FLAC Decoding Using GPU Acceleration

Haolei Ye
u5870415

Supervised by
Dr. Eric C. McCreath

May 24, 2018

A thesis submitted in part fulfillment of the degree of

Master of Computing
College of Engineering and Computer Science
The Australian National University
This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of the author’s knowledge, it contains no material previously published or written by another person, except where due reference is made in the text.

Haolei Ye

May 24, 2018

©Haolei Ye
For Eric C. McCreath,
who told me the fun of university life

For Peter Strazdins,
who helped me with the course permission code

For Fan Yu,
who led me to the gate of CUDA programming

For Luming Wang,
who provided me advice to steal time from GPU

For Haoting Xu,
who enlightened me when I was confused

For my mother, uncle and grandparents,
who allowed me to arrive here

For Sawamura Youraku Eriri,
the one who provides the name of the framework
Abstract

Free Lossless Audio Codec (FLAC) is a widely used lossless audio codec with a good compression rate. Fast decoding of FLAC is useful for audio editing applications, however, due to the sequential nature of the format it is difficult to improve performance using current single threaded approaches. This report presents a novel audio decoding algorithm for FLAC that uses GPU acceleration. The approach or algorithm takes the advantages of GPU computing power on Nvidia GeForce GTX 1080, and provides better performance than the official FLAC decoder and FFmpeg API implementations when run on a multi-core CPU. This produces the fastest currently known decoder of FLAC. Test results show 5.0 times faster than the FLAC official implementation for a 270.9MB FLAC audio file. The PRAM time complexity of the new algorithm for decoding FLAC is $O(1)$, compared to $O(n)$ serial implementation (where $n$ is the number of frames and assumes $n$ parallel processors). The approach is designed to extend to other formats easily.

Keywords: Audio, Audio Codec, FLAC, Signal processing, GPU, Decoder, CUDA, Linear-predisctive coding, Branch-bound, Branch-constructs, Parallel Processing, Accelerating, High-performance computing, Pascal, GP104
## Contents

Abstract .................................................. ii

List of Figures ........................................... v

List of Tables ........................................... vi

1 Introduction .............................................. 1
   1.1 GPU\textsuperscript{RAKU} ................................. 2
   1.2 Contribution .......................................... 2
   1.3 Limitations ........................................... 3
   1.4 Related Works ........................................ 3
   1.5 Forward .............................................. 4

2 Background ............................................... 5
   2.1 FLAC Format .......................................... 5
      2.1.1 Metadata Blocks .................................... 6
      2.1.2 Frames ............................................ 8
      2.1.3 Subframes ......................................... 11
      2.1.4 Rice Coding ....................................... 13
      2.1.5 Channel Assignments .............................. 17
   2.2 CUDA Architecture .................................... 18
      2.2.1 Pascal GPU Microarchitecture ..................... 19
      2.2.2 Pascal Memory Hierarchy ......................... 21
      2.2.3 CUDA Programming Model ......................... 23

3 Proposed Methods ...................................... 26
   3.1 Frameworks ........................................... 26
      3.1.1 CPU - FFMpeg API Framework ..................... 26
      3.1.2 GPU - GPU\textsuperscript{RAKU} Framework ............ 27
   3.2 Optimization Policies ............................... 29
      3.2.1 Replace Conditional Constructs to Tables .......... 30
3.2.2 Reduce Global Memory Access ..................................... 33
3.2.3 Lighten Thread Loads .................................................. 37

4 Experiment Results .......................................................... 39
  4.1 Experiment Environment ................................................. 39
  4.2 Results ........................................................................... 40
    4.2.1 Fixed-Linear Prediction decoding .............................. 40
    4.2.2 FIR-Linear Prediction decoding ............................... 41
    4.2.3 Normal decoding ..................................................... 42

5 Summary ............................................................................. 48
  5.1 Conclusion .................................................................... 48
  5.2 Future Works .................................................................. 49
    5.2.1 Performance Improvement ....................................... 49

Appendix ................................................................................ 51
  A Normal Decoding File Attributes ................................. 51
  B Kernel Function Profiling .............................................. 54

References .............................................................................. 65
List of Figures

2.1 LPC signal restore process .................................................. 13
2.2 CUDA Processor framework[1] ............................................ 19
2.3 Pascal Streaming Multiprocessor framework[1] ......................... 20
2.4 Hardware Model[2] ............................................................. 22
2.5 CUDA Thread Hierarchy[2] ..................................................... 23
2.6 Relationship of thread block, warp and hardware[3] ................. 24
2.7 Dynamic Parallelism[3] .......................................................... 24

3.1 FLAC decoding framework with FFMpeg API ............................ 26
3.2 FLAC decoding framework with CUDA GPU ............................ 28
3.3 Frame Finding and Header Decoding Schematic .......................... 38
3.4 Subframes Decoding and Restoring Schematic .......................... 38

4.1 Fixed-Linear Prediction decoding result ................................. 41
4.2 FIR-Linear Prediction decoding result .................................... 42
4.3 Time Usage of official FLAC, FFMpeg and GPURAKU decoders .... 43
4.4 Time Usage of the data transferring ....................................... 44
4.5 Time Usage of Each Frame on GPURAKU ............................... 45
4.6 Profiling Result from NVIDIA Visual Profiler .......................... 45
4.7 GPU Utilization of flac_cuda_find_frames ............................... 46
4.8 GPU Utilization of flac_cuda_decode_sub_frame ....................... 46
4.9 GPU Utilization of flac_cuda_restore_signal ........................... 46
4.10 GPU Utilization of flac_cuda_decorrelate_interchannel ............. 47
List of Tables

2.1 FLAC metadata block header structure ........................................ 6
2.2 FLAC STREAMINFO block structure ........................................... 7
2.3 FLAC STREAMINFO block example ............................................ 9
2.4 FLAC Frame Header structure ................................................ 9
2.5 FLAC Frame Header example .................................................. 10
2.6 FLAC LPC subframe example .................................................. 12
2.7 Unary Encoding Example ($B = 2$) .......................................... 15
2.8 FLAC LPC subframe residual decoding example .......................... 16
2.9 Memories on Nvidia GeForce GTX1080 (GP104)[4][5][3][6] ........ 22

A.1 Test FLAC audio files attributes ............................................. 51
Chapter 1

Introduction

Computers are widely used and the size of data processed by computer has increased as performance increased[7]. Various data compression methodologies are introduced to reduce the data size[8], this reduces the transfer and storage requirements. This enables the best usage of the computer resources like memory and hard disks. This is especially for audio formats. There are several well-known algorithms used for audio compression including MPEG Audio Layer III[9], WavPack[10] and Monkey’s Audio[11]. Free Lossless Audio Codec (FLAC) format is a popular audio codec first released in 2000 by Josh Coalson[12]. It is well-known for its lossless compression and being non-proprietary and open source[13]. FLAC format is now supported by various hardware and software[14]. There are two modern FLAC decoder implementations: one is the official FLAC encoder and decoder implementation by Josh Coalson and Xiph.Org Foundation[15], the other one is the implementation used in FFMpeg by FFMpeg team released in 2009[16].

There is a tradeoff in using compression formats that the decoding time of the compression file increases as the file becomes larger. As the large data machine learning and data mining of video and audio data are now becoming a hot topic of research, reducing the audio decoding time could save the time for users. This is especially important for audio and video editing users. Latest timeline-based video editing applications are using GPU accelerating for data rendering. For example, Adobe Premiere has its Mercury Streaming Engine for the solution[17]. Machines for this usage are armed with great graphics cards with definitely good performance to rendering image tracks with the help of these engines. However, these engines only process the graphics rendering and leave all the audio decoding and encoding to CPUs.
Introducing graphics processing unit (GPU) as the computing device is one possible solution. Although it was originally designed for computer graphics output, it is currently used for general-purpose computing and provides massively multithreaded executing ability[18]. As GPU with CUDA platform is already widely used for acceleration for video rendering[19], neural network generating[20] and game physic engines[21]. It would also help to use the GPU computing power for accelerating decoding if there is a method for decoding audio with CUDA.

1.1 GPUraku

In this report, a new practical audio decoding implementation, named GPUraku, is proposed for decoding FLAC file data into PCM signal and output PCM data as a WAV file. Although decoding FLAC is an entirely serial algorithm with massive branch constructs which is not suitable for the architecture of GPU, GPUraku introduced several methods to reduce or remove branches and allows it to speed up to maximum 5.0 times faster than the official CPU implementation. This framework is implemented in CUDA C and runs on Nvidia GPUs.

1.2 Contribution

The contributions made in this project are:

- Implemented the current fastest FLAC decoder;
- Summarized Nvidia’s GPU architecture memory performance parameters;
- Introduced a novel algorithm on FIXED type subframes to enable parallel execution;
- Investigated and applied several optimizations based on Pascal architecture for fully using the on-chip memory for less global memory accessing latency;
- Applied and analyzed several optimizations for reducing branch constructs in the algorithm for reducing instruction cache misses of decoding algorithm on GPUs. For instance, the constant table in frame header decoding.
- All the source codes of GPUraku can be downloaded at https://github.com/Harinlen/GPUraku and licensed under GNU General Public License version 2 or any later version. The source codes could be used to increase
the performance of the FLAC format audio decoding in all kinds of application on CUDA platform. It also enables the researchers to repeat our evaluation and built on for the research.

1.3 Limitations

GPURaku does not fully support the FLAC specification. The following parts are still not implemented.

1. **Frame UTF-8 coded Sample Number** The current implementation only supports for treating the UTF-8 coded index as frame index.

2. **Subframe Wasted bits-per-sample flag** The current implementation only supports for no wasted bits-per-sample. However, for 8-bit and 24-bit FLAC audio files, they are using this flag to encode the data. To decode these audio files, this feature needs to be implemented.

3. **RICE2_PARTITION implementation** The current implementation only supports for RICE_PARTITION. There is only a small difference on the bits of encoding parameter between these two approaches. It should be easy to implement.

4. **Use of CUDA rather than OpenCL** The current implementation is only deployed on Nvidia’s graphics card supporting CUDA. However, it could be ported to OpenCL which would enable it to be executed on a wider range of platforms.

1.4 Related Works

Many researchers have reported GPU acceleration in the audio decoding field: C. Xiaoliang, Z. Chengshi, M. Longhua, C. Xiaobin, and L. Xiaodong proposed a CUDA based MP3 audio decoder in 2010[22]. R. Ahmed and M. S. Islam introduced an optimized Apple Lossless Audio Codec (ALAC) decoder in 2016[7].

However, the only two papers focus on FLAC decoding. These are:

- Tian who shows a trial on porting the FLAC decoder to the CUDA framework[23]. However, there are many problems in this thesis: CUDA only has been used at the end phase of the entire framework[23], the maximum FLAC audio file is only 1565KB[23], the reported performance of the CPU
decoder is odd. As the official FLAC decoder is much faster than the data shown in the thesis on a lower-end processor model with a much bigger data. In Tian’s experiment data, decoding a 1565KB FLAC audio takes a Core i7 6700K processor around 250ms[23]. The new test data with official FLAC decoder and other implementations are shown in Chapter 4. These result have the CPU decodes such files in 142ms for official FLAC decoders.

- Huang also explores a CUDA FLAC decoder[24]. The framework is basically the same as Tian’s framework but moving the subframe decoding part from CPU to GPU[24]. But there are also problems in this paper involving the time complexity analysis and the experiment evaluation[24].

1.5 Forward

Chapter 2 introduces background to the FLAC format and information about CUDA including working methods, framework details and performance parameters.

Chapter 3 shows the methods and techniques that are used in the new GPU-RAKU CUDA decoding framework and provides the time complexity analysis.

Chapter 4 gives the results of the experiments with official FLAC decoder and FFmpeg FLAC decoder compare to GPURAKU, also this chapter compares the strengths and weaknesses of the candidate decoders.

Chapter 5 summarizes the report and provides the future research directions for FLAC decoding acceleration.
Chapter 2

Background

This chapter provides a description of the FLAC audio format because an exact understanding of this format is required to implement a decoder. Also, a background for GPU computing is given in the second section.

2.1 FLAC Format

FLAC was designed for fast low memory serial lossless decoding. This makes performance improvement challenging yet also interesting. In comparison to Vorbis or MP3 which uses floating point calculations transforming from frequency domains into temporal domain, FLAC only requires integer operations\[25\]. 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\[25\], this allows us to find a frame in the middle of the entire FLAC audio\[25\]. Hence, FLAC itself is a seekable format which allows finding all the frames concurrently. This is a key characteristic we exploited to let us parallelized the decoding implementation.

The format of FLAC has been documented in the FLAC format\[26\] page. FLAC audio has one FLAC stream\[26\]. Although interestingly it is not defined by this document, rather, the resulting implementation provided by Josh and Xiph.Org Foundation (the official reference FLAC decoder) defines the format.

FLAC is made of:

- A 32-bit FLAC stream marker in ASCII ("fLaC", Hexadecimal: 0x66 0x4C 0x61 0x43),
Chapter 2 Background

2.1 FLAC Format

- At least one metadata block structure (METADATA_BLOCK), and
- One or more audio frames.

In the following chapter expression, an audio sample is a integer or floating point number for expressing a specific time point of the entire audio. A channel is a set of samples (a stereo audio file has 2 channels). A block is a set of channels for a period of time. A frame is a compressed state block data in FLAC format. A stream is a set of frames to express the entire audio streams.

2.1.1 Metadata Blocks

A FLAC metadata block structure (METADATA_BLOCK) has a block header and the data of the block[26]. The component of the block structure is shown in Table 2.1[26].

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Last</td>
<td>Type (BLOCK_TYPE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Last Flag is a 1-bit flag which marks whether the current metadata block is the last metadata block before the audio frames[26]. 1 for yes and 0 for no[26].

Then it follows a 7-bit unsigned integer, describes the type of the metadata block[26]. A 7-bit unsigned integer has $2^7 = 128$ different values, which means that FLAC could have maximum 128 kinds of metadata blocks. 127 has been used for marking invalid to avoid confusion with a frame sync code[26], thus FLAC format could have 127 valid types of metadata block.

Currently, there are 7 types of metadata blocks have been defined[26]:

1. STREAMINFO (describes the information about the whole stream),
2. APPLICATION,
3. PADDING,
4. **SEEKTABLE,**

5. **VORBIS_COMMENT,**

6. **CUESHEET,** and

7. **PICTURE**

For the FLAC decoding part, block **STREAMINFO** stores the information about the stream attributes. The structure of the **STREAMINFO** block is shown in Table 2.2[26].

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Minimum block size (samples)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Maximum block size (samples)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Minimum frame size (bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Maximum frame size (bytes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Sample rate (Hz)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(number of channels)-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>(bits per sample)-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Total samples in stream</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>MD5 signature of the unencoded audio data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.2: FLAC STREAMINFO block structure**

From the document, the member of the **STREAMINFO** block has the following characteristic:
• The maximum and minimum block size might be identical. In this circumstance, it implies a fixed-blocksize stream[26]. The maximum block size is very important to allocate memory for the PCM samples.

• A single stream supports minimum 1 channel and maximum 8 channels[26].

• The bits per sample should be minimum 4 to maximum 32[26].

According to above characteristics, the memory usage of storing the final PCM data could be calculated from the parameters of STREAMINFO block. Each PCM sample of FLAC could be described as a signed 32-bit integer. Hence, it will take 4 bytes per sample. For each block, it has a maximum size (size(block)max). This value could be used as the alignment of the block samples. For each block, it has at least one channel and maximum 8 channels (ch). The aligned PCM sample array size of a n-block FLAC stream is
\[ n \times \text{size(block)}_{\text{max}} \times \text{ch} \times 4 \]

bytes.

In the format description, FLAC specifies a minimum block size of 16 samples. This would not only define the invalid of each frame, but also guarantee the frame index would be stored with a 32-bit unsigned integer. The total samples in a stream is a 36-bit unsigned integer, consider the circumstance that each frame contains only 16 samples, so the frame index would be maximum \( \frac{2^{36}}{2^{1}} = 2^{32} \), and it could be expressed in \( \log_{2}(2^{32}) = 32 \) bits.

Table 2.3 shows an example of FLAC STREAMINFO metadata block with its block header.

### 2.1.2 Frames

The audio data of a stream is separated into several Frames[26]. The structure of a Frame in FLAC contains the following parts[26]:

• Header,

• Channels of Subframes, e.g. a stereo stream has two channels, then each frames should have two Subframes.

• Zero-padding to byte alignment, and

• Footer.
Chapter 2 Background

2.1 FLAC Format

<table>
<thead>
<tr>
<th>Offset</th>
<th>00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00 00 00 22 04 80 04 80 00 00 0E 00 10 12 0B B8</td>
</tr>
<tr>
<td>0000</td>
<td>02 F0 00 BE 60 30 57 80 F3 6C 33 9C AB 7A A7 26</td>
</tr>
<tr>
<td>0000</td>
<td>2A B3 FE 16 C4 75</td>
</tr>
</tbody>
</table>

- Not the last metadata block
- Metadata block type: STREAMINFO
- Minimum block size: 1152
- Maximum block size: 1152
- Minimum frame size: 14
- Maximum frame size: 4114
- Sample rate: 48000
- 2 Channels
- Bits per sample: 16
- Total samples: 12476464
- MD5 signature

Table 2.3: FLAC STREAMINFO block example

The Header of the Frame provides the details of one frame. It includes but not limited the following attributes: block size, sample rate, and channel number[26].

The structure of the Frame Header is shown in Table 2.4[26].

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>Bk</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.4: FLAC Frame Header structure

Frame Header is a variable length structure. The length UTF-8 encoded frame number could be minimum 1 byte and maximum 8 bytes. Also, according to the block size and sample rate value, it will have minimum 0 and maximum 3 bytes after the UTF-8 encoded frame index. It is easy to find all the frames if the decoder finds frames consecutively right after the end of all the Metadata Blocks.
This makes it hard to be located in the file if the decoder starts to decode the FLAC stream from the middle because this structure might also appear in the middle of the frame. Table 2.5 shows an example of FLAC frame header.

<table>
<thead>
<tr>
<th>Offset</th>
<th>00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>FF F8 3A 18 01 14 00 00 00 00 00 00 00 00 00 00</td>
</tr>
</tbody>
</table>

Table 2.5: FLAC Frame Header example

The official guide[26] suggests to locate the frame position with the following three methods:

- Search for the sync code (‘1111111111111110’) of the frame header. The format definition itself ensures that the sync code only appears at the beginning of the frame header, and it could mark the start of a frame. However, the problem is that the sync code might appear in the middle of the frame, i.e. subframes. Hence, once a sync code has been detected, it needs further check to ensure it is a frame header or not.

- Check the parameter validation. Once a sync code has been found, it can determine a frame header. According to the definition of the frame header parameter, there are several reserved or invalid value for different parameters[26]. With the information from STREAMINFO block and parameter definition, the correctness of the frame header could be checked.

- The final check is the CRC-8 code at the end of the frame header. Generate an 8-bit CRC of the entire frame header before the last byte and compare the number to the byte.

With the rules above, the Algorithm 1 could be used to check whether a byte is a start of a frame.
Algorithm 1 FLAC Frame Header Detection

1: function IsFrameHeader(data)
2:    if data[0]=0xFF and data[1]&0xFE=0xF8 and data[3]&0x01=0x00 then
3:        if data[2]&0xF0≠0x00 and data[2]&0x0F≠0x0F then
4:            if bits per sample, channels, sample rate are same as stream then
5:                utfLength ← utf8_len(data + 4)
6:                frameLength ← 4 + utfLength + other
7:                crcResult ← crc8(data, frameLength)
8:                if crcResult=data[frameLength] then
9:                    return true
10:            end if
11:        end if
12:    end if
13: end if
14: return false
15: end function

In Algorithm 1, the other is the length of block size and sample rate length after the UTF-8 encoded frame index. utf8_len(x) is the function which could check x as the first byte of UTF-8 and return the length of the UTF-8 encoded number. crc8(data, length) is the function that calculates the CRC-8 value of first length bytes of unsigned char array data.

2.1.3 Subframes

Inside a frame, each channel has a subframe to describe the data for the channel[26]. There are four types of subframes[26]

1. VERBATIM This kind of subframe stores all the samples are not encoded. For a 16-bit 4096-sample subframe, it stores 4096 16-bit integers right after the subframe header.

2. CONSTANT This kind of subframe stores only one sample. The sample is the same in the entire subframe. For a 16-bit 4096-sample subframe, if it stores 0 as its sample, then this represents 4096 samples of 0.

3. LPC The full name is FIR-Linear Prediction subframe. In the header of the subframe, it stores the order of the subframe. This kind of subframes uses the following components to store the raw PCM samples (pcm): warm-up samples (w), coefficients (c), shift (s) and residuals (r). Table 2.6 shows an example of an LPC subframe.
Offset | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
---|---
0000 0000 | 4E 00 00 00 00 00 00 FF FF 00 00 01 00 01 00
0000 0001 | 01 B6 02 4A 1E 1B 40 2E 8B 78 9F 88 E0 F6 06 31
0000 0002 | 6B 5A D5 0B 2D 28 23 4D 2A 45 94 29 72 94 · ·

| 0100 1110 - Zero bit padding |
| 0100 1110 - Subframe type: SUBFRAME_LPC, order=8 |
| 0100 1110 - No wasted bits-per-sample in source subblock |
| 00 00 · · · 00 01 - 8 16-bit warm-up samples |
| B - Coefficients’ precision: 12 bits |
| 0110 0000 - Shift 12 bits |
| 0000 0010 4A 1E · · · F6 0000 0110 - Coefficients |
| 0000 0110 31 6B · · · - Residuals |

Table 2.6: FLAC LPC subframe example

The order of the subframe determines the length of \( w \) and \( c \). For a 16-bit 4096-sample with \( n \) warm-up samples and \( m \) level coefficients LPC subframe, it has 4096 \( - n \) residuals. Algorithm 2 shows the way to restore the LPC data to raw PCM samples.

**Algorithm 2** FLAC Restore LPC Signal

1. procedure \textsc{restoreLPC}(w, c, s, r, n, m, pcm)
2. \hspace{1em} for \( i \in [0, n) \) do
3. \hspace{2em} pcm[i] \leftarrow w[i]
4. \hspace{1em} end for
5. \hspace{1em} for \( i \in [n, \text{size}(pcm)) \) do
6. \hspace{2em} sum \leftarrow 0
7. \hspace{2em} for \( j \in [0, m) \) do
8. \hspace{3em} sum \leftarrow sum + c[j] \times pcm[i - 1 - j]
9. \hspace{2em} end for
10. \hspace{2em} pcm[i] \leftarrow pcm[i] + sum >> shift
11. \hspace{1em} end for
12. end procedure

Figure 2.1 visualizes the process of the LPC signal restore process with \( w = [-3, -5, -6], c = [3, -3, 1], s = 0 \). The PCM samples keep the first several warm-up samples, and use these warm-up samples to calculate the first PCM samples with the first residual. For all the other residuals, the
algorithm loops through all the residuals and calculates each PCM signal. Hence, each PCM signal is determined by its previous several signal numbers, coefficients, shifts and its own residual number.

4. FIXED The full name is Fixed-Linear Prediction subframe. This is a special kind of LPC. It also stores the order of subframe in its header. According to the different order of the subframe, it has five different sets of $c$: $\emptyset$, $[1]$, $[2, -1]$, $[3, -3, 1]$ and $[4, -6, 4, -1]$. The $s$ of the FIXED subframe is always 0. So it only needs to store $w$ and $r$ in the subframe.

2.1.4 Rice Coding

In FLAC format, Rice Coding is used as the residual coding for FIXED and LPC subframes[26]. Rice Coding is an improved adaptive Golomb coding, or a special case of Golomb coding[27]. Rice Coding is used in many lossless audio data compression methods as the entropy encoding stage. FLAC is one of the codecs that uses Rice Coding as the residual coding.

Rice Coding uses an adjustable parameter $D$, which should be $2^B (B \geq 0)$ and it will be used as the denominator. $D$ divides the raw value $N$ (which is the
numerator) into two parts: \( Q \) is the floor of the quotient, hence
\[
Q = \left\lfloor \frac{N}{D} \right\rfloor = \left\lfloor \frac{N}{2^B} \right\rfloor
\]
and \( R \) is the remainder, hence
\[
R = N - Q \times D
\]

The Rice Coding first stores \( Q \) in unary coding, and followed with truncated binary encoding \( R \). It is obvious that when \( B = 0 \), Rice Coding is equivalent to unary coding. As for the \( R \) part, it needs to be encoded in \( \log_2(D) \)-bit (or \( B \)-bit) binary.

For instance, suppose \( B = 4 \) and the raw value is \( N = 87 \). It implies that \( D = 2^B = 2^4 = 16 \) and \( R \) needs 4 bits to store. \( Q = \left\lfloor \frac{87}{16} \right\rfloor = \left\lfloor \frac{87}{16} \right\rfloor = 5 \) and \( R = N - Q \times D = 87 - 5 \times 16 = 7 \). The unary encoding \( Q \) is 111110, which is \( Q \) times 1 follows a 0. The 4-bit binary encoding \( R \) is 0100, which is binary \( R \) with 0 at the front to fill up blank bits. Hence, the Rice Coding of 87 with \( B = 4 \) is 1111100100.

In FLAC, it is using another type of unary coding. In the example above, \( Q \) would be stored as 000001 instead of 111110 in FLAC[26]. So, 87 would be stored as 0000010110 in FLAC as a residual. The Rice Coding is originally designed to encode unsigned integers, but it could extend to encode signed integers[28]. Reorder the integer in the following sequence: 0, -1, 1, -2, 2, \cdots. Using odd numbers \( (2n - 1) \) to map negative numbers \( (-n) \), and even number \( (2n) \) to map positive numbers \( (n) \). Table 2.7 lists the signed and unsigned integers using 8-bit two’s complement encoding (marked as Binary) and unary encoding with 2-bit remainder.

Unary encoding could use no more bits than normal 8-bits two’s complement encoding when it is used to express signed number from -12 to 12 or unsigned number from 0 to 23 as it shown in Table 2.7. As a matter of fact, unary coding with \( B \)-bit remainder could save bits from \( n \)-bit two’s complement encoding when it is expressing unsigned numbers from 0 to \( (n - B) \times 2^B - 1 \). In FLAC, it supports 4 and 5 bits remainders. When the bits per sample is 16, unary encoding would save bits for residual samples below 191 and 351 for unsigned numbers. So, FLAC would try to find the best order of LPC and FIXED type subframes which would save most of bits from the raw two’s complement encoding.
### 2.1 FLAC Format

#### Signed

<table>
<thead>
<tr>
<th>Signed</th>
<th>Unsigned</th>
<th>Unary Coding</th>
<th>Saved Length from 8-bit Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
<td>00000001 10</td>
<td>-2</td>
</tr>
<tr>
<td>14</td>
<td>28</td>
<td>00000001 10</td>
<td>-2</td>
</tr>
<tr>
<td>13</td>
<td>26</td>
<td>00000001 10</td>
<td>-1</td>
</tr>
<tr>
<td>12</td>
<td>24</td>
<td>00000001 00</td>
<td>-1</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>0000001 10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>0000001 00</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>00001 10</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>00001 00</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>0001 10</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>0001 00</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>001 10</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>001 00</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>01 10</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>01 00</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1 10</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1 00</td>
<td>5</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>1 01</td>
<td>5</td>
</tr>
<tr>
<td>-2</td>
<td>3</td>
<td>1 11</td>
<td>5</td>
</tr>
<tr>
<td>-3</td>
<td>5</td>
<td>01 01</td>
<td>4</td>
</tr>
<tr>
<td>-4</td>
<td>7</td>
<td>01 11</td>
<td>4</td>
</tr>
<tr>
<td>-5</td>
<td>9</td>
<td>001 01</td>
<td>3</td>
</tr>
<tr>
<td>-6</td>
<td>11</td>
<td>001 11</td>
<td>3</td>
</tr>
<tr>
<td>-7</td>
<td>13</td>
<td>0001 01</td>
<td>2</td>
</tr>
<tr>
<td>-8</td>
<td>15</td>
<td>0001 11</td>
<td>2</td>
</tr>
<tr>
<td>-9</td>
<td>17</td>
<td>0001 01</td>
<td>1</td>
</tr>
<tr>
<td>-10</td>
<td>19</td>
<td>00001 10</td>
<td>1</td>
</tr>
<tr>
<td>-11</td>
<td>21</td>
<td>000001 01</td>
<td>0</td>
</tr>
<tr>
<td>-12</td>
<td>23</td>
<td>000001 11</td>
<td>0</td>
</tr>
<tr>
<td>-13</td>
<td>25</td>
<td>0000001 01</td>
<td>-1</td>
</tr>
<tr>
<td>-14</td>
<td>27</td>
<td>0000001 11</td>
<td>-1</td>
</tr>
<tr>
<td>-15</td>
<td>29</td>
<td>00000001 11</td>
<td>-2</td>
</tr>
<tr>
<td>-16</td>
<td>31</td>
<td>00000001 11</td>
<td>-2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Table 2.7: Unary Encoding Example** ($B = 2$)
Table 2.8 shows the example of decoding the residuals of the subframe in Table 2.6.

<table>
<thead>
<tr>
<th>Offset</th>
<th>00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000 4E 00 00 00 00 00 FF FF 00 00 01 00 01 00</td>
</tr>
<tr>
<td>0000</td>
<td>0001 01 B6 02 4A 1E 1B 40 2E 8B 78 9F 88 E0 F6 06</td>
</tr>
<tr>
<td>0000</td>
<td>0002 6B 5A D5 0B 2D 28 23 4D 2A 45 94 29 72 94 ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>0000 0110 - Partitioned Rice coding with 4-bit Rice parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td>(0000 0110) - Rice parameter order=3, partitions=8</td>
</tr>
</tbody>
</table>

Start of a RICE_PARTITION

<table>
<thead>
<tr>
<th>Offset</th>
<th>06 31 (0000 0110 0011 0001) - Encoding parameter=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>(0011 0001) - Residual=0 (Unary=0, Remainder=0)</td>
</tr>
<tr>
<td>31 6B</td>
<td>(0011 0001 0110 1011) - Residual=2 (Unary=2, Remainder=0)</td>
</tr>
<tr>
<td>6B</td>
<td>(0110 1011) - Residual=-1 (Unary=0, Remainder=1)</td>
</tr>
<tr>
<td>6B</td>
<td>(0110 1011) - Residual=-1 (Unary=0, Remainder=1)</td>
</tr>
<tr>
<td>5A</td>
<td>(0101 1010) - Residual=-1 (Unary=0, Remainder=1)</td>
</tr>
<tr>
<td>5A</td>
<td>(0101 1010) - Residual=-1 (Unary=0, Remainder=1)</td>
</tr>
<tr>
<td>5A</td>
<td>(0101 1010) - Residual=-1 (Unary=0, Remainder=1)</td>
</tr>
<tr>
<td>D5</td>
<td>(1101 0101) - Residual=1 (Unary=1, Remainder=0)</td>
</tr>
<tr>
<td>D5</td>
<td>(1101 0101) - Residual=1 (Unary=1, Remainder=0)</td>
</tr>
<tr>
<td>D5</td>
<td>(1101 0101) - Residual=1 (Unary=1, Remainder=0)</td>
</tr>
</tbody>
</table>

... Table 2.8: FLAC LPC subframe residual decoding example

To decoding the rice encoding, decoder needs to check the first-bit many times.
To decoding an $n$-bit number, it needs to execute $n$ times conditional constructs.
Algorithm 3 shows the method to decode FLAC type unary encoding from unsigned raw data $r$.

The first-bit mask can be calculated by 1 left shift for the bit length of $r$ minus 1. If $r$ is a 32-bit integer, the first bit mask would be $1 << 31$. Suppose the result of the algorithm is $u$. As it shows in the Algorithm 3, the while loop is necessary and it could be treated as $u$ conditional constructs. But in the real implementation, it is more than is. The algorithm doesn’t show the part of reading next data to the $r$ as a bit stream. Hence it needs to do one more conditional construct for each loop which is whether $r$ reaches the end of the current byte and read the next byte of $r$ from the raw FLAC subframe data. The loop should
be treated as a set of $2 \times u$ conditional constructs. Consequently, decoding rice coding is a branch-bound type algorithm.

2.1.5 Channel Assignments

FLAC supports stereo stream to have different assignments for different frames[26]. There are four kinds of assignments supports for stereo stream audio[26]

- **Independent.** Two channels store their samples independently.
- **Left-side.** The left channel stores its original sample. The other channel stores the side channel. To get the right channel data, it needs to be restored with Algorithm 4.
- **Right-side.** The right channel stores its original sample. The other channel stores the side channel. The signal restore of the left channel is similar to Left-side channel assignment.
- **Mid-side.** It stores the middle channel and the side channel. The middle channel is the mid-point of each left and right channel samples. And the side channel stores the difference between the left and right channel to the middle channel. To get the left and right channel samples, it needs to be restored with Algorithm 5

### Algorithm 3 FLAC Read Unary Encoding

1: function READUNARY($r$)  
2: counter ← 0  
3: mask ← first-bit mask  
4: while $r \& mask = 0$ do  
5: counter ← counter + 1  
6: $r ← r << 1$  
7: end while  
8: return counter  
9: end function

### Algorithm 4 FLAC Restore Left-side Stereo Frame

1: procedure RESTORELEFTSIDEASSIGNMENT($left$, $side$, $right$)  
2: for $i \in (0, size(left))$ do  
3: $right[i] ← left[i] - side[i]$  
4: end for  
5: end procedure
Algorithm 5 FLAC Restore Mid-side Stereo Frame

1: procedure RESTOREMIDSIDEASSIGNMENT(middle, side, left, right)  
2:   for $i \in (0, \text{size}(middle))$ do  
3:     mid ← $(\text{middle}[i] \times 2)(\text{side}[i] \& 1)$  
4:     left[$i$] ← $(\text{mid} + \text{side}[i])/2$  
5:     right[$i$] ← $(\text{mid} + \text{side}[i])/2$  
6: end for  
7: end procedure

For different channel assignment, the bits per sample (bps) of the subframe would be changed. For the Independent frame, bps keeps the same. For the first subframe of Right-side frame and the second subframe of Left-side and Mid-side subframe, bps has to increase 1 bit, hence $bps + 1$ bits, for the subframe. The other subframes keep the same bps bits.

2.2 CUDA Architecture

Graphics Processing Unit (GPU) was introduced by Nvidia first in 1999 with the GeForce 256 chips[29], and General-purpose computing on graphics processing units (GPGPU) was available after 2001[30]. Compute Unified Device Architecture (CUDA) is a parallel computing platform model created by Nvidia in November 2006 with its GeForce 8800 GTX for general purpose processing[31]. It allows programmers to write parallel programs with various types of programming languages including C++ and compatible with C[31]. CUDA allows programmers to apply high-performance computing concepts to GPUs[30].

Unless otherwise specified, all the attributes, hardware architectures and software implementation described in this chapter are compatible with the Pascal architecture release by Nvidia Corporation in 2016 with CUDA 8.0. The CUDA architecture has been changed several times from the first CUDA support architecture Tesla G80 to Volta GV100[32]. The change of the architecture has been reflected in the support of different features and instruction sets called compute capability[31]. The compute capability of following description on Pascal architecture is 6.1, based on the white paper for the Pascal architecture and official guide or tutorial for Nvidia GeForce GTX 1080.
Chapter 2 Background 2.2 CUDA Architecture

2.2.1 Pascal GPU Microarchitecture

The Pascal architecture, developed by Nvidia, is the successor to the Maxwell architecture[33]. It was first released with Tesla P100 in 2016 and used in GeForce 10 series graphics cards[33].

2.2.1.1 CUDA Processor

The core of CUDA framework is its CUDA Processor, also called CUDA core. The framework of a single CUDA Processor is shown in Figure 2.2.

![CUDA Processor framework](image)

Figure 2.2: CUDA Processor framework[1]

A CUDA Processor has a fully pipelined ALU and FPU as its major computing component[1]. The ALU supports full 32-bit precision for all instructions[1]. Each cycle may issue 1 or 2 instructions[34]. Each CUDA Processor also has its own dispatch port, operand collector and result queue[1], but does not contain any branch prediction circuitry[35]. 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[36]. Although having many threads in the pipeline will hide the latency, the CUDA Processor performance would be poor when executing branch-bound programs.

CUDA Processor and other execution units inside the Streaming Multiprocessor execute the threads[1].

2.2.1.2 Streaming Multiprocessor

In CUDA, threads are organized in thread blocks[1]. The Streaming Multiprocessors are the components that execute the thread blocks[1]. Different architectures have different hardware configuration. In the Pascal architecture, each Streaming
**Multiprocessor** contains 128 *CUDA Processors*, 256 kilobytes register file capacity, 96 kilobytes shared memory, 48 kilobytes L1 cache and 8 texture units[37]. Figure 2.3 shows the diagram of the *Streaming Multiprocessor* on the Pascal architecture GP104.

![Pascal Streaming Multiprocessor diagram](image)

Figure 2.3: Pascal *Streaming Multiprocessor* framework[1]
In Figure 2.3, it clearly shows that 128 \textit{CUDA processors} have been divided into 4 groups. In each group, it has its own 64 kilobytes register file, total 16384 32-bit registers shared by 32 \textit{CUDA processors}. Average 512 registers per \textit{CUDA processor}. The \textit{Streaming Multiprocessors} schedule wrap execution[37].

All the instructions are stored in global memory[4]. Before \textit{CUDA Processors} execute the instructions, the \textit{Streaming Multiprocessor} has to load and cache them first[4].

\textbf{2.2.1.3 Pascal GP104 GPU}

In Pascal architecture, one \textit{Streaming Multiprocessors} and a \textit{Polymorph Engine} combine as a \textit{Texture / Processor Cluster} (TPC). 5 TPCs and a \textit{Raster Engine} combine as a \textit{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[37]. Therefore, GP104 provides total 20 \textit{Streaming Multiprocessors}, 2560 \textit{CUDA processors} and 320 gigabytes per second memory bandwidth.

\textbf{2.2.2 Pascal Memory Hierarchy}

The memory resources could be categorized as registers, local memory, shared memory, global memory, constant memory, texture memory[38]. The memory hierarchy model on hardware level is shown in Figure 2.4.

Registers are the fastest storage units that a thread could use. The latency of registers is basically 0 cycles for reading or around 20 cycles for R-after-W[39]. As mentioned in Section 2.2.1.2, For a single \textit{Streaming Multiprocessor}, the number of registers is limited. The usage of registers is transparent to developers and managed by compilers[5], but could still be suggested by the register keyword.

Shared memory and L1 cache are also on-chip memories[5]. They have very low latency and high throughput (aggregate more than 1 terabytes per second)[5]. L2 cache is a on-chip memories also[37]. All the global memory access would go through L2 cache to lower the latency of the global memory[5]. Global memory is a kind of off-chip memory[5]. The latency of global memory is very high (around 400 to 800 clock cycles)[5]. As mentioned in Section 2.2.1.2, accessing global memory frequently is expensive. For better performance, Nvidia suggests reducing branching for avoiding cache misses[4].
Table 2.9 summarizes the focus points of the CUDA memory hierarchy of GP104 on the official Nvidia GeForce GTX1080 graphics card.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Latency (Cycles)</th>
<th>Size (Total)</th>
<th>Location</th>
<th>Access</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>0(read) / ~20(r-a-w)</td>
<td>5120 KB</td>
<td>On-Chip</td>
<td>R/W</td>
<td>N/A</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>~28</td>
<td>960 KB</td>
<td>On-Chip</td>
<td>-</td>
<td>N/A</td>
</tr>
<tr>
<td>Shared</td>
<td>~92</td>
<td>1920 KB</td>
<td>On-Chip</td>
<td>R/W</td>
<td>N/A</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>~200</td>
<td>2048 KB</td>
<td>On-Chip</td>
<td>-</td>
<td>N/A</td>
</tr>
<tr>
<td>Local</td>
<td>~400 - ~800</td>
<td>-</td>
<td>DRAM</td>
<td>R/W</td>
<td>Yes</td>
</tr>
<tr>
<td>Constant</td>
<td>~400 - ~800</td>
<td>64 KB</td>
<td>DRAM</td>
<td>R</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture</td>
<td>~400 - ~800</td>
<td>Up to 8 GB</td>
<td>DRAM</td>
<td>R</td>
<td>Yes</td>
</tr>
<tr>
<td>Global</td>
<td>~400 - ~800</td>
<td>Up to 8 GB</td>
<td>DRAM</td>
<td>R/W</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 2.9: Memories on Nvidia GeForce GTX1080 (GP104)[4][5][3][6]
## 2.2.3 CUDA Programming Model

### 2.2.3.1 Threads, Blocks and Grids

From the programmers’ perspective, concurrency is described as threads. In CUDA, it uses Parallel Thread Execution (PTX) programming model for an explicitly parallel approach[2]. Threads are gathered as blocks, and blocks are gathered as grids[31]. The relationship of threads, blocks and grids are shown in Figure 2.5.

![Figure 2.5: CUDA Thread Hierarchy][2]

For GP104, the following limitations have been applied[6]

1. Each *thread* could use maximum 255 32-bit registers and 512 kilobytes local memory;

2. Each *block* could have 3 dimensions with the maximum of each dimension being (1024, 1024, 24). And contains a maximum of 1024 threads and use maximum 48 kilobytes of shared memory;

3. A single *Streaming Processor* could have a maximum of 32 resident blocks.

### 2.2.3.2 Warp

The *Streaming Multiprocessor* manages several threads as a group, these are called *warps* in CUDA[6]. This is the hardware view of the thread block. When the thread block is going to be executed on *Streaming Multiprocessors*, they would divide threads into *warps* and be scheduled by the *warp schedulers* shown in Pascal *Streaming Processor* framework (Figure 2.2.1.2)[6].

Figure 2.6 describes the relationship between the thread *block* and *warp*. *Warp* is the basic schedule unit in the CUDA kernel execution[6]. The *warp* size of GP104
is 32[6]. When executing an \( n \)-thread block, the number of the blocks would be \( \lceil \frac{n}{32} \rceil \)[3]. If the threads are not an even multiple of 32, it would add several inactive threads to the block to make it an even number of 32[3]. These inactive threads would still consume the resource of Streaming Multiprocessor like registers and shared memories[3]. To maximize the usage of the Streaming Multiprocessor, the number of threads in a thread block should be a multiples of 32.

However, for one Streaming Multiprocessor, it has a limitation of resident warps. This limitation on GP104 is 64[6]. So, the maximum number of running threads on a single GP104 Streaming Multiprocessor is 2048, and this is also the limitation of the maximum resident threads of one Streaming Multiprocessor[6]. These warps can come from different blocks as a Streaming Multiprocessor can hold multiple blocks which described in Section 2.2.3.1. However, threads from the same block must be executed in the same Streaming Multiprocessor[40].

### 2.2.3.3 Dynamic Parallelism
Dynamic Parallelism is a feature that allows GPU kernel functions to launch and manage other kernel functions inside itself. This has been available since the Kepler GPU[41] with 3.5 compute capability and above[3]. Figure 2.7 shows the state with and without dynamic parallelism.

Without dynamic parallelism, the GPU function could be only allowed to launch by the CPU[3]. If the GPU supports dynamic parallelism, the kernel function could be executed right in the kernel function[3].
Chapter 3

Proposed Methods

This chapter describes the structure of the new parallel FLAC decoding algorithm and the several methods for optimization the GPU decoding performance.

3.1 Frameworks

3.1.1 CPU - FFMpeg API Framework

The workflow of decoding FLAC audio with FFMpeg library API is shown in Figure 3.1. The CPU decoding version is using this workflow. The framework is modified from the official example ‘trancoding_aac.c’[42].

![FLAC decoding framework with FFMpeg API](image)

Figure 3.1: 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 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[43]. And then, use a loop to decode the entire audio data. FFmpeg provides the following two functions for decoding the data:

1. **av_read_frame** For reading an FFmpeg audio frame from the FLAC format file[44]. However, the parameters of the function are `AVFormatContext *` and `AVPacket *`. The function actually reads the frame from the format context to the packet.

2. **avcodec_decode_audio4** Decode audio frame of a specific size from packet into the audio frame[45]. In the latest version of FFmpeg it is marked as 'Deprecated' but still used in the example[45]. This function should be used after `av_read_frame` read the frame data to the packet. The samples could add to a queue and write the samples to the file. These samples could be played by audio playing backend libraries like `libsoundio` or `SDL`.

In the implementation of the FFmpeg API version, it uses a first-in first-out structure to cache the decoded frames. Once the data of the decoded samples is enough for encoding a frame, it will encode the frame into the WAV format.

The entire FFmpeg API decoding framework is running on the CPU. All the optimizations rely on the optimization configuration switches of FFmpeg itself. The compiler configuration of the example is using the official Makefile of examples.

### 3.1.2 GPU - GPUraiku Framework

FFmpeg provides the general workflow of decoding a FLAC audio. Suppose a FLAC file has $n$ frames and the First-In First-Out (FIFO) queue could always output data when the new frame decoded. The time complexity of FFmpeg API framework working on decoding FLAC is $O(n)$ where $n$ is the number of frames.

GPUraiku is a new audio decoding framework designed work with CUDA and CUDA GPUs. It reduces the time complexity of decoding FLAC to $O(1)$. The proposed framework of the CUDA decoding framework is shown in Figure 3.2.
CPU first reads the entire raw FLAC data, and parses the metadata block and skip to the start position of the start frames. Then CPU searches the last frame and records its frame index to get the frame count.

CPU transfers the frame raw data to the CUDA device, and GPU would start to find all the positions of the frames, and save the positions into an array. Next
GPU would start to decode all the frame header concurrently, and prepare the frame header data for the subframes. Then loop the number of channel times, decoding and restore the subframes. Last, GPU process the channel assignments and transfer the data back to CPU. CPU simply writes the PCM data out in WAV format.

In the new CUDA framework, the time complexity has been separated into two parts: finding frames and decoding frames. Assume that GPU has enough cores and FLAC frame data has $m$ bytes. Use $m$ threads to check whether the data is the start of the position. The time complexity of this procedure could be treat as $O(1)$. All $n$ frames have been tested now, use $n$ threads to decode all these frames and the time complexity of this procedure could be also $O(1)$. Consequently, the Parallel Random-Access Machine (PRAM) parallel algorithm could decode the FLAC file with $O(1)$ time complexity.

However, in practice, the number of threads to check the entire data area is bound. And hence, the performance becomes linear ($O\left(\frac{n}{m}\right)$, $m$ is the number of CUDA Processors) as the number of frames increases. For the high-end models GPU, like Nvidia Tesla V100, it only has 5,120 CUDA Processors[46]. The number of threads needs to be changed. In the implementation, for $n$ frames, the framework would use $\lceil \frac{n}{32} \rceil \times 32$ threads to find the frames, according to the CUDA warp model described in Section 2.2.3.2. This number would be at least $n$ when $n$ is an integer multiple of 32. Threads would divide the entire FLAC frame data into several same parts and each thread would search for its own threads. If it finds the start position of a frame it will save the position of the frame. The length of FLAC frames is not identical, but the distribution of the frames is overall average in the FLAC frame data area especially when all the frames are using the VERBATIM subframes to encode the data. Therefore, the number of frames that founded by each thread could be treated as 1.

## 3.2 Optimization Policies

Although the framework itself highly reduce the time complexity of the algorithm. Because of the special architecture of GPU, it needs many optimizations to increase the performance. These optimizations are explained in the following sections.
3.2.1 Replace Conditional Constructs to Tables

According to Section 2.2.1.1 and Section 2.2.1.2, CUDA processor performs badly at conditional constructs. However, FLAC decoding is a massive conditional procedure, such as reading the UTF-8 frame index, reading frame header information, reading rice encoding residuals, reading specific bits signed or unsigned integer and so on. Some branch constructs could be directly reduced, like the bit stream reader for the subframes. The usual implementation of reading \( n \) bits is repeating the read bit function for \( n \) times. But it would execute \( n \) conditional constructs for checking it should switch to the next bytes. For reading more than 8 bits it might go through 2 bytes or more. But most of these could be reduced to only one conditional statement by checking the position of the current bit of the byte. This kind of reduction is straight forward. There are many other means to reduce conditional constructs. One method of reducing the conditional constructs is replacing them with tables.

3.2.1.1 Constant Value Table

In the frame header structure mentioned in Section 2.1.2, all parameters are encoded under 4 bits. So each parameter could have maximum 16 different possible value. Hence, the massive branch construct of value checking (a switch statement) could be replaced by a table.

Algorithm 6 shows the original massive branch construct version of reading the block size of a specific frame with the start position of frame data.

Algorithm 7 shows the table seeking implementation of Algorithm 6. It replaces the 16 branch statements into a single calculation statement. The only thing we need to prepare is a table (blockSizeTable) which stores all the data of block size. This table could be reused in all the frame headers. It only needs 64 bytes (16 \( \times 4 \)) so it could be stored in the constant memory for lower latency (mentioned in Section 2.2.2).

The constant value table replacement could be 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 channel parameter needs two tables, because the channel assignment only exists for stereo audio, so the number of channel for the frame who has channel
Chapter 3 Proposed Methods

3.2 Optimization Policies

Algorithm 6 Decoding FLAC Frame BlockSize (Massive Branch Construct)

1: procedure GETFRAMEBLOCKSIZE(data, blockSize)
2:     rawBlockSize ← (data[2]&0xFFFF) >> 4
3:     if rawBlockSize = 0 then
4:         raise INVALID_VALUE
5:     else if rawBlockSize = 1 then
6:         blockSize ← 192
7:     else if rawBlockSize = 2 then
8:         blockSize ← 576
9:         ... 
10:    else if rawBlockSize = 15 then
11:        blockSize ← 32768
12:    end if
13: end procedure

Algorithm 7 Decoding FLAC Frame BlockSize (Table Seeking)

1: blockSizeTable[] ← [0, 192, 576, ... , 32768]
2: procedure GETFRAMEBLOCKSIZE(data, blockSize)
3:     blockSize ← blockSizeTable[(data[2]&0xFFFF) >> 4]
4: end procedure

Assignment is 2, and for those frames who have only 1 channel or more than 2 channels, they could only have independent channel assignment.

3.2.1.2 Function Table

Constant value table was used to reduce the control transfer instructions of value assignment, but it cannot apply to the UTF-8 and CRC-8 evaluation. This type of calculations was optimized with a function table.

There are two kinds situations of calculation that could be replaced by a function table: branch calculation assignment and limited loop calculation.

Branch calculation assignment stands for assigning a value by calculation with a limited possibilities parameter. One of the typical calculation is reading a number of specific bits.

In the bit stream data of decoding residuals and some other samples. With the limitations of the bits per sample, this part could be implemented as a massive-
Chapter 3 Proposed Methods

3.2 Optimization Policies

branch construct like a *switch* statement in C. Algorithm 8 describes the algorithm to fetch an *n*-bit integer using multiple branch constructs with the start position of binary data(*data*).

**Algorithm 8** Fetching *n* bits as a Number (Massive Branch Construct)

1: function FETCH UNSIGNED BITS NUMBER(*data*, *n*)
2:     if *n* = 0 then
3:         return Null
4:     else if *n* = 1 then
5:         return *data*[0] & 0x80
6:     . . .
7:     else if *n* = 32 then
9:     end if
10: end function

For the actual implementation of Algorithm 8, the data needed to be updated after every function call. Hence, the data array needs to be updated after each fetching operations. To avoid moving the entire array, an 8-bit cache is introduced to record the left bits of the current byte.

However, on GPUs, executing massive branch construct like this would cause a huge instruction cache misses. Due to the lack of branch prediction, the instruction cache would definitely miss loading the correct instruction. The control transfer instruction of the calculation cannot be avoided due to the different calculation methods are applied, but the control transfer instruction of the bit number checking could be 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. 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 only execute that function for fetching bits from binary data.

**Limited loop calculation** means the value is calculated with a limited loop. In FLAC decoding, CRC-8 calculation and the sample reading are classified as this kind of calculation.
The value of CRC-8 at different position could be easily calculated and replaced by a constant table as mentioned before. The problem is it needs a loop from the start position to the end position. Generally, it is impossible 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 here could be enumerated. All the calculations are stored in a function table and replace the original multiple branch construct by the table seeking.

### 3.2.2 Reduce Global Memory Access

The raw FLAC audio data and PCM decoded data has 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.

For the LPC and FIXED subframes, to restore its PCM data, it needs to store the warm up samples, coefficients, shift bits and residuals. Warm up samples and residuals use 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 32 32-bit signed integer. Hence, the memory space of decoding LPC and FIXED subframes is predictable.

When all these parameters have been read from raw LPC and FIXED subframes, it still needs to execute the Algorithm 2 in Section 2.1.3 to restore the original PCM samples. According to the Algorithm 2, it needs access the previous samples frequently. For example, to restore the PCM samples with Algorithm 2 in a $n$ coefficients LPC subframe with $m$ samples, it needs to access the global memory for $n(m - n)$ times, which could waste a lot of processor cycles. The following two methods are introduced to reduce the memory access to $2m - n$ times.

#### 3.2.2.1 Sample Cache Array

One simple idea is to store all the previous samples in an array and calculated them on the GPU. Algorithm 9 shows how this idea works with an array (cache) on shared memory. Suppose the first $order$ samples in $pcm$ are the warm up samples, and all the rest $blockSize - order$ samples are residuals, coefficients $c$ are all stored at shared memory.
Algorithm 9 FLAC Restore LPC Signal

1: procedure RESTORELPC(order, blockSize, c, s, pcm)
2:  for $i \in [0, order)$ do
3:      cache[$i$] $\leftarrow$ pcm[$i$]
4:  end for
5:  for $i \in [order, blockSize)$ do
6:      sum $\leftarrow$ 0
7:      for $j \in [0, order)$ do
8:          sum $\leftarrow$ sum + $c[j] *$ cache[$j$]
9:      end for
10:     cache[$order - 1$] $\leftarrow$ pcm[$i$] + sum $>>$ shift
11:  end for
12:  pcm[$i$] $\leftarrow$ cache[$order - 1$]
13: end procedure

In the first loop, the algorithm reads the first order samples and saves to the cache array. This cost order times access. And then in the other loop, the decoding loop, 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 cost $2 \times (blockSize - order)$ times access. Hence, it cost $2 \times blockSize - order$ times access. In fact, blockSize is $m$ and order it $n$. Thus the global memory access times of the Algorithm 9 is $2m - n$.

3.2.2.2 N-Level Increased Progression

Sample cache array could accelerate all the LPC calculation, but it still needs to do multiplications. However, in the FIXED subframe, it is possible to reduce it into only additions.

Consider the following coefficients: $[3, -3, 1]$. Suppose the warm up samples are $r_1, r_2, r_3$ and residuals are $r_4, r_5, r_6, \ldots$. If $p_1, p_2, p_3, \ldots$ are the final PCM samples, it is obviously that

\begin{align}
    p_1 &= r_1 \\
    p_2 &= r_2 \\
    p_3 &= r_3
\end{align}
According to Algorithm 2, it can get

\[
p_4 = r_4 + 3p_3 - 3p_2 + p_1 = r_4 + 3r_3 - 3r_2 + r_1
\]

\[
p_5 = r_5 + 3p_4 - 3p_3 + p_2 = r_5 + 3r_4 + 6r_3 - 8r_2 + 3r_1
\]

\[
p_6 = r_6 + 3p_5 - 3p_4 + p_3 = r_6 + 3r_5 + 6r_4 + 10r_3 - 15r_2 + 6r_1
\]

\[
\ldots
\]

Assign progression \( c'_x \) which

\[
c'_x = \begin{cases} 
  r_3 - r_2 & \text{if } x = 3 \\
  p_x - p_{x-1} & \text{if } x > 3
\end{cases}
\]

Then

\[
c'_3 = r_3 - r_2
\]

\[
c'_4 = p_4 - p_3 = r_4 + 2r_3 - 3r_2 + r_1
\]

\[
c'_5 = p_5 - p_4 = r_5 + 2r_4 + 3r_3 - 5r_2 + 2r_1
\]

\[
c'_6 = p_6 - p_5 = r_6 + 2r_5 + 3r_4 + 4r_3 - 7r_2 + 3r_1
\]

\[
\ldots
\]

Assign progression \( c_x \) which

\[
c_x = \begin{cases} 
  r_3 - 2r_2 + r_1 & \text{if } x = 3 \\
  c'_x - c'_{x-1} & \text{if } x > 3
\end{cases}
\]
Then

\[ c_3 = r_3 - 2r_2 + r_1 \]
\[ c_4 = c_4' - c_3' \]
\[ = r_4 + r_3 - 2r_2 + r_1 \]
\[ c_5 = c_5' - c_4' \]
\[ = r_5 + r_4 + r_3 - 2r_2 + r_1 \]
\[ c_6 = c_6' - c_5' \]
\[ = r_6 + r_5 + r_4 + r_3 - 2r_2 + r_1 \]

\( (3.4) \)

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

\[ d_4 = c_4' - c_3' \]
\[ = r_4 \]
\[ d_5 = c_5' - c_4' \]
\[ = r_5 \]
\[ d_6 = c_6' - c_5' \]
\[ = r_6 \]

\( (3.5) \)

Progression \( d_x \) is the same as the residual \( r_x \). The new algorithm is actually the reversed way to calculate \( p_x \). Algorithm 10 shows the new way to restore PCM signal of the fixed coefficients \([3, -3, 1]\).

**Algorithm 10 FLAC Restore No.3 Fixed PCM Signal**

1: **procedure** RESTOREFIXED3(blockSize, pcm)
2: \( c \leftarrow pcm[2] - 2 \times pcm[1] + pcm[0] \)
4: **for** \( i \in [3, \text{blockSize}] \) **do**
5: \( c \leftarrow c + pcm[i] \)
6: \( c' \leftarrow c' + c \)
7: \( pcm[i] \leftarrow c' + pcm[i-1] \)
8: **end for**
9: **end procedure**

In the first two assignment statements, it only needs to access global memory for 3 times to get \( pcm[0], pcm[1] \) and \( pcm[2] \). In the loop, the \( pcm[i-1] \) could be
stored and updated by a local variable in the register, so it only needs to access global memory for 2 times for reading $pcm[i]$ and update $pcm[i]$. So it needs $2 \times blockSize - 3$ times global memory access, which performs the same as the Algorithm 9. However, the loop only has addition statements now, which doesn’t need to calculate the sample multiply by the coefficients.

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


(3.6)

For $N$ coefficients, it needs $N - 1$ level progression addition up to restore the PCM signal. This is called the $N$-level increased progression. It could help to decrease both the global memory access and reduce the calculation instruction complexity of FIXED subframe signal restoring.

3.2.3 Lighten Thread Loads

To decrease the instruction cache misses, there is a suggestion from Nvidia to merge kernels because short kernels are more impacted by instruction cache misses[4]. But there is another limitation needed to be considered and that is the resource per thread.

The workflow shown in Figure 3.2 could be merged into only one kernel execution. However, the resource usage would be dramatically increased. Consider the register limitation described in Section 2.2.1.2 and the kernel executing timeout limitation[31], the workflow of FLAC has been divided into the following stages

1. Frame Finding and Header Decoding This stage goes through the entire FLAC file and search for the frames. Each thread would go through approximately one frame length and locate the position of all the frames. Most of the resources are used as function pointers and data caches. Header decoding is a very small stage. In this stage, the function uses the constant value table to get the parameters for each frame. It would also save the start position of the subframes data and prepare the bit stream for the subframe decoding. Figure 3.3 shows the schematic of this stage.
2. **Subframes Decoding and Restoring** This stage is implemented by a loop on CPU for decoding all the channels in the frame. Two kernel function are introduced: one for decoding the subframes according to the official definitions, the other one is restoring the CONSTANT, FIXED and LPC subframes to the PCM signal samples. Each subframe matches a channel. Figure 3.3 shows the schematic of this stage with a stereo audio FLAC file.

3. **Restore Channel Assignments** This is a stage that could introduce the dynamic parallelism mentioned in Section 2.2.3.3. All the channels have been decoded however stereo audio needs to be converted from the side channel encoding into independent channels. Each sample is now independent to each other inside the channel, so it is easily parallelized. A very low number of registers are used in this stage.
Chapter 4

Experiment Results

The GPU RAKU audio decoding library is the implementation on the GPU with CUDA using the model mentioned in Chapter 3. To evaluate the performance of the program, the following implementations are chosen for comparison FLAC decoders

- **FLAC official encoder/decoder** This is the command-line implementation by Josh Coalson and Xiph.Org Foundation\[15\]. It shows the result of official optimized performance for vectorized CPUs. It variously uses hard coded SSE2, SSE3 and AVX instructions for accelerating calculations\[47\].

- **FFMpeg official binary** This is the command-line binary implementation by FFMpeg organization\[16\]. The file format decoding algorithm has been changed for the FLAC decoding structure. The entire FFMpeg framework supports multiple threads decoding, but the implementation of the FLAC format does not make use of these multiple threads\[16\]. But it still has one thread for reading and writing operation and one thread for decoding provided by the FFMpeg architecture itself\[16\].

4.1 Experiment Environment

Experiments are executed on a machine with the following hardware configuration:

- Intel® Core™ i7-6700K CPU (4.00GHz, 4 cores 8 threads)
- Dual Channel 16GB DDR4-2400 Memory
- ASUS Z170 Pro Gaming/AURA Motherboard
- Kingston 240GB Solid State Drive
• Gigabyte GeForce® GTX 1080 G1 Gaming 8G (GV-N1080G1 GAMING-8GD)

and the following software environment
• Ubuntu 16.04.03 LTS operating system
• CUDA compilation tools, release 9.0, version 9.0.176
• NVRM version: NVIDIA UNIX x86_64 Kernel Module 384.90 Tue Sep 19 19:17:35 PDT 2017
• GCC version 5.4.0

The version of all experiment objects are
• FFMpeg 2.8.11, marked as FFMpeg in the chart
• FLAC official decoder 1.3.1, marked as FLAC in the chart
• GPURAKU 10.0.9, marked as CUDA in the chart

The time is measured using the Linux system call \texttt{ftime()} on the GPURAKU evaluation. For FLAC official decoder and FFMpeg binaries use the \texttt{system()} call to execute and use \texttt{ftime()} to record their executing time. All the objects are executed for 10 times and the minimum using time is recorded.

4.2 Results

4.2.1 Fixed-Linear Prediction decoding

This test is designed to test the performance of all the decoders to decode \texttt{FIXED} type \textit{subframes} encoded FLAC audio. A FLAC audio file with following attributes is chosen as the Fixed-Linear Prediction decoding experiment material

• Duration: 4,678 seconds (1 hour 17 minutes 58 seconds)
• File size: 696,822,177 bytes (696.8 MB)
• MD5 hash sum 5f7b776e25d8b2fa51bbf0b0353e35e2
• Stereo. Signed 16-bit per sample. 48,000 Hz
• 194,944 FLAC frames in total: 0 CONSTANT, 0 VERBATIM, 389, 890 FIXED and 0 LPC \textit{subframes}. 
This file contains only FIXED subframes which is encoded command line `flac -disable-constant-subframes -disable-verbatim-subframes -max-lpc-order=0 -no-seektable test_wav_file.wav`. This file is set to test the result of decoding FIXED subframes with the new implementation of FIXED subframes decoding.

The result of all decoders shows in Figure 4.1.

![Figure 4.1: Fixed-Linear Prediction decoding result](image)

The CUDA version is 2.49 times faster than FFMpeg implementation to 2.77 times faster than the FLAC official implementation on decoding FIXED subframes.

### 4.2.2 FIR-Linear Prediction decoding

This test is designed to evaluate the performance of decoding LPC type subframes encoded FLAC audio. The same PCM audio files using in the Section 4.2.1 are used in this experiment. The command line used to encode the FLAC file is `flac -disable-constant-subframes -disable-verbatim-subframes -disable-fixed-subframes -no-seektable test_wav_file.wav`. The experiment material FLAC audio file has the following attributes:

- **Duration**: 4,678 seconds (1 hour 17 minutes 58 seconds)
- **File size**: 630,939,196 bytes (630.9 MB)
- **MD5 hash sum**: 9ac0baef7d14ecb8dcb13fd76efb398f
- **Stereo. Signed 16-bit per sample. 48,000 Hz**
- **54,829 FLAC frames in total: 0 CONSTANT, 0 VERBATIM, 0 FIXED and 109,658 LPC subframes.**
The test file contains only LPC subframes for the entire file. Compare to the FIXED subframe, LPC subframes have various lengths of coefficients as the description in Section 2.1.3. However, the calculation of restoring the PCM signal is actually an integer Fused Multiply-Addition calculation. However, as the compiler shows, the CUDA fma() function only supports for float and double type. Although the precision of float is enough for 16-bit integer calculations, it is not enough for 24-bit integer calculations. Hence, the implementation on CUDA does not take advantage of fused multiply and addition instruction.

Figure 4.2 shows the experiment result.

<table>
<thead>
<tr>
<th></th>
<th>Time usage (unit: ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLAC</td>
<td>7323</td>
</tr>
<tr>
<td>FFMpeg</td>
<td>4433</td>
</tr>
<tr>
<td>CUDA</td>
<td>1615</td>
</tr>
</tbody>
</table>

Figure 4.2: FIR-Linear Prediction decoding result

According to the result, the CUDA accelerated version is 2.59 times faster than the FFMpeg implementation and 4.3 times faster than the FLAC official implementations. For the CPU implementations, time usage increase dramatically especially for FLAC implementation due to the increase of data calculation complexity. The time usage of CUDA doesn’t change much because of the power from $O(1)$ time-complexity algorithm. From the profiling, it also shows the improvement of using the shared memory to be the cache. The bandwidth of global memory accessing is just 39.878 GB/s, but it achieves 1992.195 GB/s (about 2.0 TB/s) for memory accessing.

4.2.3 Normal decoding

This test is designed to test the performance of the candidate decoders to decode FLAC audio files with default compression settings. All the following FLAC testing audio files are compressed with the official FLAC encoder. The command line is `flac –no-seektable –best test_wav_file.wav`.

All the test audio files are 16-bit signed stereo audio sampling at 48,000 Hz. The other attributes of FLAC audio files are shown in Appendix A. The original
WAV file is a 4,620 seconds long audio file. The first 76 files are split from this WAV file from 0 seconds. The frame size and duration of test files are increased linearly. The last file is the raw WAV file but encoded using FLAC encoder. The time usage results of all the candidate decoders are shown in Figure 4.3.

![Figure 4.3: Time Usage of official FLAC, FFMpeg and GPURAKU decoders](image)

From the experiment result, GPURAKU is maximum 5.09 times faster than the CPU decoders. There are two aspects need to be pointed out.

The first aspect is the executing time of the official FLAC decoder is not a linear result, even the frame number increases linearly, so is the FFMpeg decoder. The files are all increased linearly. The reason for this is still unknown.

The time of GPURAKU is not linear because of the finite CUDA cores. The first reason is that time usage of the host/device data transferring. The data transferring is taking linear time. Figure 4.4 shows the H/D and D/H data transferring time from the total data. If we removing the time from the GPURAKU time usage (for example, if the data of audio stream is already loaded at the video RAM) and only considering the calculating time, GPURAKU is maximum 7.8 times than CPU decoders. Currently, there are two ways to reduce the time usage of the data.
One of the solutions is using streaming. The algorithm didn’t use any methods for streaming the data. Using streaming method could hide the time of transferring back to the data of calculation. The other method is using the unified memory on Pascal and later GPUs with hardware page faulting and migration[6].

![Figure 4.4: Time Usage of the data transferring](image)

The second reason is the $O(1)$ time complexity could only be established when there are as many cores as frames. When the cores are not enough, warps need to wait to be executed until there are available CUDA cores. If we calculate the decoding time of each frame from the 29th file, it shows the trend of time usage decreasing on the decoding time of each frame. Just as a $O(1)$ algorithm should perform. Figure 4.5 shows the time usage of each frames since the 29th file.

There are four kernel functions to be executed on GPU:

- `flac_cuda_find_frames` for finding the position of the frames and decoding the frame header
- `flac_cuda_decode_sub_frame` for decoding one sub frame of all the frames
- `flac_cuda_restore_signal` for restoring the residuals of the subframes into WAV samples
Chapter 4 Experiment Results

4.2 Results

![Figure 4.5: Time Usage of Each Frame on GPUraKU](image)

- `flac_cuda_decorrelate_interchannel` for restoring the channel alignments.
  All of these functions are executed in the main stream.

The executing status exported from NVIDIA Visual Profiler (NVVP) is shown in Figure 4.6. The decoding file is the file which generated all the normal decoding experiment files (named as file_split_78.flac) of the normal decoding experiment.

![Figure 4.6: Profiling Result from NVIDIA Visual Profiler](image)

`flac_cuda_find_frames` would only execute once for finding all the frames. As for `flac_cuda_decode_sub_frame` and `flac_cuda_restore_signal` would be executed for several times to decode all the subframes inside a frame. Hence, if we have \( m \) channels, these two kernel would be execute \( m \) times. And if the file is a stereo audio file, `flac_cuda_decorrelate_interchannel` would be executed once for decorrelate the channel assignments into two independent channels.

In this example, the file is a stereo audio file with complex channel assignments inside each frames. After finding the frame positions, the `flac_cuda_decode_sub`
flac_cuda_find_frames and flac_cuda_restore_signal have been executed twice for decoding the data of 2 channels in each frame. flac_cuda_decorrelate_interchannel has been executed for separating the data into 2 independent channels at the end.

From the report of Nvidia Visual Profiler, all the kernel functions reach at least 90% GPU utilization. Figure 4.7, 4.8, 4.9 and 4.10 show the statistics of the Streaming Multiprocessor utilization of all the kernel function from the exported report. The horizontal axis is the index of the Stream Multiprocessor in GTX1080, and the vertical axis is the utilization from 0% to 100%.

![Figure 4.7: GPU Utilization of flac_cuda_find_frames](image)

![Figure 4.8: GPU Utilization of flac_cuda_decode_sub_frame](image)

![Figure 4.9: GPU Utilization of flac_cuda_restore_signal](image)
By reducing the branch constructs, most of these instructions have been transferred as instruction loaded from memory. The memory dependency is the bottleneck of our implementation at the moment. From the profiling result of all the four kernels, most of the samples of the PC sampling is distributed in memory dependency. The profiling from Nvidia Visual Profiler are attached in Appendix B.
Chapter 5

Summary

5.1 Conclusion

A new concurrent decoding framework on CUDA platform to decode FLAC audio is proposed and implemented. As the results show in Section 4.2, the new framework has maximum 5.0 times faster than the CPU implementations. The framework highly depends on the number of CUDA Processors. If the experiments are taken on GPU which has more cores, for instance, GV100 on TITAN V, it would be around 10 times faster than the CPU implementations running on i7-6700K.

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. Due to the lack of branch prediction, multiple methods to reduce branch constructs have been introduced.

Furthermore, the time usage of accessing memory is one of the most important things to be optimized. Two methods of reducing the FIR-Linear Prediction decoding memory access have been introduced to achieve the purpose.

This framework is also designed to easily extend a new format. To add a new decoder, the only thing that needs to be done is implemented the decoder structure for the new format and add it to the decoder list. Thus, all the previous CUDA accelerated implementation could be integrated into this framework.

The new framework is suitable for applications in the following area: real-time audio transfer via FLAC formats, FLAC audio data mining, FLAC audio
data searching. This framework is also a good choice for high-performance audio transcoding applications.

## 5.2 Future Works

There is still room to improve the CUDA implementation. Except the limitations mention in Section 1.3, the following sections are discussed from the performance aspect.

### 5.2.1 Performance Improvement

As mentioned in Section 2.1.4 that rice coding is a branch-bound algorithm and Section 2.2.1.1 that GPU is not good at conditional constructs, the time usage of decoding \texttt{FIXED} and \texttt{LPC} residuals cost a lot. A new method of decoding rice coding with less conditional constructs is a good direction to improve the performance. The following method is worth considering: in the experiment, the rice encoding is not greater than the 64. So the data could be decoded with a 64-thread block. Algorithm 11 shows the method.

\begin{algorithm}[h]
\caption{Concurrent FLAC Unary Decoding}
\begin{algorithmic}
  \Procedure{READ\textsc{Unary}}{r, threadIndex, result}
  \State \texttt{target} $\leftarrow$ 1 $\ll$ (65 $-$ threadIndex)
  \If{$r \& (\sim (\texttt{target} - 1)) = \texttt{target}$}
  \State \texttt{result} $\leftarrow$ threadIndex
  \EndIf
  \EndProcedure
\end{algorithmic}
\end{algorithm}

According to the definition of the unary decoding, only one thread could execute inside the if statement. It is a theoretical $O(1)$ algorithm compare to the original $O(n)$ which $n$ stands for the value of decoding number. But there are two problems need to be solved before using it

1. To decode the unary data, the most straightforward way to embed this function is using the \textit{dynamic parallelism} to execute it. But the time usage of launching kernel function is very expensive. Hence, it cannot be used with current framework.

2. Also, the parameter cannot point to local memory of the function. So it could only use global memory to pass the result. The memory access cost
much enough to cancel out the time saving by the concurrent algorithm
time complexity compared to the serial version with explicit register pa-
rameters.

Most of these residuals are not big. In the experiments, most of these resid-
uals are less than 5. Hence it does need to execute so much branch constructs.
Even this algorithm provides a better time complexity algorithm, but it is hard to
implement on current CUDA platform.

The function table methods described in 3.2.1.2 is now implemented by local
array inside the function. Using Shared Memory is a better choice but the problem
is to prevent multiple written by all threads of the block. This is another thing
needs to be done.

In Section 3.2.2.2, the N-Level Increased Progression is introduced for accelerating
the FIXED subframes. The possibility of applying this progression to LPC subframes
is worthy to consider. The progression might be a special case, there should be a
mathematics model could be used to remove the multiple calculations in decod-
ing LPC subframes.

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. With this method, the FLAC audio decoding could be deployed to
multiple CUDA devices.

Add multiple streams is one way to address the FLAC file size problem. The
current implementation only uses the main stream, hence it needs to copy all the
data to the device and process them at once. Stream allows to asynchronously
copy and process the FLAC audio data. Not all part of the FLAC would be trans-
ferred to the device VRAM to avoid the device memory limitation.
Appendix A

Normal Decoding File Attributes

Table A.1 shows the attributes of the FLAC files using in Section 4.2.3. All the files listed below are 16-bit signed stereo audio sampling at 48,000 Hz.

Table A.1: Test FLAC audio files attributes

<table>
<thead>
<tr>
<th>File</th>
<th>Frames</th>
<th>Duration(s)</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_split_01.flac</td>
<td>703</td>
<td>60</td>
<td>d348987cb82ffe3111e5915e3c8671d</td>
</tr>
<tr>
<td>file_split_02.flac</td>
<td>1,406</td>
<td>120</td>
<td>d2917b085809de06f1b3b392c4fa5c6</td>
</tr>
<tr>
<td>file_split_03.flac</td>
<td>2,109</td>
<td>180</td>
<td>8497e1606bdd672815f257e562839f67</td>
</tr>
<tr>
<td>file_split_04.flac</td>
<td>2,812</td>
<td>240</td>
<td>ce7172ae91509bdf23eb2737717f0b0</td>
</tr>
<tr>
<td>file_split_05.flac</td>
<td>3,515</td>
<td>300</td>
<td>99356d871d5dad6b3e9f6959d8fc146</td>
</tr>
<tr>
<td>file_split_06.flac</td>
<td>4,218</td>
<td>360</td>
<td>ebb840a75bc696731e06703684eafa</td>
</tr>
<tr>
<td>file_split_07.flac</td>
<td>4,921</td>
<td>420</td>
<td>21b219a9ed4e7d7ecf9624a41c7a214e</td>
</tr>
<tr>
<td>file_split_08.flac</td>
<td>5,624</td>
<td>480</td>
<td>29928825397cd328c9d09226fc7ad9</td>
</tr>
<tr>
<td>file_split_09.flac</td>
<td>6,328</td>
<td>540</td>
<td>a4add28f4b32567a9e9f514f23fe92b</td>
</tr>
<tr>
<td>file_split_10.flac</td>
<td>7,031</td>
<td>600</td>
<td>a1b2d5d1d3f39a7f04049c39f607872</td>
</tr>
<tr>
<td>file_split_11.flac</td>
<td>7,734</td>
<td>660</td>
<td>9a9ce62fffd32261e0714d831d2a8d04</td>
</tr>
<tr>
<td>file_split_12.flac</td>
<td>8,437</td>
<td>720</td>
<td>8dd80a5a7c5034264d70bb6920fe054f</td>
</tr>
<tr>
<td>file_split_13.flac</td>
<td>9,140</td>
<td>780</td>
<td>5b9644a4f0fced8f1ed574f9c486d939</td>
</tr>
<tr>
<td>file_split_14.flac</td>
<td>9,843</td>
<td>840</td>
<td>d0bbc1d94944ed6ba7380c96e248371</td>
</tr>
<tr>
<td>file_split_15.flac</td>
<td>10,546</td>
<td>900</td>
<td>c9e3fee105ba043d488511e4e1f96410</td>
</tr>
<tr>
<td>file_split_16.flac</td>
<td>11,249</td>
<td>960</td>
<td>419e451f1979fdd90929b4adf6562f7e</td>
</tr>
<tr>
<td>file_split_17.flac</td>
<td>11,953</td>
<td>1,020</td>
<td>91140b23156685af41f22cee0f6cdfb8</td>
</tr>
<tr>
<td>file_split_18.flac</td>
<td>12,656</td>
<td>1,080</td>
<td>70c05e0bdc40e85fcee5826f009b14c20</td>
</tr>
<tr>
<td>file_split_19.flac</td>
<td>13,359</td>
<td>1,140</td>
<td>cfbb57df01fd2539360cb080f7f57297</td>
</tr>
<tr>
<td>file_split_20.flac</td>
<td>14,062</td>
<td>1,200</td>
<td>95d0d13b335e7e52f4152727c4d362ed</td>
</tr>
<tr>
<td>file_split_21.flac</td>
<td>14,765</td>
<td>1,260</td>
<td>a7780a34aecc41639aa8a95ed2f41924</td>
</tr>
<tr>
<td>file_split_22.flac</td>
<td>15,468</td>
<td>1,320</td>
<td>51</td>
</tr>
</tbody>
</table>
## Appendix A Normal Decoding File Attributes

<table>
<thead>
<tr>
<th>File</th>
<th>Frames</th>
<th>Duration(s)</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>file_split_23.flac</code></td>
<td>16,171</td>
<td>1,380</td>
<td>b1e1b2e5c9bb33547d74c5064eeafeb5</td>
</tr>
<tr>
<td><code>file_split_24.flac</code></td>
<td>16,874</td>
<td>1,440</td>
<td>e761b18dd06f18c3df7a38a58caa41f</td>
</tr>
<tr>
<td><code>file_split_25.flac</code></td>
<td>17,578</td>
<td>1,500</td>
<td>2c3439402c22698e2189318bb59818648</td>
</tr>
<tr>
<td><code>file_split_26.flac</code></td>
<td>18,281</td>
<td>1,560</td>
<td>c8790b7a0af8d4a6990c41874492e14</td>
</tr>
<tr>
<td><code>file_split_27.flac</code></td>
<td>18,984</td>
<td>1,620</td>
<td>1aafe7f8c4a90040df229d668c1600e</td>
</tr>
<tr>
<td><code>file_split_28.flac</code></td>
<td>19,687</td>
<td>1,680</td>
<td>0ae5f859e1441c2dbf19a7348def275c</td>
</tr>
<tr>
<td><code>file_split_29.flac</code></td>
<td>20,390</td>
<td>1,740</td>
<td>7a18c1bcb000279bc1ef1381d3e63228</td>
</tr>
<tr>
<td><code>file_split_30.flac</code></td>
<td>21,093</td>
<td>1,800</td>
<td>e23173aaecfa79572178facb59f559cc</td>
</tr>
<tr>
<td><code>file_split_31.flac</code></td>
<td>21,796</td>
<td>1,860</td>
<td>6b99275c6b450963c77790d3d5e06</td>
</tr>
<tr>
<td><code>file_split_32.flac</code></td>
<td>22,499</td>
<td>1,920</td>
<td>ee89d49c8cb29bb07739e937ff69f8e1</td>
</tr>
<tr>
<td><code>file_split_33.flac</code></td>
<td>23,203</td>
<td>1,980</td>
<td>106d21a0c94252d39573370e22fb5</td>
</tr>
<tr>
<td><code>file_split_34.flac</code></td>
<td>23,906</td>
<td>2,040</td>
<td>3189063906a946cb1cd435cc5efec</td>
</tr>
<tr>
<td><code>file_split_35.flac</code></td>
<td>24,609</td>
<td>2,100</td>
<td>3884216775871302ceaf9299fba70a</td>
</tr>
<tr>
<td><code>file_split_36.flac</code></td>
<td>25,312</td>
<td>2,160</td>
<td>e2ec1d191547e34ec4eb4f9da773026</td>
</tr>
<tr>
<td><code>file_split_37.flac</code></td>
<td>26,015</td>
<td>2,220</td>
<td>f1b7cdd1c6fcd32798a743fd7a1d9084f</td>
</tr>
<tr>
<td><code>file_split_38.flac</code></td>
<td>26,718</td>
<td>2,280</td>
<td>db8a2c942430dccc544cc0197d1c09</td>
</tr>
<tr>
<td><code>file_split_39.flac</code></td>
<td>27,421</td>
<td>2,340</td>
<td>c0b488b0030432439c4838d5560cf2c0</td>
</tr>
<tr>
<td><code>file_split_40.flac</code></td>
<td>28,124</td>
<td>2,400</td>
<td>59b2d10de78c6667eaf4db83b87bfa22</td>
</tr>
<tr>
<td><code>file_split_41.flac</code></td>
<td>28,828</td>
<td>2,460</td>
<td>2dbc255cd04e9c103111092a5121e57</td>
</tr>
<tr>
<td><code>file_split_42.flac</code></td>
<td>29,531</td>
<td>2,520</td>
<td>7b5ca6dc2c2f37e30a12e62ee20a44</td>
</tr>
<tr>
<td><code>file_split_43.flac</code></td>
<td>30,234</td>
<td>2,580</td>
<td>1b3cc552a905f92a5572d54a8ca48f4</td>
</tr>
<tr>
<td><code>file_split_44.flac</code></td>
<td>30,937</td>
<td>2,640</td>
<td>5f108331a9ff84326716f797ef41d5c</td>
</tr>
<tr>
<td><code>file_split_45.flac</code></td>
<td>31,640</td>
<td>2,700</td>
<td>ca366e8d04eb3ed46686b9630ce87a</td>
</tr>
<tr>
<td><code>file_split_46.flac</code></td>
<td>32,343</td>
<td>2,760</td>
<td>7ab11ca543d2ff5c642365816c5da43d</td>
</tr>
<tr>
<td><code>file_split_47.flac</code></td>
<td>33,046</td>
<td>2,820</td>
<td>18bc1ac7c5948d767f56196e431b691</td>
</tr>
<tr>
<td><code>file_split_48.flac</code></td>
<td>33,749</td>
<td>2,880</td>
<td>0bf524d23104c47967edfb45bcce2960</td>
</tr>
<tr>
<td><code>file_split_49.flac</code></td>
<td>34,453</td>
<td>2,940</td>
<td>2cd5edc9b9d374892b920c0017ca42609</td>
</tr>
<tr>
<td><code>file_split_50.flac</code></td>
<td>35,156</td>
<td>3,000</td>
<td>0ee7671ca2961e1af8f659f103bf8c4cb</td>
</tr>
<tr>
<td><code>file_split_51.flac</code></td>
<td>35,859</td>
<td>3,060</td>
<td>acf20d91f2b0b918e4e638531f45f9</td>
</tr>
<tr>
<td><code>file_split_52.flac</code></td>
<td>36,562</td>
<td>3,120</td>
<td>4496468c1518652fc884b4eb1010127c</td>
</tr>
<tr>
<td><code>file_split_53.flac</code></td>
<td>37,265</td>
<td>3,180</td>
<td>667b0f8b97e62941cb3f595542c3fbf</td>
</tr>
<tr>
<td><code>file_split_54.flac</code></td>
<td>37,968</td>
<td>3,240</td>
<td>9bf523df8eab9ff60cf43ff59f5f9</td>
</tr>
<tr>
<td><code>file_split_55.flac</code></td>
<td>38,671</td>
<td>3,300</td>
<td>6473df8418e136b3a2b6d2c2fc3314</td>
</tr>
<tr>
<td><code>file_split_56.flac</code></td>
<td>39,374</td>
<td>3,360</td>
<td>e979ca7d007e3989c25a962061ea2ee8</td>
</tr>
<tr>
<td><code>file_split_57.flac</code></td>
<td>40,078</td>
<td>3,420</td>
<td>0be5663db22fd16e82358910d91570</td>
</tr>
<tr>
<td><code>file_split_58.flac</code></td>
<td>40,781</td>
<td>3,480</td>
<td>8873e151ec7bc26fedf8d3d15252700</td>
</tr>
</tbody>
</table>
### Appendix A Normal Decoding File Attributes

<table>
<thead>
<tr>
<th>File</th>
<th>Frames</th>
<th>Duration(s)</th>
<th>MD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_split_59.flac</td>
<td>41,484</td>
<td>3,540</td>
<td>b0ff41fcd4f2cc8fe0c5ec7bdb10ab0</td>
</tr>
<tr>
<td>file_split_60.flac</td>
<td>42,187</td>
<td>3,600</td>
<td>0cf8b92c45cfcd618d4295a16fa88c82</td>
</tr>
<tr>
<td>file_split_61.flac</td>
<td>42,890</td>
<td>3,660</td>
<td>4f615797ae4c8da864ba3c82f0223e76</td>
</tr>
<tr>
<td>file_split_62.flac</td>
<td>43,593</td>
<td>3,720</td>
<td>507a7662a6782795295578d238892776</td>
</tr>
<tr>
<td>file_split_63.flac</td>
<td>44,296</td>
<td>3,780</td>
<td>bc62290dc0571299cc66c88405e9847a</td>
</tr>
<tr>
<td>file_split_64.flac</td>
<td>44,999</td>
<td>3,840</td>
<td>2e7e9a009451f26eeec10c49022a9942c</td>
</tr>
<tr>
<td>file_split_65.flac</td>
<td>45,703</td>
<td>3,900</td>
<td>c9d777b1e89272d7702f8636551be7fa</td>
</tr>
<tr>
<td>file_split_66.flac</td>
<td>46,406</td>
<td>3,960</td>
<td>c90b6e564917bd02a793b39ee310189d</td>
</tr>
<tr>
<td>file_split_67.flac</td>
<td>47,109</td>
<td>4,020</td>
<td>6a498c839d611eca8397f9f25f5c9952</td>
</tr>
<tr>
<td>file_split_68.flac</td>
<td>47,812</td>
<td>4,080</td>
<td>c92cddca2e99031005f569b930b3192b</td>
</tr>
<tr>
<td>file_split_69.flac</td>
<td>48,515</td>
<td>4,140</td>
<td>e03dd8ef7390a5823ec5b9c0fa41c921</td>
</tr>
<tr>
<td>file_split_70.flac</td>
<td>49,218</td>
<td>4,200</td>
<td>e93b028d9874f413a3f69499154c914</td>
</tr>
<tr>
<td>file_split_71.flac</td>
<td>49,921</td>
<td>4,260</td>
<td>566f4cf436494f961a7f753cadf2d510</td>
</tr>
<tr>
<td>file_split_72.flac</td>
<td>50,624</td>
<td>4,320</td>
<td>e4982a639c877a9a1f1ea666bc41d89</td>
</tr>
<tr>
<td>file_split_73.flac</td>
<td>51,328</td>
<td>4,380</td>
<td>f237eb669b64ebc174938f2b94a369f</td>
</tr>
<tr>
<td>file_split_74.flac</td>
<td>52,031</td>
<td>4,440</td>
<td>d02129d8c539b2e75aff401ec84e94b8</td>
</tr>
<tr>
<td>file_split_75.flac</td>
<td>52,734</td>
<td>4,500</td>
<td>cd8bb7961cb69bc5931c53f4b061309</td>
</tr>
<tr>
<td>file_split_76.flac</td>
<td>53,437</td>
<td>4,560</td>
<td>61388bd2ce052c8133df0e7ee0f8fc86</td>
</tr>
<tr>
<td>file_split_77.flac</td>
<td>54,140</td>
<td>4,620</td>
<td>a159b26ee689e5fd016683bc04e55b8</td>
</tr>
</tbody>
</table>
Appendix B

Kernel Function Profiling

The following report are the profiling result for the four kernel functions used in GPUARKU

- flac_cuda_find_frames
- flac_cuda_decode_sub_frame
- flac_cuda_restore_signal
- flac_cuda_decorrelate_interchannel

All the analysis reports are done on the same machine described in Section 4.1. The file is the same as the file used in end of Section 4.2.3 for the profiling result and GPU utilization, which is the file_split_78.flac.

Please notice:

- These contents in left of this chapter are automatically generated and exported by NVIDIA Visual Profiler, including all the texts and figures.
- All the texts and figures attached following are copied from the original report without any modification.
- Not all the sections of the original report has been attached.
- The format has been modified to make the content to be cleared.
### Analysis Report of Resources

<table>
<thead>
<tr>
<th>Duration</th>
<th>flac_cuda_find_frames:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>136.42615 ms (136,426,155 ns)</td>
</tr>
<tr>
<td>flac_cuda_decode_sub_frame:</td>
<td>227.7849 ms (227,784,903 ns)</td>
</tr>
<tr>
<td>flac_cuda_restore_signal:</td>
<td>233.46405 ms (233,464,046 ns)</td>
</tr>
<tr>
<td>flac_cuda_decorrelate_interchannel:</td>
<td>289.72127 ms (289,721,274 ns)</td>
</tr>
</tbody>
</table>

| Grid Size                     | [1714,1,1]            |
| Block Size                    | [32,1,1]              |

| Registers/Thread              | flac_cuda_find_frames: |
|                               | 29                     |
| flac_cuda_decode_sub_frame:   | 44                     |
| flac_cuda_restore_signal:     | 37                     |
| flac_cuda_decorrelate_interchannel: | 24                     |

| Shared Memory/Block           | flac_cuda_find_frames: |
|                               | 512 B                  |
| flac_cuda_decode_sub_frame:   | 0 B                    |
| flac_cuda_restore_signal:     | 4 KiB                  |
| flac_cuda_decorrelate_interchannel: | 0 B                  |

| Shared Memory Requested       | 96 KiB                 |
| Shared Memory Executed        | 96 KiB                 |
| Shared Memory Bank Size       | 4 B                    |
## Device Information

### [0] GeForce GTX 1080

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU UUID</td>
<td>GPU-76a1a9bd-0a6b-8700-5923-828dc52f24bb</td>
</tr>
<tr>
<td>Compute Capability</td>
<td>6.1</td>
</tr>
<tr>
<td>Max. Threads per Block</td>
<td>1024</td>
</tr>
<tr>
<td>Max. Threads per Multiprocessor</td>
<td>2048</td>
</tr>
<tr>
<td>Max. Shared Memory per Block</td>
<td>48 KiB</td>
</tr>
<tr>
<td>Max. Shared Memory per Multiprocessor</td>
<td>96 KiB</td>
</tr>
<tr>
<td>Max. Registers per Block</td>
<td>65536</td>
</tr>
<tr>
<td>Max. Registers per Multiprocessor</td>
<td>65536</td>
</tr>
<tr>
<td>Max. Grid Dimensions</td>
<td>[2147483647, 65535, 65535]</td>
</tr>
<tr>
<td>Max. Block Dimensions</td>
<td>[1024, 1024, 64]</td>
</tr>
<tr>
<td>Max. Warps per Multiprocessor</td>
<td>64</td>
</tr>
<tr>
<td>Max. Blocks per Multiprocessor</td>
<td>32</td>
</tr>
<tr>
<td>Half Precision FLOP/s</td>
<td>73.4 GigaFLOP/s</td>
</tr>
<tr>
<td>Single Precision FLOP/s</td>
<td>9.395 TeraFLOP/s</td>
</tr>
<tr>
<td>Double Precision FLOP/s</td>
<td>293.6 GigaFLOP/s</td>
</tr>
<tr>
<td>Number of Multiprocessors</td>
<td>20</td>
</tr>
<tr>
<td>Multiprocessor Clock Rate</td>
<td>1.835 GHz</td>
</tr>
<tr>
<td>Concurrent Kernel</td>
<td>true</td>
</tr>
<tr>
<td>Max IPC</td>
<td>6</td>
</tr>
<tr>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Global Memory Bandwidth</td>
<td>320.32 GB/s</td>
</tr>
<tr>
<td>Global Memory Size</td>
<td>7.923 GiB</td>
</tr>
<tr>
<td>Constant Memory Size</td>
<td>64 KiB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>2 MiB</td>
</tr>
<tr>
<td>Memcpy Engines</td>
<td>2</td>
</tr>
<tr>
<td>PCIe Generation</td>
<td>3</td>
</tr>
<tr>
<td>PCIe Link Rate</td>
<td>8 Gbit/s</td>
</tr>
<tr>
<td>PCIe Link Width</td>
<td>16</td>
</tr>
</tbody>
</table>
Appendix B Kernel Function Profiling

Analysis Report

`flac_cuda_find_frames(unsigned char*, unsigned long, unsigned long*, CudaFrameDecode*, unsigned long, unsigned long, unsigned int, unsigned int, unsigned int, unsigned char, unsigned char)`

1. Compute, Bandwidth, or Latency Bound

1.1. Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1080". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

2. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The results below indicate that the GPU does not have enough work because instruction execution is stalling excessively.

2.1. Kernel Profile - PC Sampling

The Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Using this information you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not.

Instruction Issued - Warp was issued
Instruction Fetch - The next assembly instruction has not yet been fetched.
Execution Dependency - An input required by the instruction is not yet available.
Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.
Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.
Texture - The texture sub-system is fully utilized or has too many outstanding requests.
Synchronization - The warp is blocked at a __syncthreads() call.
Constant - A constant load is blocked due to a miss in the constants cache.
Pipe Busy - The compute resource(s) required by the instruction is not yet available.
Appendix B Kernel Function Profiling

Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.

Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.

Other - The warp is blocked for an uncommon reason.

Sleeping - The warp is blocked, yielded or sleeping.

Examine portions of the kernel that have high number of samples to know where the maximum time was spent and observe the latency reasons for those samples to identify optimization opportunities.

<table>
<thead>
<tr>
<th>Cuda Functions</th>
<th>Sample Count</th>
<th>% of Kernel Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>flac_cuda_find_frames(unsigned char*, unsigned long, unsigned long*, CudaFrameDecode*, unsigned long, unsigned long, unsigned int, unsigned int, unsigned char, unsigned char)</td>
<td>18187829</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Source Files:

/home/u5870415/GPUraku/src/.formats/flac/cuda/flac_cuda_frame.cup
Analysis Report

flac_cuda_decode_sub_frame(CudaFrameDecode*, unsigned char, unsigned long, unsigned long, unsigned int, CudaSubFrameType*, int*)

1. Compute, Bandwidth, or Latency Bound

1.1. Kernel Performance Is Bound By Instruction And Memory Latency
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1080". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

2. Instruction and Memory Latency
Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The results below indicate that the GPU does not have enough work because instruction execution is stalling excessively.

2.1. Kernel Profile - PC Sampling
The Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Using this information you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not.

Instruction Issued - Warp was issued
Instruction Fetch - The next assembly instruction has not yet been fetched.
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.
Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.
Texture - The texture sub-system is fully utilized or has too many outstanding requests.
Synchronization - The warp is blocked at a __syncthreads() call.
Constant - A constant load is blocked due to a miss in the constants cache.
Pipe Busy - The compute resource(s) required by the instruction is not yet available.
Memory Throttle - Large number of pending memory operations prevent further
forward progress. These can be reduced by combining several memory transac-
tions into one.
Not Selected - Warp was ready to issue, but some other warp issued instead. You
may be able to sacrifice occupancy without impacting latency hiding and doing
so may help improve cache hit rates.
Other - The warp is blocked for an uncommon reason.
Sleeping - The warp is blocked, yielded or sleeping.
Examine portions of the kernel that have high number of samples to know where the
maximum time was spent and observe the latency reasons for those samples to identify
optimization opportunities.

<table>
<thead>
<tr>
<th>Cuda Functions</th>
<th>Sample Count</th>
<th>% of Kernel Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>flac_cuda_decode_sub_frame(CudaFrameDecode*, unsigned char, unsigned long, unsigned long, unsigned int, CudaSubFrameType*, int*)</td>
<td>33139299</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Source Files:

/home/u5870415/GPUraku/src/./formats/flac/cuda/flac_cuda_bitstream.cup
/home/u5870415/GPUraku/src/./formats/flac/cuda/flac_cuda_subframe.cup
Analysis Report

`flac_cuda_restore_signal(CudaFrameDecode*, CudaSubFrameType*, unsigned int)`

1. Compute, Bandwidth, or Latency Bound

1.1. Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "GeForce GTX 1080". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

2. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The results below indicate that the GPU does not have enough work because instruction execution is stalling excessively.

2.1. Kernel Profile - PC Sampling

The Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Using this information you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not.

- **Instruction Issued** - Warp was issued
- **Instruction Fetch** - The next assembly instruction has not yet been fetched.
- **Execution Dependency** - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.
- **Memory Dependency** - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.
- **Texture** - The texture sub-system is fully utilized or has too many outstanding requests.
- **Synchronization** - The warp is blocked at a `__syncthreads()` call.
- **Constant** - A constant load is blocked due to a miss in the constants cache.
- **Pipe Busy** - The compute resource(s) required by the instruction is not yet available.
- **Memory Throttle** - Large number of pending memory operations prevent further
forward progress. These can be reduced by combining several memory transactions into one.
Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.
Other - The warp is blocked for an uncommon reason.
Sleeping - The warp is blocked, yielded or sleeping.

Examine portions of the kernel that have high number of samples to know where the maximum time was spent and observe the latency reasons for those samples to identify optimization opportunities.

<table>
<thead>
<tr>
<th>Cuda Functions</th>
<th>Sample Count</th>
<th>% of Kernel Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>flac_cuda_restore_signal(</td>
<td>34173625</td>
<td>100.0%</td>
</tr>
<tr>
<td>CudaFrameDecode*,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CudaSubFrameType*, unsigned int)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source Files:

/home/u5870415/GPUraku/src/. ./formats/flac/cuda/flac_cuda_channel.cup
Appendix B Kernel Function Profiling

Analysis Report

flac_cuda_decorrelate_interchannel(CudaFrameDecode*, unsigned int)

1. Compute, Bandwidth, or Latency Bound

1.1. Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of “GeForce GTX 1080”. These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.

2. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The results below indicate that the GPU does not have enough work because instruction execution is stalling excessively.

2.1. Kernel Profile - PC Sampling

The Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Using this information you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not.

Instruction Issued - Warp was issued
Instruction Fetch - The next assembly instruction has not yet been fetched.
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.
Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.
Texture - The texture sub-system is fully utilized or has too many outstanding requests.
Synchronization - The warp is blocked at a __syncthreads() call.
Constant - A constant load is blocked due to a miss in the constants cache.
Pipe Busy - The compute resource(s) required by the instruction is not yet available.
Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transac-
tions into one.
Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.
Other - The warp is blocked for an uncommon reason.
Sleeping - The warp is blocked, yielded or sleeping.
Examine portions of the kernel that have high number of samples to know where the maximum time was spent and observe the latency reasons for those samples to identify optimization opportunities.

<table>
<thead>
<tr>
<th>Cuda Functions</th>
<th>Sample Count</th>
<th>% of Kernel Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>flac_cuda_decorrelate_interchannel(CudaFrameDecode*, unsigned int)</td>
<td>44296565</td>
<td>100.0%</td>
</tr>
</tbody>
</table>

Source Files:

```
/home/u5870415/GPUraku/src/.formats/flac/cuda/flac_cuda_channel_30.cup
```
References


[34] Julien Demouth, CUDA Optimization with NVIDIA Tools, Nvidia Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, 2014.


