signal within the output.
Example Slurm submission:

```
srun -n1 -c8 ./run sample_audio_files/speech_4sec_mono.wav
sample_audio_files/bottle_hall_mono.wav
sample_audio_files/output_parallel_add.wav .3 8
```

Data parallelism of the output with OpenMP
Compiling:

```
g++ -std=c++1y -O3 -o <object_filename> output_parallel.cpp -fopenmp
```

Running:

```
./<object_filename> <input_file> <impulse_response_file> <output_file>
<wetness> <num_threads>
```

Note: <wetness> is a decimal between 0 and 1, representing the “wetness” of the reverb, which is the ratio of the volume of the reflected signals to the dry signal within the output.
Example Slurm submission:

```
srun -n1 -c8 ./run sample_audio_files/speech_4sec_mono.wav
sample_audio_files/bottle_hall_mono.wav
sample_audio_files/output_data_parallel.wav .3 8
```

Sample Audio Files
Provided along with the source code is a folder containing sample audio files to test out the programs:

- “cabinet_mono.wav” - Short-length impulse response file for simulating a cabinet.
- “bottle_hall_mono.wav” - Moderate-length impulse response file for simulating a large hall.
- “church_mono.wav” - Long-length impulse response file for simulating a church.
- “ee_mono.wav” - A short, mono vocal file to use as input.
- “speech_4sec_mono.wav” - A 4-second clip of MLK Jr.’s “I Have a Dream” speech to use as input.
- “speech_30sec_mono.wav” - A 30-second clip of MLK Jr.’s “I Have a Dream” speech to use as input.
- “output_speech_30sec.wav” - Sample output after using the “speech_30sec_mono.wav”, “bottle_hall_mono.wav”, and a wetness of .3.

Sources

[1] J. Smith, “Physical Audio Signal Processing”, *Stanford University Department of Music*, Mar. 4, 2020. [Online]. Available: https://www.dsprelated.com/freebooks/pasp/

[2] F. Wefers, “Fast convolution”, *audiolabs-erlangen.de*, Mar. 4, 2020. [Online]. Available: https://www.audiolabs-erlangen.de/content/05-fau/professor/00-mueller/02-teaching/2019s_apl/2_convolution/convolution.html