Optimizing Apples Lossless Audio Codec Algorithm using NVIDIA CUDA

Supervisor: **Dr. Jia Uddin**

Rafid Ahmed 13101209

Md. Sazzadul Islam 13201081

Department of Computer Science and Engineering,

BRAC University

Submitted on: 14th December 2016

We would like to dedicate this thesis to our

Parents
and
Teacher...

DECLARATION

We hereby declare that this report is our own work and effort and that is has not been submitted anywhere for any award. All the contents provided here is totally based on our own labor dedicated for the completion of the thesis. Where other sources of information have been used, they have been acknowledged and the sources of information have been provided in there reference section.

Signature of Supervisor

Dr. Jia Uddin

Signature of Author

Md. Sazzadul Islam

Rafid Ahmed

ACKNOWLEDGEMENTS

First of all, we would like to express our deepest sense of gratitude to almighty Allah. Secondly, I would like to express my sincere gratitude to my advisor DR. Jia Uddin for the continuous support of my research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time.

Finally, we would like to express our sincere gratefulness to our beloved parents, brothers and sisters for their love and care. We are grateful to all of our friends who helped us directly or indirectly to complete our thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

As majority of the compression algorithms are implementations for CPU architecture, the primary focus of our work is to exploit the opportunities of GPU parallelism in audio compression. We present an implementation of Apples Lossless Audio Codec (ALAC) algorithm by using NVIDIA GPUs Compute Unified Device Architecture (CUDA) Framework. The core idea is to identify the areas where data parallelism can be applied and parallel programming model CUDA is used to execute the identified parallel components on Single Instruction Multiple Thread (SIMT) model of CUDA. The dataset is retrieved from European Broadcasting Union, Sound Quality Assessment Material (SQAM). Faster execution of the algorithm leads 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.

Chapter 1

Introduction

In 21st century use of computer increased exponentially and also increasing size of data increasing comparatively. To cope up with the data size there are different compression technique we can see. There are different kind of compression for example, Multimedia data such as audio, video, image compression, text compression etc. Therefore, importance compression is increasing due to demand. Some algorithms are really slow and some are not. As we can use GPU to compress data or files to make those algorithm faster will help to execute big size of data at a time.

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 compression; FLAC [1], ALAC [2], WavPack [3], Monkey's Audio [4], OptimFrog [5], TTA [6] and others being used by programs to alleviate space usage. 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.

In this paper, we propose an implementation of Apples Lossless Audio Codec audio compression on NVIDIA GPUs. It is a highly serialized algorithm which is not efficient enough to be used on GPU. For example, computing the separate channels from the input file is done by using a number of conditional branches which limits the efficiency of a GPU thread. Our redesigned implementation for the CUDA framework aims to reduce the effect of compression time compared to CPU based compression implementations.

1.1 Problem Definitions

We will implement ALAC in parallel using CUDA C. But for that we need to know how the algorithm works, which part is data dependent which part is data independent because to implement in parallel using CUDA data independent plays a vital role in performance

optimization. Furthermore, we need to know how to algorithm works by which audio file such as .wav or .mp3 can convert into playable codec .caf . In ALAC, there is encoder and decoder. For both we need to break the system or algorithm into small segments. After that we will define the data dependent and independent part. In other word we will look for the codes that is possible to parallel and those codes whom are sequential and then will be able to optimize according to its need.

1.2 Motivations

We understood the importance of compression in modern age because it is easy to keep or delete big size of data but storage remain kind of same for certain time therefore, compression enhance that possibility to store more data. Furthermore, execution time plays a vital role in performance of a algorithm and implementing ALAC in CUDA will help to reduce the execution time which will be useful for converting big size file into .caf format. Furthermore, as it is codec so it is playable for after converting the file. Considering the fact, we implemented ALAC in CUDA using CUDA C from NVIDIA GPU

1.3 Contribution Summary

The summary of the main contributions is as follows:

- We used CUDA toolkit and CUDA C programming language for implementing
- We applied all possible optimization over CUDA to make as much parallel as possible. Such as using technique memory coalescing etc.
- Then we tried to Optimize power consumption so that this algorithm use less power than usual
- By using CPU-GPU based implementation of our proposed work, the whole process become more faster in execution time and also use less power consumption.

1.4 Thesis Orientation

The rest of the thesis is organized as follows:

- Chapter 02 includes the necessary background information regarding the proposed approaches of Apple Lossless Audio Codec (ALAC) in parallel using CUDA.
- Chapter 03 presents the methods and implementation details for ALAC in parallel.
- Chapter 04 demonstrates the experimental results and comparison.

- Chapter 05 concludes the thesis and states the future research directions.

Chapter 2

Background Information

2.1 GPU Architecture

Heterogeneous architectures that are consist of Central Processing Units (CPU) and computational accelerators, for example, Graphics Processing Units (GPU) have been adopted in large number of supercomputer, desktop for engineering or scientific work or workstations. This gives massive parallel computing capabilities to the users while preserving the flexibility given by CPU for various workload. However, efficiently exploiting GPUs' full performance may become complex due to the programming challenges faced when mapping computational algorithms to hybrid and heterogeneous architectures [7]. One of the trends in GPU is parallel global optimization. In last decade, productivity of GPU has increased notably. Today a GPU is high performance flexibly programmable and massively parallel processor that provide solution to many intensive problems [8].

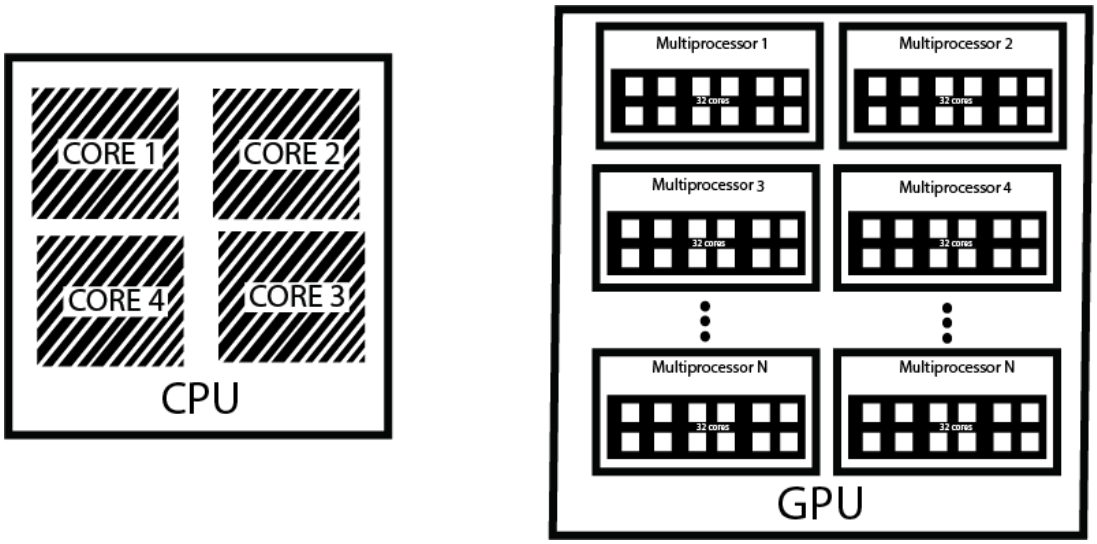

Figure 2.1: GPU Architecture.

Table 2.1: CPU VS GPU [9]

CPU	GPU
Really fast caches (great for data reuse)	Lots of math units
Fine branching granularity	Fast access to onboard memory
Lots of different processes/threads	Run a program on each fragment/vertex
High performance on a single thread of execution	High throughput on parallel tasks
CPUs are great for task parallelism	GPUs are great for data parallelism
CPU optimized for high performance on sequential codes (caches and branch prediction)	GPU optimized for higher arithmetic intensity for parallel nature (Floating point operations)

2.2 NVIDIA GPU Architecture

There are few differences between GPU and CPU processor architecture. NVIDIA's GPU consist of multiple streaming multiprocessor (SMs) and each streaming multiprocessor consist of many scalar processor also known as cores. NVIDIA provided different architecture of GPU such as Kepler, Fermi etc.

2.2.1 Till NVIDIA G70

Till NVIDIA's G70 GPU and there last generation of architecture handled by vertex and pixel shading in multiple dedicated units. They implemented array system where top of array was used to handle to vertex processing of 8 shaders and pixel processing was managed in middle of the array of 24 shaders. Sometime pixel shaders remain idle until data were passed through from vertex shaders. This counting problem, idle sitting of hardware is the reason that NVIDIA had to shift their directions to new architecture [10].

2.2.2 G80 and Tesla

After G70 GPU architecture failed to cope up with the speed NVIDIA come up with G80 architecture. It helps to solve many issue which g70's could solve, such as unused shader hardware etc. This GPU was the first GPU which has unified shader with 128 processing elements which distributed in 8 shader core [10]. but in G80 architecture the scheduler can prioritize and allocate shader to all execution unit. Due to we could increase the number of

vertex shader in cores easily the opportunity for increasing performance was available [11]. In this time to keep with the speed and performance increase they introduced Compute Unified Device Architecture (CUDA). This was C based development environment for GPU [12]. To bring more efficiently and advantage Tesla product's came in [10].

2.2.3 Fermi

The Fermi architecture is one of the most significant step forward towards GPU architecture since the Original G80. Whole new approach was taken for creating first computational GPU. It was improved in few area such as, Improve Double Precision Performance, True Cache Hierarchy, ECC support, Faster Context Switching, Faster Atomic Operations and More Shared Memory.

The Fermi allows increased compute capacity with new innovation which increased programmability and computational ability.

2.2.3.1 Key Notes of The Fermi Architecture

- First Fermi based GPU had 3 billion transistor which consist of 512 CUDA cores. Per CUDA core can executes a floating point or integer instruction per clock for a tread.
- The CUDA cores are organized in 16 streaming Multiprocessor (SM) of 32 cores each.
- GPU has a six 64 bit memory partition for a 384 bit memory interface.
- At maximum supports 6GB GDDR5 DRAM memory.
- The GPU to CPU for host interface are connects with PCI-Express.
- Thread blocks to SM tread Scheduler has been distributed by Giga Thread Global Scheduler.

2.2.4 Kepler Architecture

Kepler GPU microarchitecture was introduced by NVIDA after Fermi. This architecture was focused on power efficiency. GeForce series from 600 to 700 and some of 800 series used Kepler architecture. Later on Kepler was replaced by Maxwell architecture. In this GPU microarchitecture NVIDIA developer focused on efficiently use of power and also programmability and performance meanwhile architecture before this was focusing on increasing performance [13][14]. To be precise, two Kepler cores use around 90% of power of a Fermi cores where unified GPU clock reduce 50% power consumption [15].

Kepler GPU has different configuration of Graphics Processing Cluster (GPC), Streaming Multiprocessor (SM) and memory controller.

2.3 CUDA Overview

In this part we are going to talk about CUDA and some of its basic units.

2.3.1 CUDA

CUDA's full form is Compute Unified Device Architecture which is NVIDIA GPU architecture that is in GPU card. It has positioned itself as a whole new meaning for general purpose computing with GPUs. CUDA uses extension of c++ known as CUDA C for programming purpose. CUDA provides advantage of huge computational power to the programmer [9].

CUDA has 128 co-operating cores. Here cores can communicate and also they can exchange information with each other so that, running multithreaded application there is no need for streaming computing in GPU. CUDA is popular because it provides huge freedom for programmer to work on but also programmer need to have skill to use that efficiently. CUDA is not useful for some serial algorithm but algorithms which we can break into small parts and send it to thread to work on it enhance power of it a lot [9].

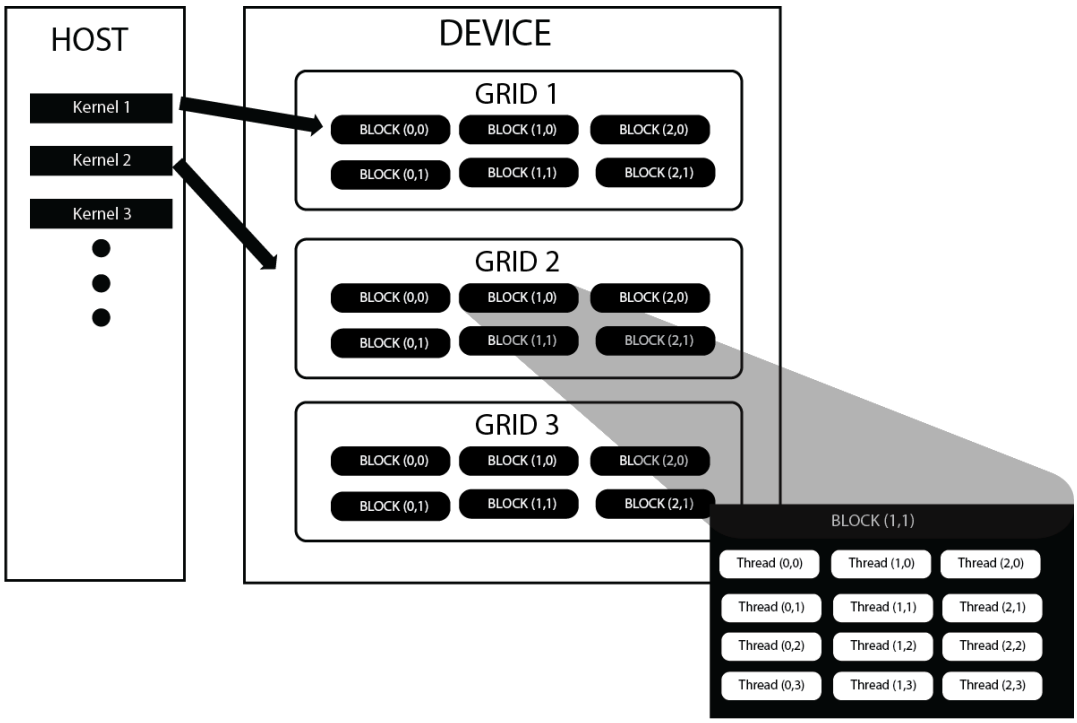

Figure 2.2: CUDA architecture

As previously mentioned CUDA uses C programming language and main idea of CUDA is that GPU consist of thousands of thread that can execute in parallel. All those thread can execute same function or code known as kernel. Here all the treads are executed using same code but different data, for example multiplying 1 to 7 with 2. CUDA program consist of one or two part that is executed on either on host (CPU) or on device (GPU). When code consist of little parallelism it is better to use host because copying data from host to device takes time but when we can do huge parallelism it is better to use device because it overcome copy speed and come up with performance boost. The code in host is normal c code as it uses CPU to executes where codes in device uses different keywords such as "*__global__, __device__"* etc. For some cases kernels can also be executed if there is no GPU available. CUDA software development kit provides this options [9].

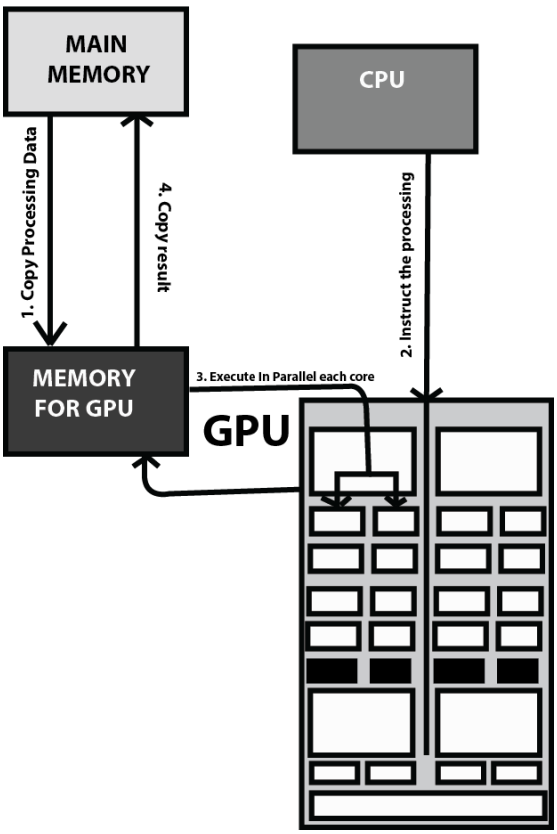

Figure 2.3: processing flow on CUDA

2.3.2 Processing flow on CUDA

Data independent part is send to GPU for parallelism where dependent part can be executable in CPU because compare to the speed of copy takes time which is no use for data

dependent code or sequential code so it is better to use them in CPU or host. We can see processing flow on CUDA in fig 2.3 [16].

2.3.3 Basic Units of CUDA

In this part we are going to talk about some basic units of CUDA that it uses to execute program and run a program.

2.3.3.1 Kernels

In CUDA C kernel is a special function that is been called to execute parallel functions. When kernel is call is executed program is executed N times in parallel by N different CUDA threads.

To differentiate call of kernel form C function <<<……>>> this is used in execution configuration syntax. Here configuration shows how many CUDA threads and block it will be needed to run in parallel. To make CUDA kernel we need to write *"__global__"* keyword in the start of function. There is a variable called threadIDX, blockIDX etc. which helps to keep count of thread or block etc. It is needed because all reads are given unique thread ID. Fig 2.4 shows how a kernel works for a simple addition function [17].

2.3.3.2 The Grid

A grid is a group of threads all running in the same kernel. Using one grid all call from CPU to CUDA is made. Even though starting a grid on CPU is a synchronous operation but multiple grids can run at once using multi GPU system. Grids cannot be shared between GPUs [9]. Fig 2.2 shows how grid works in GPU architecture.

2.3.3.3 The Block

Grid is combination of multiple blocks. Each block consist of multiple thread which run co-operatory. Similar to grid blocks are not shared in multiprocessors but all threads in a single block uses same program. There is a variable called *"blockIdx"* in CUDA to specify the current block number. Different blocks uses a number to specify and it can be 1D or 2D and using 1D or 2D depends of programs need [9]. Fig 2.2 shows how block works in GPU architecture.

2.3.3.4 The Thread

A block consist of different composed and co-operative threads. Threads are run on individual cores of multiprocessor but like grids or blocks it is not restricted to single core. Thread has its unique id where current thread can be identify in *"threadIDX"* variable. Thread can be 1D, 2D or 3D depends on blocks dimension. Threads have a decent number of register memory which is usually 512 per block [9]. Fig 2.2 shows how thread remains in GPU architecture.

2.3.4 Basic Memories of CUDA

There are some basic types of memories in CUDA. We are going to talk about them below.

2.3.4.1 Global Memory

It is read and write memory and it is slow on copying. It is uncached and needed sequential and aligned 16 byte read and write to be fast [9].

2.3.4.2 Texture Memory

It T is read only memory. For 2D spatial access patter its cache is optimized.

2.3.4.3 Constant Memory

Constant and arguments are stored here. It is slow but it has cache.

2.3.4.4 Shared Memory

All threads in a block can used shared memory for read and write operation. It is common between all threads in a block and its size is smaller than global memory [9].

2.3.4.5 Local Memory

It is generally used for whatever does not fit into register but it is slow and do not have cache. Allows automatic coalesced reads and writes [9].

2.3.4.6 Registers

Among these this is the fastest memory here. One set of register memory is given to each thread and it uses them for faster storage, retrieval or data's like counter etc. Mostly it is used by thread [9].

2.4 Optimization Possibility

GPUs are intensively parallel computing units that provides high parallelism and memory bandwidth in a low lost, energy efficient platform [3]. Within the NVidia's impressively power-efficient Maxwell architecture GPU, there are 1024 CUDA cores with 128-bit memory bus width and 112.16 GB/s total memory bandwidth [4].

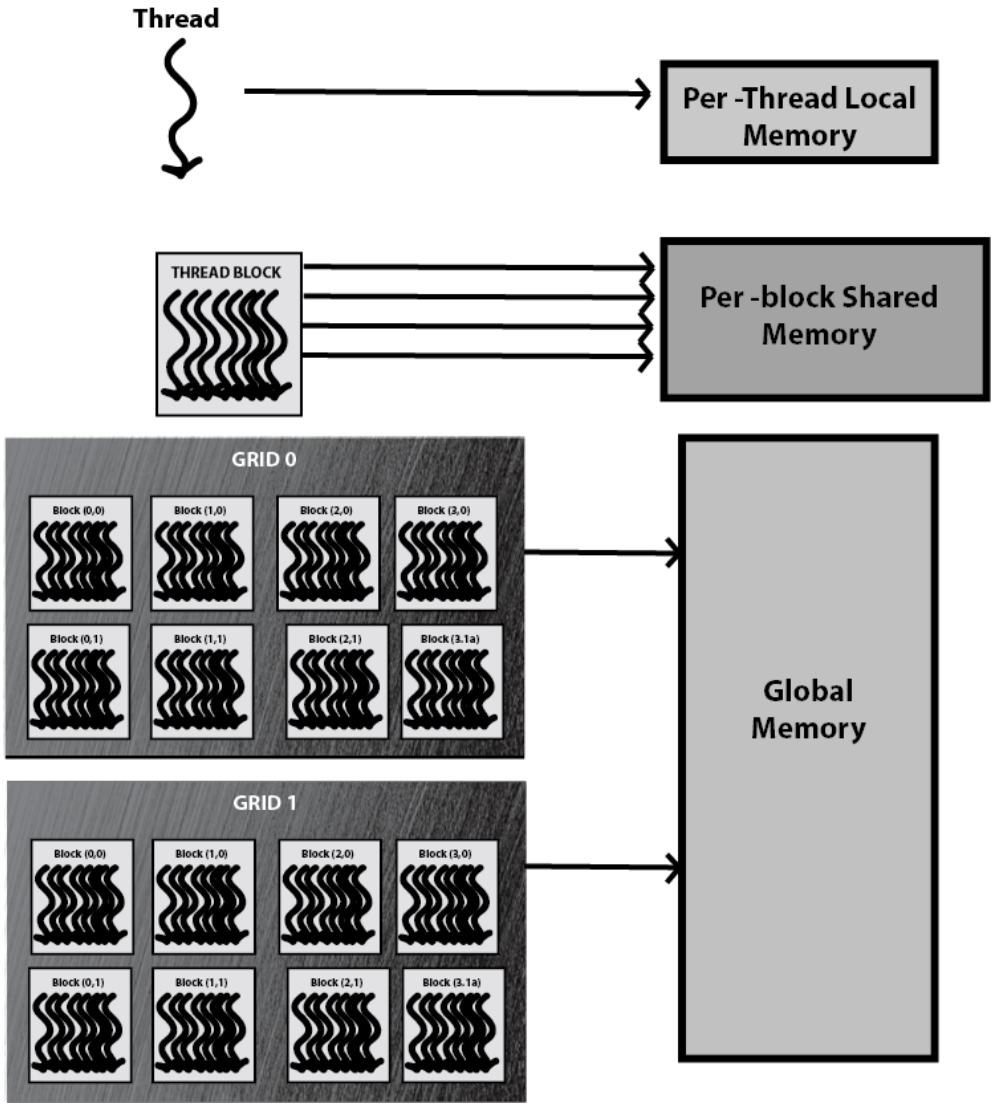

Figure 2.4: Memory Hierarchy

For accessing global memory, forming warps by a group of threads is necessary. Global memory loads and stores issued by threads of a warp are coalesced by the device into as less transactions as possible to minimize DRAM bandwidth. For strided global memory access, the effective bandwidth is poor regardless of architecture version. When concurrent threads simultaneously access memory addresses that are very far apart in physical memory, there is no chance for coalescing the memory access [5] as shown in Figure 2.5.

According to Schaa, D. and Kaeli, D., if dataset size is more than RAM's capacity then performance will degrade notably. Pinned memory makes code less portable as well. Therefore, pinned memory should only be allocated when RAM's memory space can fit dataset size [20].

CUDA drivers uses pinned memory, so it's better to use pinned memory rather than use paged memory and copying to pinned memory in order to reduce copying cost. Moreover, the device memory and device has more bandwidth than device memory and host memory therefore to achieve best performance, accesses of these memory have to be coalesced. Using of pinned memory instead of explicit copies between device and host memory can give better performance when mapped memory is read or written only once [18].

All data transfer has some overheads therefore, it's a problem for small transfers as it will result in more overheads. To reduce this overhead, it is convenient to batch many small transfers together into a single transfer [19].

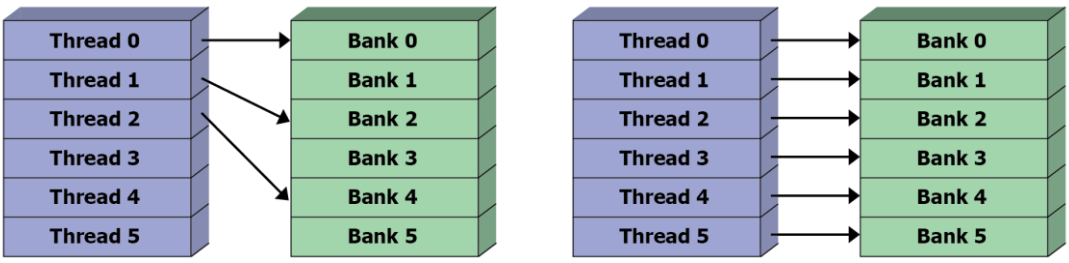

Figure 2.5. Uncoalesced and coalesced memory access

2.5 ALAC Algorithm

Apple Lossless, also known as Apple Lossless Audio Codec (ALAC), or Apple Lossless Encoder (ALE), is an audio coding format, and its reference audio codec implementation, developed by Apple Inc. for lossless data compression of digital music introduced in the following [18]. Other lossless codecs, such as FLAC and Shorten, are not

natively supported by Apple's iTunes software (either the Mac OS or Windows versions) or by iOS devices, so users of iTunes software who want to use a lossless format which allows the addition of metadata (unlike WAV/AIFF or other PCM-type formats, where metadata is usually ignored) have to use ALAC. Apple Lossless supports bit depths of 16, 20, 24 and 32 bits and any arbitrary integer sample rate from 1 to 384,000 Hz.

ALAC algorithm follows the same basic principle of other lossless audio compression by first separating the main audio file to packets of a fixed length and then mixing/converting input packet data depending on audio type. Then the algorithm remove redundancy in third step of figure 1 by a linear predictive modeling mathematical operation given by,

$$x'(n) = \sum_{i=1}^{p} a_i \, x(n-i) \qquad (1)$$

Where **x(n-i)** the previous observed values, and ai the predictor coefficients in (1). The error generated by this estimate is,

$$e(n) = x(n) - x'(n) \qquad (2)$$

Where **x(n)** is the true signal value in (2). The differences are found in the way the predictors ai are chosen. Finally, lossless compression is executed.

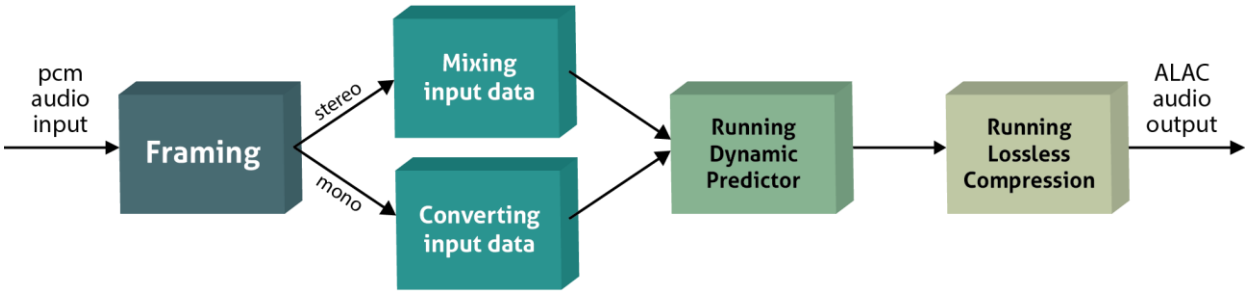

Figure 2.6. The basic operations of ALAC encoder

Chapter 3

Methods and Implementation Detail

In the CUDA implementation of ALAC algorithm, we have decided to exploit the framing and mixing phase of encoding discussed in the background section. The decoding of an ALAC audio to pcm (pulse code modulation) data are done by following the steps of Figure 3.1 reversely. So, for the decoding section, we attempted to parallelize the un-mixing and concatenating phase.

The steps of CUDA implementation in encoding and decoding stages discussed below.

3.1 Encoding

In serial CPU implementation of ALAC, the input data is divided to several frames by each encoding phase, where a frame is split into even smaller pieces to carry out mixing operation.

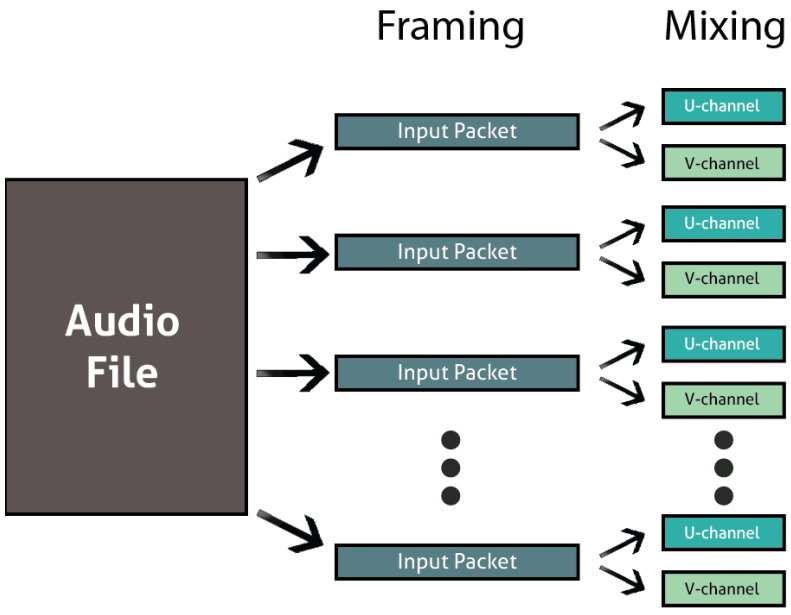

Figure 3.1. CPU implementation of Framing and Mixing

As shown in Figure 3.1, 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 3.2.

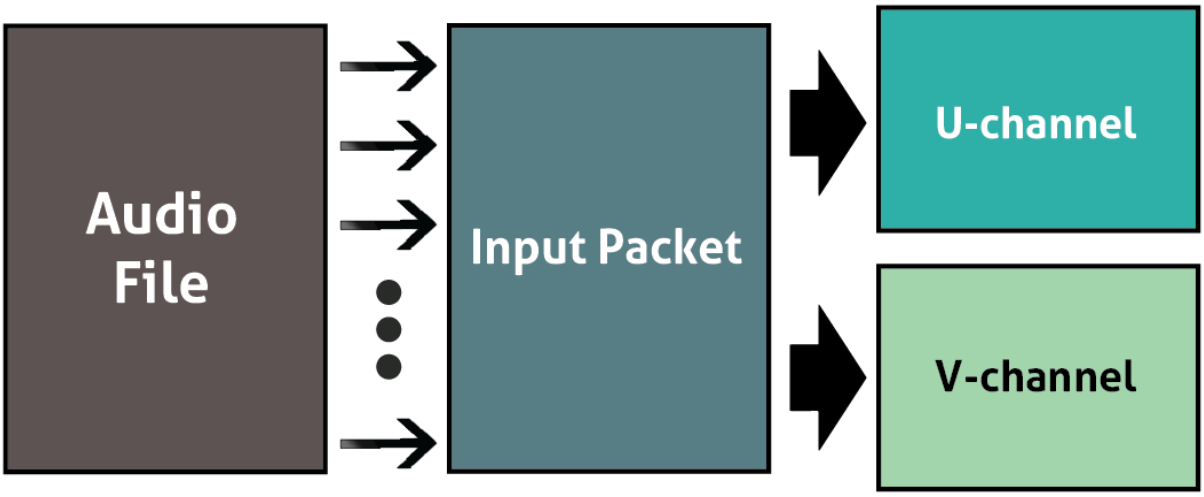

Figure 3.2. GPU implementation of Framing and Mixing

3.1.1 Steps of Encoding

Step 1: Copying pcm data from the host memory to the device memory via small batch transfers

Step 2: Calculating the grid size and block size for maximum parallelism

Step 3: Execute kernel for separating the U (right) channel and V (left) channel from the pcm data

Step 4: Copying the separate channel buffers back to host memory for running dynamic predictor and lossless operation.

For mono input, there is no optimization loop, so we convert the pcm data to 32-bit data for predictor in **step 3.**

3.2 Decoding

In the decoding process is reverse of the encoding process by first decompressing the ALAC audio data. Then the predictors are over the data to convert it to PCM data. Finally, 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. For this purpose, we keep an array of frame compression sizes that were recorded during encoding

process to get the actual length of the PCM data while concatenating U and V channels. In case of decoding mono channel, the uncompressed 32-bit data is converted to PCM data in place of un-mixing.

Chapter 4

Experimental Results

4.1 Testbed Configurations

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.

4.2 Datasets

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

- Track 5 of [1] (28 s): Electronic Gong 5kHz (Mono)
- Track 20 of [1] (39 s): Saxophone
- Track 50 of [1] (22 s): Male speech (English)
- Track 68 of [1] (2 min 44 s): Orchestra
- Track 70 of [1] (21 s): Song by Eddie Rabbitt

The audio files are collected from Sound Quality Assessment Material (SQAM) recordings for subjective tests [21]. 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 Figure 4.1.

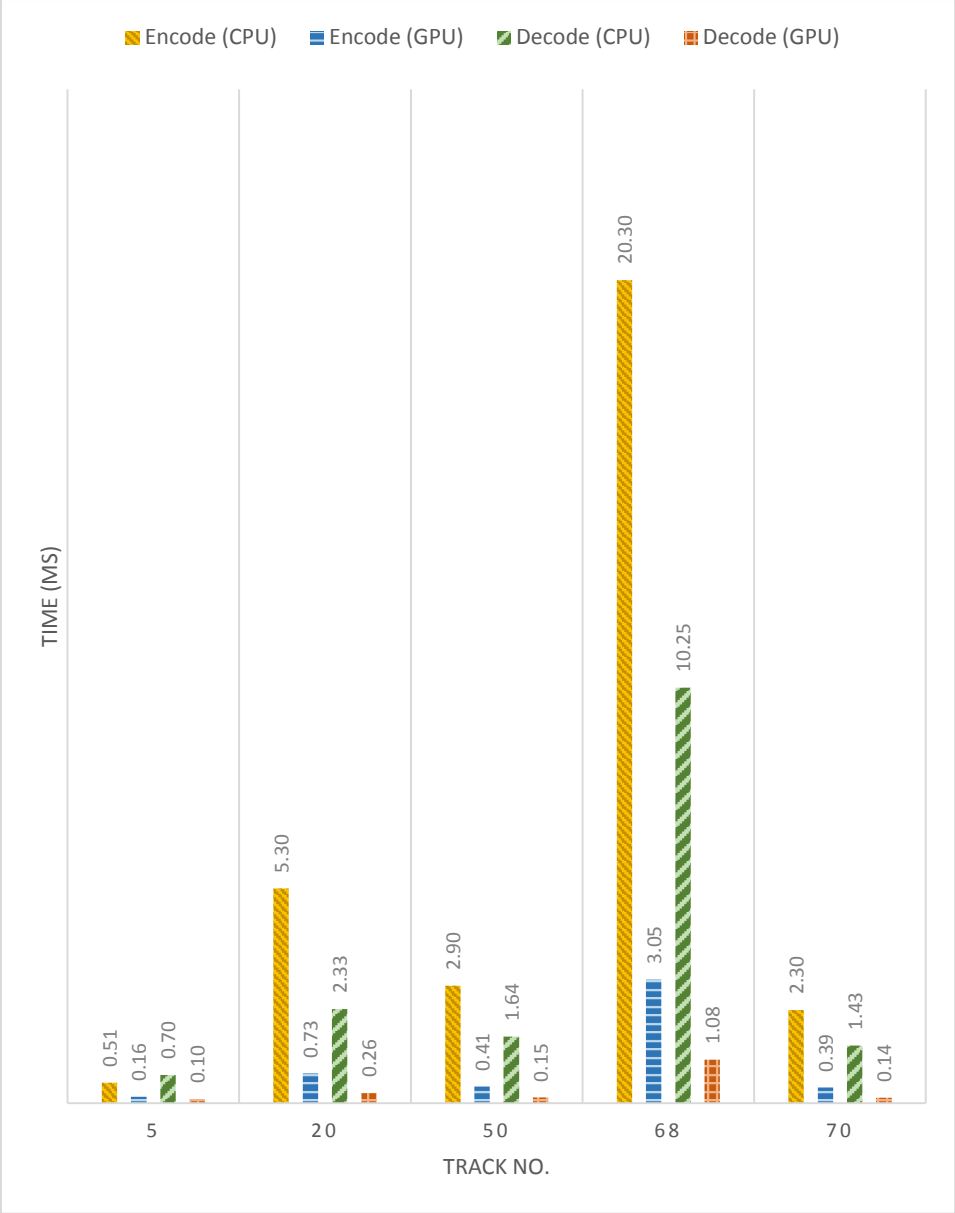

Figure 4.1. Mixing/Converting phase speed up for encoding and decoding process against CPU using dataset

4.3. Encoding Results

Here we are going to discuss about the execution speed gain for mono and stereo audio also power consumption for encoding.

4.3.1 Execution Speed Gain

In Figure 4.1, we see for first test file (Track 5) we achieve 3x speed up, for second and third test file (Track 20 and 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.

4.3.1.1 Mono Audio

Figure 4.2 shows the average running speed results of mono audio (Track 68) for CPU and GPU separately. For 16bit audio we achieve speed gain about 67% for mono audio type. For 24bit audio the speed up is around 93% 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.

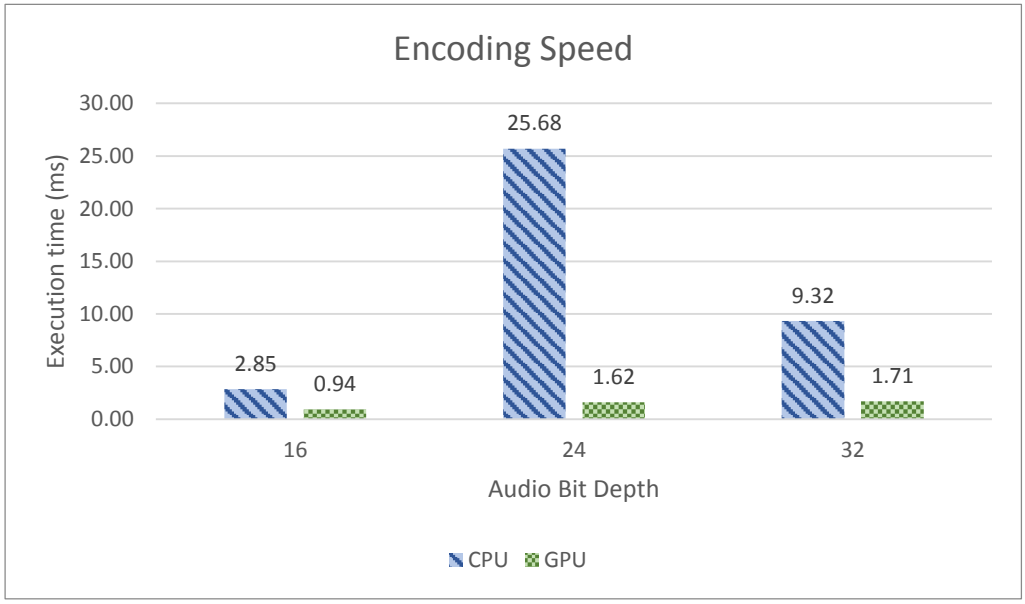

Figure 4.2. Converting phase execution time comparison for various bit depth audio encoding process

To show the consistency of the speed up, we measure the results shown in Figure 4.3 by running on cropped versions of classical piano music, 10 times for each set. The selected audio is a 44.1 kHz, 32-bit audio. The results on Figure 3 shows the average running speed results of mono audio file. For 10MB file size we achieve 80% encoding speed up, for 20 MB file the speed gain was 83% and for later file sizes the speed up was around 81% for mono audio file.

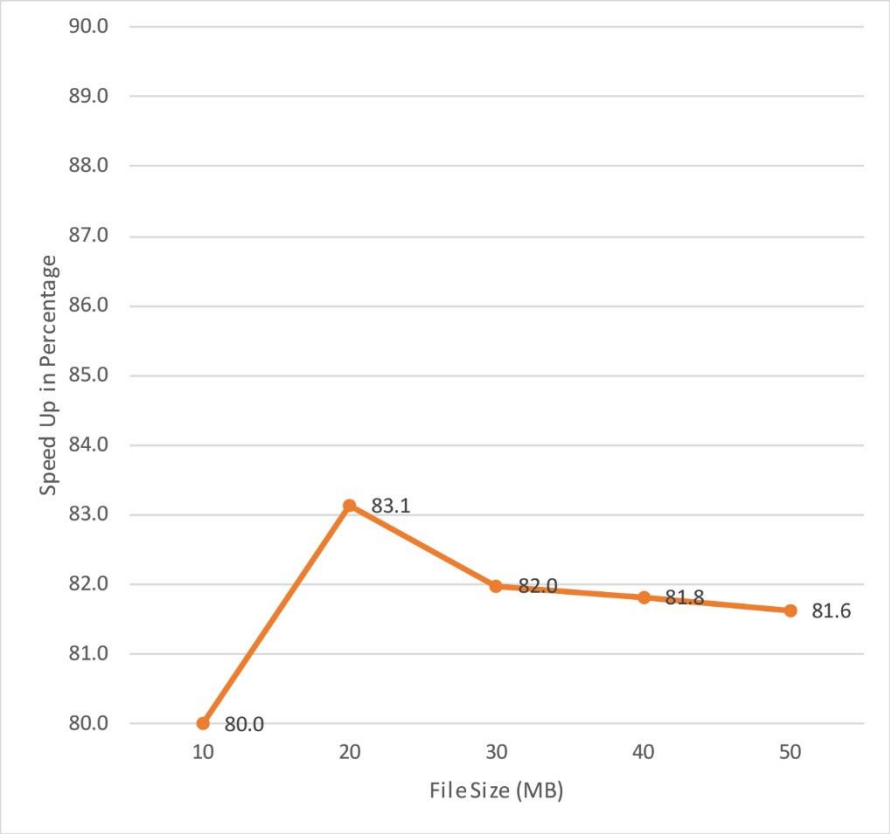

Figure 4.3. Converting phase speed up for various 32-bit audio size encoding process

4.3.1.2 Stereo Audio

Figure 4.4 shows the average running speed results of stereo audio (Track 68) for CPU and GPU separately. For 16bit audio we achieve speed gain about 85% for stereo audio type. For 24bit audio the speed up is around 91%. For 32bit audio, the speed increase is around 82%. Here we can also infer that 24bit audio conversion results in faster encoding speed for GPU where for 16bit audio it is slower than the others.

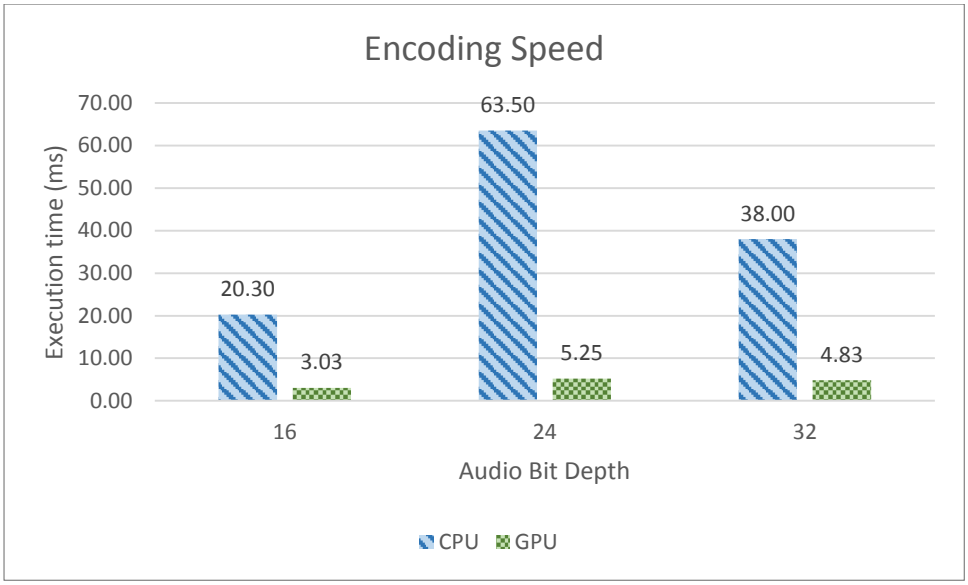

Figure 4.4. Mixing phase execution time comparison for various bit depth audio encoding process

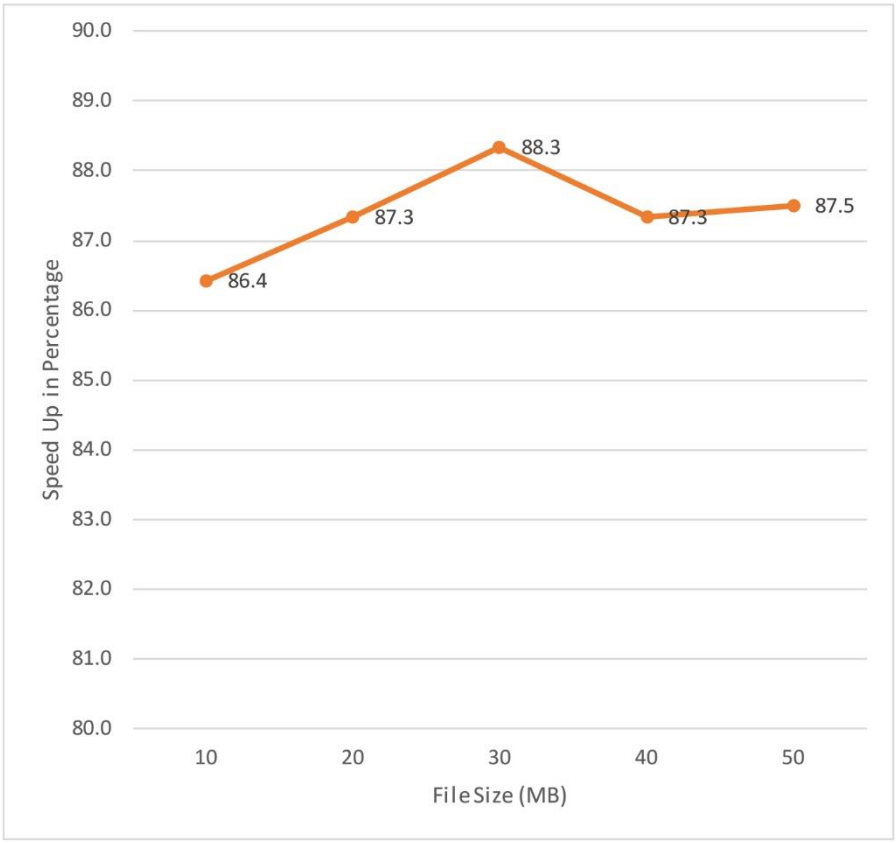

Figure 4.5. Mixing phase speed up for various 32-bit audio size encoding process

Figure 4.5 shows the average running speed results of stereo audio for CPU and GPU separately. For 10MB file size we achieve 86.4% encoding speed up, for 30 MB file the speed gain was 88% and for other file sizes the speed up was around 87%.

4.3.2. Power Consumption Saving

Looking at Table 4.1, the power consumption saving is least for 16bit audio for both mono and stereo audio files where for 24bit audio saving is the highest.

Table 4.1. CPU power saving (in Watts) for various bit depth audio in encoding process

Type	Bit Depth		
	16	24	32
Mono	0.66792W	5.49444W	2.0544W
Stereo	3.636927W	9.209082W	7.257163W

Looking at Table 4.2, the power consumption saving increases with the corresponding file size for both mono and stereo audio files where power saving for encoding stage of stereo audio file is 2x to 3x than the mono audio file.

Table 4.2. CPU power saving (in Watts) for various audio size in encoding process

Type	File Size (MB)				
	10	20	30	40	50
Mono	1.19W	2.09W	2.6W	2.89W	3.84W
Stereo	3.76W	4.28W	6.94W	7.53W	11.3W

4.4. Decoding Results

Here we are going to discuss about the execution speed gain for mono and stereo audio also power consumption for encoding.

4.4.1. Execution Speed Gain

In Figure 4.1, we see for first test file (Track 5) we achieve 6x speed up, for second and fourth test file (Track 20 and 68) we achieve 9x speed up, for third test file 11x speed is gained and lastly for fifth track (Track 70) we get 10x speed up.

4.4.1.1. Mono Audio

Decoding results for mono files are shown in Figure 4.6. According to the results, we get around 85% speed increase for 16bit mono audio. For 24bit audio, we achieve more than 92% speed up. Lastly for 32bit audio, around 87% 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.

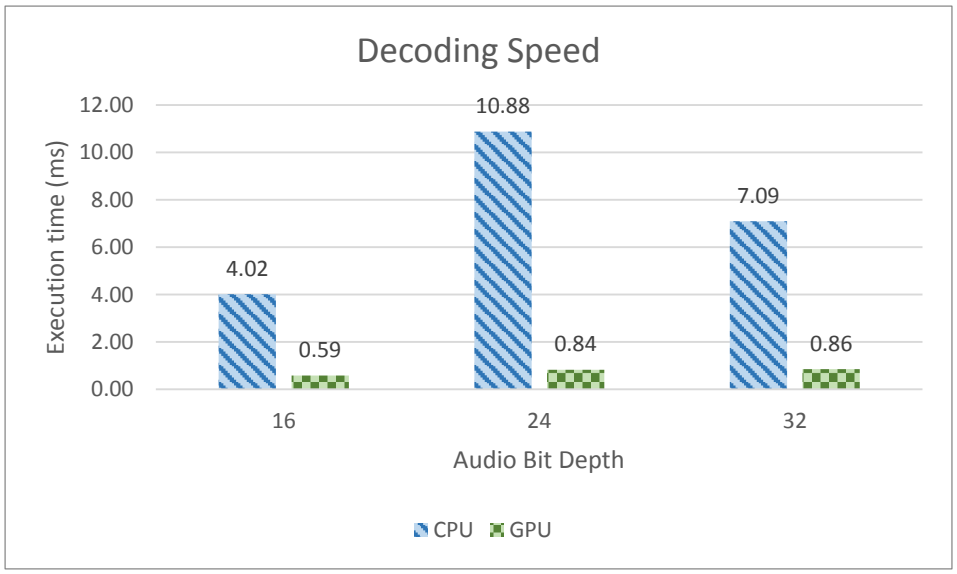

Figure 4.6. Converting phase execution time comparison for various bit depth audio decoding process

According to our results in Figure 4.7, performance gain in converting data to pcm in mono audio file types is around 88% for 10 and 30 MB file size, 86% speed gain for 20 MB audio and 88% speed up for 40 and 50 MB file.

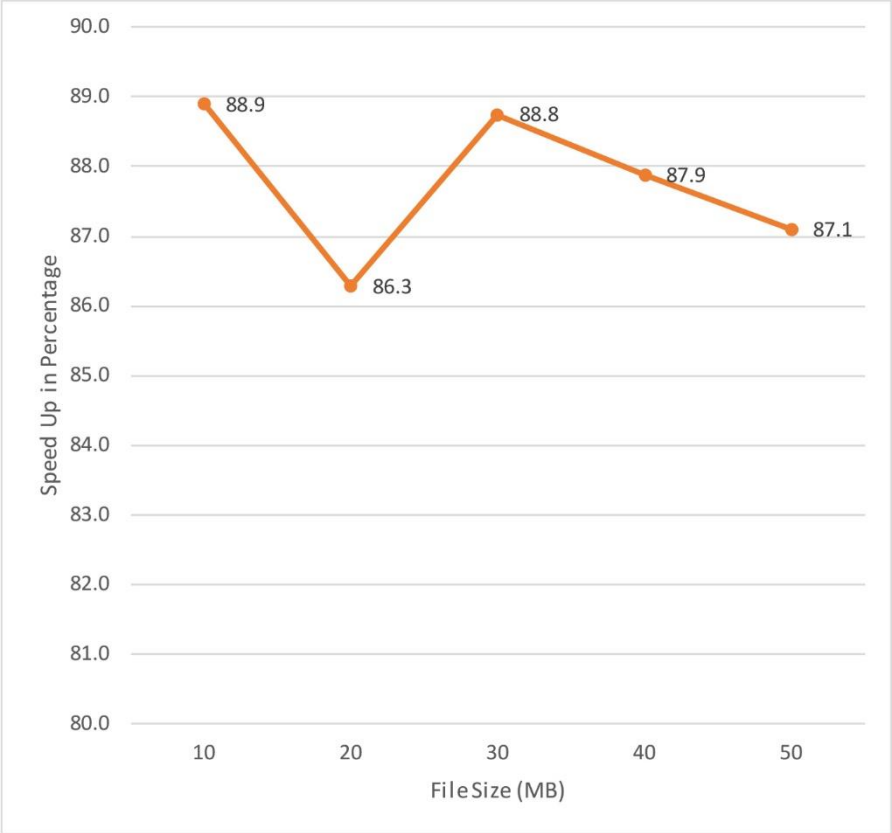

Figure 4.7. Converting phase execution time comparison for various 32-bit audio size decoding process

4.4.1.2 Stereo Audio

Decoding results for stereo files are shown in Figure 4.8. According to the results, we get around 89% speed increase for 16bit mono audio. For 24bit audio, we achieve more than 93% speed up. Lastly for 32bit audio, around 91% speed up is gained. From this table, we can also see that 24bit audio decoding is faster in GPU than the others where 16bit audio conversion is slowest.

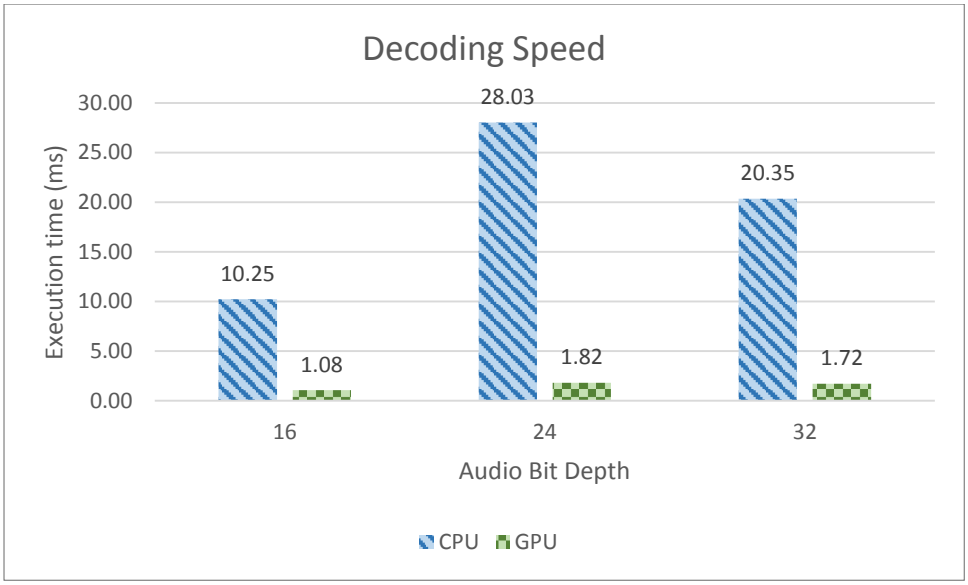

Figure 4.8. Mixing phase execution time comparison for various bit depth audio decoding process

According to our results in Figure 4.9, performance gain for un-mixing phase of decoding stereo files, the speed gain is around 95% for 10 MB file, 92% speed gain for 20 and 50 MB file and for other file sizes the speed up is about 90 %. The speed up in decoding is higher than the speed up we gained in encoding stage. One possible reason for that is while decoding the brute force optimization loop was not implemented unlike in case of encoding step.

4.4.2. Power Consumption Saving

For decoding stage, we get less power saving than that of encoding stage. Here also 16bit audio has the least power saving where 24bit audio has the highest shown in Table 4.3.

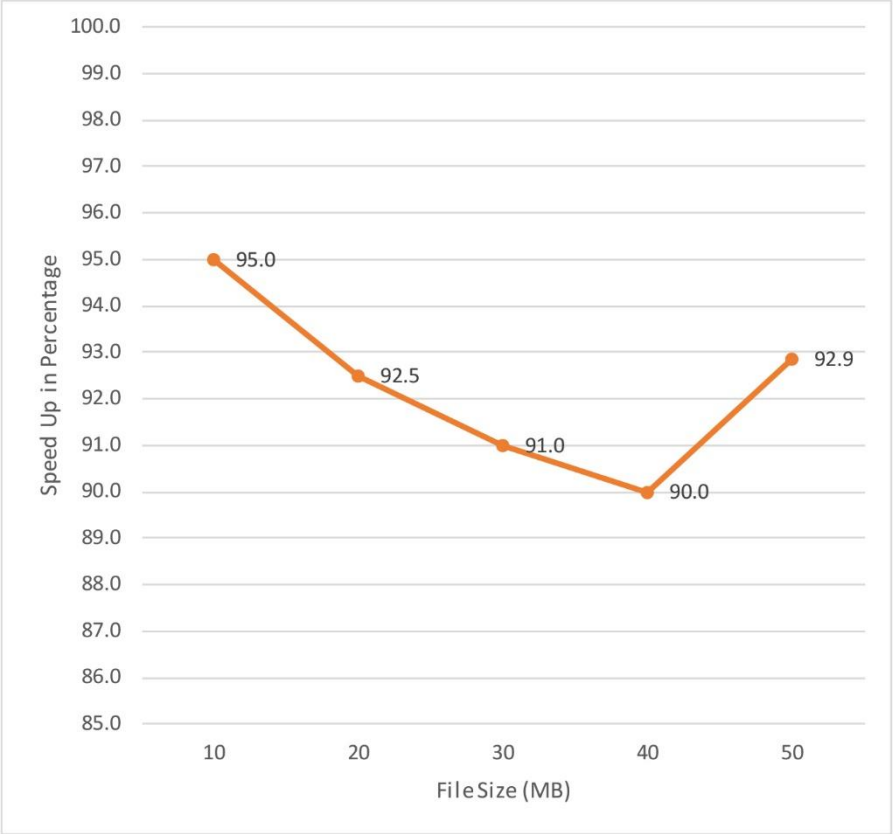

Figure 4.9. Mixing phase execution time comparison for various 32-bit audio size decoding process

Table 4.3. CPU power saving (in Watts) for various bit depth audio in decoding process

Type	Bit Depth		
	16	24	32
Mono	0.690414W	1.8539W	1.124886W
Stereo	1.577814W	4.379118W	3.087194W

In terms of power saving for different file sizes in decoding stage, we get almost similar result to that of encoding stage shown in Table 4.4

Table 4.4. CPU power saving (in Watts) for various audio size in decoding process

Type	File Size (MB)				
	10	20	30	40	50
Mono	1.13W	1.87W	2.26W	2.92W	3.17W
Stereo	5.3W	7.09W	5.49W	7.2W	8.01W

Chapter 5

Conclusion and Future Works

5.1 Conclusion

In this paper, we analyzed the feasibility to use CUDA framework for Apples 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. Our tests show that we get average of 80-95% speed up for mixing/un-mixing audio data. This work, to our best knowledge, is the 2nd implementation of lossless audio compression on CUDA.

5.2 Future Work

This Software by apple has huge potential as lossless audio codec by we can see its compression ratio. We tried our best to optimize this algorithm in parallel using CUDA but there are so many complicated calculation and method that executes on CPU also possible to optimize.

We focused in parallel implementation and parallel optimization for ALAC in these paper but in future we can focus on data dependent part to use different optimization technique there to get better results.

REFERENCES

[1] "FLAC -Free Lossless Audio Codec,". [Online]. Available: https://xiph.org/flac/. Accessed: Nov. 16, 2016.

[2] T. Gu, "TimothyGu- alac," GitHub, 2016. [Online]. Available: https://github.com/TimothyGu/alac. Accessed: Nov. 16, 2016.

[3] "WavPack audio compression,". [Online]. Available: http://www.wavpack.com/. Accessed: Nov. 16, 2016.

[4] M. Ashland, "Monkey's audio - a fast and powerful lossless audio compressor," 2000. [Online]. Available: http://www.monkeysaudio.com/index.html. Accessed: Nov. 16, 2016.

[5] F. Ghido, "OptimFROG - main," 1996. [Online]. Available: http://losslessaudio.org/. Accessed: Nov. 16, 2016.

[6] T. Software, "Tau projects," 2003. [Online]. Available: http://tausoft.org/wiki/Main_Page. Accessed: Nov. 16, 2016.

[7] X. Tian, R. Xu, Y. Yan, S. Chandrasekaran, D. Eachempati, and B. Chapman, "Compiler transformation of nested loops for general purpose GPUs," Concurrency and Computation: Practice and Experience, vol. 28, no. 2, pp. 537–556, Aug. 2015.

[8] K. Barkalov and V. Gergel, "Parallel global optimization on GPU," Journal of Global Optimization, vol. 66, no. 1, pp. 3–20, Feb. 2016.

[9] J. Ghorpade, "GPGPU processing in CUDA architecture," Advanced Computing: An International Journal, vol. 3, no. 1, pp. 105–120, Jan. 2012.

[10] Wiggins, N. Schultz, and P. Schmidt, "Nvidia's gpgpu architecture: Fermi and cuda," Oregon State University, Tech. Rep., 2010.

[11] NVIDIA, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, NVIDIA Corporation, 2009.

[12] P. N. Glaskowsky, "NVIDIA's Fermi: The First Complete GPU Computing Architecture," NVIDIA Corp.

[13] M. Harris, "In the trenches at GTC: Inside Kepler," Parallel Forall, 2012. [Online]. Available: https://devblogs.nvidia.com/parallelforall/trenches-gtc-inside-kepler/. Accessed: Nov. 19, 2016.

[14] J. Wang, "Introducing the GeForce GTX 680 GPU," 2012. [Online]. Available: http://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-680-gpu. Accessed: Nov. 19, 2016.

[15] R. Smith, "NVIDIA GeForce GTX 680 review: Retaking the performance crown," 2016. [Online]. Available: http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review. Accessed: Nov. 19, 2016.

[16] "CUDA," in Wikipedia, Wikimedia Foundation, 2016. [Online]. Available: https://en.wikipedia.org/wiki/CUDA#/media/File:CUDA_processing_flow_(En).PNG. Accessed: Nov. 19, 2016.

[17] "NVIDIA - Kernel,". [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#kernels. Accessed: Nov. 19, 2016.

[18] M. Hans and R. W. Schafer, "Lossless compression of digital audio," IEEE Signal Processing Magazine, vol. 18, no. 4, pp. 5–15, Jul. 2001.

[19] M. Boyer, D. Tarjan, S. Acton, and K. Skadron, "Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors," in Parallel Distributed processing 2009. IEEE International Symposium on, May 2009, pp. 1 –12.

[20] B. Jang, D. Schaa, P. Mistry, and D. Kaeli, "Exploiting memory access patterns to improve memory performance in data-parallel architectures," IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 1, pp. 105–118, Jan. 2011.

[21] Sound quality assessment material: recordings for subjective tests ; user's handbook for the EBU - SQAM compact disc in Sound quality assessment material recordings for subjective tests users' handbook for the EBU SQAM CD. Geneva: Technical Centre, 2008. [Online]. Available: https://tech.ebu.ch/docs/tech/tech3253.pdf. Accessed: Nov. 14, 2016.

[22] D. Salomon, A concise introduction to data compression. London: Springer London, 2008, pp. 227–244.

[23] L. Firmansah and E. B. Setiawan, "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), May 2016.

[24] B. Waggoner, Compression for great video and audio: Master tips and common sense, 2nd ed. Oxford, United Kingdom: Elsevier Science, 2009, pp. 341–347.

[25] R. Yu, S. Rahardja, L. Xiao, and C. C. Ko, "A fine granular scalable to lossless audio coder," IEEE Transactions on Audio, Speech and Language Processing, vol. 14, no. 4, pp. 1352–1363, Jul. 2006.

[26] D. Salomon, G. Motta, and D. Bryant, Data compression: The complete reference, 4th ed. London: Springer London, 2006, pp. 773–783.

[27] "Apple Lossless - Wikipedia," in Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Apple_Lossless. Accessed: Nov. 14, 2016.

[28] B. Waggoner, Compression for great video and audio: Master tips and common sense, 2nd ed. Oxford, United Kingdom: Elsevier Science, 2009, pp. 215–221.

[29] S. Tsutsui and P. Collet, Eds., Massively parallel evolutionary computation on GPGPUs. Berlin: Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2013, pp. 149–156.

[30] M. Al-Mouhamed and A. ul H. Khan, "Exploration of automatic optimisation for CUDA programming," International Journal of Parallel, Emergent and Distributed Systems, vol. 30, no. 4, pp. 309–324, Sep. 2014.