Analysis of CPU and GPU Implementations of Convolution Reverb Effect

Marko J. Mišić, *Member, IEEE*, Dušan V. Nikolov, and Milo V. Tomašević

***Abstract* — Rapid development of modern central processing units (CPUs) and graphics processing units (GPUs) has allowed a significant increase in computing power for different engineering applications. Audio signal processing is an example of such a computationally demanding application. Fast Fourier Transform (FFT) is often a core part of these processing algorithms, and it is efficiently implemented on the CPUs and GPUs through available libraries. In this paper, we present an implementation of the convolution reverb effect using OpenMP and FFTW library on the CPU, and CUDA and cuFFT library on the GPU. Implemented effect is tested with the set of four different audio signals and ten impulse responses of different lengths. We observed speedups in the range of 2 to 3 times over CPU implementation. The results of the analysis are briefly discussed with emphasis on the benefits and drawbacks of using GPUs in such an application.**

***Keywords* — convolution reverb, CUDA, graphics processing units, OpenMP, parallel programming.**

I. Introduction

THE progressive evolution of computer architecture and microprocessor technology in the past decade has provided necessary computing power for many demanding applications. Raw computing power is mostly increased through the introduction of innovative parallel architectures, such as multicore architectures in central processing units (CPUs), and manycore architectures in graphics processing units (GPUs). Contemporary CPUs are task-parallel processors, consisting of up to dozen of cores that are able to execute a program code concurrently. On the other hand, accelerators, such as GPUs and Intel Xeon Phi, offer abundant data-level parallelism with hundreds or thousands of cores available for execution. Typically, a CPU needs several threads to fully utilize hardware resources, while a GPU needs thousands of them.

Nowadays, the way compute-intensive applications are implemented has changed. Compute-intensive parts are offloaded to the accelerator, while a CPU takes care of I/O, management tasks, or smaller portions of work. Those applications are programmed through different, often heterogeneous programming models. In that sense, extracting parallelism from existing problems becomes an important and interesting topic for research community. Accelerators are used in many commercial and scientific applications, such as signal processing, computational physics, chemistry and life sciences, as well as in financial applications and data mining [1]. GPUs are the most mature of accelerator technologies, with steady software support and growing user community. For that reason, we used them in our research.

Audio signal processing could be regarded as an intentional modification of sound signals. It is widely employed in music and video production, computer games, virtual reality. Audio signals are usually represented in a digital form and different effects are used to measure, store, compress, or enhance audio signals through filtering. Popular effects include equalization, noise reduction, reverberation (echo), delay, etc.

Reverberation is an effect of prolonging the sound by the environment, caused by the reflectivity of surfaces and by the slow speed of sound in the air [2]. Artificial reverberation through different computational methods is known for more than 50 years. It has a widespread usage in audio and video production to create different artistic effects and improve impressions of sound in different environments. Convolution reverb is an audio effect used to digitally produce the reverberation of a physical or virtual space such as concert halls, churches, etc. The effect is produced by convolving the input audio signal with the recorded or estimated impulse response of the space being simulated. To reduce overall time complexity, fast convolution is often performed using Fast Fourier Transform (FFT). In such a case, FFT is usually the most compute intensive part of the algorithm.

A significant computing power is needed in digital audio processing, which presents a limiting factor for larger projects. Multicore CPUs and manycore GPUs can be used to provide more computing power through parallel implementations of audio signal processing algorithms. In this paper, we present the CPU- and the GPU-based implementations of the convolution reverb effect, following the similar approaches presented in [3], [4], [5], and [6]. The CPU implementation is based on OpenMP and FFTW library, while GPU implementation is based on

Paper received May 15, 2016; revised August 31, 2016; accepted September 1, 2016. Date of publication November 20, 2016. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Jovan Đorđević.

This paper is a revised and expanded version of the paper presented at the 23rd Telecommunications Forum TELFOR 2015 [4].

This work has been partially funded by the Ministry of Education and Science of the Republic of Serbia (III44009 and TR32047). The authors gratefully acknowledge the financial support.

Marko J. Mišić is with the University of Belgrade, School of Electrical Engineering, Serbia (phone: 381-11-3218-392; e-mail: marko.misic@etf.bg.ac.rs).

Dušan V. Nikolov is with the University of Belgrade, School of Electrical Engineering, Serbia (e-mail: nikolovdusan92@gmail.com).

Milo V. Tomašević is with the University of Belgrade, School of Electrical Engineering, Serbia (phone: 381-11-3218-302; e-mail: mvt@etf.bg.ac.rs).

CUDA and cuFFT library. We evaluated those implementations with different test signals and impulse responses. We observe significant speedups over CPU implementation and present our experiences with porting a convolution reverb effect on the GPU. Since fast convolution is performed using FFT, our experiences with the implementation of reverb effect on the GPU can be used in broader contexts where efficient convolution implementation is necessary.

The paper is organized as follows. The second section presents related work in the domain of audio signal processing on contemporary parallel architectures, especially GPUs. In the third section we give a short overview of the GPU architecture, CUDA and OpenMP programming models. Section IV describes convolution reverb effect, while implementation details with emphasis on parallelization using CUDA are given in section V. Experimental results are discussed in the sixth section. Conclusion and directives for future work are given in the final section.

II. Related work

Parallel processing is used in digital signal processing to decrease processing time and increase throughput. Multiple function units, such as processor cores, either operate on different tasks (signals) simultaneously or jointly operate on the different parts (samples) of the same task [7]. CPUs have been used to exploit task and data parallelism in signal processing for a long time, and different programming models and frameworks have been used in the past [8].

Modern GPUs have been used for signal processing in the last decade with the emergence of general-purpose computation on GPUs. Hardware features such as efficient memory hierarchy and a standardized programming model, make them suitable for compute-intensive audio applications. High performance Fast Fourier transforms are among the first examples of successful implementations of signal processing algorithms on the GPU [9], [10].

GPUs have been successfully applied to computationally intensive problems in acoustics and audio processing in several studies. Study [5] reports 5 to 100 times performance improvement compared to the CPU implementations in audio rendering, speech recognition, sound synthesis, acoustic simulation, etc. Implementation of dynamic range reduction of audio signals with multiple allpass filters is described in [11], reporting two times speedup over 6-core CPU. Efficient implementations of FIR filtering are also reported [10]. The authors claim that GPUs are eligible for real-time applications, such as software plugins for professional audio production or interactive audio rendering for virtual acoustics reality. Multichannel acoustic signal processing is also suitable for implementation on GPUs [12], [13]. It has been shown that, for a real-time application with 22 inputs and 64 outputs, a GPU-based system is capable of managing 1408 filters of 2048 coefficients with a latency time less than 6 ms [12].

Several directions for future work in the domain of audio signal processing using heterogeneous architectures are given in [14]. Most of those are concentrated on 3D audio and real-time applications. On the other hand, the main challenge for high performance audio applications is concurrent execution on CPU and GPU. An offloaded audio algorithm cannot run exclusively on GPU, but it has to be partitioned between GPU and CPU, and integrated into a specific application. OpenCL is suggested as a platform to support data-parallelizable audio tasks on different GPU architectures.

III. Programming models

Multicore CPUs are mainly programmed through shared memory programming models, although other programming models, such as message passing, are also exploited. Parallelism is usually implemented using a number of threads. Different threading supports exist, where thread management is handled implicitly by the compiler (OpenMP) or explicitly by the programmer (Pthreads). We opted for the former in our research, since OpenMP is a widely adopted programming model for multicore CPUs. OpenMP is a directive-based programming model that offers productivity of a high-level programming language with an extensive compiler support. Programmer is responsible to identify parallelism in the code and annotate it with OpenMP directives. Compiler and runtime environments are responsible for code execution and multithreading.

GPUs evolved from specialized processors in the domain of computer graphics to powerful accelerators programmed through commodity programming languages such as C, C++, and many others. GPUs have a specific, manycore architecture, thus code execution differs considerably from the execution on the CPU. Nowadays, CPU and GPU can work together in a heterogeneous application. Sequential parts of the application are executed on the CPU, while compute-intensive parts (kernels) are offloaded to the GPU for parallel execution.

NVIDIA Compute Unified Device Architecture (CUDA) [15] is the most mature programming model for GPUs. Other popular models include OpenCL and OpenACC, while OpenMP 4.0 recently added support for accelerators. Both CUDA and OpenCL are low-level programming models that require explicit programming using library calls and specially written computing kernels. OpenACC is a directive-based programming model similar to OpenMP. OpenACC is developed specifically for GPUs. OpenMP 4.0 supports accelerator programming, but still lacks stable compiler implementation. Since CUDA offers excellent software stack, library support, and the highest potential for code optimizations, we chose it for our implementation.

A typical GPU consists of several streaming multiprocessors (SMs) that are able to execute a large number of lightweight threads. Threads are executed in a SIMD fashion, as each SM consists of a number of scalar processors, depending on the architecture. To improve scalability, threads are organized in blocks executed on a

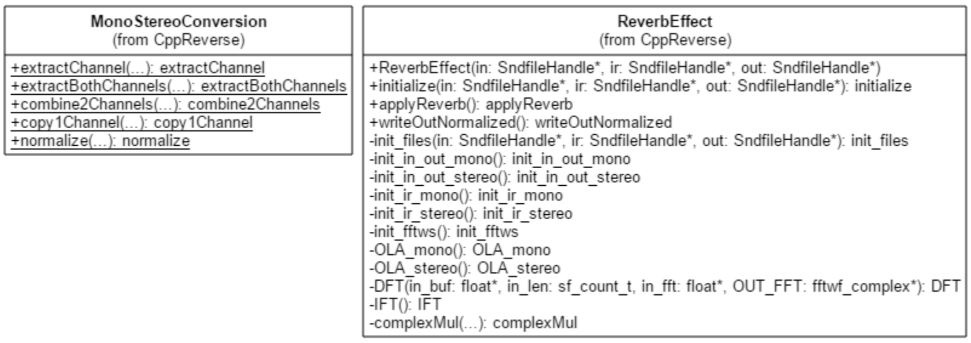
MonoStereoConversion
(from CppReverse)
+extractChannel(...): extractChannel
+extractBothChannels(...): extractBothChannels
+combine2Channels(...): combine2Channels
+copy1Channel(...): copy1Channel
+normalize(...): normalize
ReverbEffect
(from CppReverse)
+ReverbEffect(in: SndfileHandle*, ir: SndfileHandle*, out: SndfileHandle*)
+initialize(in: SndfileHandle*, ir: SndfileHandle*, out: SndfileHandle*): initialize
+applyReverb(): applyReverb
+writeOutNormalized(): writeOutNormalized
-init_files(in: SndfileHandle*, ir: SndfileHandle*, out: SndfileHandle*): init_files
-init_in_out_mono(): init_in_out_mono
-init_in_out_stereo(): init_in_out_stereo
-init_ir_mono(): init_ir_mono
-init_ir_stereo(): init_ir_stereo
-init_fftws(): init_fftws
-OLA_mono(): OLA_mono
-OLA_stereo(): OLA_stereo
-DFT(in_buf: float*, in_len: sf_count_t, in_fft: float*, OUT_FFT: fftwf_complex*): DFT
-IFT(): IFT
-complexMul(...): complexMul

Fig. 1 UML class diagram of the implemented effect

single SM. Kernel execution is organized as a grid of thread blocks. Threads in a thread block can be synchronized using a barrier, but global synchronization is achieved only when kernel execution terminates. GPU memory architecture is designed to support high throughput and execution of a number of threads in parallel. It consists of a hierarchy of memories that differ in speed and capacity. Global DRAM memory accesses are slow, so threads can utilize other smaller memories to speed up the execution, such as registers, shared memory, constant memory, etc. Typically, CPU and GPU use different, physically separated memory spaces, so explicit transfers of data between CPU and GPU are needed. Overlapping of kernel execution and memory transfers is possible through the concept of CUDA streams.

Performance optimization is a challenging task on the GPU. Performance can vary greatly depending on the resource constraints of the particular device architecture, and it is much on the developer to exploit all parallelism available [1].

IV. THE CONVOLUTION REVERB EFFECT

Digital signal processing usually involves transformation of processed signals from time to frequency domain and vice-versa. Those transformations are performed through direct and inverse Fast Fourier Transform (FFT). FFT is a rather compute-intensive operation, and for that reason it is one of the most time-consuming parts in the implementation of a sound effect [10]. On the other hand, the signal samples are often suitable for processing in parallel [16], thus making the algorithms convenient for execution on contemporary multicore and manycore architectures.

Many sound effects are performed through the convolution of input audio signal and impulse response of the effect. Impulse responses are recorded responses of an acoustic space when ideal impulse or appropriate approximation is played. The same method can be applied in the case of the convolution reverb effect.

Convolution is a complex, time-consuming operation, when done in time domain with average complexity of O(n^2). The convolution theorem states that convolution in time domain equals point-wise multiplication in frequency domain [17]. For that reason, it is possible to implement the convolution or FIR filtering in the FFT domain. The average complexity of this approach is O($n \log n$).

In theory, the lengths of input audio signal and impulse response are arbitrary. Implemented algorithm partitions the input audio signal into segments of length L, and the impulse response into segments of length M. This method is known as *Overlap and Add* method, and it is used to ensure a stable performance of the algorithm for different input signal lengths. Since M and L are not equal by default, it is needed to extend them in order to apply the Fourier transform. Usually, M and L are powers of two in order to use the Fast Fourier Transform (FFT). After multiplication in frequency domain, the resulting segment will be longer in time domain than the original segment. To produce correct results, it is needed to overlap and add extension of one segment to the beginning of the next segment.

Since impulse response is constant in time, it is possible to parallelize multiplication operation of one segment of input audio signal with all segments of impulse response. *Overlap and Add* method is also suitable for parallelization, as even and odd segments can be processed independently. The described method is suitable for parallelization using both CPUs and GPUs.

V. IMPLEMENTATION DETAILS

The convolution reverb effect was implemented in an object-oriented fashion, as a stand-alone application. Both CPU and GPU implementation are based on the same initial design shown in Fig. 1. The application uses *libsndfile* library [18] to manipulate the sound samples through a standardized interface. *MonoStereoConversion* class contains utility methods for extraction, recombination, and normalization of the sound signals. The implemented effect works with mono and stereo sound files in WAV format, both for input signals and impulse responses. The sound samples are stored in one-dimensional arrays for each channel.

The core part of the implementation is placed in *ReverbEffect* class. *DFT* and *IFT* methods implement discrete and inverse FFT using FFTW library on the CPU [19] and cuFFT library on the GPU [20]. *OLA_mono* and *OLA_stereo* methods implement the reverb effect. Those methods are basically equivalent. A pseudocode for *OLA_stereo* method is shown in Fig. 2 for the CPU version and in Fig. 3 for the GPU version of the code.

In both versions, the input signal is processed in segments of length $L=4096$, while the impulse response is partitioned into segments of $M=8192$. Those values have been tuned experimentally. *ComplexMultiplyStereo* method wraps up the convolution operation in frequency domain, as it is performed as complex multiplication. Since we support both mono and stereo signals, the result of this operation depends on their nature. *TrueStereo* signal is produced if both input signals and impulse response are stereo. Left and right channels of the input signal are multiplied with left and right channels of impulse response, producing four complex multiplication operations. *QuasiStereo* is produced with two complex multiplication operations, when the impulse response is mono, but the input signal is stereo.

```
for (long i = 0; i < in->frames(); i += L) {
  DFT();
  for (long j = 0; j < IR_blocks; j++) {
  ComplexMultiplyStereo(...)
    IFT();
   /* Overlap and Add */
  #pragma omp parallel for schedule(static) ...
  for (long k = 0; k < N; k++) {
      out_l[i + j * M + k] += (in_src_l[k] / N);
      /* find signal amplitude */
      findMax();
}}}
```

Fig. 2 Pseudocode of the implemented effect on the CPU using OpenMP.

Overlap and add method requires that the time domain signals of lengths L and M are zero-padded before the calculation of the corresponding FFTs and multiplication of the FFTs. For that reason, inverse FFT produces the result of length $L + M - 1$ in time domain. Since the input signal is partitioned into segments of length L, $M-1$ samples of the resulting signal overlaps with the samples in the beginning of the next segment. Those samples should be overlapped and recombined to produce the correct result. Excessive $M-1$ samples from previous iteration are saved for usage in the next iteration of *Overlap-Add* method which is shown in Fig. 4.

```
for (long i = 0; i < in->frames(); i += L) {
  /* FFT left and right channel input block */
  DFT();
  ComplexMultiplyStereo(...)
  /*  perform  batched  IFFT  from  OUT_SRC  to
  cache_padded (left and right) */
  IFT();
  OverlapAdd(multiple args, 0, stream_l);
  OverlapAdd(multiple args, 0, stream_r);
  OverlapAdd(multiple args, 1, stream_l);
  OverlapAdd(multiple args, 1, stream_r);
  if (i + L > in_sz / 2) {
    size = sizeof(float)* (out_sz / 2 - i);
    cudaMemcpyAsync(out_l+i,cache_l,
                   size, DToH, stream_l);
    cudaMemcpyAsync(out_r+i,cache_r,
                   size, DToH, stream_r);
  } else {
    size = sizeof(float)* L
    cudaMemcpyAsync(out_l + i, cache_l,
                   size, DToH, stream_l);
    cudaMemcpyAsync(out_r + i, cache_r,
                   size, DToH, stream_r);
    BackupCache(multiple args, stream_l);
    BackupCache(multiple args, stream_r);
    swap(temp_cache_l, cache_l);
    swap(temp_cache_r, cache_r);
  } }
  /* find signal amplitude */
  findMax();
}
```

Fig. 3 Pseudocode of the implemented effect on the GPU using CUDA.

On the CPU, *ComplexMultiplyStereo* and *Overlap and Add* operations are implemented using OpenMP *for* a work sharing construct. Those are the most time-consuming operations of the algorithm. A *static* schedule is used for work distribution, since loop iterations are well balanced.

The GPU implementation is more complex due to memory transfers involved and fine-grain nature of the programming model. Memory transfers of already processed segments are overlapped with the computation kernels execution using CUDA streams. Moreover, left and right channels are processed independently, thus offering another level of parallelism in the problem solution. Complex multiplications are efficiently implemented using one CUDA thread per sound sample, as each sample is processed independently. In the GPU implementation, excessive *M-1* samples from previous iteration are saved using *BackupCache* method for usage in the next iteration. To increase parallelism, *Overlap-Add* operation is performed independently for even and odd segments of both channels. Even and odd segments are processed in this fashion to avoid synchronization overheads in CUDA kernels. *Overlap-Add* method is illustrated in Fig. 4.

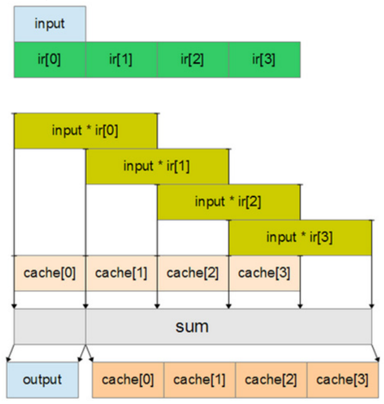

Fig 4 *Overlap-Add* method.

Several other utility functions such as *ComplexMul* and *ComplexAdd* were implemented to encapsulate operations with complex numbers. Those functions are optimized to use register storage in order to avoid expensive global memory accesses.

VI. Evaluation and Discussion

Implemented effect is evaluated on Intel Core i7 5820K 3.30GHz 6-core CPU with 16GB RAM using NVIDIA GTX Titan Black graphics card with 2880 CUDA cores and 6GB RAM under Ubuntu 14.04 OS. We used four different input signals and ten impulse responses for testing, as shown in Table 1 and Table 2. All signals were in WAV (PCM16) format with 44.1 KHz sampling rate.

The results are shown in figures 5-8. Used impulse responses are shown on the horizontal axis. Suffix FFTW is used for executions on the CPU, while cuFFT is used for executions on the GPU. The vertical axis shows execution time in milliseconds. In most cases the obtained speedup of GPU implementation is in the range of 2 to 3 times over reference, CPU implementation. Generally, execution time increases linearly with the length of impulse response for both CPU and GPU implementations, but the GPU implementation shows higher speedups for longer impulse responses,. Therefore, execution time on the GPU increases at a much slower pace.

TABLE 1: INPUT SIGNALS

Name	*Channels*	*Samples*	*Duration [s]*
gtr	1	273135	6.19
vocal	1	476021	10.79
guitar_clean	2	176400	4.0
horror_bells	2	1176597	26.68

TABLE 2: IMPULSE RESPONSES

Name	*Channels*	*Samples*	*Duration [ms]*
microCab	1	342	0.10
minicab	1	1035	0.40
cabinetIR	1	3540	0.80
Edge	2	8821	0.20
Sony MIC	1	16459	0.37
FredmanCab	2	27313	0.62
PlateIR	2	61806	1.40
T65 telephone	2	129439	2.94
LongEchoHallMono	1	176584	4.01
LongEchoHall	2	176584	4.01

It is clear that for some combinations of the input signal and the impulse response, the CPU implementation is faster than the GPU implementation. For smaller impulse responses, memory transfer overheads fairly exceed the kernel execution time, as shown in Fig. 9. However, in practical cases used in audio and video production, the impulse responses are typically 0.5 to 2 seconds long. Thus, GPU implementation is faster than CPU implementation in real-life scenarios. This is consistent with the findings in [6], where the author implemented several audio effects, including the reverb effect on GPU.

GPU Impulse Reverb VST [21] presents another effort similar to ours. It is written in OpenCL as a plugin for VST compatible host software, such as Cubase, Nuendo, Ableton Live, etc. They reported low CPU usage and low-latency, as one of the most important features. However, due to the commercial nature of this plugin, we have not been able to evaluate it and compare with our implementation.

The CPU implementation benefited from OpenMP-based parallelization, as shown in Fig. 10. We compared sequential and OpenMP implementations on the CPU and observed speedups of up to 3 times for longer impulse responses. OpenMP directives offer high programmer productivity, as threading support is completely managed by a runtime library. Thus, significant speedups can be achieved with a modest programming effort.

The GPU implementation benefited from several optimizations. As mentioned before, to reduce memory transfer overheads, we used streams for left and right channel. Kernels were run independently. The performance gains are small, but not negligible, especially for longer impulse responses. Table 3 shows the execution times for particular GPU kernels using guitar_clean.wav input signal and LongEchoHall stereo impulse response. It can be observed that *Overlap-Add* operation is dominant and consumes most of the execution time.

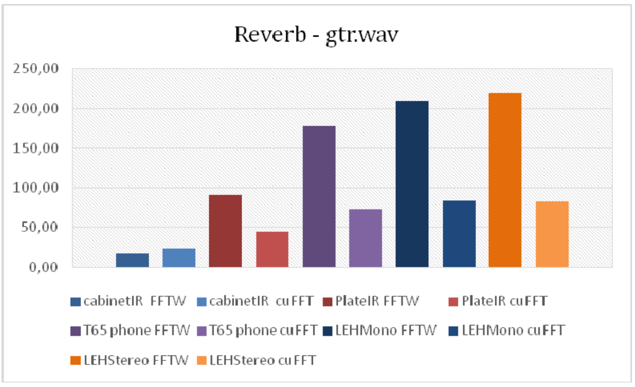

Fig 5 Comparison of CPU and GPU implementations for gtr.wav input signal.

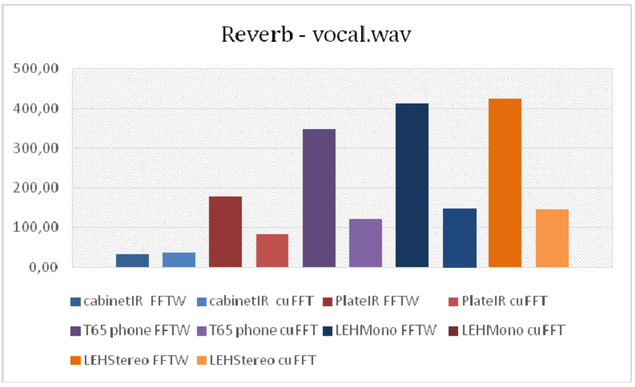

Fig 6 Comparison of CPU and GPU implementations for vocal.wav input signal.

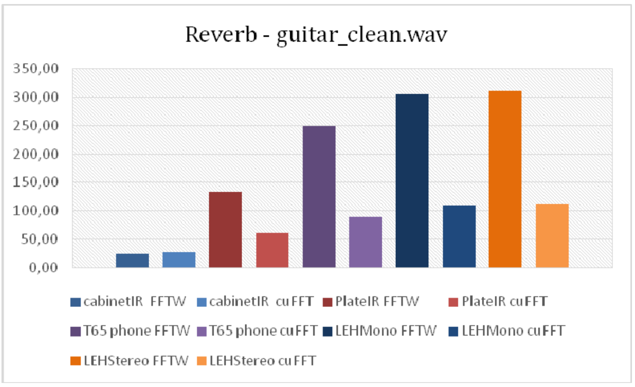

Fig 7 Comparison of CPU and GPU implementations for guitar_clean.wav input signal.

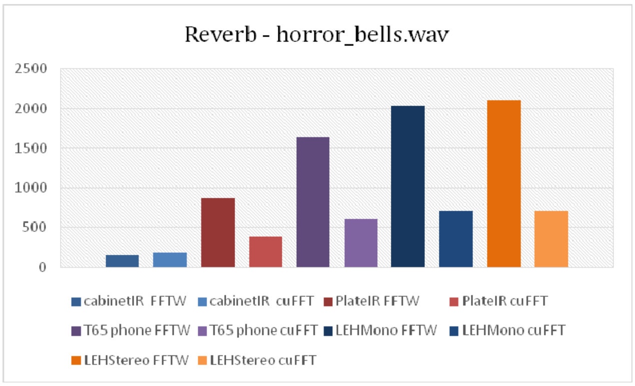

Fig 8 Comparison of CPU and GPU implementations for horror_bells.wav input signal.

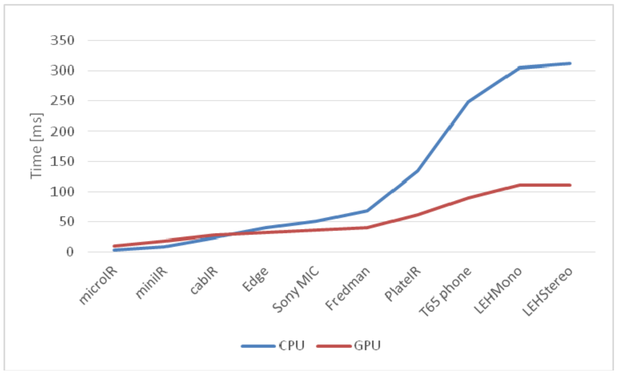

Fig 9 Comparison of CPU and GPU implementations for different impulse response sizes.

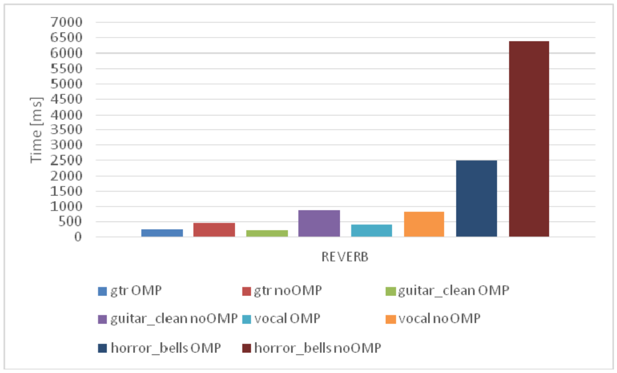

Fig 10 Comparison of sequential and OpenMP-based CPU implementations.

As mentioned in section V, even and odd segments of both channels are processed independently to avoid synchronization overheads in CUDA kernels. Before this optimization was introduced, this kernel consumed up to 70 percent of total kernel execution time. *ComplexMultiplyStereo* kernel also consumed more time in initial implementation. We also used some register-level optimizations (mostly variable privatizations), which reduced the execution time of this kernel up to three times. The same optimization gave good results in the CPU implementation.

TABLE 3: EXECUTION TIMES FOR PARTICULAR GPU KERNELS

Time %	***Time [ms]***	***No. of calls***	***Average time per call [ms]***	***Method name***
47.66	59.36	176	336.97	OverlapAdd
14.18	17.647	88	200.54	spVector [cuFFT]
13.01	16.188	86	188.23	BackupCache
12.3	15.305	44	347.85	Complex MultiplyStereo
7.88	9.8075	88	111.45	spRealComplex [cuFFT]
2	2.4833	132	18.812	spVector [cuFFT]
0.94	1.1752	88	13.354	[CUDA memcpy DtoH]
0.92	1.1405	136	8.385	[CUDA memset]
0.6	0.74069	132	5.611	spRealComplex [cuFFT]
0.52	0.64365	132	4.876	[CUDA memcpy HtoD]

VII. CONCLUSION

In this paper, we presented our experience with the implementation of convolution reverb effect on the CPU and GPU. CPU-based version is parallelized using OpenMP. Audio signals are suitable for processing on the GPU, and CUDA offers an efficient implementation of FFT. Observed speedups over the CPU implementation are significant, especially for longer impulse responses.

There are several directions for future work, mostly in GPU-based implementation. Memory transfers could be overlapped with FFT for subsequent segments of the input signal. The impulse responses significantly vary in size, so it would be beneficial to use a shared or constant memory for shorter impulse responses, while longer responses might better fit in a texture memory. *Overlap-Add* method can be also examined for further optimization.

REFERENCES

[1] D. B. Kirk, and W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2010.

[2] V. Valimaki, J. D. Parker, L. Savioja, J. O. Smith and J. S. Abel, “Fifty Years of Artificial Reverberation”, *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 20, no. 5, 2012., pp. 1421-1448.

[3] V. Ilic, M. Misic, and M. Tomasevic. “Application of graphics processing units in audio signal processing”, 20th Telecommunications Forum (TELFOR), IEEE, 2012., pp. 1616-1619. (In Serbian)

[4] D. V. Nikolov, M. J. Misic, and M. V. Tomasevic, “GPU-based implementation of reverb effect”, 23rd Telecommunications Forum Telfor (TELFOR), IEEE, 2015., pp. 990-993.

[5] N. Tsingos, W. Jiang, and I. Williams, “Using programmable graphics hardware for acoustics and audio rendering”, *Journal of the Audio Engineering Society*, vol. 59, no. 9, 2011., pp. 628-646.

[6] F. Fabritius, *Audio processing algorithms on the GPU*, MSc. thesis, Technical University of Denmark, DTU, DK-2800 Kgs. Lyngby, Denmark, 2009.

[7] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and implementation*, John Wiley, 2007.

[8] J. Kepner, and J. Lebak, J., “Software technologies for high-performance parallel signal processing”, *Lincoln Laboratory Journal*, vol. 14, no. 2, 2003., pp. 181-198.

[9] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, “High performance discrete Fourier transforms on graphics processors”, *Proc. ACM/IEEE Conf. Supercomputing*, November, 2008., p. 2.

[10] F. Wefers, J. Berg, “High-performance real-time FIR-filtering using fast convolution on graphics hardware”, *Proc. of the 13th Conference on Digital Audio Effects*, September, 2010.

[11] J. A. Belloch, J. Parker, L. Savioja, A. Gonzalez, V. Valimaki, “Dynamic range reduction of audio signals using multiple allpass filters on a GPU accelerator”, Proceedings of the 22nd European Signal Processing Conference (EUSIPCO), IEEE, 2014., pp. 890-894.

[12] J. A. Belloch, A. Gonzalez, F. J. Martínez-Zaldívar, A. M. Vidal, “Multichannel massive audio processing for a generalized crosstalk cancellation and equalization application using GPUs”, *Integrated Computer-Aided Engineering*, vol. 20, no. 2, 2013., pp. 169-182.

[13] J. A. Belloch, A. Gonzalez, F. J. Martínez-Zaldívar, A. M. Vidal, “Real-time massive convolution for audio applications on GPU”, Journal of Supercomputing, vol. 58, no. 3, 2011., pp. 449-457

[14] C. Wakeland, “Audio benefits and challenges of heterogeneous computing”, *AMD Fusion Developer Summit*, June 2011.

[15] *NVIDIA CUDA C Programming Guide*, version 7.0, NVIDIA Corporaton, 2015.

[16] E. Battenberg, A. Freed, and D. Wessel, “Advances in the parallelization of music and audio applications”, *Proc. of the International Computer Music Conference*, New York, 2010.

[17] Dutilleux, P., Zölzer, U., “Filters“, DAFX: Digital Audio Effects, John Wiley & Sons, 2002.

[18] *Libsndfile C library*, http://www.mega-nerd.com/libsndfile/, Accessed October 4th, 2015.

[19] *FFTW project - The Fastest Fourier Transform in the West*, http://www.fftw.org/, Accessed October 4th, 2015.

[20] *NVIDIA cuFFT Library User's Guide*, version 7.0, NVIDIA Corporaton, 2015.

[21] *GPU Impulse Reverb VST*, http://gpuimpulsereverb.de/, Accessed October 4th, 2015.