FLAC Decoding Using GPU Acceleration

Haolei Ye and Eric C. McCreath
Research School of Computer Science,
The Australian National University,
Canberra, Australia
Email: haolei.ye@anu.edu.au, eric.mccreath@anu.edu.au

Abstract—**Free Lossless Audio Codec (FLAC) format is a widely used format for audio storage. Using a lower performance single threaded approach, FLAC is easily decoded faster than the rate at which it is played at. However, if you wish to transcode or edit long FLAC audio files then decoding times using single thread CPU approaches becomes significant. The FLAC format contains a sequence of frames, these frames vary in size so start locations are unknown until the previous frame is decoded. This complicates parallelizing decoding. However, frames start with known fixed bit patterns and each frame contains a frame index, it is possible to locate and decode frames in parallel. In this paper, we present an approach that exploits this characteristic enabling all the frames to be decoded in parallel. This approach is implemented and evaluated using an NVIDIA GeForce® GTX 1080 graphics card showing a 5 times performance improvements than the widely used official implementation running on an Intel Core™ i7-6770K CPU.**

Index Terms—**Audio, FLAC, Codec, GPU, Decoder, CUDA, Linear-predictive coding, Parallel Processing, Pascal, GP104, Signal processing**

I. Introduction

Free Lossless Audio Codec (FLAC) format is a popular audio codec first released in 2000 by Josh Coalson [1]. As the name suggests it is a lossless compression format and is also non-proprietary and open source [2]. The FLAC format is supported by a variety of hardware and software [3]. There are two main FLAC decoder implementations: one is the official FLAC encoder and decoder implementation by Josh Coalson from Xiph.Org Foundation [4], the other is used in FFMpeg and implemented by the FFMpeg team, this implementation was first released in 2009 [5]. These implementations are basically serial so as FLAC files become larger the decoding time will also increasing linearly. If a user is intending to just play FLAC audio then this can be simply accomplished with a serial implementation as the time for playing the audio increases linearly with the size of the file to be decoded, moreover, these serial implementations can easily decode faster than the rate at which they are played. However if your wish to decode FLAC audio for other purposes, such as data mining or audio editing, then the decoding time can become significant. Graphics processing units (GPUs) provide computing devices which clearly have the potential to improve decoding performance. However, because of the serial nature of the format implementing a FLAC decoder on a GPU is non-trivial.

A number of researches have explored the use of GPU acceleration for audio decoding. These including Xiaoliang et al. [6] who proposed a CUDA based MP3 audio decoder. In this paper a CPU is used to locate the MP3 frames within the entire file and the frame data is transfer to the GPU device, then the GPU is used to decode to a PCM format, and finally the decoded PCM data is transferred back to primary memory. Their results showed a more than five times performance improvement by using an NVIDIA GeForce® GTX 260 graphics card compared to the CPU version on Intel Quad 2.33 GHz CPU [6]. The approach taken has many advantages as the part of the decoding that can be easily parallized are done using GPU acceleration, and the parts that are not easily parallized are done using a general purpose processor. However, as audio files become longer the serial aspect of this approach will mean the overall performance will grown linearly no matter how much parallel resources one has.

R. Ahmed and M. S. Islam introduced an optimised Apple Lossless Audio Codec (ALAC) decoder in 2016 [7]. They explored an architecture for decoding a mono or stereo ALAC format audio. Their decoder, running on an NVIDIA GeForce® GTX 960 graphics card, obtained an average of 80% to 95% speed up for un-mixing audio data compared to the CPU version running on Intel Core i5-4590 processor. The decoding phase contains three steps: decompressing the ALAC data, convert the data into PCM data, and concatenate the channels into a single output buffer. In a similar way to Xiaoliang et al., Ahmed et al. method only deploys the last stage to the GPU.

More broadly a number of researchers have explored the use of GPUs for high performance decoding. So Shen et al. used techniques for real-time playback of high definition video using a GPU [8]. Johnson and McCreath investigated a fast and scalable parallel decoding algorithm for Huffman decoding on a GPU [9]. Also Falcão et al. implemented an efficiently algorithm on GPUs for decoding LDPC codes [10].

In this paper, a new practical audio decoding implementation, named GPURaku, is proposed for decoding FLAC file data into a PCM signal using a GPU. FLAC was designed for fast and low memory serial lossless decoding and also only requires integer operations [11]. A significant part of FLAC decoding code is branch constructs. This is well suited to modern CPUs allowing full use of pipeline and branch prediction. FLAC also supports streamable audio which means each frame stores all the parameters, this allows us to find a frame in the middle of the entire FLAC audio. Hence, FLAC itself is a seekable format, this enabling us to finding all the frames concurrently. This is a key characteristic we exploited to let us parallelize the decoding implementation. We also introduced several methods to reduce the number or eliminate branches which improves overall performance. The framework is implemented in CUDA C/C++ and runs on NVIDIA GPUs.

In the next section we provide a background to GPU computing along with an overview of the FLAC format. Section III states the parallel algorithm used and also evaluates the complexity of this algorithm. In Section IV a summary of the framework along with the CUDA implementation is given. This is followed by Section V which describes the various optimisations used in our implementation. Section VI gives the experimental evaluation of our implementation. Finally a conclusion is provided in Section VII.

II. Background

A. GPU Computing

NVIDIA has been instrumental in the introduction of Graphics Processing Units (GPUs). In in 1999 NVIDIA introduced the GeForce® 256 chips [12], and in 2001 General-purpose computing on graphics processing units (GPGPU) was made available [13]. In 2006 the Compute Unified Device Architecture (CUDA), which is a parallel computing platform model created by NVIDIA, was first released with its GeForce® 8800 GTX [14]. This allowed programmers to more easily write parallel programs making use of the computing power of GPUs [14]. CUDA allows programmers to launch kernels from a variety of languages, including c++ and c, and write parallel kernels in a constrained c language [13]. The Pascal architecture, developed by NVIDIA, is the successor to the Maxwell architecture [15]. It was first released with Tesla P100 in 2016 and used in the GeForce® 10 series graphics cards [15]. A *CUDA Processor* has a fully pipelined ALU and FPU as its major computing components [16]. The ALU supports full 32-bit precision for all instructions [16]. Each *CUDA Processor* also has its own dispatch port, operand collector and result queue [16], but does not contain any branch prediction circuitry [17]. With no branch prediction, before the next instruction is fetched, the *CUDA Processor* has to wait the jump instruction pass the execute stage in the pipeline [18]. Although having many threads in the pipeline will hide most of this latency, however, the *CUDA Processor* may perform poorly when executing branch-bound programs.

In the Pascal architecture, one *Streaming Multiprocessors* and a *Polymorph Engine* combine as a *Texture / Processor Cluster* (TPC). 5 TPCs and a *Raster Engine* combine as a *Graphics Processing Clusters* (GPC). On NVIDIA GeForce® GTX 1080, it uses a Pascal architecture GP104 GPU consists of 4 GPCs, runs at 1607 MHz, boosts up to 1733 MHz and provides 8873 GFLOPS calculation capability [19]. Therefore, GP104 provides in total 20 *Streaming Multiprocessors*, 2560 *CUDA processors* and 320 gigabytes per second memory bandwidth.

The memory resources could be categorized as registers, local memory, shared memory, global memory, constant memory, texture memory [20]. Table I summarizes the characteristics of the CUDA memory hierarchy of the GP104 on the official NVIDIA GeForce® GTX 1080 graphics card.

Memory	Latency	Size (Total)	Location	Access
Registers	0(r) / 20(raw)	5120 KB	On-Chip	R/W
L1 Cache	28	960 KB	On-Chip	-
Shared	92	1920 KB	On-Chip	R/W
L2 Cache	200	2048 KB	On-Chip	-
Local	400 - 800	-	DRAM	R/W
Constant	400 - 800	64 KB	DRAM	R
Texture	400 - 800	Up to 8 GB	DRAM	R
Global	400 - 800	Up to 8 GB	DRAM	R/W

TABLE I: Memory characteristics of the GeForce® GTX 1080 [21] [22] [23] [24] (The Latency column is in clock cycles. For Constant and Texture memory, the latency given in the table includes cache misses.)

B. The FLAC format

FLAC [25] was designed for fast low memory serial lossless decoding. A FLAC encoded audio is made of several metadata block structures and an audio stream. The STREAMINFO block structure stores the minimum and maximum block size, channels and bits per sample of the audio stream.

The audio stream is separated into several frames. Each frame contains: a frame header, subframes, zero-paddings, and a frame footer. The frame header contains a 14-bit sync code (`11111111111110`) which marks the start of a frame, the information of the frame (including block size, sample rate, channel assignment and sample size), the sample or frame index (this uses from 8 to 56 bits to provide either a sample or frame index), and an 8-bit CRC value of the frame header. The combination of the sync code, checking for invalid values within the frame header, and the CRC result provides an approach to detect the frame header within a stream.

The FLAC format supports up to 8 channels, hence there are at most 8 subframes inside a frame. Except stereo audio (2 channels), all the other channel types of subframes are all stored independently. For stereo audio, there are four channel assignments: independent, left/side stereo, right/side stereo and mid/side stereo. Each subframe should be one of the four following types: `CONSTANT` (filling the entire subframe block with one value), `VERBATIM` (storing the unencoded samples), `LPC` (unencoded warm-up samples with FIR-Linear Prediction encoded residuals) and `FIXED` (unencoded warm-up samples with Fixed-Linear Prediction encoded). [25]

The LPC subframes stores the un-coded warm-up samples followed by the Rice encoded residuals in `LPC` and `FIXED` type subframes [25]. FLAC uses 0 for counting and 1 for ending [25], and expressing the signed values with the odd unsigned notation. For example, with order 2 rice encoding, 0001 10 (14 in unsigned value) stands for 7 in FLAC rice coding notation, and 0001 11 (15 in unsigned value) stands for -8. [25]

C. FFMpeg Framework for Decoding

The workflow of decoding FLAC audio with FFMpeg library API is shown in Figure 1. The CPU decoding version is using this workflow. The framework is modified from the official example 'trancoding_aac.c' [26].

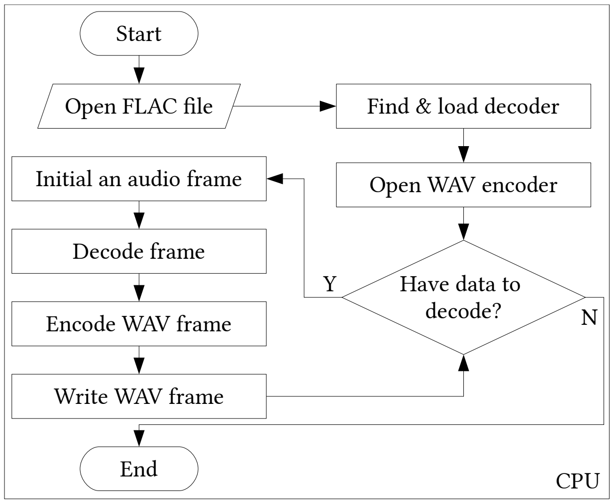

Fig. 1: FLAC decoding framework with FFMpeg API

The workflow of FFMpeg framework is straightforward. It first opens the FLAC file and finds the FLAC decoder from its decoder list. Then allocates and sets the decoder context for the specific file. In the FFMpeg API, the user has to prepare the decoder first for decoding a specific file. It provides functions like `avcodec_find_decoder` and `avcodec_open2` for preparing the decoder [27]. And then, it uses a loop to decode the entire audio data. In the implementation of the FFMpeg API uses a first-in first-out structure to cache the decoded frames. Once the data of the decoded samples is enough for encoding the resulting frame, it will encode the frame into the WAV format. The entire FFMpeg API decoding framework is running on the CPU.

III. Parallel FLAC decoding algorithm

The FLAC decoding algorithm proposed has three stages. Firstly the number of frames is determined, this can be done in constant time by working back from the end of the file until the last frame in the file is found. This enables us to know the number of frames that will be decoded in the following stages. Secondly the frame header locations are found, this is done by searching for sync markers and then checking for correctly formatted headers in parallel. Once this is done the frames can be decoded independently with the decoded data placed directly into its final location. Algorithm 1 gives the pseudo code where F denotes the FLAC data to be decoded, H is the minimum size of the FLAC frame headers, S_i is the start byte location of frame i, and P is the decoded PCM data.

```
input  : $F$ - FLAC to decode
output : $P$ - decoded PCM
// Stage 1 - determine the number of frames
$pos \leftarrow |F|$
while frame header does not start at position pos do
    $pos \leftarrow pos - 1$
od
$n \leftarrow$ index of header starting at $pos$
// Stage 2 - Find header locations
for $i = 0$ to $n-1$ dopar
    for $j = \lceil i \times |F|/n \rceil$ to $\lceil (i+1) \times |F|/n \rceil - 1$ do
        if frame header starts at byte $j$ then
            $w \leftarrow$ index of this header
            $S_w \leftarrow j$
        end
    od
odpar
// Stage 3 - Decode frames
for $i = 0$ to $n-1$ dopar
    decode frame $i$ which starts at byte $S_i$
odpar
```

Algorithm 1: Parallel FLAC decoding

Using a Parallel Random Access Machine (PRAM) model of computation and assuming we have access to n processors and also assuming the number of samples in each frame is bound, which we denoted s_{max}, then the decoding algorithm above is $O(1)$ where n is the number of frames in the FLAC file to decode. If there is a bound on the number of samples in a frame then the number of bytes in a frame will also be bound as the number of bytes in the frame and subframe header and footer sections is bound and the bits per sample is also bound. The algorithm is $O(1)$ as each of the stages in the algorithm is O(1). So in the first stage the while loop goes around at most the maximum size of bytes in a frame times. Stages 2 and 3 have parallel for loops which can be executed concurrently by the n processors. The code within these parallel for loops are both $O(1)$ as the size of each frame has a fixed maximum size and fixed maximum number of samples.

IV. CUDA Implementation

FFMpeg provides the general workflow of decoding FLAC audio. Basically each frame is decoded sequentially and as such the execution time of this implementation will grow linearly with the size of the file to decode.

GPURaku is our audio decoding framework which is implemented using CUDA. This framework is shown in Figure 2.

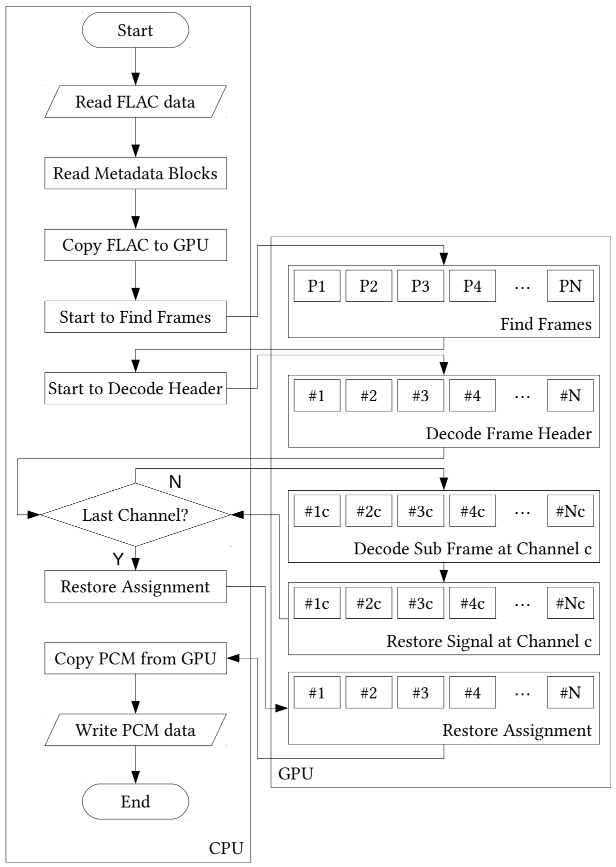

Fig. 2: FLAC decoding framework with CUDA GPU

CPU first reads the entire raw FLAC data, and parses the metadata block and then jumps to the start position of the start frames. Next, the CPU searches the last frame and records its frame index to get the frame count. After that, CPU transfers the frame raw data to the GPU. The GPU finds all frame start positions, these positions are saved into an array. Then, the GPU decodes all the frame headers concurrently, and prepares the frame header data for the subframes. Our implementation then loops the number of channel times, decoding and restoring the subframes. Finally, GPU processes the channel assignments and transfers the data back to CPU. CPU simply writes the PCM data out in the WAV format.

The algorithm is $O(1)$ assuming we have n processors, however, on any real system, such as our GPU implementation, the number of processing elements is fixed. And hence, the performance becomes linear when n becomes larger than the number of processors. Now on modern high-end GPUs the number of cores is very large, so for example on the NVIDIA Tesla V100 GPU there is 5,120 cores [28]. Thus as we implement our algorithm on the GPU we are focused on improving overall performance.

V. Optimization Policies

Although the approach implemented directly should improve overall performance, there are still many optimizations that are important in improving performance. These optimizations are explained in the following sections.

A. Replace Conditional Constructs with Tables

GPUs perform poorly when the kernel code has conditional constructs. Now a lot of FLAC decoding processes have conditional code, these include reading: the UTF-8 frame index, frame header information, rice encoding residuals, and specific bits integers. Some branch constructs could be directly transferred, like the bit stream reader for the *subframes*. The usual implementation of reading n bits is repeating the read bit function for n times. The program would execute n conditional constructs for checking if it should switch to the next bit. For reading more than 8 bits it might go through 2 bytes or more, however, most of these could be reduced to only one conditional statement by checking the position of the current bit of the byte. This reduction is straight forward. There are many other means to reduce the number conditional constructs. One method of reducing the conditional constructs is replacing them with tables. We use this approach in a number of places within our implementation these are now described.

1) Constant Value Table: In the FLAC *frame* header structure, all parameters are encoded in under 4 bits. So each parameter has maximum 16 different possible values. Hence, the massive branch construct of value checking (a *switch* statement) could be replaced by a look-up table. The only aspect we need to prepare is a table which stores all the possible values. This table is reused in all the frame headers. It only needs 64 bytes (16×4) for one parameter so it is stored in the constant memory for lower latency.

The constant value table replacement is used for all the frame header parameters (block size, sample rate, channels and bits per sample) and part of the CRC-8 calculation. For the channel parameter, it contains two things: one is the number of channels, and the other is the channel assignment for stereo audio. So the channel parameter needs two tables, because the channel assignment only exists for stereo audio, so the number of channels for the frame that has channel assignment is 2, and for those frames that have only 1 channel or more than 2 channels, they only have independent channel assignment.

2) Function Pointer Table: The constant value table was used to eliminate the control transfer instructions of the value assignment, but it cannot apply to the UTF-8 calculation and specific bits integer fetching. This type of calculations was optimized with function tables. One of the typical calculation is reading a number of specific bits. Another calculation done is in the bit stream for the decoding residuals. With the limitations of the bits per sample, on a general purpose CPU, this part could be implemented as a massive-branch construct like a *switch* statement in C. However, on GPUs, executing massive branch constructs like this would cause divergence. It is difficult to avoided due to the different calculation methods that are applied, but the control transfer instruction of the bit number checking are simplified and replace the calculation with a function pointer lookup table. The length and calculation result of fetching specific bits are both needed in parsing the rice encoding. This is done with the help of the function pointer lookup table, it will only execute the branch construct once to get the correct function pointer. From then on, it would executes that function for fetching bits from binary data. This removes repeating conditional code.

The similar situation could be found in CRC-8 calculation. In FLAC decoding, CRC-8 calculation and the sample reading is this kind of calculation. The value of CRC-8 at different positions have been calculated and replaced by a constant table as mentioned before. The problem is it needs to loop from the start position to the end position. It is difficult to enumerate all the possibilities of the CRC-8 calculation. However, the length of the frame header is limited. The minimum size of a FLAC frame header is 4 basic bytes and 1 UTF-8 byte, total 5 bytes. The maximum size is 4 basic bytes, 7 UTF-8 bytes, 2 block size bytes and 2 sample rate bytes, total 15 bytes. So the CRC-8 calculation in our implementation is enumerated. All the calculations are stored in a function tables replacing the original multiple branch construct by that table.

B. Reduce Global Memory Access

The raw FLAC audio data and PCM decoded data have to be stored in global memory. It is impossible to avoid global memory access, but still possible to reduce the access times for better performance.

To restore the PCM data for `LPC` and `FIXED` *subframes*, it requires warm up samples, coefficients, shift bits, and residuals. Warm up samples and residuals are stored in the space of final PCM data. The shift bits are stored in an 8-bit unsigned integer. The maximum coefficients size appears in `LPC` *subframe*, which is an array of 32-bit signed integers containing at most 32 elements. Hence, the memory space required for decoding `LPC` and `FIXED` *subframes* is predictable.

When all these parameters have been read from raw `LPC` and `FIXED` *subframes*, it still needs to use the coefficients with the known samples to restore the original PCM samples. Calculating the polynomial needs accessing the previous samples frequently. Assuming restoring PCM samples in an l coefficients `LPC` *subframe* with m samples and the samples are stored in global memory, it needs to access the global memory for $l(m-l)$ times, which will waste many processor cycles. The following two methods are introduced to reduce the memory access to $2m-l$ times.

1) Sample Cache Array: One simple idea is to store all the previous samples in an array of on-chip memory, e.g. shared memory. The decoder reads the first l samples and saves them to the cache array at beginning. This costs l access. And then in the decoding loop from the first residual to the last, it only accesses the PCM samples twice per loop: one for reading the current PCM samples, and the other for writing the new samples. This costs $2(m-l)$ access. Hence, the overall number of memory accesses is $2m-l$.

2) N-Level Increased Progression: Although sample cache array could accelerate all the `LPC` calculation, it still needs to do multiplications. However, it is possible to reduce them to only additions when restoring the `FIXED` *subframes* signal. Consider the following coefficients: $3, -3, 1$. Suppose the warm up samples are r_1, r_2, r_3 and residuals are $r_4, r_5, r_6, \cdots$. If $p_1, p_2, p_3, \cdots$ are the final PCM samples, according to the definition of `FIXED` *subframes*, it is obviously that:

$$\begin{aligned} p_1 &= r_1 \\ p_2 &= r_2 \\ p_3 &= r_3 \end{aligned} \tag{1}$$

Now according to the definition of FIR-Linear Prediction,

$$\begin{aligned} p_4 &= r_4 + 3p_3 - 3p_2 + p_1 \\ &= r_4 + 3r_3 - 3r_2 + r_1 \\ p_5 &= r_5 + 3p_4 - 3p_3 + p_2 \\ &= r_5 + 3r_4 + 6r_3 - 8r_2 + 3r_1 \\ p_6 &= r_6 + 3p_5 - 3p_4 + p_3 \\ &= r_6 + 3r_5 + 6r_4 + 10r_3 - 15r_2 + 6r_1 \\ &\cdots \end{aligned} \tag{2}$$

Assign progression c'_x which

$$c'_x = \begin{cases} r_3 - r_2 & (x = 3) \\ p_x - p_{x-1} & (x > 3) \end{cases}$$

Then we have

$$\begin{aligned}
c'_3 &= r_3 - r_2 \\
c'_4 &= p_4 - p_3 \\
&= r_4 + 2r_3 - 3r_2 + r_1 \\
c'_5 &= p_5 - p_4 \\
&= r_5 + 2r_4 + 3r_3 - 5r_2 + 2r_1 \\
c'_6 &= p_6 - p_5 \\
&= r_6 + 2r_5 + 3r_4 + 4r_3 - 7r_2 + 3r_1 \\
&\dots
\end{aligned} \tag{3}$$

Assign progression c_x which

$$c_x = \begin{cases} r_3 - 2r_2 + r_1 & (x = 3) \\ c'_x - c'_{x-1} & (x > 3) \end{cases}$$

thus

$$\begin{aligned}
c_3 &= r_3 - 2r_2 + r_1 \\
c_4 &= c'_4 - c'_3 \\
&= r_4 + r_3 - 2r_2 + r_1 \\
c_5 &= c'_5 - c'_4 \\
&= r_5 + r_4 + r_3 - 2r_2 + r_1 \\
c_6 &= c'_6 - c'_5 \\
&= r_6 + r_5 + r_4 + r_3 - 2r_2 + r_1 \\
&\dots
\end{aligned} \tag{4}$$

Assign progression $d_x = c_x - c_{x-1}$ and $x > 3$, then

$$\begin{aligned}
d_4 &= c_4 - c_3 \\
&= r_4 \\
d_5 &= c_5 - c_4 \\
&= r_5 \\
d_6 &= c_6 - c_5 \\
&= r_6 \\
&\dots
\end{aligned} \tag{5}$$

Progression d_x is the same as the residual r_x. The new algorithm is actually the reversed way to calculate p_x. Algorithm 2 shows the approach we take to restore PCM signal of the fixed coefficients $[3, -3, 1]$.

In the first two assignment statements, it only needs to access global memory 3 times to get $pcm[0]$, $pcm[1]$ and $pcm[2]$. In the loop, the $pcm[i-1]$ is stored and updated by a local variable within a register, thus it only accesses global memory for 2 times for reading $pcm[i]$ and update $pcm[i]$. Now it needs $2m - 3$ global memory accesses, which performs the same as the Sample Cache Array. However, the loop only has addition statements, removing the multiplication by the coefficients. In our implementation, array pcm and r share the same memory for better memory consumption.

The other two fixed coefficients have the same regular pattern. For $[2, -1]$, it needs 1 variable c initialized with $pcm[1] - pcm[0]$. For $[4, -6, 4, 1]$, it needs 3 variables.

```
input : $m$ - FLAC frame block size (sample size), $r$ -
        FLAC warm-up samples and residuals array
output: $pcm$ - FLAC frame PCM samples
$pcm[0] \leftarrow r[0]$
$pcm[1] \leftarrow r[1]$
$pcm[2] \leftarrow r[2]$
$c \leftarrow r[2] - 2 \times r[1] + r[0]$
$c' \leftarrow r[2] - r[1]$
for $i \in [3, m)$ do
    $c \leftarrow c + r[i]$
    $c' \leftarrow c' + c$
    $pcm[i] \leftarrow c' + pcm[i-1]$
od
```
Algorithm 2: FLAC Restore No.3 Fixed PCM Signal

$$\begin{aligned}
c'' &= pcm[3] - 3 \times pcm[2] + 3 \times pcm[1] - pcm[0] \\
c' &= pcm[3] - 2 \times pcm[2] + pcm[1] \\
c &= pcm[3] - pcm[2]
\end{aligned} \tag{6}$$

For N coefficients, it needs $N - 1$ level progression additions to restore the PCM signal. We call this the *N-level increased progression*. This approach helps decrease both global memory access and reduce the amount of calculation for restoring the `FIXED` *subframe* signal.

VI. Experiment Results

A series of tests are now presented which show the performance of our implementation in comparison to the official FLAC 1.3.1 and FFMpeg 2.8.15 decoders on Ubuntu 16.04.3 LTS from the Canonical software sources. The GPURAKU is compiled by NVCC with parameter `-O3 -lineinfo` for the best optimization. These tests are all conducted on an Intel Core™ i7-6770K CPU, 16GB DDR4 2400 RAM, 240GB SATA-3 SSD with an NVIDIA GeForce® GTX 1080 graphics card. To construct the FLAC test sets the official FLAC encoder implementation is used with parameters set to `-no-seektable --best` for the smallest compression and no seek table for no seeking optimization.

All the test audio files are 16-bit signed stereo audio data sampled at 48,000 Hz. The original WAV file is a 4,620 seconds (1 hour 17 minutes) long. 76 test FLAC files are created from this WAV file, this is done by increasing linearly the frame size and duration of the original audio. The last test set in this series is the original file encoded as FLAC. The time usage results of all the candidate decoders are shown in Figure 3.

From the experiment result, GPURAKU is at best 5.09 times faster than the official FLAC decoder, and at best 2.62 times faster than the FFMpeg decoder. The time of GPURAKU is not linear for two reasons. Firstly there is a constant set-up time associated with host/device data transfers, although once the transfer starts the time is

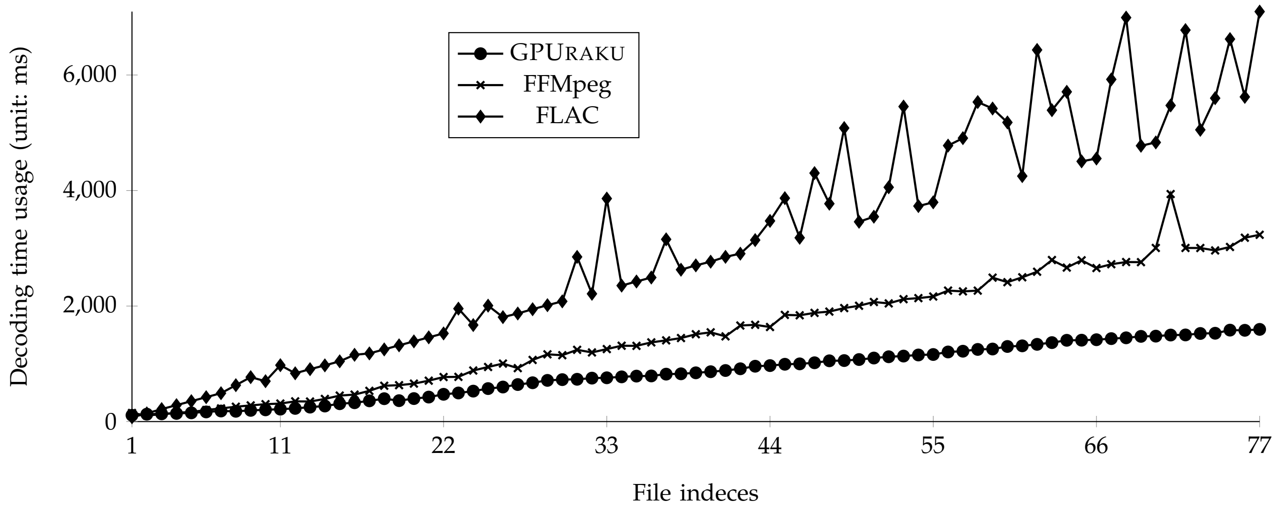

Fig. 3: Time usage of official FLAC, FFMpeg and GPURAKU decoders

linear with respect to the amount of data transferred. Figure 4 shows the H/D and D/H data transferring time from the total data. If we remove the time from the GPURAKU time usage (for example, if the data of audio stream is already loaded at the video RAM) and only consider the computation time, then GPURAKU is maximum 7.8 times the CPU decoders. Currently, there are two ways to reduce the time usage of the data transferring. One of the solutions is using streaming. Our implementation did not use any streaming approaches. Streaming would enable the transfer time to overlap is computation, improving the overall performance. Another method that could be explored is the use the unified memory which is available on Pascal and later GPUs with hardware page faulting and migration [24].

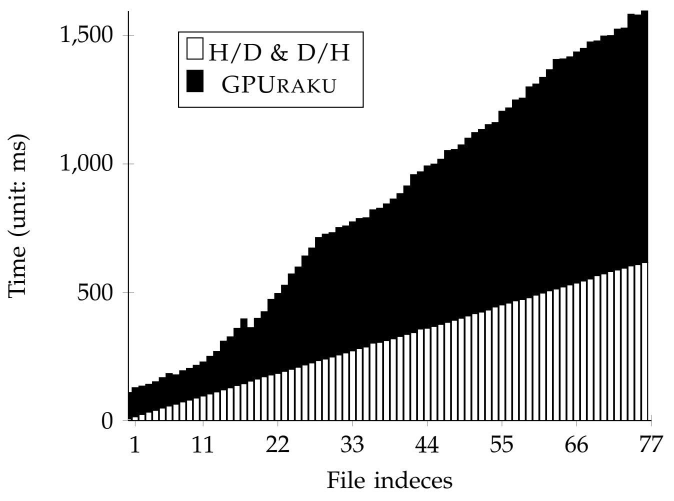

Fig. 4: Data transfer time compared to overall time

Also our GPU has only a fixed number of CUDA cores this also limits the overall performance. The $O(1)$ time complexity could only be established when there are as many cores as frames. When the number of multi-processors (each having a fixed number of CUDA cores) is insufficient, blocks need to wait until there are available multi-processors. Figure 5 shows the time usage of each frame.

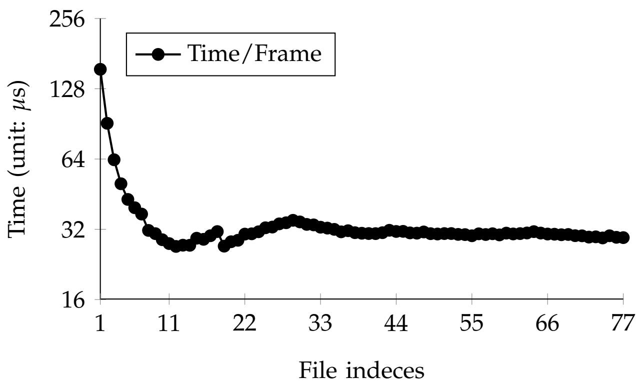

Fig. 5: Time for decoding a single frame

VII. Conclusion

A new concurrent decoding framework using CUDA to decode FLAC audio is proposed and implemented. As the experiment results show, the new framework is approximately five times faster than the CPU implementations (for large files). The results are dependent on the number of *CUDA cores*. If the experiments had been undertaken on a GPU with more cores, for instance, GV100 on TITAN V, it would have performed better.

The $O(1)$ algorithm for FLAC decoding is the key part of the framework presented. This algorithm lets the time usage increase related to the single frame computing complexity but not the entire audio stream compared to the CPU implementations. Given divergence issues in GPUs kernels, multiple methods to reduce branch constructs have been introduced. Furthermore, the time of accessing memory is one of the most important aspects to be optimized. Two methods of reducing the FIR-Linear Prediction decoding memory access have been introduced.

There is still room to improve the CUDA implementation. Rice coding is a branch-bound algorithm and the time for parsing Rice encoding residuals of `FIXED` and `LPC` subframes take a significant proportion of the GPUs time. A new method of decoding rice coding with less conditional constructs is a possible direction to explore for improving performance. As the Rice codes are less than 64 bits long, a single warp (made up of 32 threads with each thread associated with 2 bits) could be used with CUDA warp level primitives to decode concurrently. Also most of these residuals are not large, with most begin less than 5 bits. Hence it does need to execute many branches. So another approach to explore would be using tables for small Rice codes.

The *N-Level Increased Progression* is introduced for accelerating the `FIXED` *subframes*. The possibility of applying this progression to `LPC` *subframes* is worth considering.

The framework needs to copy the entire FLAC frame data to the CUDA device. The situation when the audio file is larger than the memory of CUDA device should be considered. Hence, a method of dividing the large FLAC audio file is needed. The first approach to explore would be to implement a streaming approach, this would also hide transfer times. Once this is done, the use of multiple GPUs could also be investigated.

Acknowledgment

We would like to thank the Research School of Computer Science at the Australian National University for providing the hardware and supporting this research. We would also like to thank the reviewers for their valuable feedback.

References

[1] Josh Coalson and Xiph.Org Foundation, "FLAC - news," https://xiph.org/flac/news.html, 10 2014, online; access 30-Jan-2018.

[2] ——, "FLAC - Free Lossless Audio Codec," https://xiph.org/flac/index.html, 10 2014, online; access 30-Jan-2018.

[3] ——, "FLAC - links," https://xiph.org/flac/links.html, 10 2014, online; access 30-Jan-2018.

[4] ——, "FLAC - download," https://xiph.org/flac/download.html, 10 2014, online; access 31-Oct-2017.

[5] FFMpeg team, "FFmpeg/libavcodec at master FFmpeg/FFmpeg," https://github.com/FFmpeg/FFmpeg/tree/master/libavcodec, 2010, online; access 31-Oct-2017.

[6] C. Xiaoliang, Z. Chengshi, M. Longhua, C. Xiaobin, and L. Xiaodong, "Design and implementation of MPEG audio layer III decoder using graphics processing units," in *2010 International Conference on Image Analysis and Signal Processing*, April 2010, pp. 484–487.

[7] R. Ahmed and M. S. Islam, "Optimizing apples lossless audio codec algorithm using NVIDIA CUDA," 66 Mohakhali, Dhaka 1212, Bangladesh, 12 2016.

[8] G. Shen, G.-P. Gao, S. Li, H.-Y. Shum, and Y.-Q. Zhang, "Accelerate video decoding with generic gpu," *IEEE Transactions on circuits and systems for video technology*, vol. 15, no. 5, pp. 685–693, 2005.

[9] B. Johnston and E. McCreath, "Parallel huffman decoding: Presenting a fast and scalable algorithm for increasingly multicore devices," in *Ubiquitous Computing and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on.* IEEE, 2017, pp. 949–958.

[10] G. Falcão, L. Sousa, and V. Silva, "Massive parallel ldpc decoding on gpu," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming.* ACM, 2008, pp. 83–90.

[11] Josh Coalson and Xiph.Org Foundation, "FLAC - features," https://xiph.org/flac/features.html, 10 2014, online; access 31-Oct-2017.

[12] Nvidia Corporation, "NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256," http://www.nvidia.com/object/IO_20020111_5424.html, 8 1999, online; access 30-Jan-2018.

[13] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson and Jack Dongarra, "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming," *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012.

[14] Jason Sanders and Edward Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*, 1st ed., ser. 1. Rights and Contracts Department, 501 Boylston Street, Suite 900 Boston, MA 02116: Addison-Wesley Professional, 7 2010, vol. 1.

[15] Sumit Gupta, "NVIDIA Updates GPU Roadmap; Announces Pascal," https://blogs.nvidia.com/blog/2014/03/25/gpu-roadmap-pascal/, 03 2014, online; access 11-Mar-2018.

[16] Nvidia Corporation, *NVIDIAs Next Generation CUDA™Compute Architecture: Fermi™Whitepaper*, Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2009.

[17] Basheer Ershaad Ahamed, *Advanced School in High Performance and GRID Computing - Introduction to GPU programming in the nvidia CUDA environment*, Jawaharlal Nehru Centre for Advanced Scientific Research Centre for Computational Materials Science, Jakkur P.O., Bangalore 560064 Karnataka India, 11 2008.

[18] Ben Lee, Alexey Malishevsky, Douglas Beck, Andreas Schmid and Eric Landry, *Dynamic Branch Prediction*, Oregon State University, 1500 SW Jefferson St., Corvallis, OR 97331, 541-737-1000, 12 2001.

[19] Nvidia Corporation, *NVIDIA GeForce GTX 1080 Whitepaper*, Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2016.

[20] Gregory Ruetsch and Massimiliano Fatica, *CUDA Fortran for Scientists and Engineers: Best Practices for Efficient CUDA Fortran Programming*, 1st ed., ser. 1. Rights and Contracts Department, 501 Boylston Street, Suite 900 Boston, MA 02116: Elsevier, 2013, vol. 1.

[21] Julien Demouth and Cliff Woolley, *CUDA Optimization with Nvidia NSIGHT™Eclipse Edition*, Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2014.

[22] Justin Luitjens, *Global Memory Usage and Strategy*, Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2011.

[23] John Cheng, Max Grossman and Ty McKercher, *Professional CUDA C Programming*, 1st ed., ser. 1. Rights and Contracts Department, 501 Boylston Street, Suite 900 Boston, MA 02116: John Wiley & Sons, Inc., 2014, vol. 1.

[24] Nvidia Corporation, "CUDA C Programming Guide," Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2017.

[25] Josh Coalson and Xiph.Org Foundation, "FLAC format, Xiph. Org Foundation Std," https://xiph.org/flac/format.html, 10 2014, online; access 31-Oct-2017.

[26] Andreas Unterweger and FFmpeg team, "FFmpeg: transcode_aac.c," http://www.ffmpeg.org/doxygen/2.8/transcode_aac_8c-example.html, 10 2015, online; access 28-Jan-2018.

[27] Fabrice Bellard and Martin Bohme, "An ffmpeg and SDL Tutorial or How to Write a Video Player in Less Than 1000 Lines," http://dranger.com/ffmpeg/tutorial03.html, 2 2015, online; access 28-Jan-2018.

[28] Nvidia Corporation, *Nvidia Tesla V100 GPU Architecture*, Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 8 2017.