15-418: Profiling Partitioned Convolution on a GPU for Real-Time Applications

Melinda Chen, Ruslana Fogler

May 2024

1 Summary

We implemented partitioned audio convolution on the GPU using several strategies for memory management and pipelining. We compared the performance and characteristics of GPU accelerated linear convolution, naive overlap-add, overlap-add with streaming, and overlap-add with streaming *and* pinned memory to hide memory transfer latency.

2 Background

2.1 Setup, Inputs, Outputs

When performing convolution for the sake of audio processing (i.e. convolution reverb), there are generally 2 important inputs: signal and impulse. The impulse is usually a sharp sound which captures the frequency response of an environment (i.e. someone clapping inside of a concert hall). When convoluted against the signal, it outputs a result which sounds like the signal played in the impulse's environment. All the algorithms described below follow the general structure of taking in an impulse and signal and returning a convoluted wav file. While we were not able to set up a real-time streaming interface in time, we imposed constraints on the ways in which we could parallelize the problem (i.e. disallowing parallelizing across signal chunks) so that our approach could more easily extend to real-time applications.

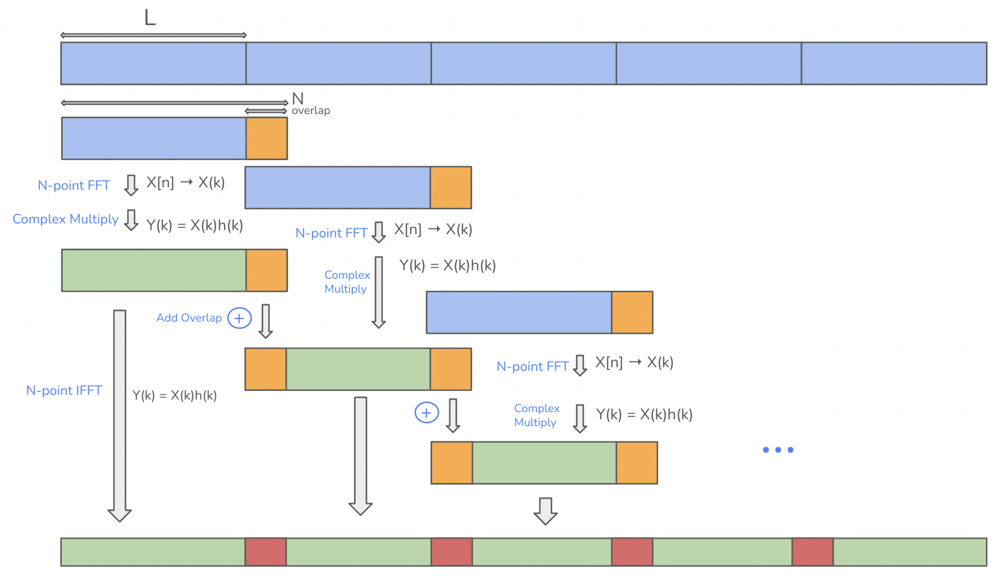
2.2 Core Algorithms

The main strategies and algorithms we investigated were: linear convolution, overlap-add, and overlap-add using CUDA streams. Linear convolution was parallelized simply by assigning each CUDA thread to a single element in the output and iterating over pairs of values in the signal and impulse around that element, summing the multiple of the pairwise multiplication to yield the convoluted output.

Unfortunately, the process of linear convolution is highly expensive, as it is an $O(n^2)$ operation, and also relies on the entirety of the input signal and impulse signal to be known. Hence, state-of-the-art convolution algorithms utilize the fact that multiplication in the frequency domain yields the same result as a time domain convolution. This process requires a fast fourier transforms to turn time domain signals to the frequency domain and back, which have complexity of $O(nlog(n))$. As a result, this process is computationally far more scalable than basic linear convolution. Furthermore, to this end, these algorithms partition their inputs into specifically-sized chunks that are suitable for the fast fourier transform in use(otherwise, they become susceptible to circular aliasing). This chunk size is determined by the length of the impulse reverb signal.

For the basis of our partitioned convolution implementation, we used the overlap-add algorithm (Figure 1). In overlap-add, the input signal is split up into overlapping chunks which are then padded, Fourier transformed, multiplied point-wise against the Fourier transform of the impulse, then inverse Fourier transformed to yield the convoluted output. Overlapped sections are added together to yield a seamless sounding result.

Figure 1: Overlap Add Algorithm



2.3 Key Operations and Data Structures

Across all algorithms, audio information was represented using `float` arrays. Before performing Fourier transforms on the data, it was converted into complex number form (functionally, `float2` arrays, where the first element represents the

real component and second element represents the imaginary component).
The major operations performed on the audio inputs include:

- Fast Fourier Transform (and its inverse): Conversion from time domain to frequency and vice-versa, allows us to carry out convolution via pointwise multiplication
- Complex Multiply: Actual convolution step - multiplication of frequency-domain impulse against signal
- Overlap-Add: Combination of outputs in overlapping sections of chunks.

For our overlap-add implementation, we had buffers on the host side to store the complex conversion of our input signal chunks and the convoluted output. On the device side, we had buffers to store the output of the impulse and signal chunk FFTs, as well as 2 buffers to store inverse FFT outputs from adjacent chunks which would then be overlap-added together.

2.4 Workload and Dependencies

All of the operations mentioned above could benefit from parallelism. Additionally, since performing a FFT on the signal eliminates dependencies between elements when performing the actual convolution, parallelizing the complex multiplication becomes very simple. Given that elements in the signal chunk are laid out temporally, there is also plenty of locality to exploit. Overall, the lack of dependencies and between elements (excluding performing the actual FFT), lack of divergence, and simple, coalesced data layout make this problem a very good fit for GPU acceleration.

While the overall data layout and operations are well suited for the GPU, the main challenge (which we will delve into significantly more detail later) comes from getting the data to the GPU in an efficient manner. In particular, we approached this problem with the goal of being able to extend our work to real-time applications, which meant that we couldn't parallelize over all the chunks of the input signal (which would be analogous to pulling sound from the future if we were to use this algorithm to carry out real-time convolution) and would have to deal with constantly copying memory back and forth from the GPU to CPU and vice versa. Thus, the dependency of all the GPU operations on fresh data ended up being the most important and difficult part of our pipeline to optimize.

3 Approach

3.1 Libraries/API's/Machines used

To load in audio files in a format that we could manipulate, we used the AudioFile library to read in wav files. This gave us `AudioFile<float>` objects

which contained information about the input's channels, bit depth, sample rate, and length. Each `AudioFile` object also contained an `AudioBuffer` which stored the actual samples associated with the audio file in a `float` array.
For our GPU implementation, we used CUDA-11.7 on the GHC machines (with NVIDIA GeForce RTX 2080 B GPUs). The FFT and inverse FFT operations were performed using the cuFFT library. We also used some utils functions (i.e. `CUDA_RT_CALL`) from the `cufft_utils.h`. All other kernels were implemented by us.
For our sequential implementation, we used the kissfft library.

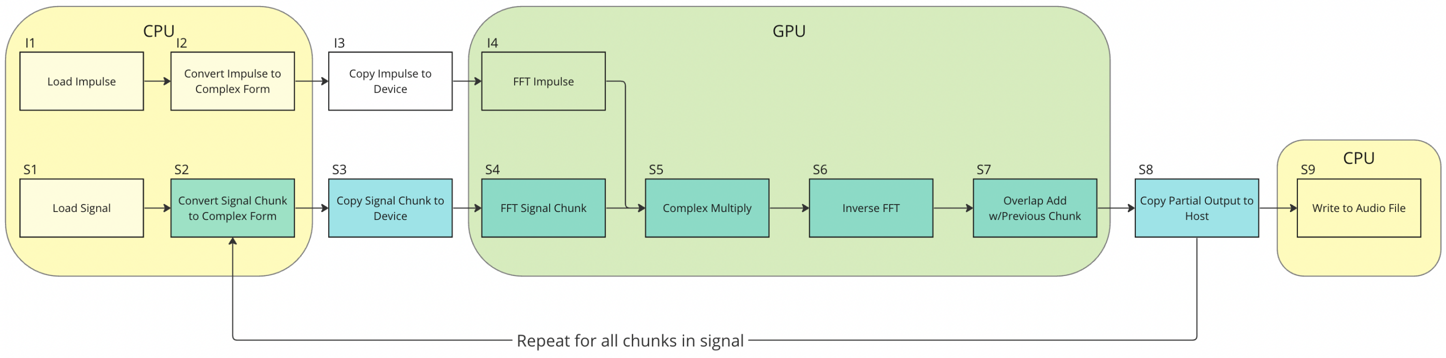
3.2 Core Implementations

A high-level overview of how we implemented overlap-add on the GPU can be seen below (Figure 2). After loading in both the impulse and signal, for each output audio channel, the impulse was copied to device memory, and Fourier transformed. Chunk size was calculated based on the size of the impulse rounded to nearest power of 2.

For each signal chunk, the section of the audio buffer was converted into a `complex<float>` type before being copied to device memory. The chunk was then Fourier transformed using `cufftExecC2C()`. The signal chunk and impulse were pointwise multiplied against each other using a kernel with each thread handling one element. That is, if the chunk (and by extension the complex-to-complex FFT output) contained `N` elements, we would launch N (rounded up to multiple of 512) threads.

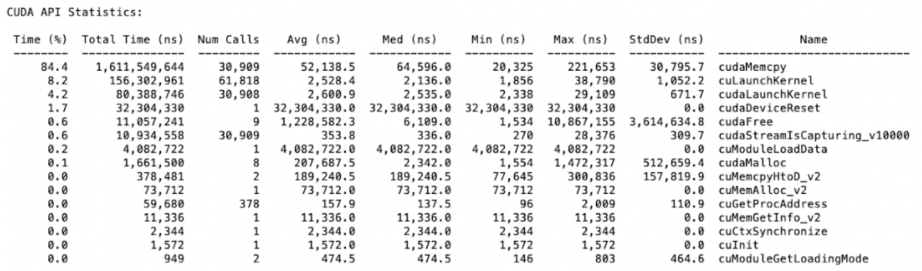
The convoluted chunk was then inverse Fourier transformed using another call to `cufftExecC2C()`, with the output being written to either an `even` or `odd` buffer depending on the parity of the chunk and combined with the prior chunk's result using another kernel (with threads being assigned to individual elements in the same way as the complex multiplication kernel). The combined output was then copied over back into host memory.

Figure 2: Overlap Add CPU/GPU Breakdown


CPU
I1
Load Impulse
I2
Convert Impulse to Complex Form
I3
Copy Impulse to Device
GPU
I4
FFT Impulse
S1
Load Signal
S2
Convert Signal Chunk to Complex Form
S3
Copy Signal Chunk to Device
S4
FFT Signal Chunk
S5
Complex Multiply
S6
Inverse FFT
S7
Overlap Add w/Previous Chunk
S8
Copy Partial Output to Host
CPU
S9
Write to Audio File
Repeat for all chunks in signal

Overall, the structure of the convolution portion of our implementation was quite similar to the sequential version. However, the way in which we managed memory transfer was significantly more involved.
Notably, since steps S4, S5, S6 and S7 all wrote their outputs into separate buffers, it would not impact the correctness of the algorithm to overlap memory transfers with computation for adjacent chunks. Additionally, after profiling our initial parallel overlap add implementations, we observed that memory transfer was undeniably our main bottleneck (Figure 3).

Figure 3: Memcpy Bottleneck



```
CUDA API Statistics:

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)     Min (ns)    Max (ns)    StdDev (ns)              Name
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------------------
     84.4    1,611,549,644     30,909      52,138.5      64,596.0      20,325     221,653     30,795.7  cudaMemcpy
      8.2      156,302,961     61,818       2,528.4       2,136.0       1,856      38,790      1,052.2  cuLaunchKernel
      4.2       80,388,746     30,908       2,600.9       2,535.0       2,338      29,109        671.7  cudaLaunchKernel
      1.7       32,304,330          1  32,304,330.0  32,304,330.0  32,304,330  32,304,330          0.0  cudaDeviceReset
      0.6       11,057,241          9   1,228,582.3       6,109.0       1,534  10,867,155  3,614,634.8  cudaFree
      0.6       10,934,558     30,909         353.8         336.0         270      28,376        309.7  cudaStreamIsCapturing_v10000
      0.2        4,082,722          1   4,082,722.0   4,082,722.0   4,082,722   4,082,722          0.0  cuModuleLoadData
      0.1        1,661,500          8     207,687.5       2,342.0       1,554   1,472,317    512,659.4  cudaMalloc
      0.0          378,481          2     189,240.5     189,240.5      77,645     300,836    157,819.9  cuMemcpyHtoD_v2
      0.0           73,712          1      73,712.0      73,712.0      73,712      73,712          0.0  cuMemAlloc_v2
      0.0           59,680        378         157.9         137.5          96       2,009        110.9  cuGetProcAddress
      0.0           11,336          1      11,336.0      11,336.0      11,336      11,336          0.0  cuMemGetInfo_v2
      0.0            2,344          1       2,344.0       2,344.0       2,344       2,344          0.0  cuCtxSynchronize
      0.0            1,572          1       1,572.0       1,572.0       1,572       1,572          0.0  cuInit
      0.0              949          2         474.5         474.5         146         803        464.6  cuModuleGetLoadingMode
```

Thus we decided to focus most of our effort on an implementation that used streaming to hopefully overlap the latency of memory transfers with computation from the previous chunk (Figure 4). Ultimately, to accomplish this we allocated buffers in pinned memory to copy the complex input signals into, then used two CUDA streams to handle convolution and memory transfer for alternating chunks.

Figure 4: Overlap Add Streaming Strategy

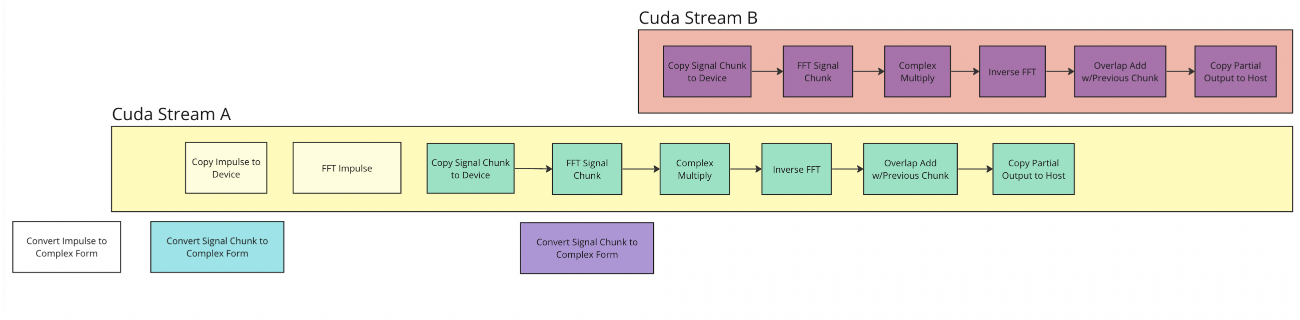
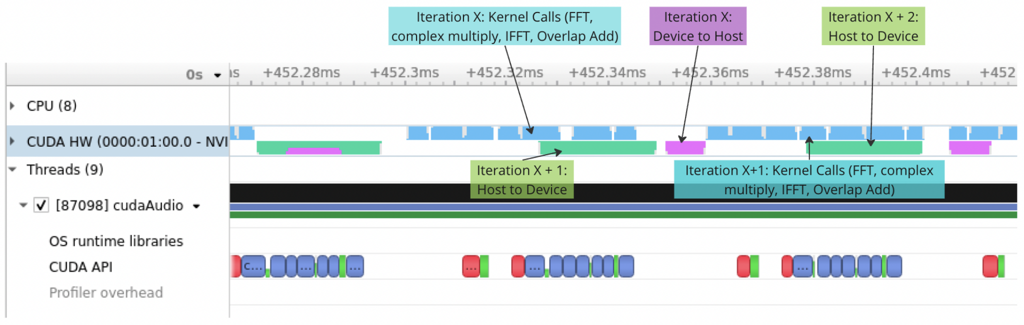

Cuda Stream B
Copy Signal Chunk to Device
FFT Signal Chunk
Complex Multiply
Inverse FFT
Overlap Add w/Previous Chunk
Copy Partial Output to Host
Cuda Stream A
Copy Impulse to Device
FFT Impulse
Copy Signal Chunk to Device
FFT Signal Chunk
Complex Multiply
Inverse FFT
Overlap Add w/Previous Chunk
Copy Partial Output to Host
Convert Impulse to Complex Form
Convert Signal Chunk to Complex Form
Convert Signal Chunk to Complex Form

Figure 5: Overlapping Memory Transfer with Kernel Calls


Iteration X: Kernel Calls (FFT, complex multiply, IFFT, Overlap Add)
Iteration X: Device to Host
Iteration X + 2: Host to Device
0s
+452.28ms
+452.3ms
+452.32ms
+452.34ms
+452.36ms
+452.38ms
+452.4ms
CPU (8)
CUDA HW (0000:01:00.0 - NVI
Threads (9)
Iteration X + 1: Host to Device
Iteration X+1: Kernel Calls (FFT, complex multiply, IFFT, Overlap Add)
[87098] cudaAudio
OS runtime libraries
CUDA API
Profiler overhead

3.3 Optimizing our Results: Tricks played

To analyze our code, we used a combination of `nvprof`, timing kernel/algorithm execution with `<chronos>`, and nsys-ui. From `nvprof` outputs, we saw that the majority of our time was spent in `cudaMemcpy`, which led us down the path of focusing on ways to reduce the impact of memory transfer latency. Importantly, just coalescing the `cudaMemcpy`s together was not an option given our constraint of not parallelizing across signal chunks.

We tried many different configurations of streaming before reaching our final implementation. Our initial implementation of streaming actually ended up performing worse than the naive parallel overlap-add algorithm for a couple of reasons. Firstly, to prevent other streams from modifying a buffer before it was finished being copied to/from, we used a `cudaStreamQuery` followed by a `cudaStreamSynchronize`. Since this would wait for the whole stream to finish execution, even if stream A were to query stream B after B had finished

its memory transfer (i.e. now performing some computation on the GPU), it would not be able to start its memory transfer until after stream B finished its computation, effectively eliminating any possibility of overlapping computation with memory transfer (while also adding some overhead).
We also played around with the mapping of `cuFFT` plans to streams. Initially, we had allocated two `cuFFT` plans to use for transforming the impulse and signal, however we realized that it would be a waste of memory given that the impulse only needed to be Fourier transformed once, and the overhead from allocating working sets for multiple plans was not trivial.
We also realized that in order for streaming to be truly effective, we would have to allocate buffers in pinned memory for both copies to and from device, otherwise the memory copies would block further kernel calls. In a similar vein, we played around with the position of the memory copy to device relative to writing the input values into the pinned memory buffer such that waiting on copying values to the buffer on the host would not block a call to a memory copy to/from device on a different stream.

4 Results

4.1 Performance Profiling

Our performance was measured by the speedup between our new Cuda implementations and our original sequential Overlap-Add Algorithm. In addition, to improve our first Cudafied Overlap-Add Algorithm, our primary benchmark was measuring the speedup of our streaming/pinned memory + streaming implementations.

We aimed to achieve the fastest convolution while keeping the real-time constraint of expecting our inputs to be processed in block chunks over time.

4.2 Experimental Setups

Our primary experimental setup was initializing an input audio buffer of random float values that were sizes of powers-of-two, running Cuda implementations on each sized buffer, and timing the execution. We used two different wav file reverb impulse responses against each randomly-initialized audio buffer to also keep track of how impulse response size affected our result.

For the impulse response wav file "hm.wav", which was a 3-second concert room impulse response, we measured our Cudafied linear convolution, Cudafied Overlap-Add, Cudafied Streaming Overlap-Add, and finally Cudafied Streaming Overlap-Add with pinned memory buffers. All these programs were compared against our sequential Overlap-Add algorithm on a sweep of audio buffer sizes from 2^{16} to 2^{24}.

Then, for the impulse response wav file "Muladhara.wav", a 4-minute guitar track from the video game *Digital Devil Saga 1*, we repeated the same measurement process with a sweep of audio buffer sizes from 2^{25} to 2^{29}.

Then, we calculated speedup of each test run and graphed our results.

4.3 Speedup Results

Our speedup results are as follows. Our configurations included testing our Cuda implementations over two vastly different sized impulse responses over many different audio buffer sizes.

4.3.1 Over Shorter Reverb Impulses

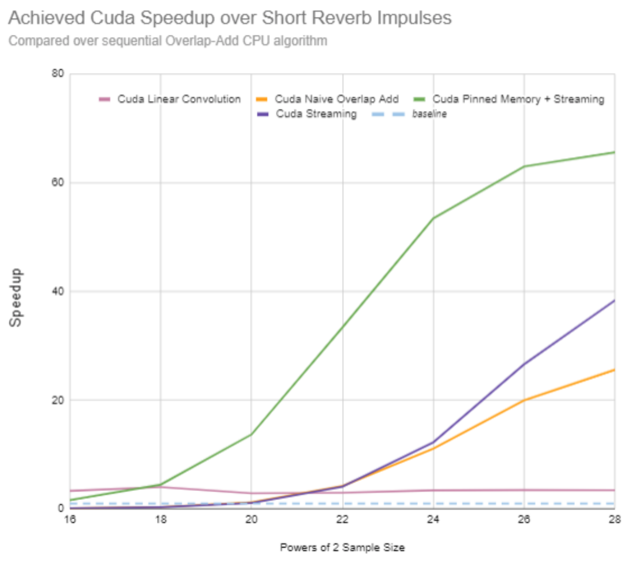


Figure 6: Sequential Speedup observed over Convolving against *hm.wav*, a shorter impulse response

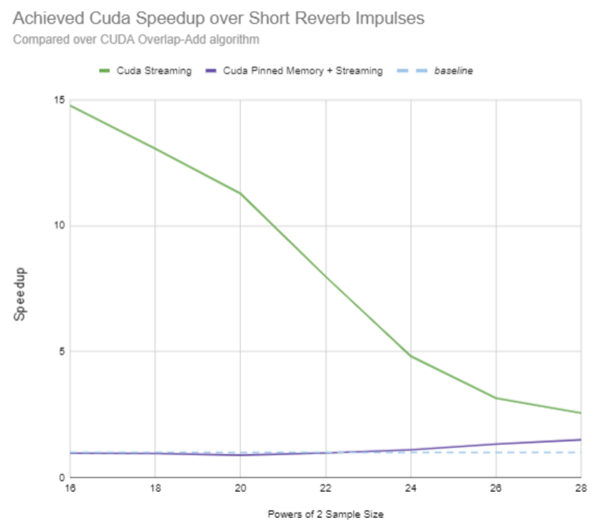


Figure 7: Basic Cuda OLA Speedup observed over Convolving against *hm.wav*, a shorter impulse response

Above, it can be seen that we encountered huge speedup between our Cudafied versions of Overlap Add against the first sequential algorithm. This was very expected, as the GPU is capable of multiplying the input and impulse frequency responses in parallel, and the cuFFT library was also faster than kissfft's sequential approach. We then sought to make our algorithms better by using the Cuda Overlap Add as our baseline. Our final implementation–Cuda Overlap Add with streaming and pinned memory–performed considerably better than our other implementations.

Our second graph conveys the speedup we acquired over improving the first CUDA Overlap Add algorithm.

Our pinned memory + streaming implementation's speedup gets worse over shorter reverbs, but it is still consistently far better than our implementation with only having attempted to integrate streaming. Pinned memory is first initialized by copying over paged memory to an unpageable memory location, which is an amortized process to ensure later data transfers are faster.

Another interesting observation is that our Overlap Add algorithm's speedup is significantly better than our naive Cuda linear convolution, in spite of the GPU's ability to multiply points massively in parallel.

4.3.2 Over Longer Reverb Impulses

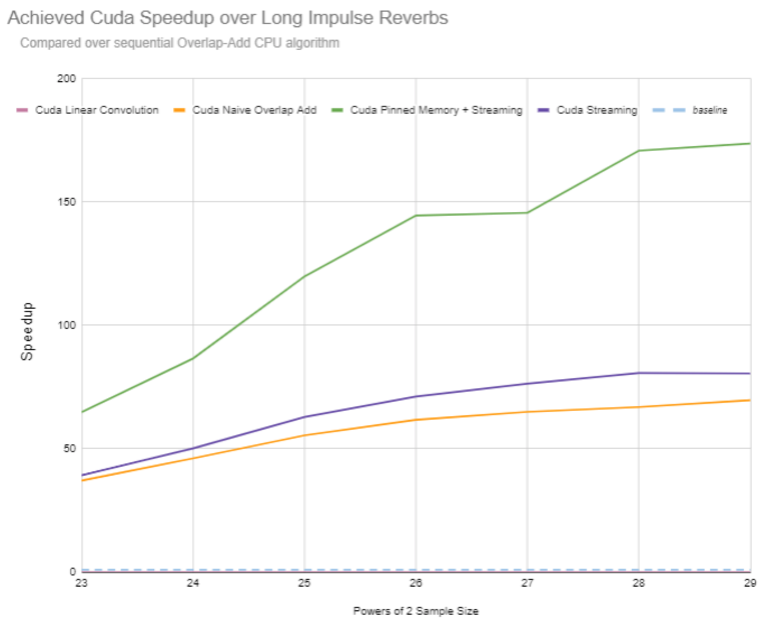


Figure 8: Sequential Speedup observed over Convolving against *Muladhara.wav*, a longer impulse response

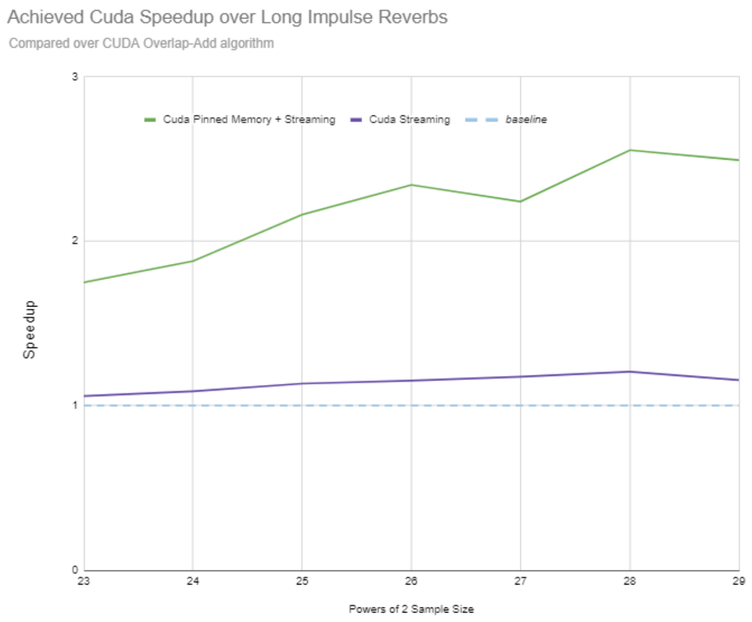


Figure 9: Basic Cuda OLA Speedup observed over Convolving against *Muladhara.wav*, a longer impulse response

Again, as expected, we encountered massive speedup for our Cuda implementations of convolution over our sequential implementations. In fact, this speedup is even greater than before because the larger impulse response. Above, it can be seen that using streaming and pinned memory strategies produced considerable speedup over our initial implementation of Overlap Add and our naive GPU linear convolution code.

Across larger impulse responses and larger audio buffer sizes, our pinned memory + streaming algorithm is consistently way better than our implementation that only does streaming.

4.4 Problem Size–Changing the Workload

Changing the workload of impulse response sizes produced a notable difference between our pinned memory + streaming implementation. While our combination of pinned memory and streaming had consistently larger speedup over large impulse responses and larger input sizes, the speedup actually decreased for shorter impulse responses and larger input sizes. We suspect that this may come down to initialization behavior sensitivity differences on very small input sizes.

Another observation is that our speedup begins to taper off more and more on larger sample sizes.

4.5 Speedup Limitations

Over larger and larger sample sizes, we began to see the speedup taper off.

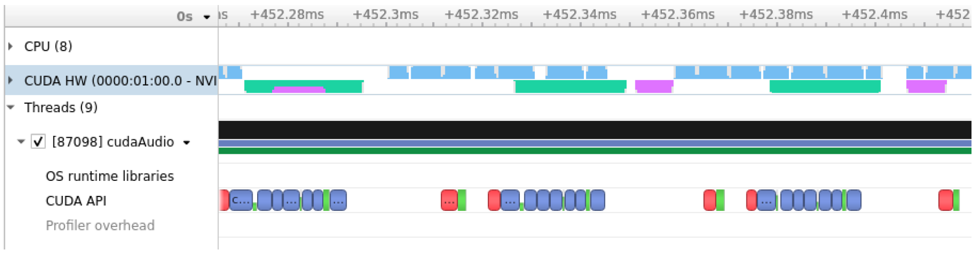


Figure 10: Cuda Stream Pipeline as obtained from `nsys-ui`

Since our kernels (cuFFT, combine, complex-multiply) are highly optimized for speed and simplicty of operation (no branching), they are relatively trivial to any speedup ceiling. Instead, waiting on memory copies and transfers are our main source of bottlenecks. There are three crucial roofs to our ability to obtain more speedup: the bandwidth/time limitation in Cuda's `memcpy` processes and the need for our pipeline's Cuda Streams to synchronize. In addition, overlap add is inherently a sequential algorithm. A barrier to greater parallelization is that we are choosing to develop this algorithm for a real-time purpose, so the assumption is that the input becomes available as block chunks over time; the entirety of the input is not known at the start of the program.

4.5.1 `memcpy`

One of the most expensive processes of our algorithm is the need to call memcpy frequently to move data between the cuFFT forward/inverse plans and the kernel to perform the complex multiply. Since this is an inherent, crucial part of the Overlap Add algorithm, it can't be removed. However, we were able to optimize the time consumption of the `memcpy` routine by incorporating streams and `memcpyAsync`. This enables us to perform a forwards fourier transform right after the current section is finished with the forwards cuFFT, without waiting for the entire section to be finished processing. Hence, the cost of the `memcpy`'s are more staggered and consume less overall program time. Regrettably though, `memcpy` routines cannot be reduced.

4.5.2 Synchronization

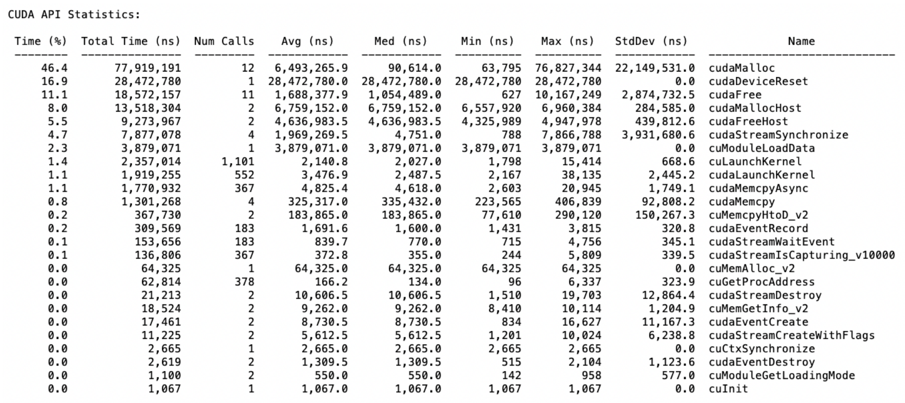
Another major blocker to speedup for our algorithm is the need to synchronize the cuda streams. In order for our pipeline to function, we need to ensure that the streams are coordinated with each other, otherwise audio aliasing occurs. One area where are synchronization is necessary is to allow the forwards cuFFT `memcpyAsync` and fast fourier transform to complete before performing the complex multiply between its frequency response and the impulse response.

Another area where synchronization is necessary is before our `combine` kernel, where the overlapping sections of the section convolutions are summed together. Since the completed convolution of each section depends on the prior section's overlapping values, failing to synchronize this region also impacts the convolution's correctness. Since this pipeline dependency cannot be eliminated, it actively prevents us from easily achieving more speedup.

4.5.3 Real-Time Inhibitions

A major constraint on our attempt to optimize convolution is that we intend this pipeline to be in use for real time convolution purposes. In this circumstance, we must assume that the entirety of the input signal is not entirely available at the start of the program. This means that we can't simply partition every X[n] section and lauch more cuFFT plans for sections in the future; in a realtime context, those future sections have not arrived yet. This sequential limitation also places a major constraint on our efforts to parallelize. This limitation is also visible when looking at Figure 10. The gaps between CUDA API calls correspond to time spent copying from the audio buffer into pinned memory and vice versa. When dealing with real-time applications, this constraint would still exist, except instead of copying from an audio buffer that is read in from a file, we would need to copy from an input audio stream.

4.6 Execution Time Breakdown



```
CUDA API Statistics:

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)      Min (ns)    Max (ns)    StdDev (ns)              Name
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  ------------  ----------------------------
     46.4       77,919,191         12   6,493,265.9      90,614.0      63,795  76,827,344  22,149,531.0  cudaMalloc
     16.9       28,472,780          1  28,472,780.0  28,472,780.0  28,472,780  28,472,780           0.0  cudaDeviceReset
     11.1       18,572,157         11   1,688,377.9   1,054,489.0         627  10,167,249   2,874,732.5  cudaFree
      8.0       13,518,304          2   6,759,152.0   6,759,152.0   6,557,920   6,960,384     284,585.0  cudaMallocHost
      5.5        9,273,967          2   4,636,983.5   4,636,983.5   4,325,989   4,947,978     439,812.6  cudaFreeHost
      4.7        7,877,078          4   1,969,269.5       4,751.0         788   7,866,788   3,931,680.6  cudaStreamSynchronize
      2.3        3,879,071          1   3,879,071.0   3,879,071.0   3,879,071   3,879,071           0.0  cuModuleLoadData
      1.4        2,357,014      1,101       2,140.8       2,027.0       1,798      15,414         668.6  cuLaunchKernel
      1.1        1,919,255        552       3,476.9       2,487.5       2,167      38,135       2,445.2  cudaLaunchKernel
      1.1        1,770,932        367       4,825.4       4,618.0       2,603      20,945       1,749.1  cudaMemcpyAsync
      0.8        1,301,268          4     325,317.0     335,432.0     223,565     406,839      92,808.2  cudaMemcpy
      0.2          367,730          2     183,865.0     183,865.0      77,610     290,120     150,267.3  cuMemcpyHtoD_v2
      0.2          309,569        183       1,691.6       1,600.0       1,431       3,815         320.8  cudaEventRecord
      0.1          153,656        183         839.7         770.0         715       4,756         345.1  cudaStreamWaitEvent
      0.1          136,806        367         372.8         355.0         244       5,809         339.5  cudaStreamIsCapturing_v10000
      0.0           64,325          1      64,325.0      64,325.0      64,325      64,325           0.0  cuMemAlloc_v2
      0.0           62,814        378         166.2         134.0          96       6,337         323.9  cuGetProcAddress
      0.0           21,213          2      10,606.5      10,606.5       1,510      19,703      12,864.4  cudaStreamDestroy
      0.0           18,524          2       9,262.0       9,262.0       8,410      10,114       1,204.9  cuMemGetInfo_v2
      0.0           17,461          2       8,730.5       8,730.5         834      16,627      11,167.3  cudaEventCreate
      0.0           11,225          2       5,612.5       5,612.5       1,201      10,024       6,238.8  cudaStreamCreateWithFlags
      0.0            2,665          1       2,665.0       2,665.0       2,665       2,665           0.0  cuCtxSynchronize
      0.0            2,619          2       1,309.5       1,309.5         515       2,104       1,123.6  cudaEventDestroy
      0.0            1,100          2         550.0         550.0         142         958         577.0  cuModuleGetLoadingMode
      0.0            1,067          1       1,067.0       1,067.0       1,067       1,067           0.0  cuInit
```

Figure 11: Cuda Stream Pipeline as obtained from `nsys-ui`

By using `nvprof` throughout our Cuda implementation, we were able to see the areas of program time consumption rather well. Through `nvprof`, it seems like the greatest part of time consumption of our program is `cudaMalloc`. Since our kernels include an already-optimized Cuda fast fourier transform algorithm and a massively parallelized complex multiply operation with no branching, our kernels occupy infinitesimal time percentage in our program.
Since much of our program's operations of cuFFT buffer copying, complex multiply, and cuIFFT buffer are pipelined through usage of Cuda streaming and pinned memory implementations, the latency of such operations are hidden.

Since a non-negligible amount of program time consumption is due to `cudaMalloc`, a small optimization we could further do is to use complex-to-real and real-to-complex fast fourier transforms instead of complex-to-complex ones. This would allow us to cut the size of our fourier transform output in half, because we can exploit the fact that real signals are symmetric in the discrete frequency domain. While a fast fourier transform of a P-length input for a complex-complex operation yields P values, a fast fourier transform of a P-length input for real to complex is optimized to yield $\lceil \frac{P}{2} \rceil$ values. Of course, the inverse(real to complex) would once again yield P values. Hence, the size of our fft output buffer would use less memory our `cudaMalloc` and `cmul` kernels could be slightly smaller. However, our larger bottlenecks reduce the capability for additional optimization.

4.7 Justifying Machine Choice

Since the process of massive parallel multiplications is key to the overlap add algorithm and the GPU is amazing at performing identical multiplications on a monumental scale, we feel that choosing the GPU for convolution reverb was very justified. No other tool that we've seen seems better suited for this purpose.

However, another interesting implementation that we encountered while researching the subject was a multi-threaded realtime convolver on the CPU. With this approach, the researcher was able to have some threads manage work large granularity work queues for the bulk of the convolution reverb process, and other threads manage small granularity work queues to hide the real-time latency of the convolution reverb. While we didn't ultimately have time to explore this approach, it would have been interesting to investigate the CPU's ability to perform realtime convolution. Arguably, this non-uniform partitioning includes some amounts of branching, which would have been disadvantageous for a GPU to perform.

Another justification for using a CPU could have been exploring how the CPU handled openMP implementations. The *kissFFt* library chosen for our sequential overlap-add prototype included openMP support for parallel butterfly fast fourier transform routines, and we could have also used those to see some startling speedup.

Furthermore, much research into GPU realtime convolution deals with the idea of needing to convolve larger and larger signals against each other. In circumstances like this, we realized that through our GPU sweep performance experiments that 2^{29} was the maximum float buffer that we could allocate on a single machine. With MPI, the possibility of doing more forms of incredibly large convolution with the cooperation of several GPUs becomes more possible.

5 References

- Bui, Cindy. GPU Acceleration of Massive Convolution, NYU Steinhardt Music Technology, 14 Dec. 2018, cindybui.me/images/Capstone_Report.pdf.
- Belloch, Jose A, et al. "Real-Time Multichannel Audio Convolution — GPU ... - Nvidia." Real-Time Multichannel Audio Convolution, Nvidia, 2009, www.nvidia.com/content/gtc-2010/pdfs/2116_gtc2010v2.pdf.
- Sadreddini, Maryam. "Non-Uniformly Partitioned Block Convolution on Graphics Processing Units." (2013).
- Kundur, Deepa. Overlap-Save and Overlap-Add, ww2.comm.utoronto.ca/ dkundur//course_info/real-time-DSP/notes/8_Kundur_Overlap_Save_Add.pdf. Accessed 5 May 2024.

- Harris, Mark. “How to Optimize Data Transfers in CUDA C/C++.” NVIDIA Technical Blog, 21 Aug. 2022, developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

6 ~~Distribution of Work~~ Load Balancing

Ruslana (50%):

- Sequential Overlap Add Algorithm Design
- Kissfft, Audiofile library integration
- PortAudio for Realtime Tests (did not end up in project, sadly)
- Setting up Random Vectorized buffers for Profiling setup
- Profiling Cuda Streaming, Pinned Memory + Streaming Implementation
- Profiling Sequential Overlap Add Algorithm
- Report Writing: Results, References, Work Distribution

Melinda(50%):

- Cuda Overlap Add Implementation
- Setting up Cuda library
- Cuda Overlap-Add Streaming Implementation
- Cuda Nsight/Nsys tool setup
- Cuda Overlap Pinned Memory + Streaming Implementation
- Report Writing: Summary, Background, Approach