International Journal of Electrical and Computer Engineering (IJECE)
Vol. 8, No. 1, February 2018, pp. 70~75
ISSN: 2088-8708, DOI: 10.11591/ijece.v8i1.pp70-75

Optimizing Apple Lossless Audio Codec Algorithm using NVIDIA CUDA Architecture

Rafid Ahmed, Md. Sazzadul Islam, Jia Uddin
Department of Computer Science & Engineering, BRAC University, Bangladesh

Article Info

Article history:

Received May 4, 2017
Revised Oct 3, 2017
Accepted Oct 17, 2017

Keyword:

ALAC
Audio decoding
Audio encoding
CUDA
PCM
SIMT

ABSTRACT

As majority of the compression algorithms are implementations for CPU architecture, the primary focus of our work was to exploit the opportunities of GPU parallelism in audio compression. This paper presents an implementation of Apple Lossless Audio Codec (ALAC) algorithm by using NVIDIA GPUs Compute Unified Device Architecture (CUDA) Framework. The core idea was to identify the areas where data parallelism could be applied and parallel programming model CUDA could be used to execute the identified parallel components on Single Instruction Multiple Thread (SIMT) model of CUDA. The dataset was retrieved from European Broadcasting Union, Sound Quality Assessment Material (SQAM). Faster execution of the algorithm led to execution time reduction when applied to audio coding for large audios. This paper also presents the reduction of power usage due to running the parallel components on GPU. Experimental results reveal that we achieve about 80-90% speedup through CUDA on the identified components over its CPU implementation while saving CPU power consumption.

Corresponding Author:

Jia Uddin,
Department of Computer Science & Engineering,
BRAC University.
66 Mohakhali, Dhaka 1212, Bangladesh.
Email: jia.uddin@bracu.ac.bd

1. INTRODUCTION

Best use of expensive computing resources such as memory, network bandwidth or processing units is growing day by day. As a result, consumption of those resources needs to be carefully planned in order to achieve maximum performance. Data compression helps to utilize space limited resources more effectively. There are several algorithms on lossless audio codecs; Free Lossless Audio Codec (FLAC) [1], Shorten [2], Windows Media Audio (WMA) [3], MPEG-4 [4], WavPack, Monkey's Audio [5] and others being used by programs to alleviate space usage. Lossless audio compression detects and discards statistical redundancy in audio file in order to get reduced bit-rate and this process is generally obtained by implementing linear prediction for redundancy removal and the entropy encoder for coding the predictor output [6]. There are also some tradeoffs on the decision of using compression. One of the main issues is increase in encoding/decoding time as well as growth of power consumption.

Apple Lossless Audio Codec (ALAC) is an audio coding format developed by Apple Inc. for lossless data compression of digital music. Filename extension .m4a is used for Apple Lossless data to store within an MP4 container. This extension is also used by Apple for lossy AAC audio data which is not a variant of ALAC [7]. An iTunes .m4p file is a DRM-encrypted M4A which isn't interoperable outside of the Apple ecosystem. As a result, other lossless codecs, such as FLAC and Shorten, are not natively supported by Apple's iTunes software. So, users of iTunes software who want to use a lossless format have to use ALAC [8]. Apple Lossless supports bit depths of 16, 20, 24 and 32 bits and any arbitrary integer sample rate from 1 to 384,000 Hz.

In comparison with the traditional GPGPU techniques, CUDA offers several advantages, such as shared memory, faster downloads, fully support for integer and bitwise operations and also CUDA code is easily understandable than OpenCL [9]. The process flow of CUDA programming is described below [10]:

a. Data is copied from the main memory to the GPU global memory
b. CPU sends the processing instructions to the GPU
c. Each core at the GPU memory parallelly executes the data
d. Result from GPU global memory is copied back to main memory

In this paper, we propose an implementation of ALAC audio compression on NVIDIA GPUs. ALAC algorithm follows the same basic principle of other lossless audio compression illustrated in Figure 1 [11]. It is a highly serialized algorithm which is not efficient enough to be used on GPU. For example, as the input audio file is separated into packets, the encoding of next packet depends on the results of previous encoded packet. Our redesigned implementation for the CUDA framework aims to parallelly implement the parts of the algorithm where the computation of a method is not serialized by previous computations.

The paper is presented as follows. Section 2 describes the implementation of CUDA model on ALAC Encoding and Decoding process. The experimental result with hardware and software setups are discussed in section 3 and finally conclusions will be made in section 4.

2. RESEARCH METHOD

2.1. SIMT Model

The GPU core contains a number of Streaming Multiprocessors (SMs). On each SM, execution of each instruction follows a model like SIMD which is called SIMT referred by Nvidia. In SIMT, the same instruction is assigned to all the threads in the chosen warp. All threads in a warp are issued the same instruction, although not every thread needs to execute that instruction. As a result, threads in a warp diverging across different paths in a branch results in a loss of parallelism [12]. In our implementation, the branching factor is handled in CPU before calling the kernel to gain total parallelism. The kernel block size must be chosen less than or equal to tile size such that one or more elements of a tile is loaded into shared memory by each thread in a block. Resultantly, the device performs one instruction fetch for a block of threads which is in SIMT manner. This shortens instruction fetch and processing overhead of load instruction [13]. In the tests, we determine that 512 threads per block configuration is giving the best performance. In the CUDA implementation of ALAC algorithm, we have decided to exploit the framing and mixing phase of encoding. The decoding of an ALAC audio to pcm (pulse code modulation) data are done by following the steps of Figure 1 reversely. So, for the decoding section, we attempted to parallelize the un-mixing and concatenating phase.

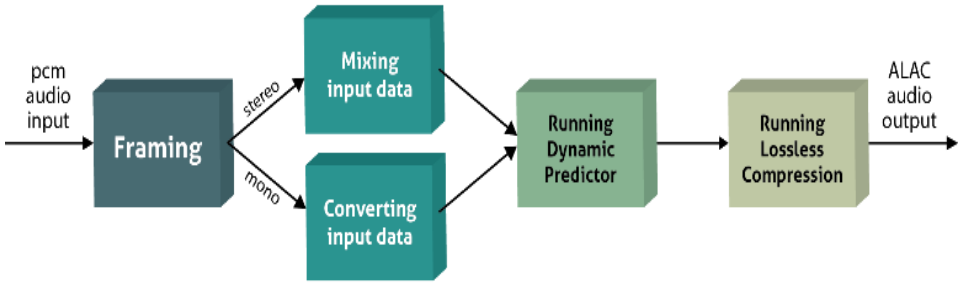

Figure 1. The basic operation of ALAC encoder

2.1.1. SIMT Model in Encoding

In serial CPU implementation of ALAC, the input data is divided to several frames by the encoding phase, where a frame is split into even smaller pieces to carry out mixing operation. As shown in Figure 2, the possible way to utilize this serialization of framing and mixing of encoding a stereo audio is to batch all the input packets into CUDA global memory for a single parallel operation of mixing the data as shown in Figure 3 taking advantage of SIMT nature.

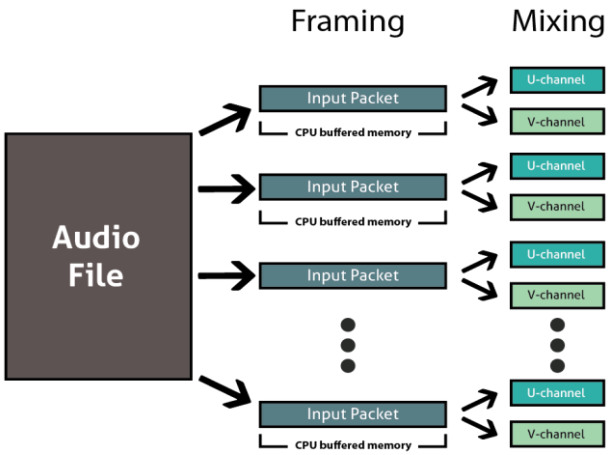

Figure 2. CPU implementation of Framing and Mixing phase

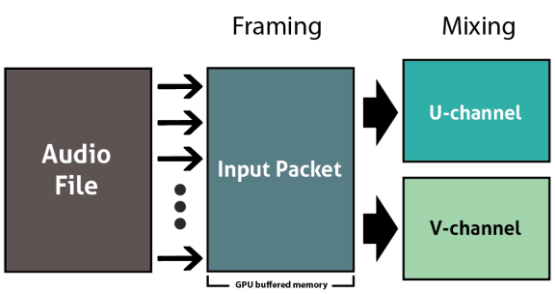

Figure 3. GPU implementation of Framing and Mixing phase

2.1.2. SIMT Model in Decoding

The decoding process is reverse of the encoding process. First ALAC audio data is decompressed. Then the predictors are run over the data to convert it to pcm data. Finally, for stereo audio, un-mix function is carried out to concatenate the 2 channels into a single output buffer. As the same independent behavior exist in the decoding process to make use of the data parallelism in CUDA, we distribute the work of the end of the decoding process across the GPU.

2.2. Memory Coalescing

According to Jang B, Schaa D, et al, GPU memory subsystems are designed to deliver high bandwidth versus low-latency access. To gain highest throughput, a large number of small memory accesses should be buffered, reordered, and coalesced into a small number of large requests [14].

For strided global memory access, the effective bandwidth is always poor. When concurrent threads simultaneously access memory addresses that are located far apart in physical memory, there is no possibility for coalescing the memory access [15]. Thus, we restructure the buffer in such a way that global memory loads issued by threads of warp are coalesced in encoding process as show in Figure 4.

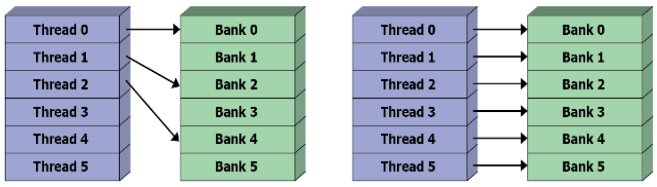

Figure 2. Uncoalesced and coalesced memory access

2.3. Pinned Memory

By default, CPU data allocations are pageable. As CUDA driver's uses page-locked or pinned memory, GPU cannot directly access data prom pageable host memory. Consequently, the GPU first allocates a temporary pinned memory, copies host data into it and then transfers data to the device memory. So, it is better to use pinned memory beforehand rather than using paged memory in order to reduce copying cost [16]. As we have to transfer channel buffers from GPU to CPU for each iteration while encoding frames, we kept the channel buffer of CPU as pinned memory for fast transfer of data.

3. RESULTS AND ANALYSIS

To analyze the performance of CUDA implementation, we used a GeForce GTX 960 card with CUDA version 7.5 installed on a machine with Intel(R) Core(TM) i5-4590 CPU running at 3.30GHz. The

CPU implementation of ALAC is also tested on the same testbed. To compare the performance of our GPU implementation of ALAC algorithm, we selected representative of five audio files for testing. All these files have the CD format, i.e. 44.1kHz, 16 bits.

a. Track 5 of [17] (28 s): Electronic Gong 5kHz (Mono)
b. Track 20 of [17] (39 s): Saxophone
c. Track 50 of [17] (22 s): Male speech (English)
d. Track 68 of [17] (2 min 44 s): Orchestra
e. Track 70 of [17] (21 s): Song by Eddie Rabbitt

The audio files are collected from Sound Quality Assessment Material (SQAM) recordings for subjective tests 9. The flies were converted to WAV from FLAC to work with pcm data. To measure the results, we ran our test 10 times on each dataset for each reading and showed the average running results in this section.

3.1. Encoding Results

3.1.1. Execution Speed Gain

The results on Table 1 shows the average running speed results of both mono and stereo audio (Track 68) for CPU and GPU separately. For 16bit audio we achieve speed gain about 67% for mono and 85% for stereo audio type. For 24bit audio the speed up is around 90% for both audio types. For 32bit audio, the speed increase is around 81%. Here we can infer that 24bit audio conversion results in faster encoding speed for GPU where for 16bit audio it is slower than the others.

In Figure 5, we see for first test file (Track 5) we achieve 3x speed up, for second and third test file (Track 20 & 50) we achieve 7x speed up, for fourth test file (Track 68) we gain 6.5x speed and lastly for fifth file (Track 70) we get 5x speed up.

Table 1. Mixing/Converting phase execution time comparison for encoding process

Bit Depth	CPU execution time for mono (ms)	GPU execution time for mono (ms)	CPU execution time for stereo (ms)	GPU execution time for stereo (ms)
16	2.85	0.94	20.3	3.0259
24	25.68	1.62	63.5	5.246
32	9.32	1.71	38	4.8266

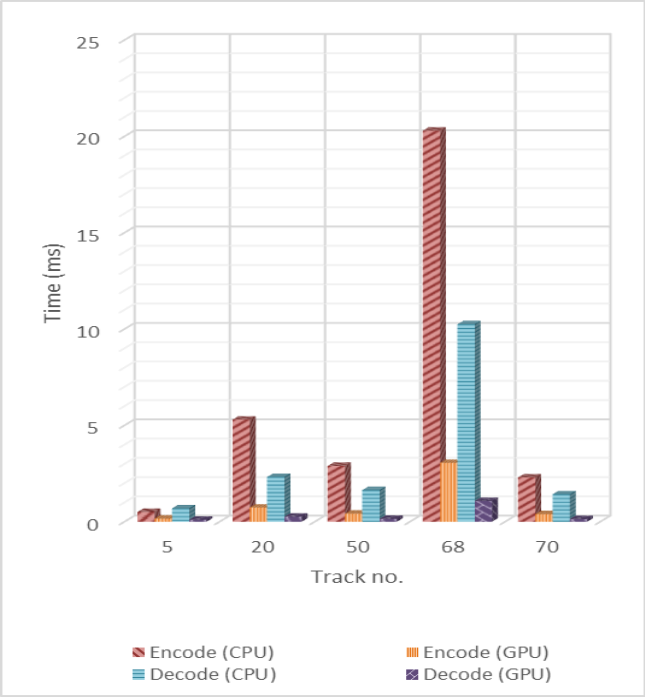

Figure 3. Mixing/Converting phase speed up for encoding and decoding process against CPU using dataset

3.1.2. Power Consumption Saving

Looking at Table 2, the power consumption saving is least for 16bit audio for both mono and stereo audio files where for 24bit audio power saving is the highest.

Table 2. CPU power saving in encoding process

Bit Depth	Power consumption saving for mono (Watt)	Power consumption saving for stereo (Watt)
16	0.66792	3.636927
24	5.49444	9.209082
32	2.0544	7.257163

3.2. Decoding Results

3.2.1. Execution Speed Gain

Decoding results for both mono and stereo file (Track 68) are shown in Table 3. According to the results, we get around 85% speed increase for 16bit mono audio and 90% for 16bit stereo. For 24bit audio, we achieve more than 90% speed up. Lastly for 32bit audio, around 89% speed up is gained. From this table, we can also state that 24bit audio decoding is faster in GPU than the others where 16bit audio conversion is slowest.

Table 3. Un-mixing/Converting phase execution time comparison for decoding process

Bit Depth	CPU execution time for mono (ms)	GPU execution time for mono (ms)	CPU execution time for stereo (ms)	GPU execution time for stereo (ms)
16	4.02	0.5892	10.25	1.0764
24	10.884	0.8352	28.032	1.8192
32	7.093	0.864	20.354	1.715

3.3.2. Power Consumption Saving

For decoding stage, we get less power saving than that of encoding stage. Here also 16 bit audio has the least power saving where 24 bit audio has the highest shown in Table 4.

Table 4. CPU power saving in decoding process

Bit Depth	Power consumption saving for mono (Watt)	Power consumption saving for stereo (Watt)
16	0.690414	1.577814
24	1.85396	4.379118
32	1.124886	3.087194

4. CONCLUSION

In this paper, we analyzed the feasibility to use CUDA framework for Apple Lossless Audio Codec compression algorithm. Our primary focus was on outperforming the mixing/un-mixing speed of the CPU based ALAC implementation by using NVIDIA GPUs without losing any compression ratio. We tested our implementation on several datasets and made comparison. Experimental results demonstrate that we achieve average of 80-95% speed up for mixing/un-mixing audio data.

REFERENCES

[1] Firmansah L, Setiawan EB, *"Data audio compression lossless FLAC format to lossy audio MP3 format with Huffman shift coding algorithm,"* 2016 4th International Conference on Information and Communication Technology (ICoICT). 2016 May.

[2] Salomon D. A concise introduction to data compression. London: Springer London; 2008 Jan 14. 227–44 p. ISBN: 9781848000711.

[3] Waggoner B, "Compression for great video and audio: Master tips and common sense," 2nd ed. Oxford, United Kingdom: Elsevier Science; 2009 Nov 23. 341–7 p. ISBN: 9780240812137.

[4] Yu R, Rahardja S, Xiao L, Ko CC, "A fine granular scalable to lossless audio coder," IEEE Transactions on Audio, Speech and Language Processing. 2006 Jul; 14(4):1352–63.

[5] Salomon D, Motta G, Bryant D, "Data compression: The complete reference," 4th ed. London: Springer London; 2006 Dec 19. 773–83 p. ISBN: 9781846286025.

[6] Gunawan T, Zain M, Muin F, Kartiwi M, "Investigation of Lossless Audio Compression using IEEE 1857.2 Advanced Audio Coding," *Indonesian Journal of Electrical Engineering and Computer Science*, 2017;6: 422 – 430

[7] Apple Lossless [Internet]. Applelossless.com. 2017 [cited 23 September 2017]. Available from: http://www.applelossless.com/

[8] Waggoner B, "Compression for great video and audio: Master tips and common sense," 2nd ed. Oxford, United Kingdom: Elsevier Science; 2009 Nov 23. 215–21 p. ISBN: 9780240812137.

[9] Guo S, Chen S, Liang Y, "A Novel Architecture of Multi-GPU Computing Card," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 2013;11(8)

[10] Nasreen A, G S, "Parallelizing Multi-featured Content Based Search and Retrieval of Videos through High Performance Computing," *Indonesian Journal of Electrical Engineering and Computer Science*, 2017;5(1):214

[11] Hans M, Schafer RW. Lossless compression of digital audio. IEEE Signal Processing Magazine. 2001, Jul: 18(4):5–15.

[12] Sutsui S, Collet P, editors, "Massively parallel evolutionary computation on GPGPUs," Berlin: Springer-Verlag Berlin and Heidelberg GmbH & Co. K; 2013 Dec 6. 149–56 p. ISBN: 9783642379581.

[13] Al-Mouhamed M, Khan A ul H, "Exploration of automatic optimization for CUDA programming," *International Journal of Parallel, Emergent and Distributed Systems*, 2014 Sep 16;30(4):309–24.

[14] Jang B, Schaa D, Mistry P, Kaeli D, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, 2011 Jan;22(1):105–18.

[15] Harris M. [Internet]: Parallel For all. How to access global memory efficiently in CUDA C/C++ kernels; 2013 Jan 7 [cited 2016 Nov 14]. Available from: https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/

[16] Harris M. [Internet]: Parallel For all. How to optimize data transfers in CUDA C/C++; 2012 Dec 4 [cited 2016 Nov 14]. Available from: https://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[17] Sound quality assessment material: recordings for subjective tests; user's handbook for the EBU - SQAM compact disc. Geneva: Technical Centre; 2008. [cited 2016 Nov 14]. Available from: https://tech.ebu.ch/docs/tech/tech3253.pdf.