Title	Fast Search of Audio Fingerprint using K40 GPGPU
Author(s)	Nguyen, Mau Toan
Citation	
Issue Date	2016-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/13741
Rights	
Description	Supervisor:井口　寧，情報科学研究科，修士

Fast Search of Audio Fingerprint using K40 GPGPU

Nguyen Mau Toan

School of Information Science
Japan Advanced Institute of Science and Technology
September, 2016

Master's Thesis

Fast Search of Audio Fingerprint using K40 GPGPU

1410215 Nguyen Mau Toan

Supervisor :	Professor Yasushi Inoguchi
Main Examiner :	Professor Yasushi Inoguchi
Examiners :	Professor Mineo Kaneko
	Associate Professor Kiyofumi Tanaka

School of Information Science
Japan Advanced Institute of Science and Technology

August, 2016

Contents

This dissertation was prepared according to the curriculum for the Collaborative Education Program organized by Japan Advanced Institute of Science and Vietnam National University, Ho Chi Minh City (VNU-HCM).

List of Figures

List of Tables

Abstract

Nowadays, there are millions of audio and video contents uploaded to the Internet, so the searching speed and database organization are the problems for the audio management system. Audio fingerprint is the digital fingerprint that can help to identify the audio content. With the advantages of audio fingerprint, we can reduce the size of data to hundreds of times less than storing original audio raw data. And with audio fingerprint, we have a standard format that supports to compare or structuralize the database. In this thesis, we propose a new hierarchy searching system that can detect the meta information for fingerprint in real time by using the advantages of K-modes and Locality Sensitive Hashing (LSH). The K-modes is used as Level 1 in our method and works in CPU. K-modes supports in clustering the big database into sub-databases that can store to GPGPU devices. In searching step, K-modes is responsible for finding the nearest centroid of every query and send this query to suitable GPGPU device. LSH will handle the data structure of GPGPU devices' sub-database and respond for management the kernel that is compatible with parallel in single GPGPU. Our method can combine the advantages of both CPU and GPGPUs by putting together in the same computer system. With the power of multiple GPGPU devices, we can obtain the meta information for a query within 2 milliseconds for 10 million songs' database.

Acknowledgment

I would first like to thank my thesis advisor Professor Yasushi Inoguchi of the Research Center for Advanced Computing Infrastructure at JAIST. Prof. Inoguchi was always ready to help me whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this thesis to be my own work but steered me in the right the direction whenever he thought I needed it.

I would also like to thank my second supervisor Professor Mineo Kaneko who were involved in the validation survey for this research project. Without his passionate participation and input, the validation survey could not have been successfully conducted.
I also wish to express my honor to my advisor for minor research Professor Ryuuhei Uehara for his valuable advice and suggestions.

I would also like to acknowledge all members of Inoguchi-laboratory of the Information Science School at JAIST for well supporting and giving good comments during the time I did my research.

I am eagerly acknowledge the JAIST-scholarship for the joint program with Vietnam National University, Ho Chi Minh City (VNU-HCM), which makes my study and life feasible in Japan, and also the Japan Student Services Organization (JASSO)'s scholarship for supporting me the living expenses for 6-month study in Japan.

Finally, I must express my very profound gratitude to my parents and my brother for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Chapter 1

Introduction

Audio Fingerprint is a technique to standardize the music based on the content of music that can summarize an audio recording. The main application of audio fingerprint is that it can extract the audio or a short-clip of audio and represent it into a feature-base vector. After that, the fingerprints can store in database with labels. Then, people can identify the unlabeled fingerprint by finding the most similar labeled-fingerprint in the database and set its label to the query input [3].

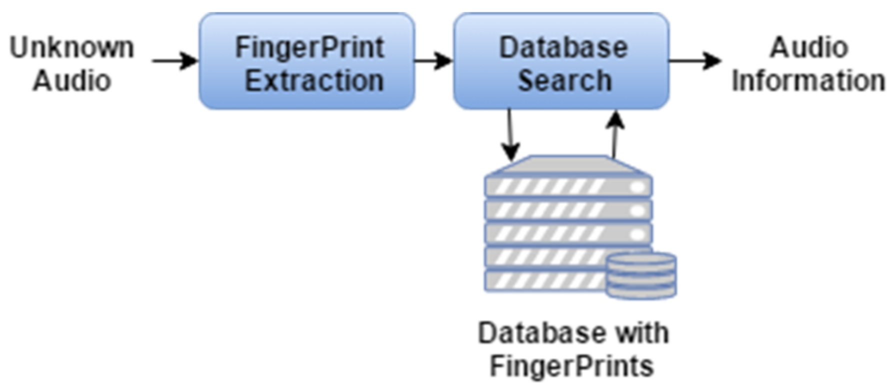


Figure 1.1: Audio Fingerprint Searching Problem

In Figure 2.1 There are two main problems in this field: fingerprint extraction and fingerprint searching [7]. The accuracy of whole system depend of the algorithm of audio extraction and searching algorithm. Both algorithm are important so with a bad algorithm of audio fingerprint extraction we can not build a good searching system using the fingerprints extracted by a bad algorithm.

In this thesis, we focus on building a massively parallel searching system that can work on multiple GPGPUs for handling the database with 10 million songs. Settling this problem, we need K-modes - an effective algorithm for clustering the database to divide the whole database into several sub-databases and we also need LSH that can support approximate searching of the nearest neighbors for multiple queries in parallel.

Our searching system includes two main stages 'Choosing the device' and 'Parallel hashing searching' which we call 'level 1' and 'level 2', respectively. Level 1 works in CPU with the use of K-modes to help detect the most potential device that can store current query. In addition, the LSH will be used in level 2 in each GPGPU device for the high performance in parallel searching of multiple queries.

1.1 Background

In this thesis, I inherit the HiFP2.0 fingerprint extraction algorithm for getting the audio features. The feature of HiFP2.0 can achieve many advantages for storing and hashing by decoding with small and standard binary vector [1]. For level 1 working on CPU, we choose to use K-modes to handle all processing on clustering and detecting the nearest cluster. K-modes is a extension algorithm of K-means, it focuses on handling the category data such as binary vector [16]. The architecture of CUDA platform is also considered for optimizing the parallel processing in GPGPU device [26]. For Level 2 on GPGPU, LSH is the main method for hashing the database to detect the most suitable bucket. With the stable database structure stored in GPGPU devices, the searching job can be easily paralleled by query threads [6, 21].

1.2 Problem Statement

The database F is already known with a large number of points (audio fingerprint) n: $F_1, F_2, F_3, ..., F_n$. The meta information for all audio fingerprints in the database is also given. Our target is to give the meta information for multiple audio fingerprints queries $F_{q1}, F_{p2}, F_{q3}, ..., q_T$ with unknown meta information by searching the nearest audio fingerprint in the database and use its meta information for the corresponding query. The algorithm for audio fingerprint extraction and the standard format for audio fingerprint are given. The goal of this research is to find the nearest audio fingerprint for every query in limited time (10 milliseconds) and support for parallel queries using multiple GPGPU devices.

There are two big problems to be solved in this research. Firstly, it is the synchronous of threads in CPU because in CPU the threads are using the same memory and need a resource from other threads. Secondly, the ε-NNS problem will be considered for level 2 for getting the higher result for every query.

1.3 Research Objective

Audio fingerprint extraction and audio fingerprint searching strategy are two main stages of identifying information for unknown audio waveform problem when database and meta data are known. We inherit the goodness of HiFP2.0 for extracting the audio fingerprint. In this thesis, we focus on proposing new computer structure using multiple GPGPU devices for intelligent storage fingerprints and supporting for fast multiple searching fingerprint queries in parallel.

There are two aspects in the objective of this research. Firstly, the accuracy of the whole system should be considered. For the real data on the Internet, queries can be transformed by many ways by noise, changing of volume/pitch or mixing with other sounds. Our method should have reliability for giving the meta information for every query by comparing with the original method of HiFP2.0 and other researchers working in this field. Secondly, the searching speed of every single audio fingerprint is requisite for real world data. For 300 hours of videos uploaded to the Youtube every minute, our system should be able to handle 6000 fingerprints within one second.

1.4 Approach

For the parallelism searching, we need to choose an algorithm that can run in parallel when using the same resources and the device supporting to run multiple kernels at the same time. We choose to use GPGPU having a great number of cores that can run thousands of threads at the same time with a large amount of memory size per device. Now that the size of single GPGPU memory is not enough for real data, we propose to use multiple GPGPU devices that can storw the parts in the whole database. To do that, the compatibility of managing and tasking CPU will be used to manage the jobs of GPGPUs. For clustering the database of managing the queries, we choose to use K-modes as the main algorithm that can work in CPU. The LSH algorithm well support parallelism in multiple threads in GPGPU.

1.5 Scope of the Thesis

Our main target is to focus on searching speed and ability of the parallel system for the problem of the nearest audio fingerprint searching in big database system. The organization supports for fast search only with storing of hash value and corresponding audio fingerprints. Audio fingerprint is based on the binary array with containing the content of source audio and supporting for specific hash functions. Due to the requirements for big database, almost audio fingerprints in tested database are the random generation followed by the standardized format of audio fingerprint extracted by HiFP2.0, but the accuracy of the whole system with random data is as same as the accuracy of system using small real database.

The testing queries for the testing system are based on distorting audio fingerprints from the original audio. The size of an audio fingerprint is limited by 4096 bits and extracted from first 2.97 seconds of an audio song.

1.6 Organization of the Thesis

In this Chapter, we provide the research backgrounds used in my thesis and also our research objective. In chapter 3, we show two types of research related to our research in order to understand the advantages and disadvantages of methods and facilitate comparing their approachs with our method. In chapter 2, we show the research backgrounds and parallel architecture that we use in this research. Chapter 4 will show our strategy of storing and searching in single GPGPU, which is the key factor for massively parallel for multiple GPGPUs. In Chapter 4, we also show the result of searching unbder comparison with previous works using FPGA. Chapter 5 will be our main proposed method in this thesis, it will inherit the advantages of the method in Chapter 4 for building a new massively parallel system using K-modes and LSH. Chapter 6 will show our result to compare with the research objective and related researches also. Finally, we will have several discussions about our research in Chapter 7 based on our result and objective. Chapter 7 also shows our current problem and the solution for the future works in this thesis.

Chapter 2

Research Background

2.1 GPU, GPGPU Architecture and CUDA (Compute Unified Device Architecture)

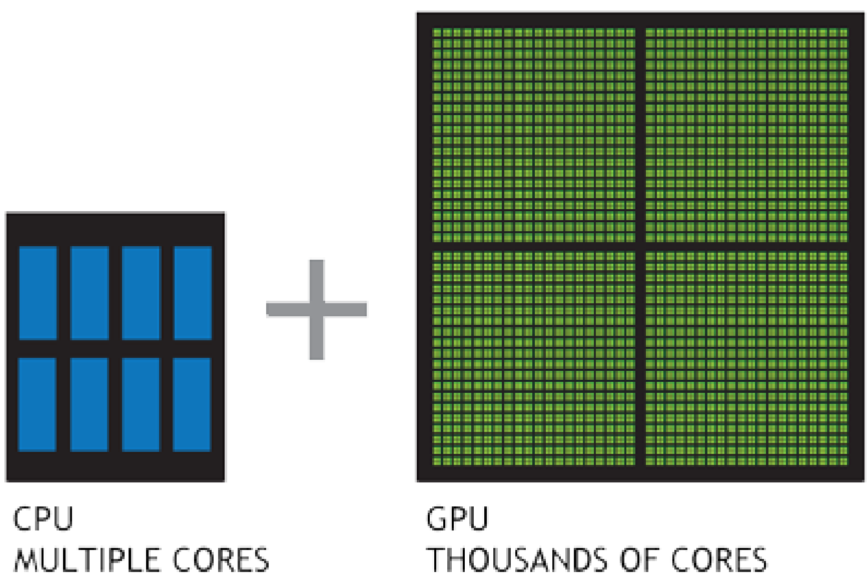


Figure 2.1: Core comparison between CPU and GPU [29, p.2]

GPU (Graphics Control Unit) is an electronic circuit designed for manipulating the frame buffer of images used for display's output in personal computer, mobile phone, game console or embedded system. GPU has the power of thousands of cores working in parallel for handling multiple fractal of graphic images. In Figure 2.1, when comparing with the CPU, GPU has overwhelming numbers of cores supporting for running kernels in parallel. Although CPU has less core than GPU but it can work with general task with different kernels. Basically, GPU has processing cores and memory for graphics purpose

Table 2.1: Comparison between CPU and GPU [29, p.3]

CPU	GPU
fast caches (great for data reuse)	Lots of math units
Good Fine branching granularity	Fast access to onboard memory
Lots of different processes/threads	Run a program on each fragment/vertex
High performance on a single thread of execution	High throughput on parallel tasks
Good for task parallelism	Good for data parallelism
high performance on sequential codes	optimised for higher arithmetic intensity for parallel nature

including vertex processors and fragment processors. Figure 2.2 shows that the input for graphics rendering is the raw vertices and primitives. First of all vertex processor will deploy the raw data into 3D environment with triangles with vertices. Following is rasterizer step that will put vertices for filling the current triangles for creating meshes for the whole environment. Then, the fragment processor will be responsible for coloring the meshes by the shader primitives and material using the image textures from texture memory from GPU. Finally, it needs merging the output for converting to 2D array that fits with the monitor's resolution.

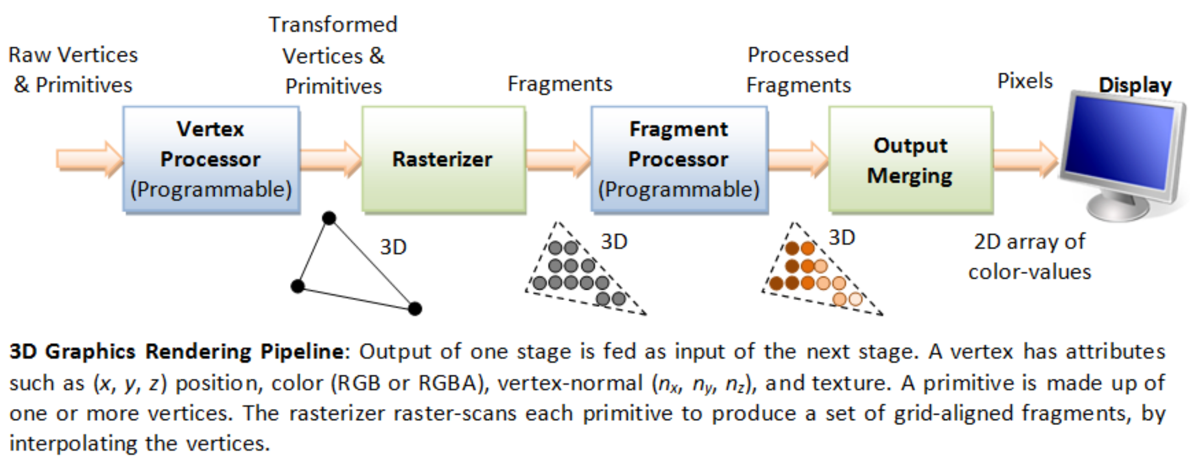


Figure 2.2: Phases of the GPU graphics processing [25]

GPU has focused on parallel for handling graphics kernel only for years. In Table 2.1, it is very clear that GPU has potential strength in parallel processing. GPU architectures are ALU-heavy and contain multiple-vertex & fragment pipelines for solving similar jobs in parallel. Besides, there are many problems having multiple similar works like graphics processing in computer science. Using the GPU for processing nongraphical entities is known as the General Purpose GPU or GPGPU, this will take advantages of GPU in parallelism but we should accept the trade-off of lacking multitasking, otherwise can not handle shift bits, bitwise, integer data operands. For GPGPU, researchers now can access the texture memory of GPU and put or removing the non-graphics data. Besides, researcher can change the graphics kernel to general purpose using non-graphics data and come with different output with non-image data.

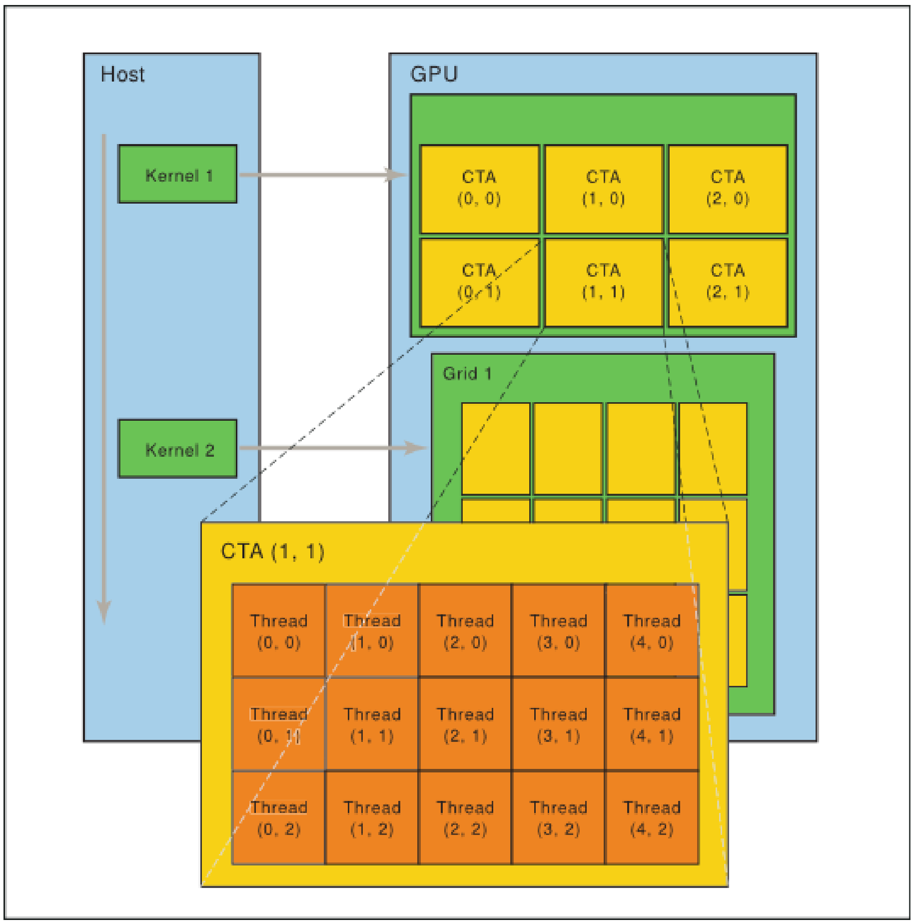


Figure 2.3: Thread Batching for CUDA's flow [26]

Compute Unified Device Architecture (Cuda) is NVIDIA's architecture for GPGPU cards. It supports for managing the GPGPU organizational structure by itself. Programmer can access and handle the NVIDIA's GPGPU by CUDA C/C++ or CUDA Fortran, those are the extensions of C/C++ and Fortran programming language [25]. In Figure 2.4, every kernel of CUDA are handled by a grid with blocks with 2D addressing $(0, 0)$ to $(block^x_{max}, block^y_{max})$. And every block also has a 3D structure for threads indexing from $(0, 0, 0)$ to $(thread^x_{max}, thread^y_{max}, thread^z_{max})$. Every threads in grid are deployed by one kernel sent from host. Threads are only distinguished by thread index tid including $(tid.x, tid.y, tid.z)$. The indexes of thread are the keys for variability purpose of CUDA [26].

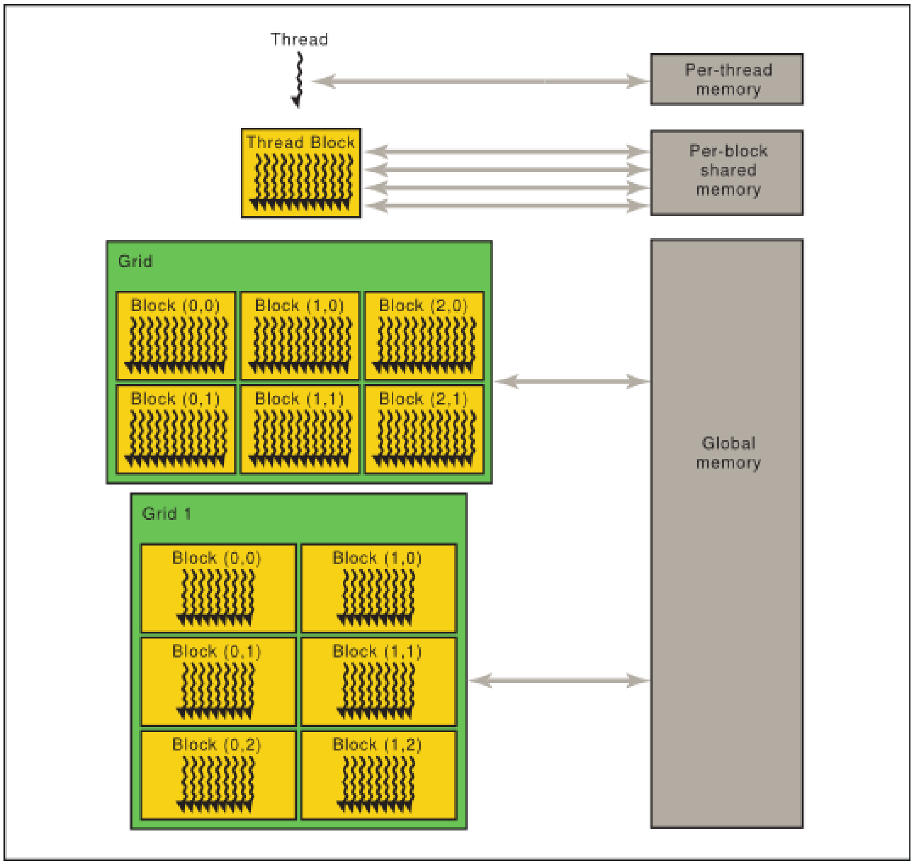


Figure 2.4: Memory Hierarchy in CUDA device [26]

Another important feature of CUDA is memory architecture. Figure 2.4 proves that CUDA device has different structure with main memory. Each thread has a small memory for its processing flow such as indexing numbers or temporary variables. Global memory is very similar to main memory, Global memory can be accessed by every thread and block in grid. Basically, programmer can access this by transferring input data from main memory for storing for output data to copy to main memory. Per-block shared memory is a special feature in CUDA architecture, it helps to gain performance by using the same resource of parallel threads in the same block. Beside that, CUDA also has Texture memory (read only) for cache optimized 2D spatial access pattern and a high speed Constant memory for storing data accessed with high frequency [26].

With their advantages of parallelism capability and easily availability, CUDA represents ability of parallel processing research. It becomes the preferred choices for researchers who are working with parallel processing and massively parallel.

2.2 Audio Fingerprint

2.2.1 Introduction of Audio Fingerprint

Audio fingerprint is a feature with content-based extracted from the audio/song waveform that can standardize the content of audio/song recording. Audio fingerprint can help to compare the similarities and differences of two songs. Beside that, audio fingerprint can support for storing normalized format/structure data with the size far smaller than the original audio waveform. For the fingerprint database system, fingerprints are stored alongside with its meta information so as to gain the information retrieval system.

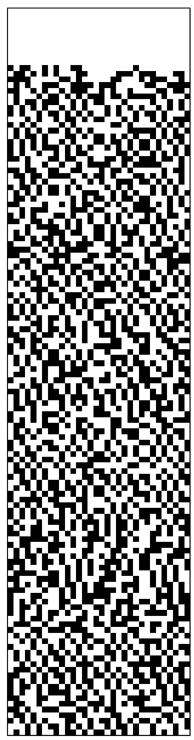


Figure 2.5: A example of 4096-bits fingerprint extracted by HiFP2.0 fingerprint extraction

In figure 2.5, we can see a bits-sequence of fingerprint extracted by HiFP2.0 fingerprint extraction. This fingerprint is content-based extraction algorithm for the first 2.97 seconds of a song. We can see all zeros in the first part on this audio fingerprint, which indicates the beginning of the original song is empty [1].

In technical way, audio fingerprint represents the information corresponding segment of original audio content. So, the similarity of audio can be showed by the distance of fingerprints. For our problem, due to the transformation of source audio content, the audio fingerprint can be different by the distance between of source audio's fingerprint and transformed audio's tends to be closer to each other. For this problem, we can metaphor it a problem for finding the nearest fingerprint in the database for the input that is a audio fingerprint query.

Using audio waveform for comparing differences have many problem about un-normalize, we favor using audio fingerprint due to numerous of advantages and described as follow:

- Robustness: Fingerprint can identify the audio even if the audio has been transformed in many ways or has noise
- Fingerprint size: The number of songs is plenty but the memory of device is limited, so it is very important when we build the real system
- Granularity: Due to the normalized format, the comparing of audio fingerprints can be much simpler than comparing the audio waveform.
- Search Speed and Scalability: Searching time depends much on the size of database and the method to search. And audio fingerprint can be extracted form multiple variance of music for conducting a normalized audio fingerprint.
- Efficient comparison: Audio fingerprint extraction algorithm focuses on removing the irrelevance information from content, so this will be more efficient than comparing the information with distortion data.

2.2.2 Mathematical Definition of Audio Fingerprint

From audio content **A**, we use a function (audio fingerprint extraction algorithm) **HF** to map audio/song content **A** to a audio fingerprint F_j. Audio fingerprint F_j is a bits sequence and for normalization we must use fingerprint with the same number of bits. The distance function of audio fingerprint should be a norm distance $Distance(F_1, F_2) = ||F_1 - F_2||$. In term of comparing bits array like audio fingerprint, in this thesis we choose hamming distance for calculating the similarity between two fingerprints:

$$D_h(F_1, F_2) = \sum_{i=1}^{d} |F_1^i - F_2^i| \tag{2.1}$$

For the equation 2.1, d is the number of bits of audio fingerprint, the hamming distance technique is used to count the number of different bits between two arrays. Especially, hamming distance will return the number of different bits in two audio fingerprints indicated by bit indexing.

We also can define audio fingerprint as a cryptography method with using hash function family. A cryptography function is a algorithm of merging the arbitrary data block for returning a same length of bits sequence. Hash function $\mathbf{H}()$ can help to normalize the audio waveform object $\mathbf{X}$ to vector-domain. For comparing two audio waveform objects $\mathbf{X}$ and $\mathbf{Y}$, we can transfer those into hash-value content before making a simpler comparison between $\mathbf{H}(\mathbf{X})$ and $\mathbf{H}(\mathbf{Y})$. With using the hash values only for determining the difference, we need to accept the error probability of loss information from original objects. By the properly of designed cryptographic functions, with the chance of error is 2^{-n}, and n is the number of hash functions. By assigning the number of hash functions, we can decide the size of hash value for reducing the storing size of hash values instead of storing objects. The biggest drawback of using hash function is it is algorithm based on the random hash function to reduce the dimensions of original object, beside that the audio wave can exist in many forms of waveform but still sound similar with human ear. In this case, we should normalize the original audio to the same format of waveform such as bit-rate, sample bits. For discriminating between two hash values, we need to choose threshold **T** for indicating the probability of the same object content.

$$\text{If audio waveforms } \mathbf{X} \text{ and } \mathbf{Y} \text{ are similar, } ||\mathbf{X} - \mathbf{Y}|| \leq \mathbf{T} \tag{2.2}$$

$$\text{If audio waveforms } \mathbf{X} \text{ and } \mathbf{Y} \text{ are dissimilar, } ||\mathbf{X} - \mathbf{Y}|| > \mathbf{T} \tag{2.3}$$

Threshold **T** in equation 2.2 and 2.3 is the threshold of hamming distance are selected by specific of database [2].

2.2.3 Applications of Audio Fingerprint

Audio Fingerprint has many roles in real-world, especially for identifying the meta information of unknown audio content. Many of them were already deployed for applications and here are some examples of how audio fingerprint is helpful in human life:

- **Identifying illegal audio content**
  As mentioned earlier, there are 300 hours of videos uploaded to youtube per minute. In this large amount of video content, there are many illegal songs/segments with no license or copied from other licensed songs/tracks. And Youtube has a system called "Content ID" for storing audio fingerprint for every audio uploaded. And when handling the newly uploaded audio-content, this system will match this queries fingerprint with fingerprints in database to claim whether this new audio content is legal or not [28]. This application is one of the most convenient usage for human life. It can help artist/composer protect their intellectual property perfectly, and processing speed of "Content ID" is also acceptable for real-world requirements.

- **Automatic Music Library Organization**
  Music Library Organization is a big problem for all online music storage systems such as Itunes or Amazon Music Store. The organization by category helps the users minimize complexity when searching in browsers or easily find their favorite music. It is becoming a serious problem when there are thousands of new audio contents uploaded to store every day. Automatic Music Library Organization by using fingerprint will help the store measure the intonation of music and detect which kind/category new song/track should belongs to. For example, Rock and Jazz will have different intonation, beats speed or rhythm.

- **Identifying Unknown Song/Rhythm**
  This is an example of an interesting application for smartphone users. Imagine that when users are listening to radio/public sound and they hear great tunes, how they can get the title of the song they are listening to. There are several applications for

smartphone that can support in these case like MIDOMI, SHAZAM or musiXmatch. That application can show the name of song/track on the phone's screen to users. Users just need to record the sound, then those application will extract the audio fingerprints using their algorithm on user's phone. After that, those audio fingerprint will be sent to their audio fingerprint database server for finding the most similar audio fingerprint. Finally, this server will send back the meta information of the output fingerprints to users.

2.3 Audio Fingerprint Extraction Algorithm and HiFP2.0

2.3.1 Audio Fingerprint Extraction Algorithm

There are several algorithms for audio fingerprints extraction like Mel Frequency Cepstral Coefficients (MFFC) [27], Linear Predictive Coding (LPC) [3]. Most of them use Fast Fourier Transform (FFT) to transfer audio waveform to spectra domain before collecting features from its spectra. Using spectra of audio is a good way for extracting the content by the values of frequency on its audio (audio frequency is easily distinguished by the human ear).

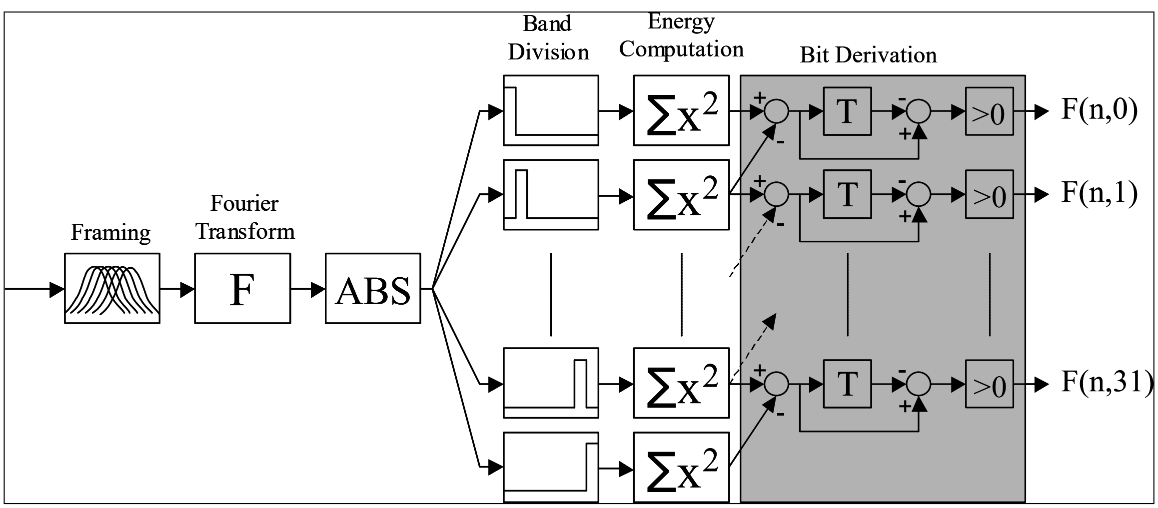


Figure 2.6: Overview fingerprint extraction scheme using FFT [7, p.4]

Figure 2.6 demonstrates the stages of audio fingerprint extraction using FFT method. In this Figure, audio waveform is transformed to multiple Band Division by using a positive amount of frequency after using FFT. Each Band Division has an Energy by the sum of square of every value in this band. Finally, Now Derivation is a final step for computing the mutual information for bands and exporting bits for output audio fingerprint [7].

However, FFT is a complicated algorithm with complexity of $O(nlog(n))$ where n is the length of input vector. Beside that, FFT uses many floating-point operations, which is not compatible for a fast system we want to build. In order to avoid using floating-point operations and reduce the complexity of algorithm, we prefer using algorithm which uses Haar Wavelet Transform (HWT) instead of FFT method. HWT transfers PCM to time-sequence domain and only uses integer operations that help increase speed of extraction method [2].

2.3.2 HiFP2.0

We choose to use HiFP2.0 for the extraction of the fingerprint from raw songs. HiFP2.0 of Yang is a good algorithm for extracting the audio fingerprint without using the floating point numbers. The size of a fingerprint is not too large (512 bytes) for a normal system. The results show that HiFP2.0 can make right 100 percent for the query with 0.05 percentage distortion query [1, 2].

As principle of HiFP2.0 decribed in Figure 2.7a, it first uses HWT to decompose the signal of input waveform to low-frequency and high-frequency having half-length of the original waveform data. There are multiple levels of sub-bands and the higher level of sub-band will carry out the result of previous sub-band. The more levels of decomposition there are, the smaller the compress size of audio fingerprint after extraction is. In our case, we choose to use 3-level of decomposition for optimal speed and fingerprint size. After that in Figure 2.7b, similar to MFFC, HiFP2.0 calculates the energy of subbands for extracting the reliable factors. By calculating the gradient of the subband, HiFP2.0 features are only storing the gradient directions for up/down to one or zero. Technically, HiFP2.0 use the difference of subband value with value followed. The value of finger is true if the sign is positive and otherwise is false [1].

After two stages of decomposition, the output is a binary vector representing the audio fingerprint of the input waveform object. When comparing the current original audio's size content when using 3-level decomposition, HiFP2.0 features can reduce the size of the original to 512 times [1].

(a) Algorithm of multi-level subband decomposition using Haar Wavelet Transform

```
MHWT( wav[] ← input waveform data,  n ← number of input signal, m ← number
of output samples)
{
while (TRUE) do
    n ← n/2;
    for ( i = 0; i <n/2; i++ ) do
        Hi[i] ← ( wav[2*i] − wav[2*i+1] ) / 2;
        Lo[i] ← ( wav[2*i] + wav[2*i+1] ) / 2;
    end for
    wav[] ← Lo[i];
    if (n < m)
        break
    end if
end while
return (Hi, Lo);
}
```

(b) Algorithm of fingerprint generation

```
HiFP2.0( wav[] ← PCM data )
{
n ← Number of PCM data samples;
m ← Number of output samples;
Hi[], Lo[] ← MHWT( wav[], n, m );
for ( i = 0, j = 0; i <m-4; i += 4, j++ ) do
    if ( Lo[i] - Lo[i+4] >0 ) then
        FP[j] ← 1;
    else
        FP[j] ← 0;
    end if
end for
FPID[m-1] ← 0;
return FP;
}
```

Figure 2.7: HiFP2.0 audio fingerprint extraction algorithm [2]

2.4 Localitive Sensitive Hashing

Similarity search problem includes a collection of objects represented by vector and several queries that need the most similar object in the collection [6].

Theorem 1 (Nearest Neighbor Search(NNS)) *Given a set P of objects represented as points in a normed space l_p^d, preprocess P so as to efficiently answer queries by finding the point in P closest to query point p [6, 22, 8].*

NNS is an important problem in many fields of science and engineering. There are many researches of algorithms that are already proposed to handle the NNS. However, complexity of algorithms grows exponentially with the dimensions (curse of dimension), which is a big difficulty for real-time system with high dimensions. By a simple trade-off, we can deal with the curse of dimension by using a technique for approximating the NNS [23].

Theorem 2 (ε-Nearest Neighbor Search(ε-NNS)) *Given a set P of objects represented as points in a normed space l_p^d, preprocess P so as to efficiently return a point $p \in P$ for any given query point q that $d(q,p) \leq (1+\varepsilon)d(q,P)$ where $d(q,P)$ is the distance of q to its closest point in P [6, 9].*

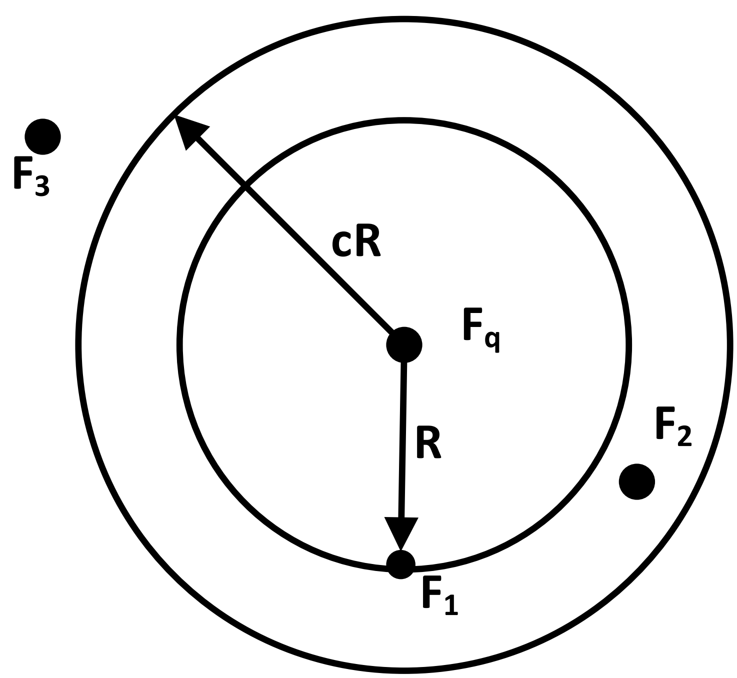


Figure 2.8: An illustration of nearest neighbor and approximate nearest neighbor [6, p.2]

In Figure 2.8, we can see there are three points in the database F_1, F_2, F_3 and a query point F_q. For the nearest neighbor problem, F_1 should be the chosen one. However, when we consider approximate nearest neighbor problem on this database, suppose that the distance between F_q and F_1 is R and the approximate factor is c, there are two points

F_1, F_2 will meet the requirement $|F_q - F_i| \leq cR$. In this case, we can also return F_1 or F_2 both of which are fine.

LSH is one of the best well-known methods for ε-NNS problem in big data using approximate nearest neighbor. In simple way, LSH devides the data into buckets, the number of buckets depends on numbers of hash functions to hash the vector. Vectors in the same buckets tend to be similar to each other because of the continuity of the selection of hash functions. Therefore, instead of comparing the input vector with all of the vectors in database, now we just need to compare with the vectors in several buckets.

In Figure 2.9, LSH uses hash functions to choose l subnets $I_1, I_2, ..I_l$ of database vectors. Let p_I be the projection of vector F_j on the coorinate positions. Denoting $g_j(p) = p_I$, we store each $F_j \in F$ in the bucket $g_j(F_p)$. We also need another table for saving the map of buckets because the number of buckets may be large or numbers of points in each bucket are different.

In Figure 2.10, for the searching problem in LSH, we also use the same hash functions for every query. For the query F_q, we determine all $g_1(q), g_2(q), ...g_l(q)$, and let $F_1, F_2, ...F_t$ be the points in bucket on current process. We need to compute the distance $l_1(F_p, F_q)$ for every point in this bucket. For KNN problem, we stop when reaching K points in different or same buckets. However, for the audio fingerprint, we can return at the first F_p having the $l_1(F_p, F_q) < P_1$ to archive a good result and a better performance, where P_1 is a threshold of the maximum distance when two points are close.

Figure 2.9: Algorithm of LSH Preprocessing [1]

Input A set of n points(fingerprints) F
Input A hash table maps the points with buckets, $T_1, T_2, ...T_l$
For each $i = 1, 2, ...l$
    Build a i-th bucket by randomly generating the hash function $g_i(.)$
For each $i = 1, 2, ...l$
    For each $j = 1, 2, ...n$
        Store the F_j to the bucket $g_i(F_j)$
Store the hash-table

In Figure 2.11, LSH chooses a family of hash function for handling database and query also. In the preprocessing stage, hash function is used for dividing the database into buckets, there are three buckets with the different colors in figure. Each bucket will have its hash value by the principle of hash function. In term of Searching stage, the query also needs calculating the hash value by the previous hash function. This value of hash function will indicate to a bucket that holds the similar points to the query (purple). Next, there is another step for comparing the distance from query to all points in the purple buckets and returning the closest one.

Figure 2.10: Algorithm of LSH Approximate Nearest Neighbor Query

Input A query point F_q
Output Point F_p that is approximate nearest neighbor of F_q
For each $i = 1, 2, ...l$
    Find the bucket B in hash function $g_i(F_q)$
    Return the first point $F_p \in B$ that $d(F_p, F_q) < \mathbf{P_1}$

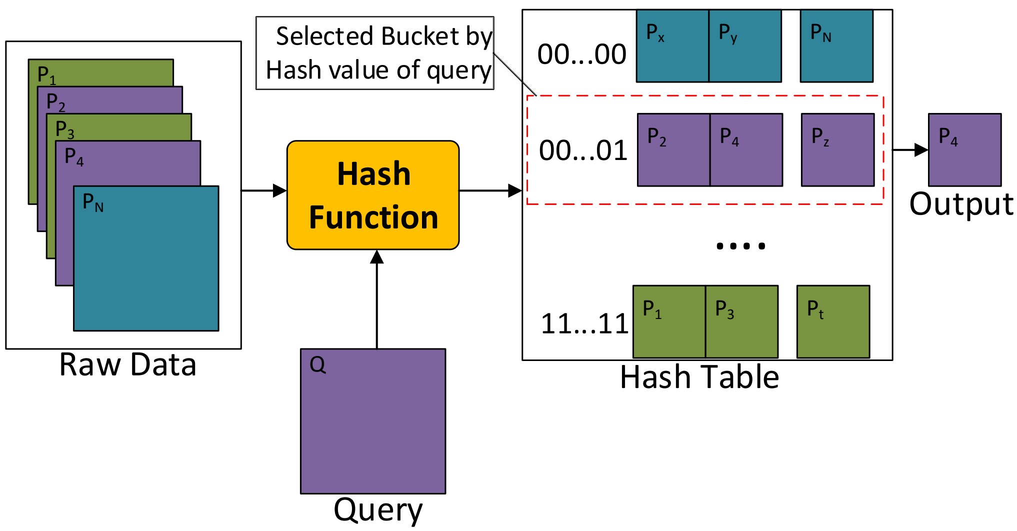


Figure 2.11: An illustration of locality-sensitive hashing

2.5 K-means

The goal of clustering is to partition database points into distinguished groups. Clustering has a big role in machine learning, pattern recognition, image processing and data mining. K-means is one of the popular algorithms for data clustering [15].

Suppose that we have dataset of fingerprints $F = F_1, F_2, F_3, ..., F_n \in Binary^d$, we want to divide this dataset into K groups (clusters) $C_1, C_2, ...C_K$. K-means finds local optimal solutions with minimized error functions defined by sum of Euclidean distances between each data point F_i with m_j, where m_j is the mean vector of cluster C_j. The error function is defined as bellow [15]:

$$\mathbf{E}(\mathbf{C_1}, \mathbf{C_2}, ..., \mathbf{C_K}) = \sum_{\mathbf{j=1}}^{\mathbf{K}} \sum_{\mathbf{F_i} \in \mathbf{C_j},} ||\mathbf{F_i} - \mathbf{m_j}||^{\mathbf{2}} \tag{2.4}$$

The target of K-means clustering is to find the model $C_1, ...C_K$ that minimizes the error function 2.4:

$$\underset{\mathbf{C}}{\operatorname{argmin}} \sum_{\mathbf{j=1}}^{\mathbf{K}} \sum_{\mathbf{F_i \in C_j},} ||\mathbf{F_i} - \mathbf{m_j}||^{\mathbf{2}} \tag{2.5}$$

Figure 2.12: Algorithm of K-Means [21]

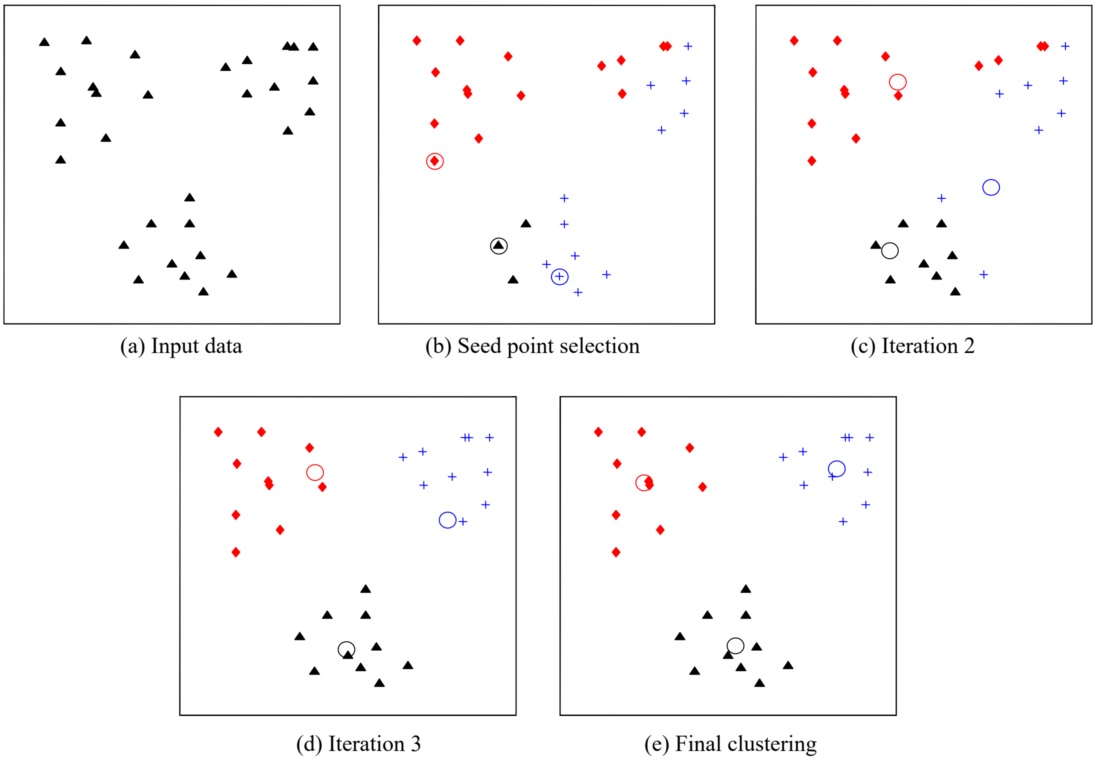
Input Dataset $F = F_1, F_2, ..., F_N$ and K
Step 1 Random initialization $m_1, m_2, ..., m_K$ for
clusters $C_1, C_2, ..., C_K$, set $t = 0$
Step 2 For each point F_p in F
Assign label for F_p :
$\mathbf{L^t(F_p) = \{C^t_p : ||F_p - m_p||^2 \leq ||F_p - m_i||^2 \forall i, 1 \leq i \leq K\}}$
Step 3 For each point C_i in $C_1, C_2, ..., C_K$
Re-update the position m_i for cluster C_i
$\mathbf{m_i^{t+1}} = \dfrac{\mathbf{1}}{|\mathbf{C_i}|} \sum_{\mathbf{F_j \in C_i}} \mathbf{F_j}$
Step 4 $t \leftarrow t + 1$
Step 5 **If** number of loop t reaches its threshold or the error function is converged then **exit**
Else Go to step 2

In Figure 2.12, K-means has a random step at the beginning, which makes it have several local optimization outcomes for clusters. That is the reason why we should use K-means multiple times for getting the best result. K-means consists of two main steps that are done consecutively by loops. The first step is finding the label for every point $F_j \in F$ by the nearest centroid m_i. And second step is updating the positions $m_1, m_2, ..., m_K$ for every cluster $C_1, C_2, ..., C_K$ by calculating the mean of all points in each cluster. Normally, K-means uses Euclidean for computing the distance of points or centroids:

$$\mathbf{L_j(F_x, F_y)} = (\sum_{\mathbf{i=1}}^{\mathbf{d}} |\mathbf{F_x^i} - \mathbf{F_y^i}|^{\mathbf{j}})^{\mathbf{1/l}} \tag{2.6}$$

In the Figure 2.13, at the beginning, the centroids are choose randomly from data points. And the final solution have the best local optimal in term of the average distance form points to its clusters.

According to the second step of K-means, we know that the binary vector is not appropriate for calculating the mean because value domain of bits is not allowed.



(a) Input data (b) Seed point selection (c) Iteration 2

(d) Iteration 3 (e) Final clustering

Figure 2.13: An illustration of K-means Iterations [30]

2.6 K-modes

K-modes is one of the extensions for K-means that focuses on clustering the discursive data. Binary vector is a kind of categorical data with two categories TRUE or FALSE. The labeling step K-means prefers using l_1 distance (hamming distance) for measuring the different of categorical values.

$$\mathbf{L_1(F_x, F_y)} = \sum_{\mathbf{i=1}}^{\mathbf{d}} |\mathbf{F_x^i} - \mathbf{F_y^i}| \tag{2.7}$$

For the labeling step, K-modes uses the same method with K-means by minimizing the hamming distance for the point F_j to all modes (centroids) m_i. In updating modes step, K-modes finds the dominant attributes in every cluster for all dimensions to set the attribute to centroids vectors. Denote $F = \{F_1, F_2, ..., F_n\}$ is a cluster (set) we want to find the centroid, Q is the expected centroid for Y. The goal is to minimize the sum of distance between points and its centroids.

$$\mathbf{D(F, Q)} = \sum_{\mathbf{i=0}}^{\mathbf{n}} \mathbf{d_1(F_i, Q)} \tag{2.8}$$

To find the dominant attributes in set F, let $n_{c,k}$ be the number of objects in F having the k^{th} category $c_{k,j}$ in attribute A_j and $fr(A_j = c_k|F) = \frac{n_{c,k}}{n}$ be the frequency of category

$c_{k,j}$ in F. The $D(F, Q)$ is minimized if and only if:

$$\mathbf{fr(A_j = q|F) \geq fr(A_j = c_k|F) \forall q \neq, c_k \forall j = 1, 2, ..., d} \tag{2.9}$$

Where d is the dimensions of vector.

Chapter 3

Related works

3.1 Streaming Similarity Search over one Billion Tweets using Parallel Locality-Sensitive Hashing(PLSH) [13]

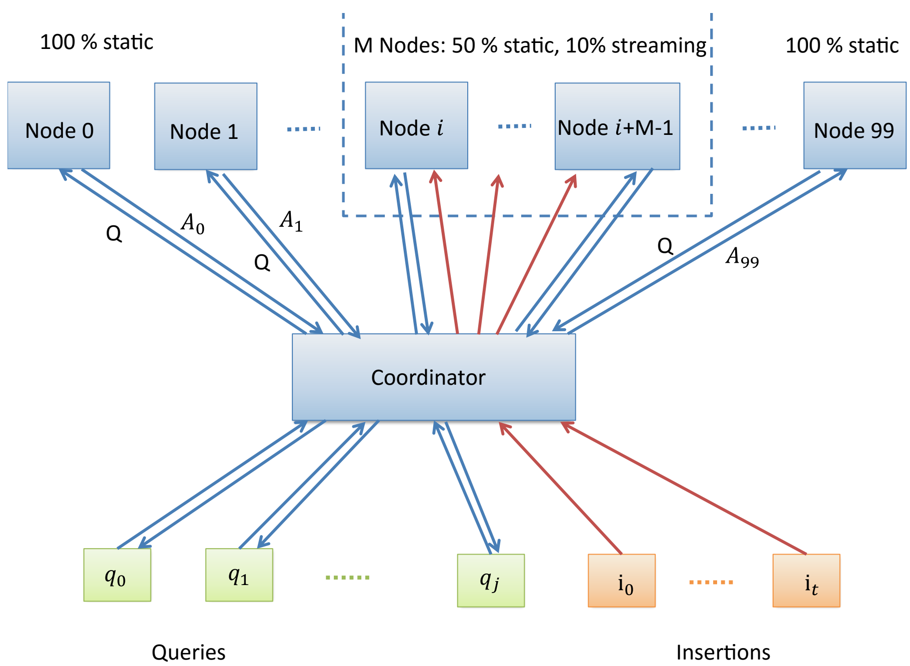


Figure 3.1: PLSH System [13, p.2]

PLSH system shows that it can handle database with billions of records by building two levels of LSH. Database is stored in different parts kept in distinguished nodes. PLSH focuses on dealing with the big data such as Tweets's database, the constantly updated data is a problem of LSH due to the transform of hash table and index of record in database. Another advantage of PLSH is handling the queries in real-time for the requisite of Tweets's users [13].

According to Figure 3.2, the coordinator will receive the data from inserting data or search query. For inserting, PLSH uses a filter window to choose M nodes from i to $i+M-1$ to handle the inserts in round-robin fashion.

The authors propose to use 2-level hashing to reduce the number of hash construction instead of using many pointers to indicate the address of every bucket. Besides that, partitioning the database can help store the big hash table on several nodes; so the PLSH can work in nodes with small memory. Another strength of PLSH is parallel querying on multiple nodes, each node holds an independent part of the database; so they can search at the same time before sending the result to coordinator. With these advantages, PLSH can speed up the inserting and query stage of Tweet system to 1.5X [13].

3.2 Bi-level Locality Sensitive Hashing for K-Nearest Neighbor Computation [11]

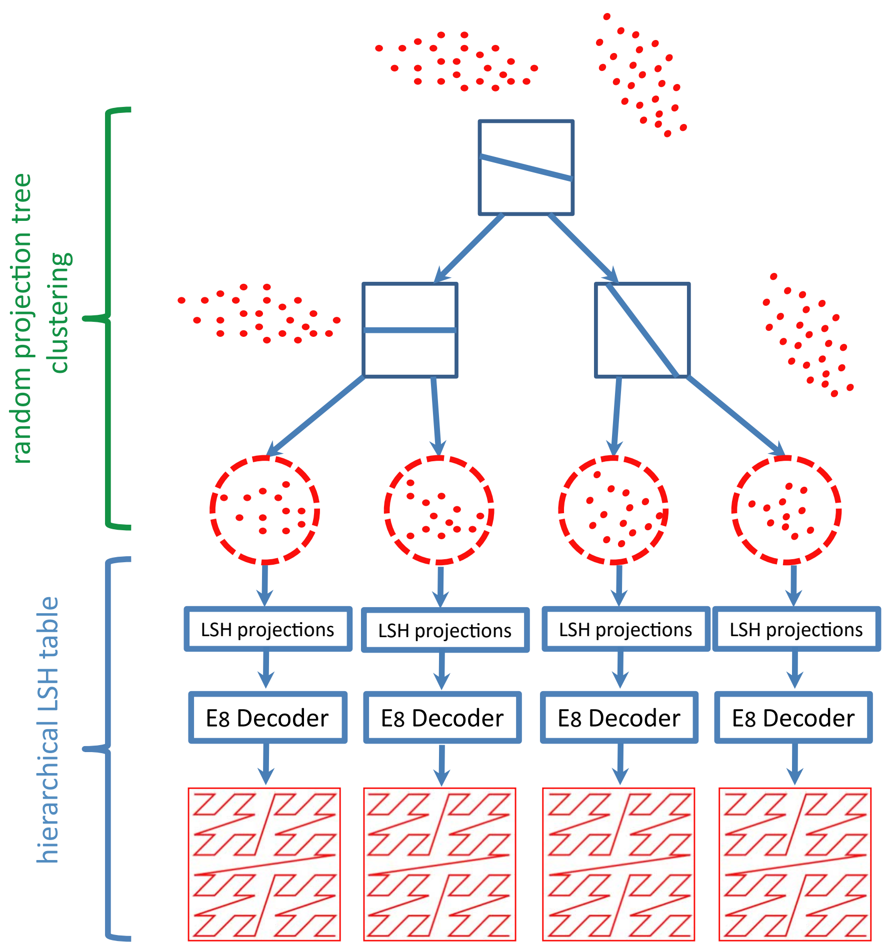


Figure 3.2: Bi-level LSH using RP tree and hierarchical lattice [11, p.3]

Bi-level LSH algorithm includes two levels of processing proposed by Jia Pan [11]. Random projection tree (RP-Tree) is used at the first level to divide the dataset into subnets. Due to the strengths of RP-Tree, the points on the same subnets will tend to be similar to each other. On the second level, authors build an LSH table for each subnet. Especially, the authors use a Morton curve to create a hierarchal LSH so as to increase speed performance of LSH query. For k-nearest neighbors problem, when the query has hash value to the bucket with high data density, the algorithm just needs to search the few nearby buckets for getting k-nearest neighbors, and for the query bucket with low data density or being empty, the algorithm needs to search in farther buckets to get enough number of nearest neighbors [10, 11].

In the query step, for the query F_q, Bi-level LSH needs to calculate the RP-tree leaf node that contains F_q first. And for the second level, LSH is used to find the buckets $H(F_q)$ that hold F_q with the hash table indicated in the RP-tree leaf node at the level 1. Therefore, the address of output of a query includes two part $\hat{H}(F_q) = (RP - tree(F_q), H(F_q))$, where the $RP - tree(F_q)$ is address of leaf node containing F_q, and $H(F_q)$ is the index of bucket having F_q in the corresponding subnet. Bi-level LSH has better locality coding than the original LSH hash function. It also has smaller deviation because of the random projection of RP-tree. Bi-level LSH shows that it is an algorithm that is compatible with GPU because running on GPU can be 40 times faster than running on CPU [11, 12, 18].

3.3 Fast k Nearest Neighbor Search using GPU [14]

k Nearest Neighbor problem is a similar problem with the approximate nearest neighbor problem with the same input data $F = F_1, ..., F_n$ and a query F_q. But k Nearest Neighbor will return k outputs $F_1^p, ..., F_k^p$ that nearest while ε-NNS return the first F_p that meet the requirements of approximate neighbor.

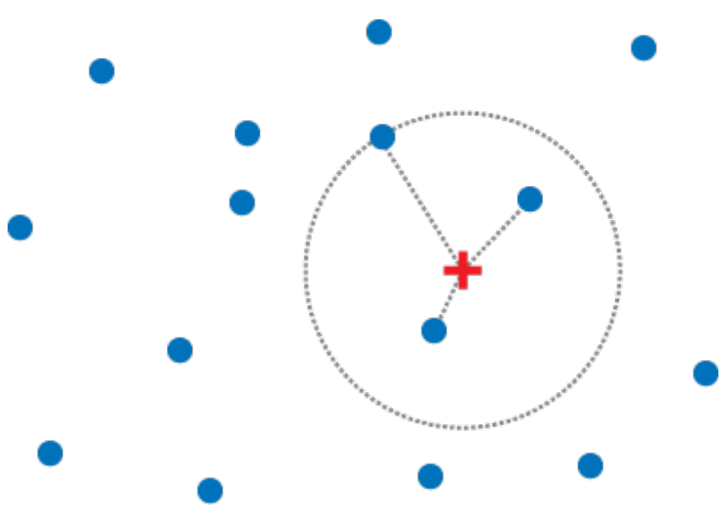


Figure 3.3: Example of k Nearest Neighbor with k =3 [14]

In Figure 3.3, With $k = 3$ the algorithm will choose the three most nearest neighbors for the query. As the original algorithm of k Nearest Neighbors, we need to compare the distance from the query for all the points in database. Calculating the distance for every points will have complexity $O(nd)$, where d is the number of dimensions. The second step of searching for k Nearest Neighbors is the sorting, sorting the distance have more complexity $O(nlogn)$ 3.3.

Bruce force for kNN has highly complexity, there are several methods for reducing the complexity of kNN by change it to approximate k-Nearest Neighbors problem. This will decrease the searching time but the accuracy will decrease also 3.3.

In this research of Fast k Nearest Neighbor Search using GPU, the authors choose to use comb ($O(nlogn)$) for implementation in GPGPU because with QuickSort they need to handle the recursive in CUDA. An important thing is they can reduce the complexity of comb sort in there research because of the requirement of kNN is not the full-sort problem. Authors need only searching for the first k elements in the array.

For the implementation of GPGPU with CUDA, Based on the easily parallel of Bruce Force method, authors can easy storing the data in the main memory that can be accessed by every threads in CUDA's cores. And the calculation of distance will be assigned for the threads 3.3.

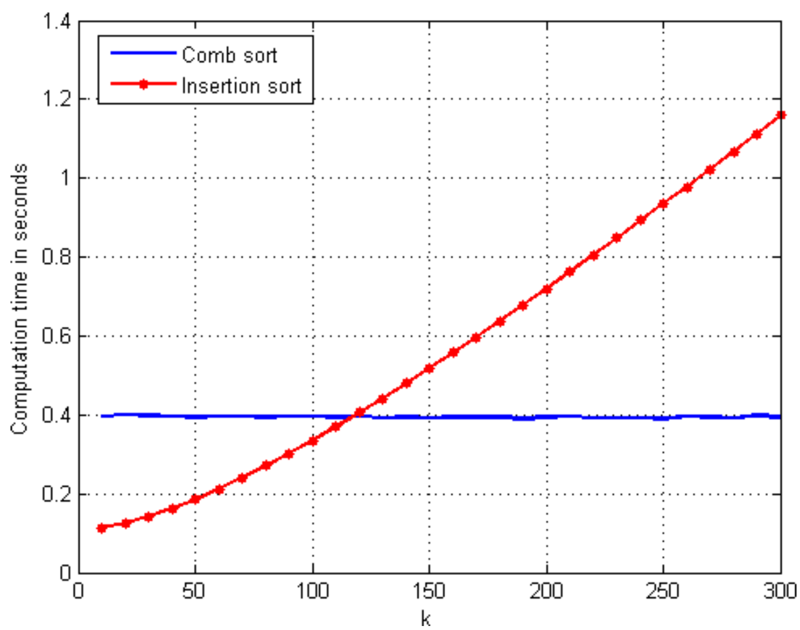


Figure 3.4: Time Comparison of Sorting using Comb sort vs Insertion sort [14]

Result in Figure 3.4 show the highly capacity for parallel of Comb sort versus Insertion sort. With the good result in Figure 3.4, authors built a parallel searching system with multiple queries for k-NN problem. In this system, each thread will handle the the whole searching stages for one query. Because the the data is stable and can storing in the main memory in device's memory, all the threads can easily access all the data without confliction.

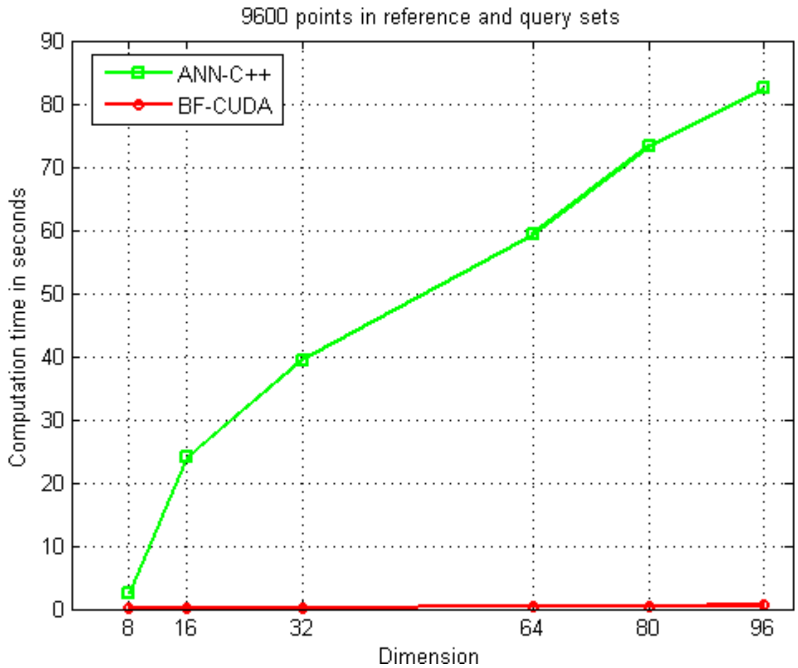


Figure 3.5: Time Comparison of Searching using GPU vs CPU [14]

In Figure 3.5, With using of parallel processing on GPGPU, authors are achieve good performance when parallel searching the queries. With these highly throughput performance, this system adapt with the system of numerous numerous queries such as audio fingerprints searching for detection the illegal songs/track are uploaded to the Internet.

Also having the well result of parallel of searching for k-NN problem, this system will have many problem with the complexity when using brute force searching and not support for using multiple GPGPU devices. Because the complexity of brute force is $O(nd)$ it is can not deal the data using big data such as data with 10,000,000 audio fingerprints. In addition, with the large number of data, they can not storing it in single GPGPU device.

Chapter 4

Proposed method: Parallel Audio Fingerprint Searching using Single GPGPU

4.1 Previous Research

HiFP2.0 of Yang is a good algorithm for extracting the audio fingerprint without using the floating point numbers. The size of a fingerprint is not too large (512 bytes) for a normal system. As the results show, HiFP2.0 can make right 100 percent for the query with distortion rates = 0.05. Using LSH (Locality-Sensitive Hashing) is an advantage to speed up the query time. However, because the number of compares and the dimensions of fingerprint is high, it spends large amount of time to return the result for a query [1][2].

In Yang's thesis, the searching time is quite good for the small database. Specifically, Yang's method can find the cR-near neighbor for a query in 0.7 milliseconds in the database with 300 fingerprints.

4.2 Problem Definition for Parallel Audio Fingerprint Searching using Single GPGPU

The audio fingerprint data holding n audio fingerprints $F = F_1, F_2, F_3, ..., F_n$, the storing size of fingerprint data base are limited by the memory size of GPGPU device, for example the limited size of Tesla K40 is 13GB. And the known meta information for every audio fingerprint in the data set F. The set of audio fingerprint query holding T queries $Q = (F_{q1}, F_{p2}, F_{q3}, ..., q_T)$ with unknown meta information. The requirement is build a audio fingerprint system that returning the information for unknown queries meet with the following conditions:

Accuracy: The meta information should match with content of unknown queries by using the meta information of approximate nearest neighbor even when the query audio fingerprint have highly distortion compare to to original audio fingerprint.

Throughput Parallel Searching: The system must support for parallel searching, can help searching thousands of queries searching at same time using multiple GPGPU cores.

Limited Database size: The requirement of this chapter is using only 1 GPGPU device for searching. And every GPPGU device, the memory size is limited by the total amount of device's memory. We should have a compress database that can supporting for fast searching and also.

Limited Searching Time: Based of the requirements of real-time system, the searching must small enough for returning the output song's meta information for user. specific in our proposal report, for 10,000,000 audio fingerprint data, the limited searching of every audio fingerprint query is 0.1 millisecond.

4.3 Preprocessing Stage (Building the Audio Fingerprint Database)

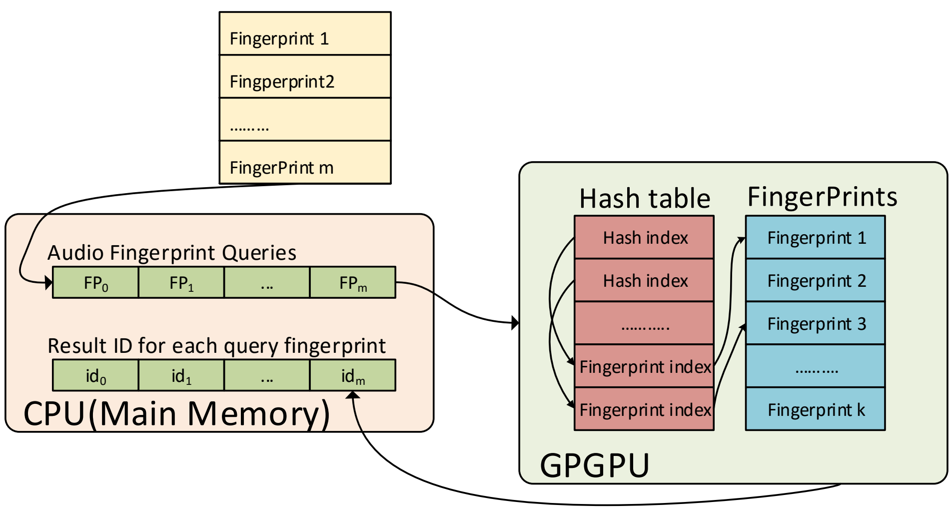


Figure 4.1: System Overview of Audio Fingerprint Searching using single GPGPU

Well organization Audio Fingerprint Database is the key for fast searching. In Figure 4.1, our database support for LSH searching. Which need the total data audio fingerprints storing in the GPGPU. And also the hash table that storing the address of buckets and address of fingerprints for every bucket.

For building the hash table for audio fingerprint data, we need to choose the number of hash function, this number of family hash function will be used in the searching stage also.

Using same family hash function for every audio fingerprint in data for getting the hash value corresponding audio fingerprint. Each different value of hash value will indicate to same bucket address.

Algorithm 1 Algorithm for Generating the Hash Table for audio fingerprint data

Require: Audio Fingerprint Data, Number of hash function
  HT=null {Hash Table}
  for Every audio fingerprint F_i in F **do**
    for j=0; j <126;j++ **do**
      frame[i] ← sub_fingerprint at j of F_i {sub-fingerprint}
      hash ← 0 {Hash value for current sub-fingerprint}
      for function h in family hash function **do**
        hash ← hash <<1
        hash ← hash OR ((frame >>h) & 1)
        HT[hash].append(128 * i + j)
      end for
    end for
  end for
  return Store HT to hard drive

Figure 4.2: Algorithm for Generating the Hash Table for audio fingerprint data

We show Algorithm 1 in Figure 4.2 for creating the hash-table for the audio fingerprint data. In which, we use the original principle of Staged-LSH by dividing the fingerprint into 126 sub-fingerprints. Each sub-fingerprint will indicate to a bucket. We can see the variable *HT* is the pointer array that storing all the address of audio fingerprints for corresponding bucket.

The algorithm 1 in Figure 4.2 is used for the main algorithm for preprocessing stage in Audio Fingerprint Searching System using single GPGPU. Which showed in Figure 4.3. We can see in Figure 4.3, after running the Algorithm 1, we get a hash table that related with the input data. The database of searching stage is the combining of the raw data with audio fingerprints and the hash-table generated by the raw data by Alrogithm 1.

Finally step in preprocessing stage is the loading the database that including the audio fingerprint and hash table into GPPGU device. With this well developed database structure, we can easily find the address of buckets and use it to find the addresses of every audio fingerprint on this bucket.

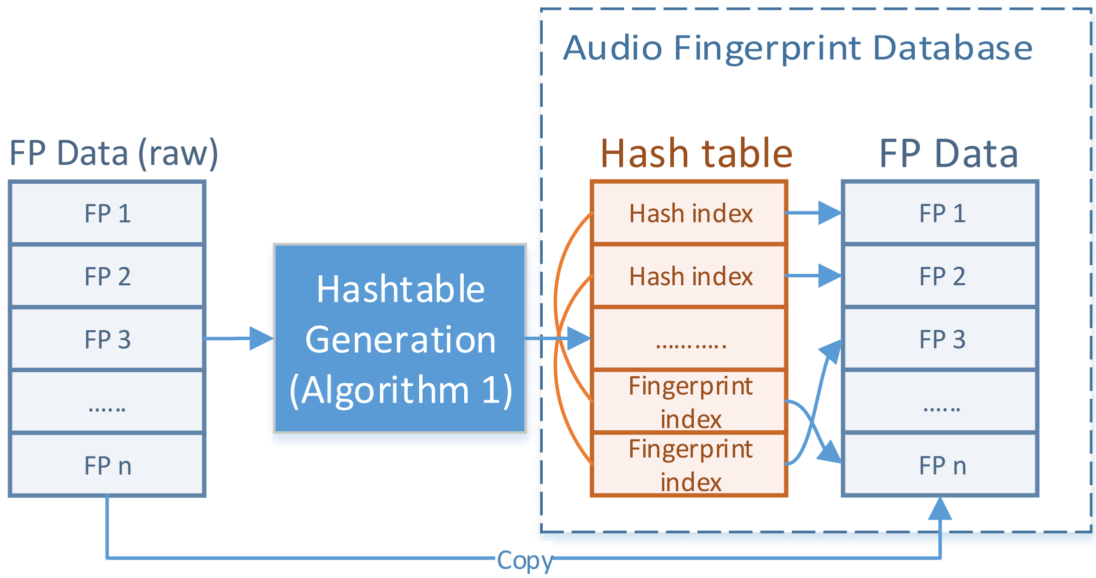


Figure 4.3: Preprocessing Flow for Audio Fingerprint Searching System using single GPGPU

4.4 Searching Method on Single GPGPU

Audio finger of HiFP2.0 is a vector having 512 bytes with the binary information. The original audio clip to extract must have 4096x128 samples (2.97s) [1]. As the principle of LSH, when applying to the data of HiFP2.0, we must calculate the hash strings for the fingerprint using the hash functions. With each value of hash string, we have a bucket correspondingly. Because the hash function has binary output, a number of buckets will be 2 exponents $hashfunctionnumber$. Because the number of hash functions will affect the searching time.

Each query fingerprint has two steps. The first one is using the LSH hash functions to detect the buckets that should have its nearest fingerprint. And the second step is finding the nearest fingerprint in the specific buckets already gotten from the first step.

In Figure 4.4, In every thread we handle the searching for one query. And the searching stage in one thread return only one ID for corresponding query. The first step, Using the same hash family functions H in preprocessing step and same number of hash function, the threads will calculate the hash strings for sub-query from F_q. For each hash string will point to a bucket B that hold several audio fingerprint $F = F_1, F_2, ..., F_N$. For the second step, the thread will compare the distance of query F_q to every audio fingerprint in B to find the distance that meet the threads hold $\mathbf{P_2}$ of LSH. This thread will stop when a approximate nearest neighbor is found on any sub-fingerprint's bucket.

In the figure 6.2, the fingerprint must be divided into 126 sub-fingerprints. The sub-fingerprint has its hash string and been pointed to a bucket. The LSH will stop when the first sub-fingerprint gets the satisfied fingerprint in its bucket. We test our new method

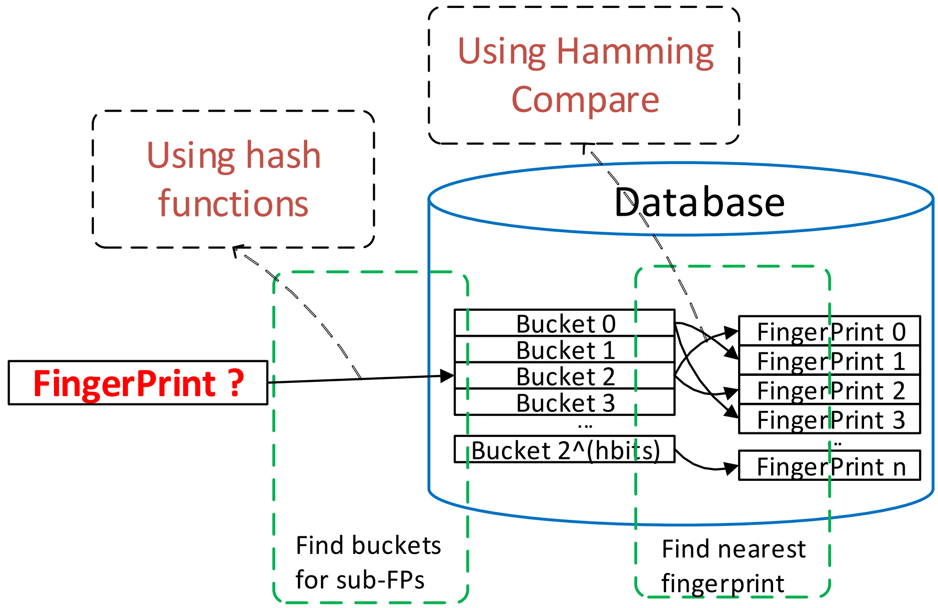


Figure 4.4: Principle of Audio Fingerprint Searching Flow in 1 Thread

using the same database and test cases as the original method.

Algorithm 2 Algorithm for Audio Fingerprint Parallel Searching in single GPGPU

Require: Database including Fingerprints and Hash table, queries

Preprocessing:
Copy database (audio fingerprints and hash table) from main memory to GPGPU's memory.
Searching:
Copy queries to GPGPU (num=cores number)
Initialize an array A (length=num) for storing the result IDs.
Copy queries to GPGPU device
Assign number threads equal number queries
Start Searching kernel for every query by 1 thread **(Algorithm 3)**
After all threads stop, copy A to main memory
return The Result audio fingerprint IDs for queries in CPU

Figure 4.5: Algorithm for Audio Fingerprint Parallel Searching in single GPGPU

In Algorithm 2 in Figure 4.5, we show the steps of system from preprocessing to search-

Algorithm 3 Algorithm for Kernel of searching for 1 query

```
Require: Database stored in global memory, ID of current thread,
  for qidx = 0; qidx <126; qidx++ do
    hash ← lsh(query[qidx], hbits) {Hash value for current sub-fingerprint}
    ht_array ← indexof(hash) {Index of bucket}
    num ← lengof(hash) {Length of bucket}
    A[ID] ← lengof(hash) {Output for current query}
    for i = 0; i <num; i++ do
      if Current bucket is empty then
        memcpy(&addr, &ht_array[addr2 + i], 4)
        memcpy(frame, &fp_array[addr], 12)
      end if
      if hd(frame, &query[qidx], 3) <24 then
        memcpy(fpdat, &fp_array[addr / 128 * 128], 512)
        tmp_hd ← hd(fpdat, query, 128)
        if tmp_hd <T1 and <tmp_hd <min_hd then
          min_hd ← tmp_hd {Current minimal hamming distance}
          mid ← addr >>7; {Save index if change minimal fingerprint}
        end if
      end if
    end for
    if mid != 0xFFFFFFFF then
      A[ID]← mid;
    end if
  end for
  return The Result audio fingerprint ID for querie to array A
```

Figure 4.6: Algorithm for Kernel of searching for 1 query

ing with multiple queries. And the specific steps of algorithm of kernel for every single query are showed in Algorithm 3 in Figure 4.6.

In Figure 4.7, The algorithm 2 is worked in CPU for management the working of GPGPU. It also use for access the data from main memory and write the result IDs to main memory. The Algorithm 3 is based on the kernel for the working of all threads in GPGPU.

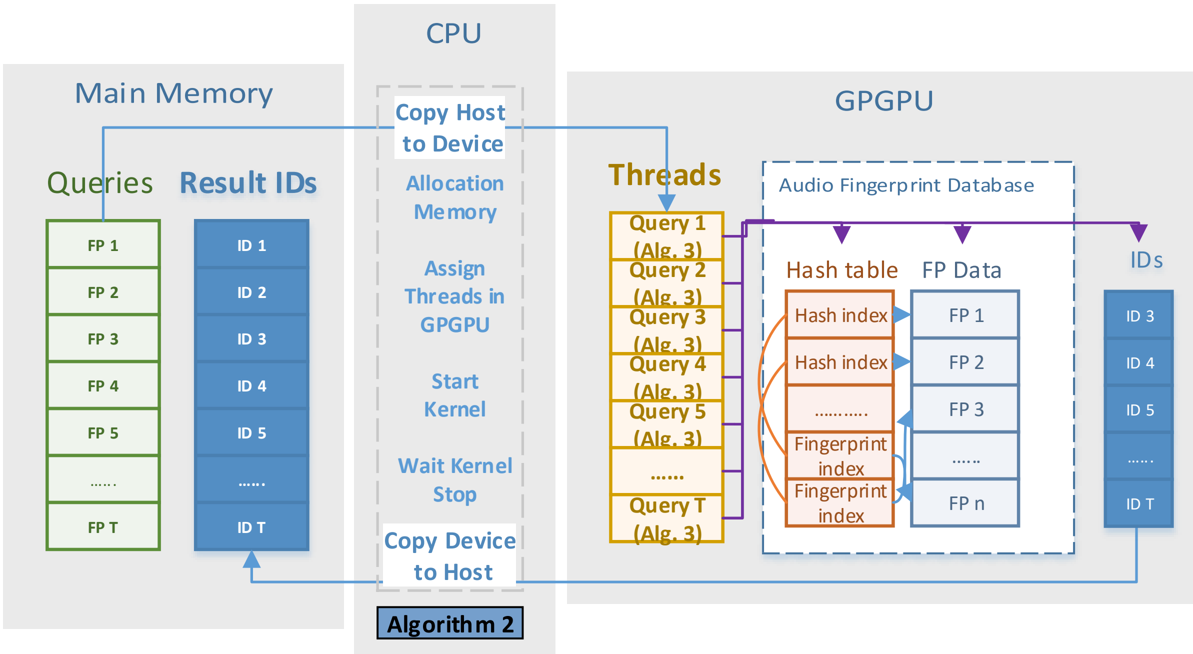


Figure 4.7: Searching Flow for Audio Fingerprint Searching System using single GPGPU

4.5 CUDA Threading Allocation

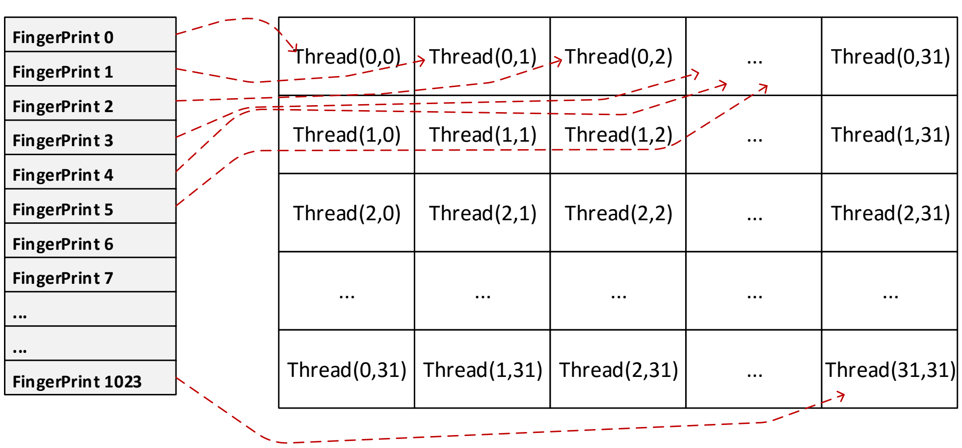


Figure 4.8: Example for parallel searching on single GPGPU using 1 Warp.

In Figure 4.8, We use multiple cores of GPU to handle the multiple queries. To do that, first we need to copy all the data including the hash-table and the labeled-fingerprint into GPU memory. We also need to copy the queries fingerprint into device, then allocate the threads number equal to the number of queries. Threads' ID in block will be

$[(query_{id})div32, (query_{id})mod32]$. And we need to create an array in device to save all the returned nearest neighbor for each query.

We also test our searching system for different CUDA threading allocation for revising with threading allocation method is most suitable for LSH hashing on parallel searching.

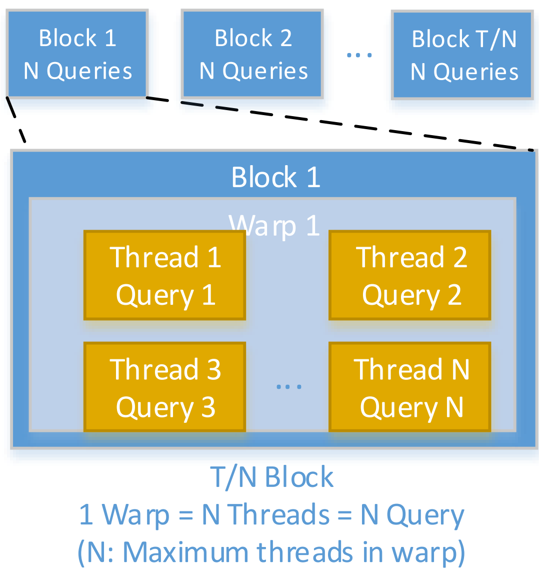


Figure 4.9: CUDA Threading Allocation with using 1 thread - 1 query on single SM

With the method on Figure 4.9, we fill the query for the warp on each SM first. For example, the K40 have limitation of number threads per warp is 2048. If we handing the less than 2049 throughput queries we can use only one core in K40. However, in this method, the rest core will do nothing. Also we can increase the number of working SM for increasing the performance of searching system.

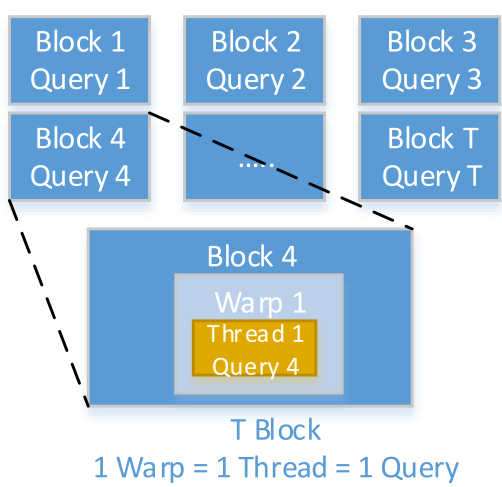


Figure 4.10: CUDA Threading Allocation with using 1 thread - 1 query on multiple SM

For the problem of method on Figure 4.9, we can easily change the system for using multiple SM. The Figure 4.10 show we can reduce the number of thread for every warp and send it to other SM. In this method, we can take advantage of the powerful of multiple CUDA core processors. However the maximum number of throughput queries will be reduced.

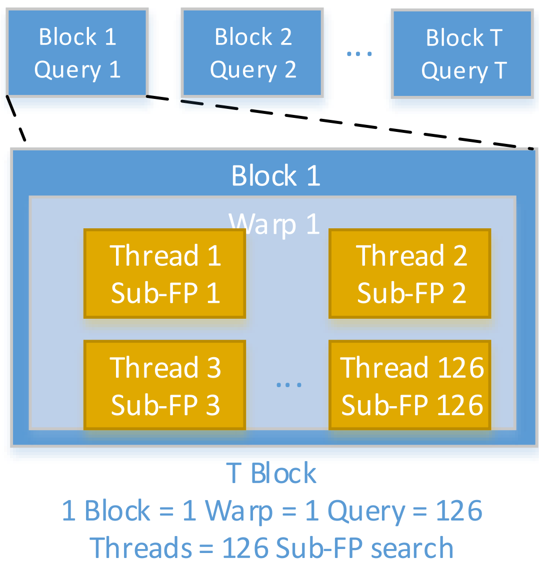


Figure 4.11: CUDA Threading Allocation with using 126 thread - 1 query on multiple SM

We also can continue to reuse the un-used threads in every SM for the method on Figure 4.10, we able to divide the job of a queries into multiple small job for multiple threads. And in this case, with the staged-LSH, it already divide the audio fingerprint to 126 sub-fingerprints. In Figure 4.11, one query will be handled by 126 threads on same warp. However when using the unit thread for sub-fingerprint, it will raise the problem for management the shared-memory for the threads in same block. Especially, 126 sub-fingerprints for 1 fingerprint, but we only need return one of the approximate nearest neighbor for the input query. So, we need the shared-memory for temporary current approximate nearest neighbor. Then 126-threads are have permission for change the value of this shared-memory. To avoid this we have to handle the writing process of every threads to this shared-memory. This trick will reduce the searching of total system.

In order to increase the performance of GPU system with the same number of threads, We choose to use the system with blocks and assign the query threads to the GPGPU cores. In Figure 4.12, We use T cores (block) for T queries. In each block there is one thread for its query.

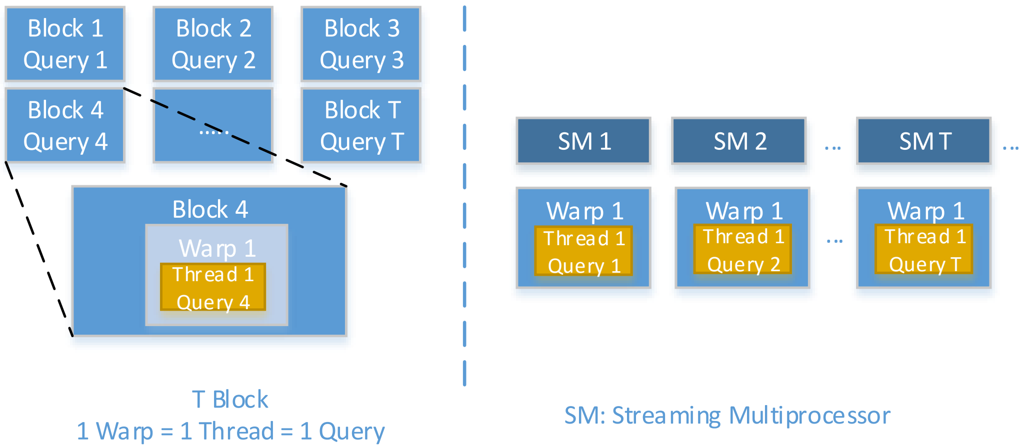


Figure 4.12: Example for parallel searching on single GPGPU Multiple Streaming Multiprocessor.

4.6 Conclusion

With the same database and queries, our method can run faster than the original method of Yang 50-60 times. This shows the great result of the parallel system. Our searching time for each fingerprint is very little (less than 0.1 millisecond), this is the advantage for us when deploying the system for almost of copyrighted music.

A number of hash functions is very important for the LSH. A larger number of hash functions will get the better searching time, but it will reduce the accuracy of the system.

Based on our research proposal, our method has a good result for the searching time. However, we need to increase the quantity of songs in database to meet our expectation.

In this research, we improve Yang's method to be suitable with the large database (millions of fingerprints) and help to search the multiple fingerprints in parallel. Our method also increases the searching speed, it will be helpful to the real system with millions of songs on the Internet.

Chapter 5

Proposed method: Parallel Audio Fingerprint Searching using Multiple GPGPUs

5.1 Problem Definition for Parallel Audio Fingerprint Searching using Multiple GPGPUs

The audio fingerprint data holding n audio fingerprints $F = F_1, F_2, F_3, ..., F_n$, the storing size of fingerprint data base are un-limited by using multiple GPGPU devices. And the known meta information for every audio fingerprint in the data set F. The set of audio fingerprint query holding T queries $Q = (F_{q1}, F_{p2}, F_{q3}, ..., q_T)$ with unknown meta information. The requirement is build a audio fingerprint system that returning the information for unknown queries meet with the following conditions:

Accuracy: The meta information should match with content of unknown queries by using the meta information of approximate nearest neighbor even when the query audio fingerprint have highly distortion compare to to original audio fingerprint.

Throughput Parallel Searching: The system must support for parallel searching, can help searching thousands of queries searching at same time using multiple GPGPU cores.

Adaptive with the real-world database: When using the system with single GPPGU, we have problem with the limited memory size. But in this chapter, we need to expand from using limited memory size to unlimited memory size.

Performance: Based of the requirements of real-time system, the searching must small enough for returning the output song's meta information for user. specific in our proposal report, for 10,000,000 audio fingerprint data, the limited searching of every audio fingerprint query is 0.1 millisecond.

Massively Parallel: When come the system using multiple GPGPUs, we need to consider the Massively Parallel problem. Because different devices have different hardware controller or different thread flow. When multiple GPPGU devices working at same time we should proposed the synchronous algorithm for management the jobs of every devices and avoid the dead-look happen.

5.2 Massively Parallel System Overview

In this thesis, we propose a new method for the approximate nearest neighbor for big database in massively parallel GPGPU system using K-modes and LSH. Our hierarchy searching system includes two levels which can be compatible with computers having multiple small memory GPGPUs. In Figure 5.4, our system has one level 1 using K-modes for clustering the data and several levels 2 for localizing search in respective subnet-database. The first level will find the most potential second level for passing the query. Every device having the second level will parallel search its query at the same time.

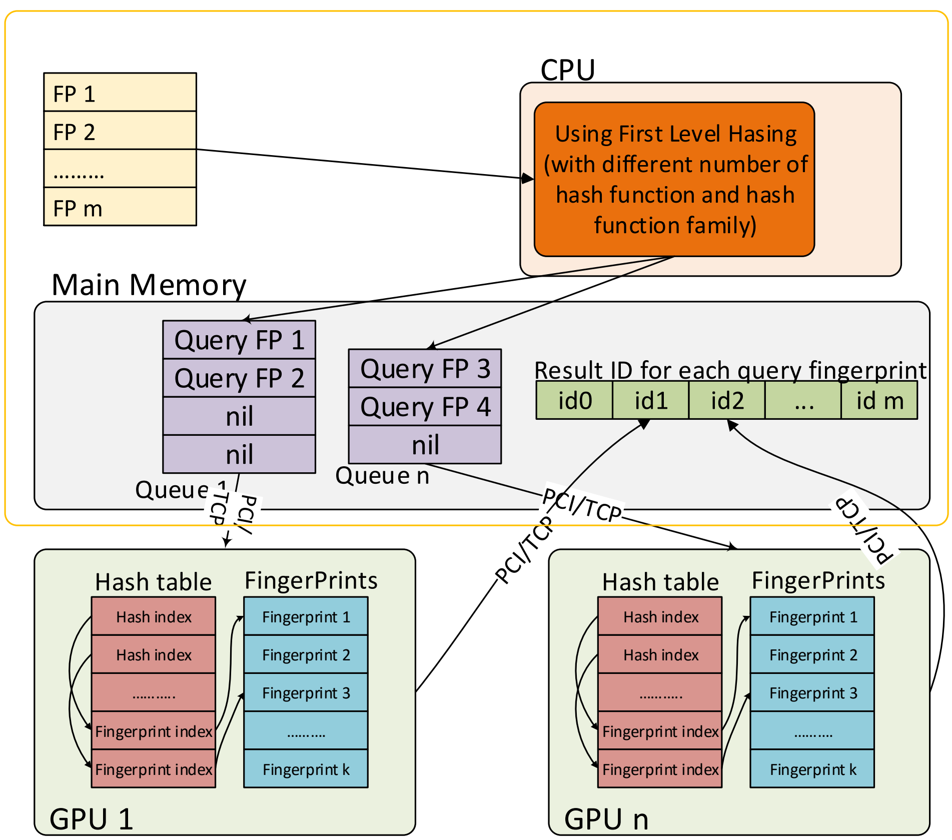


Figure 5.1: Overview of Proposed Searching System

Figure 5.1 shows the combination of algorithms and hardware in our system. The queries and output IDs of audio fingerprints are stored in main memory. The sub-databases are stored in device's memory. Communication of the host (CPU) and devices (GPGPUs) through TCP or PCI protocol depends on the hardware's configuration.

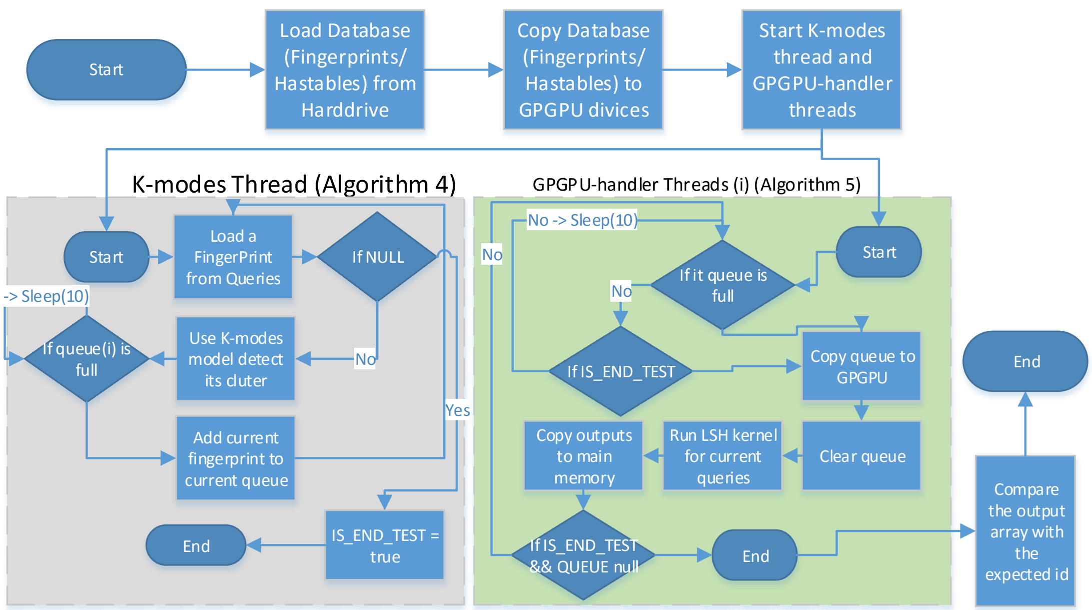


Figure 5.2: FlowChart of System Massively Parallel System

Figure 5.2 decribes the flow of threads in our method. 1 single K-modes and multiple GPGPU-hander threads. We can see CPU and GPGPUs work in the same time on this flowchart and only need to sleep when needing other resources or the current queue is full.

5.3 K-modes Level (First Level)

For sharing the database into multiple computers/devices, we use K-modes for getting the converging clusters which have similar points in the same cluster [16, 17].

5.3.1 K-modes Preprocessing

The aim of our method is to use multiple GPGPUs for multiple subnets of database. We consider the number of subnets as the number of GPGPU devices in system. Because K-modes is an algorithm that finds the local optimum, we need to build 10 K-modes models before choosing the most optimal one. In Figure 5.3, for the binary vectors, it is simple to generate the initialization centroid in range of database. A subnet of database will be represented by a binary vector $m_i = \{x_1, x_2, ..., x_d\}$ that minimizes the distance from it to all points in its cluster. After having K subnets of database, each sub-database will be transferred to LSH module for generating the buckets hash table [16].

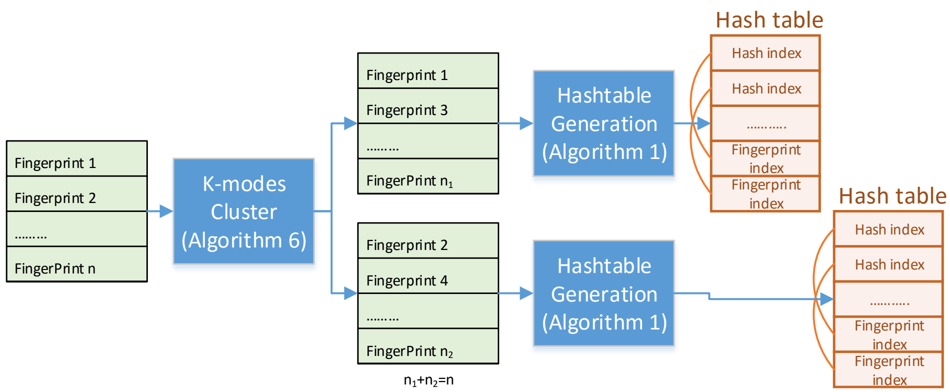


Figure 5.3: Preprocessing database for Hierarchy Searching for 2 devices

5.3.2 K-modes Querying

K-modes Querying is the first level when searching the query in database. There must be a controller that handles to query the input value and distributes it to the suitable level 2. The controller needs to calculate the hamming distance of the query F_q to every centroid $m_1, m_2, ..., m_K$ for finding closest cluster m_i [16, 21].

5.4 LSH Level (Second Level)

LSH also has an important contribution in both preprocessing and querying stages. In preprocessing stage, LSH is used for building the LSH hash table and in the second stage, LSH help to indicate the points that belong to one bucket [19, 20].

5.4.1 LSH Preprocessing

The number of hash functions is an important parameter of LSH, which directly affects the searching speed and accuracy of system. The trade-off of speed and accuracy can find the optimal solution when we know the size of database and average distortion of queries. The second level uses l hash functions by $g_j = (h_{j1}, h_{j2}, ..., h_{jk}), j \in [1; l]$ selected randomly from LSH hashing family, and builds l buckets $L_1, L_2, ..., L_l$ that have the same hash value for points $F_1, F_2, ..., p_n$. The address to every point in all buckets will be stored for querying step.

5.4.2 LSH Querying

In this stage, the LSH query handles independently in every device since each device has different sub-database and LSH hash table. For the query F_q, we address the respective bucket B by the same hash value by functions $g_j = (h_{j1}, h_{j2}, ..., h_{jk})$ in preprocessing step.

For every point $F_1, F_2, ..., F_n \in F$, we need to compute the hamming distance $d(F_q, F_j)$ to find which one meets the threshold $\mathbf{P_2}$ for returning the approximate near neighbor. In case none of the points $F_j \in P$ is not less than $\mathbf{P_2}$, our system will find again in l nearest buckets by changing 1 bit in bucket's hash string.

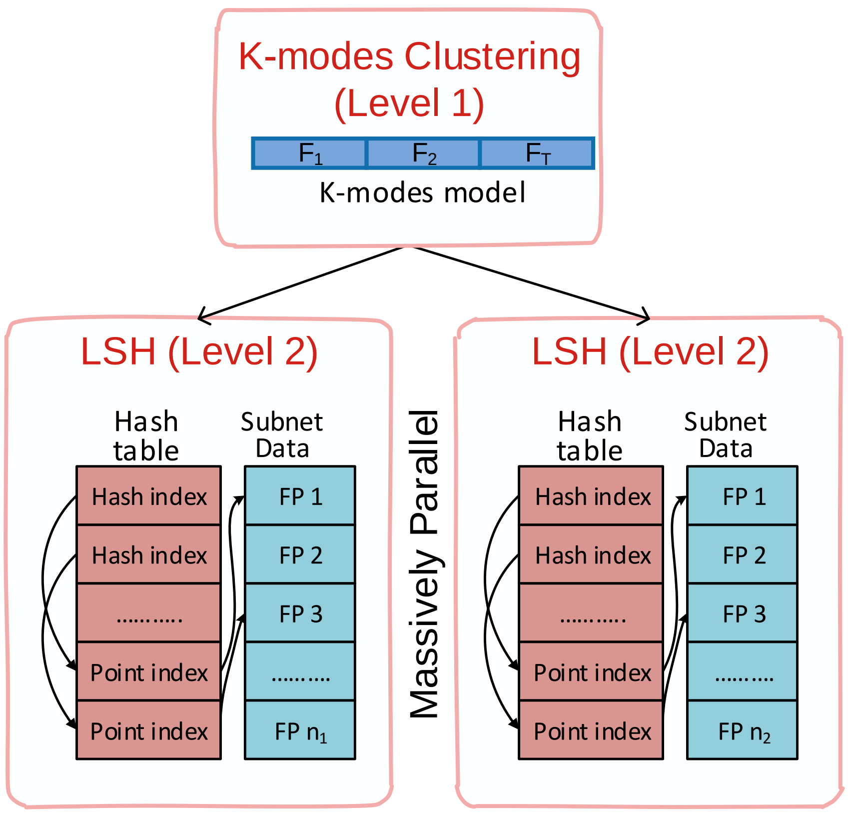


Figure 5.4: Overview Hierarchy Searching for Querying stage

In Figure 5.5, the system quite similar with the system using single GPPGU. But in this system, we need to have a parallel threads for management the query for each device. Two GPPGUs will have different queries and queue, that will increase the number of throughput queries and increase the total amount of database size. Each device also have its output array for storing the output ID for every query come to this device. The Last step is the merging the output data of GPGPUs devices to a global output ID's array.

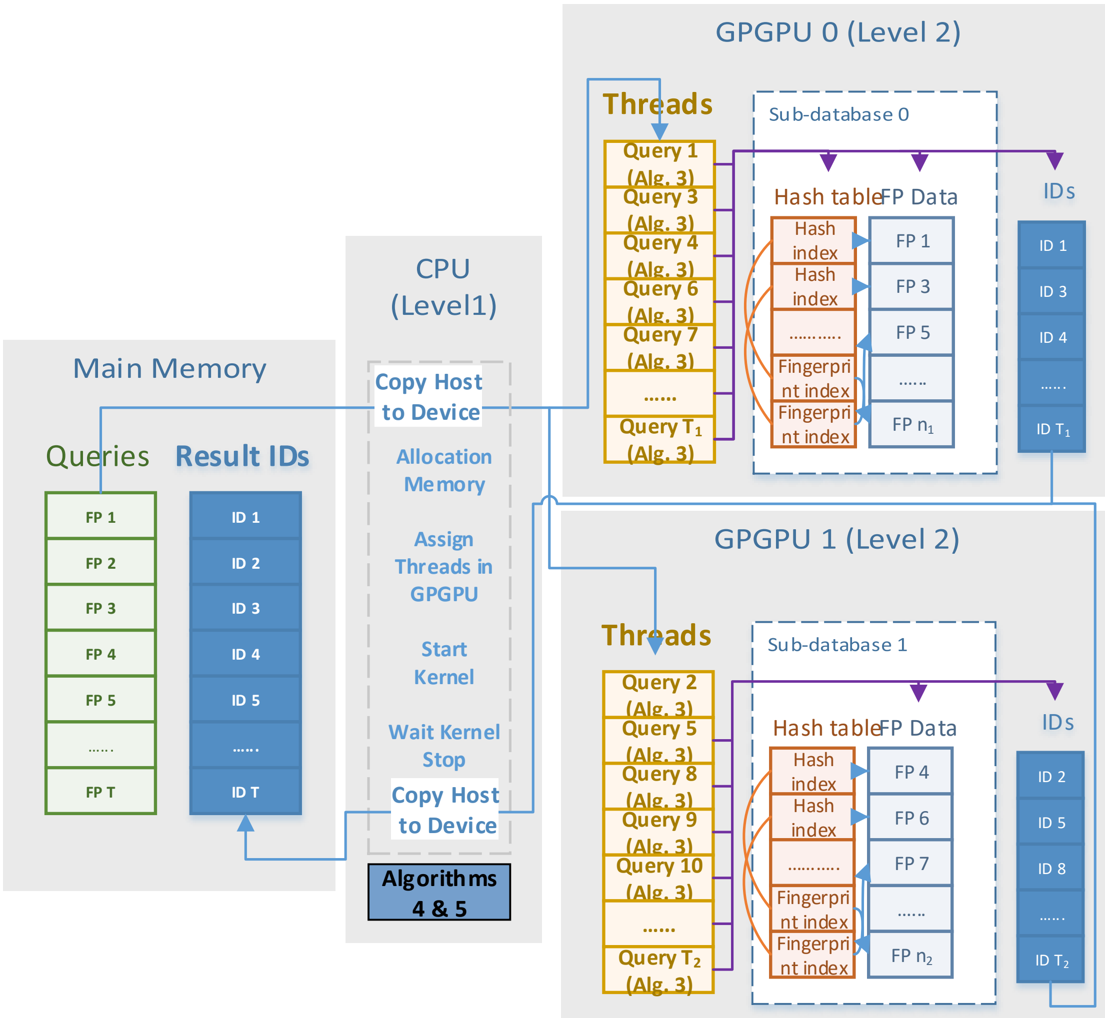


Figure 5.5: Hardware Overview Hierarchy Searching using 2 GPGPU devices

5.5 Algorithm for Audio Fingerprint Parallel Searching in multiple GPGPUs

Because we have multiple threads for a higher performance, the Deadlock will occur due to using of the same memory among threads. In algorithms 4 and 5 in Figure 5.6 and 5.7 we need to have MUTEX variables *MUTEX_IS_FULL_QUEUE* and *MUTEX_IS_KERNEL_RUNNING* for every GPGPU device to ensure that there is only one thread that can change the data in the queue or the global memory of GPGPGUs.

In algorithm 5, we also use the algorithm 2 for handling the multiple threads query for single GPGPU. We have to re-run the kernel for the last case when the main thread is stopped and there are remaining audio fingerprints in the device's queue.

Algorithm 4 K-modes Management Algorithm for Level 1 (Main Thread)

```
Require: Sub databases stored in GPGPU devices
  for size_t i = 0; i <TESTINGFINGERPRINTS->DATA_N; i++ do
    vector ← audio fingerprint query at i ;
    device_id ← KMODES->FindCluster(vector);
    if device_id != KMODES->FINGERPRINTS_INDEX[i] / KMODES->DATA_N
    then
      TOTAL_MISS++; // For counting the miss ratio
    end if
    while LEVEL2[device_id]->MUTEX_IS_FULL_QUEUE do
      Sleep(10);
    end while
    LEVEL2[device_id]->AddVector(vector, ExpectedID(vector))
  end for
  end_of_test ← true;
  return   The Result audio fingerprint ID for queries to array A
```

Figure 5.6: K-modes Management Algorithm for Level 1 (Main Thread)

Algorithm 5 K-modes Management Algorithm for Level 1 (Queries Thread)

```
Require: Sub databases stored in GPGPU device with ID = id
  while true do
    if LEVEL2[id]->MUTEX_IS_FULL_QUEUE then
      while LEVEL2[id]->MUTEX_IS_KERNEL_RUNNING do
        Sleep(10);
      end while
      LEVEL2[id]->CopyDataFromQueueToLSHLEvel2();
      LEVEL2[id]->StartTestDATA(); // Algorithm 2
      LEVEL2[id]->CalcTheAccuracyAndUpdate();
      LEVEL2[id]->QUEUE->Clear();
      LEVEL2[id]->MUTEX_IS_FULL_QUEUE ← false;
    else if end_of_test then
      while LEVEL2[id]->MUTEX_IS_KERNEL_RUNNING do
        Sleep(100);
      end while
      LEVEL2[id]->CopyDataFromQueueToLSHLEvel2();
      LEVEL2[id]->StartTestDATA(); Algorithm 2
      LEVEL2[id]->CalcTheAccuracyAndUpdate();
      LEVEL2[id]->QUEUE->Clear();
      LEVEL2[id]->MUTEX_IS_FULL_QUEUE ← false;
      break;
    end if
  end while
  return The Result audio fingerprint ID for queries to array A
```

Figure 5.7: K-modes Management Algorithm for Level 1 (Queries Thread)

Chapter 6

Evaluation

In this section, we show our system implementation using PC with Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz with two GPGPU devices Tesla K40m (13GB memory). Our database features are based on HiFP2.0 fingerprint of Yang with 4096 binary bits per vector. There are several variations of database from 100,000 fingerprints to 10,000,000 fingerprints. To evaluate the accuracy, we also choose the queries with different distortions for the experiments. We refer distortion by transformation of original audio by noise, aliasing, and flutter and distortion ratio is the percentage of error bits in audios. For level 2 implementation, we use Compute Unified Device Architecture (Cuda) as platform for the parallel blocks and threads to search the approximate nearest fingerprint in GPGPU devices[4]. We also compare our performance result with the original LSH and related works.

6.1 Experiment Design

At the initialization, the subnet databases are loaded into devices' memory and the K-modes model is loaded into controller. In Figure 6.1, our system has a controller in CPU that handles the K-modes clustering the queries. For the number of GPGPUs $K = 2$ we need to use 2 threads for managing the flow of each GPGPU. Due to the operation of Cuda, we need to use the number of query threads equal to the number of cores in GPGPU [5]. Hence, there are K queues for K GPGPUs for getting a large number of queries. Besides massively parallel among GPGPUs, our system also has good performance when searching multiple queries in parallel for each device.

The K-modes thread is responsible for getting the queries and computing the nearest centroid in model. After that, every query will be added to the corresponding queue. The queue size of each GPGPU depends on the number of cores of its GPGPU. If the current queue is full, then the K-modes thread need to sleep to wait for that its thread finishes current kernel. For the LSH's threads at the second level, it only starts the kernel only if its queue is full. When the kernel is started, its queue will be clear immediately. So when the kernel is working in GPGPU, the K-modes can add the queries to its queue in parallel. In case the queue is not full and the current GPGPU is not working, the LSH

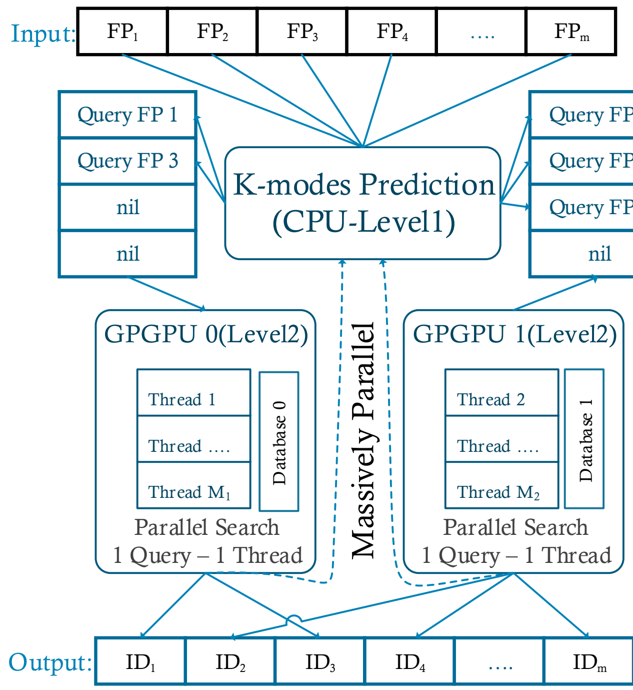


Figure 6.1: Hierarchy Searching for 2 GPGPUs

thread needs to sleep and check its queue again. Every query transferred to device will be marked the index in main memory and there is a list to store the meta data for every query. After finishing kernel in every GPGPU, the meta data will be copied to the list for retrieval or evaluation.

6.2 Result of Parallel Audio Fingerprint Searching using Single GPGPU

From Table 6.1, 6.2, we can see the size of hash table is greater than the raw size of fingerprints. However, when the number of fingerprints is large, the size of the hash table is approximately equal to the raw size. Hence, the total size of database is simply double the raw fingerprint size and can be calculated with the following equation:

Table 6.1: Raw Data size of different amounts of Fingerprints

Number of FPs	1,000	10,000	100,000	1,000,000	10,000,000
Raw Size	500KB	4.88MB	48.828MB	500MB	5GB

Table 6.2: Hash table size for different amounts of Fingerprints and different hash function number

Number of FPs	Number of Hash Functions							
	10	12	14	16	18	20	22	24
1,000	497KB	509KB	557KB	749KB	1.4814MB	4.4814MB	16.49MB	46.48MB
10,000	4.81MB	4.82MB	4.86MB	5.05MB	5.8MB	8.8MB	20.8MB	68.8MB
100,000	48.06MB	48.06MB	48.12MB	48.31MB	49.06MB	52.06MB	60.06MB	112MB
1,000,000	500MB	500MB	500MB	500MB	500MB	500MB	500MB	500MB
10,000,000	5GB	5GB	5GB	5GB	5GB	5GB	5GB	5GB

Table 6.3: Database and Queries Transfer time (millisecond) from Host to Device

Number of FPs	Number of Hash Functions							
	10	12	14	16	18	20	22	24
1,000	3	20	22	30	57	156	548	2,082
10,000	336	178	178	188	213	313	699	2,223
100,000	3,206	1,661	1,654	1,684	1,697	2,206	2,184	4,572
1,000,000	1,080	1,081	1,062	1,065	1,070	1,080	1,080	1,113
10,000,000	212,596	210,257	244,795	163,674	167,848	268,657	240,226	218,633

$$HashTableSize = RawSize * 126 * sizeof(int) + (2^{hbits}) * sizeof(int)$$
$$DatabaseSise = HashTableSize + RawSize$$

The transfer time from host to GPGPU is showed in Table 6.3, the transfer time does not depend much on the number of hash function. It only depends on the size of the database or the number of fingerprints. Since a number of test queries are the same on every case of database size. So in Table 6.4, the amounts of transfering time for output fingerprint IDs are similar. In Table 6.5, we calculate the average searching time for a single query. For different database size, we should have a suitable number of hash functions for hashing data. For example, if we have 1,000 fingerprints data, we should choose 18 hash functions. But when dealing with 10,000,000 fingerprints data, we prefer

Table 6.4: Result Transfer time (millisecond) from Device to Host

Number of FPs	Number of Hash Functions							
	10	12	14	16	18	20	22	24
1,000	0.39	0.38	0.38	0.38	0.38	0.42	0.42	0.47
10,000	0.56	0.55	0.53	0.53	2.07	0.53	0.54	0.57
100,000	0.62	0.61	0.61	0.6	0.61	0.65	0.6	0.64
1,000,000	1.20	1.12	1.10	1.16	1.16	1.16	1.18	1.20
10,000,000	4.98	4.92	4.9	4.90	4.90	4.92	4.90	5.00

Table 6.5: Average Searching time (millisecond) for single query using single GPGPU

Number of FPs	**Number of Hash Functions**							
	10	**12**	**14**	**16**	**18**	**20**	**22**	**24**
1,000	0.09	0.07	0.05	0.04	0.03	0.03	0.03	0.03
10,000	0.22	0.10	0.05	0.04	0.03	0.03	0.03	0.03
100,000	1.60	0.46	0.14	0.07	0.04	0.03	0.03	0.03
1,000,000	3.85	2.54	1.52	0.06	0.17	0.07	0.04	0.04
10,000,000	102.50	45.80	20.20	7.80	1.90	1.80	0.17	0.06

to choose 24 hash functions.

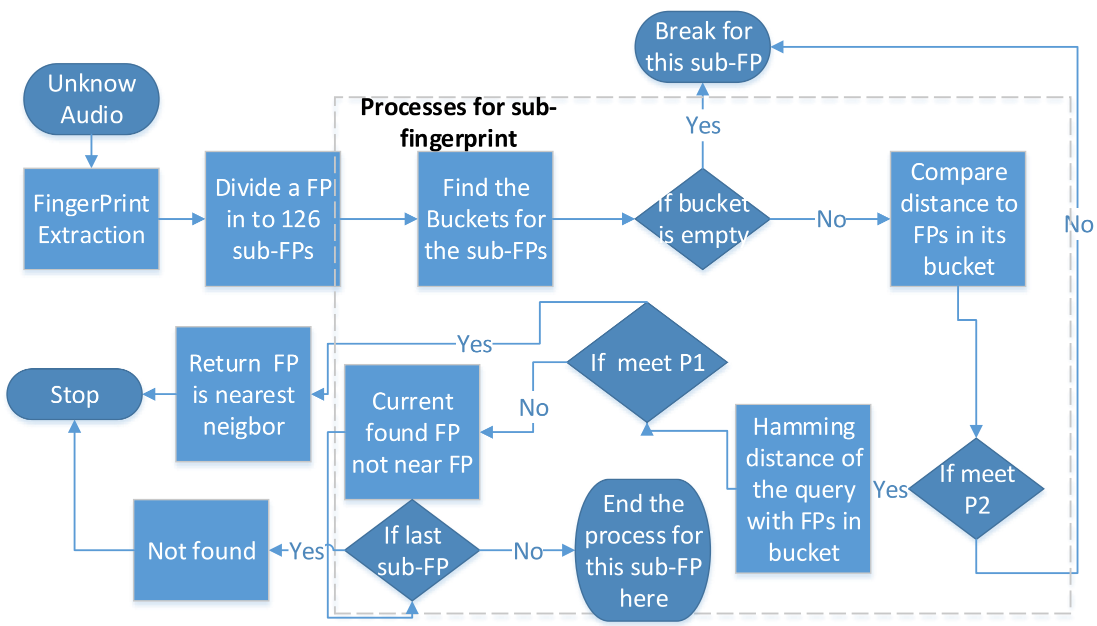


Figure 6.2: Flowchart of LSH for finding the cR-near neighbor for a query.

In Figure 6.3, we tested our method using single GPGPU using different size of data with the GPGPU detail in Table 6.6. Because the number of queries are same so the size of data copied from host to device and data's size of IDs copied from device to host are same for any database. So the transfer time are similar for every size of database. We can see, the executed time depend much on the database size and the number of hash functions. For different size of data we should choose different hash function number for getting the best performance. For example, if we have 10,000 audio fingerprint in the database we should choose to use 20 hash functions. But when dealing with database have 10,000,000 audio fingerprints, we should use 24 hash functions for getting the best result.

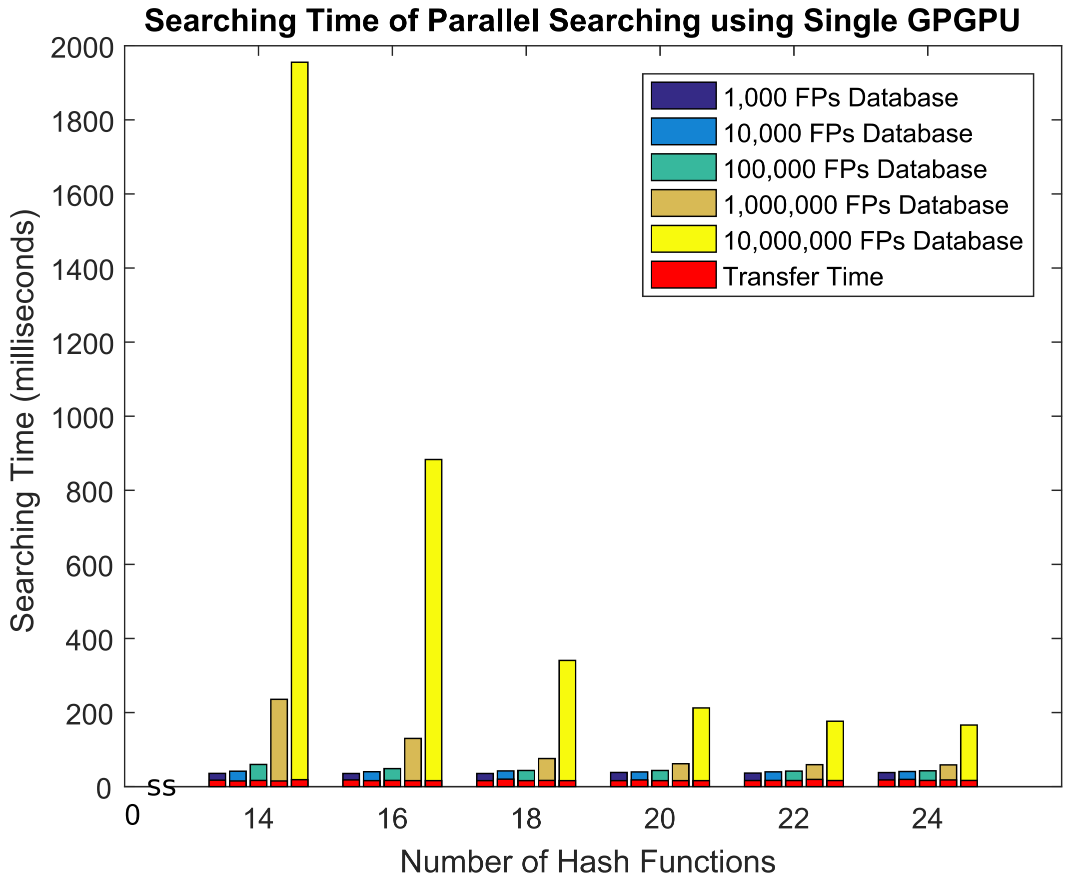


Figure 6.3: Transfer and Executed Time (Milliseconds) using single GPGPU (1024 Queries)

However, using same conditions with the testing on Figure 6.3, but in Figure 6.3 the accuracy depend much on the number of hash functions. With the higher of hash function number, the accuracy of system will be reduced. It is can easily explain by the number of buckets affected by number of hash functions. With more buckets, the average number of audio fingerprint in a single bucket will be reduces. In this case, the chance for getting the approximate nearest neighbor will also reduced. We can see with 20 hash functions it can archive the highly accuracy with acceptable search time. for the next coming result we will use 20 hash functions for other testing.

In the Figure 6.5, following the result above, we implement Yang's and our method with the same data and queries but different environment followed by the Table 6.6. The bars represent CPU_TIME/GPU_TIME in different conditions which are different fingerprint size and a different number of hash function. Not that, The searching time of our method already including the executed time and transfer time. Our method and the Yang's method have the same condition, the same database and also the same queries inputs. For database with 10 million fingerprints, our method can get the cR-near neighbor of 1024 queries in 170 milliseconds.

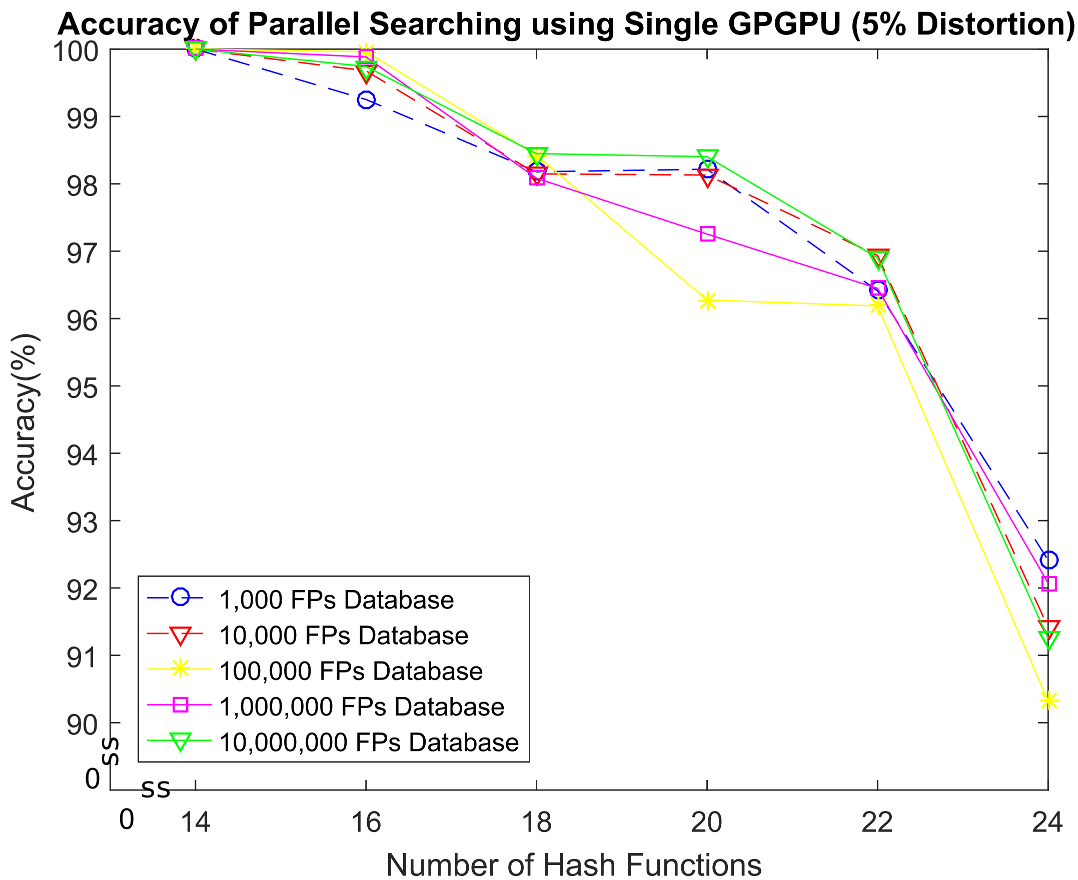


Figure 6.4: Accuracy using single GPGPU (1024 Queries - 5% Audio Fingerprint Distortion)

Table 6.6: Detail of CPU and GPPGU information are used for comparison

	CPU	**GPGPU**
Name	Intel Xeron E5-2620 v2	Tesla K40
Frequency	2.1 GHz	745 MHz
Memory	62 GB	13 GB
Language	C	Cuda
Queries	1024	1024
Threads	1	1024
Compiler	gcc 4.3	nvcc 7.0
OS	CentOS 6.4	CentOS 6.4

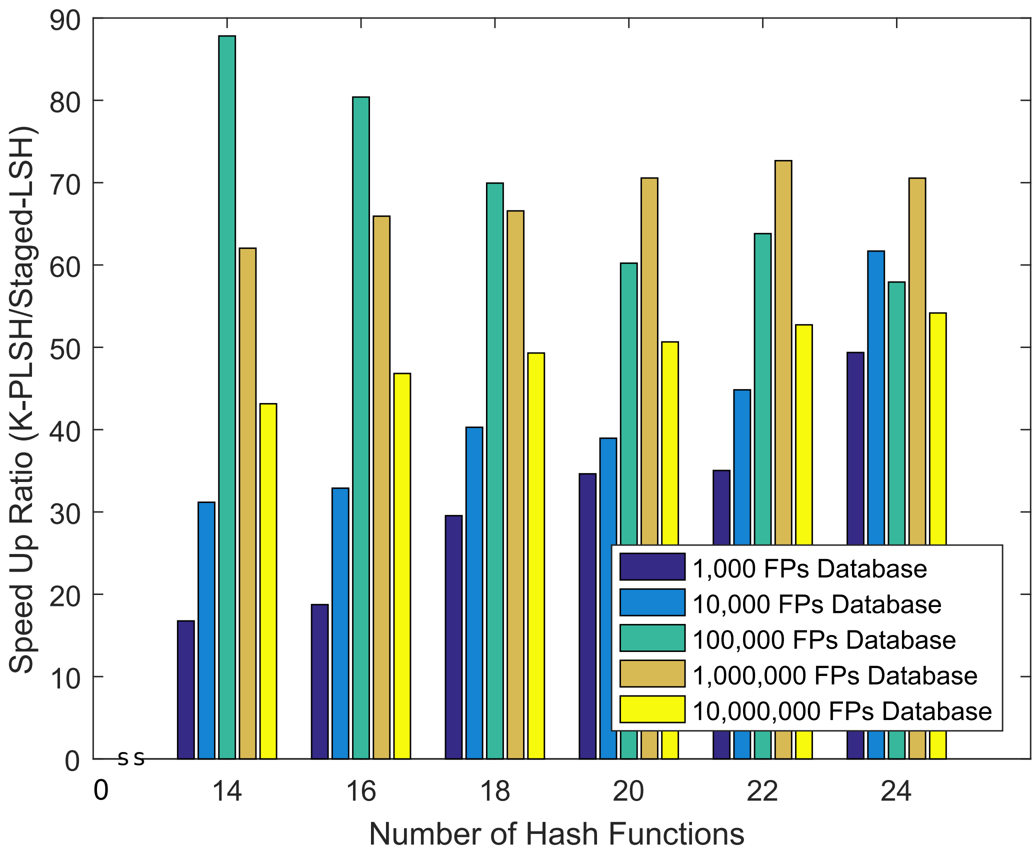


Figure 6.5: The optimization times of our method with the original.

6.3 Result of Parallel Audio Fingerprint Searching using Multiple GPGPUs

In Table 6.7, the flow of preprocessing includes two main steps : Clustering using K-modes and Generating hash table using LSH hash function family. Preprocessing time is one of drawback of our system. It takes almost one day to finish process of handling 10 million fingerprints database. However after preprocessing step, the database will be good organization and optimal for searching.

In Table 6.8, the size of sub-databases after preprocessing step are nearly equal for two clusters. For problem of dividing into 4 clusters, we get the problem for dividing size equally. In that case, we propose a method called Extended K-modes for adding a new condition to K-modes and forcing all clusters that have a limited number of vectors and we accept the reduced accuracy. The algorithm for Extended K-modes can be seen in Appendix A. In Table 6.9, we showed the detail information of GPGPGU devices we used for implementing our experiments.

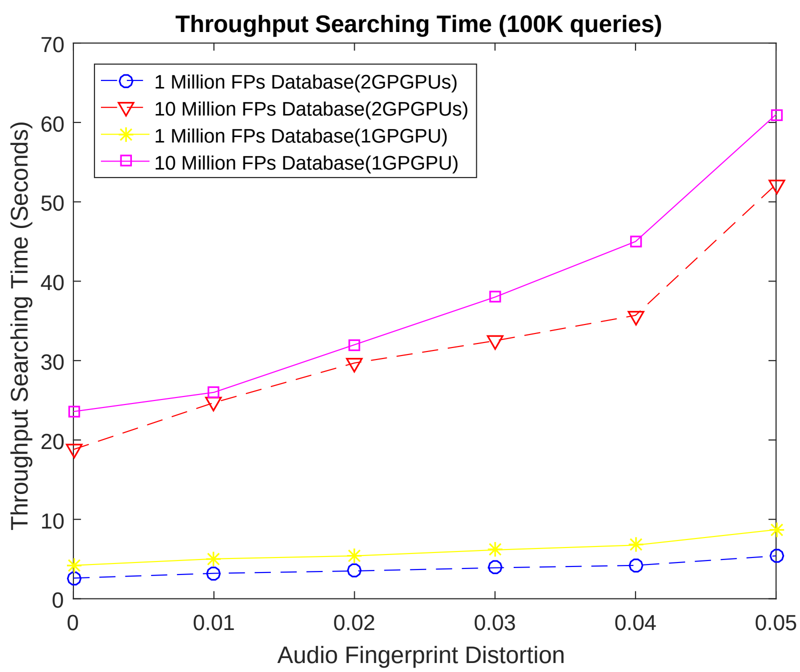


Figure 6.6: Result: Searching time of hierarchy searching follow the changing of database size and distortion ratio

Our proposed system focuses more on searching time. Figure 6.6 shows that our sys-

Table 6.7: Preprocessing time (millisecond) for clustering database into 2 dub-databases

Number of FPs	1,000	10,000	100,000	1,000,000	10,000,000
Preprocessing Time (millisecond)	3,134	31,387	424,520	4,219,329	63,173,936

Table 6.8: Sub-Databases size after preprocessing step

Number of FPs	**Sub-database size**	
	sub 1(KB)	**sub 2(KB)**
300	93	57
1,000	259	241
10,000	2,443	2,557
100,000	5,363	2,463
1,000,000	253,228	246,773
10,000,000	2,499,615	2,500,385

Table 6.9: GPPGUs information are used for testing the K-PLSH

	GPGPU
Name	Tesla K40
Frequency	745 MHz
Memory	13 GB
Language	Cuda
Cores	2880
Threads	2880
Compiler	nvcc 7.0
OS	CentOS 6.4

tem achieves impressive performance when getting the approximate nearest neighbor for 100,000 queries (10% distortion) in 110 seconds. Distortion and database size is two factors that affect directly searching time. Database size decides the buckets size, then the big database leads to a large number of audio fingerprints in buckets. It makes LSH need time to calculate the hamming distance between query and every fingerprint in each bucket. In addition to that, with the distorted queries, the distance of query to fingerprints will be affected by distortion and LSH needs to search in different buckets for the current query.

Figure 6.7 demonstrates that the accuracy of FingerPrint hierarchy is not affected by the database size. Accuracy depends on the distortion of query. Query with higher distortion can lead hash functions to have different values due to the error bits in query. That is the reason why it is harder to find the approximate nearest fingerprint in the error hash value.

In Figure 6.8, when the system starts, the subnets database needs to copy from main memory to devices' memory. By using 2 GPGPUs at the same node, we can transfer data by taking advantages of speed of PCI-Express serial expansion bus. The database with 10 million fingerprints (10GB) can be transferred to 2 devices in 4 seconds.

In our system, for the better performance K-modes thread and LSH threads need to sleep until other threads are done or queue is full. Figure 6.9 and Figure 6.10 give information about how many threads in our system wait for each other. It is important

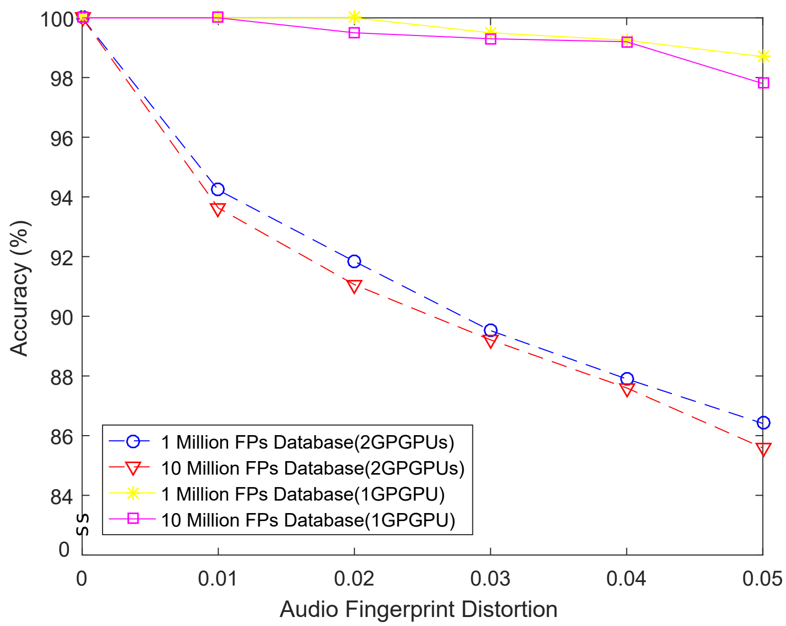


Figure 6.7: Result: Accuracy

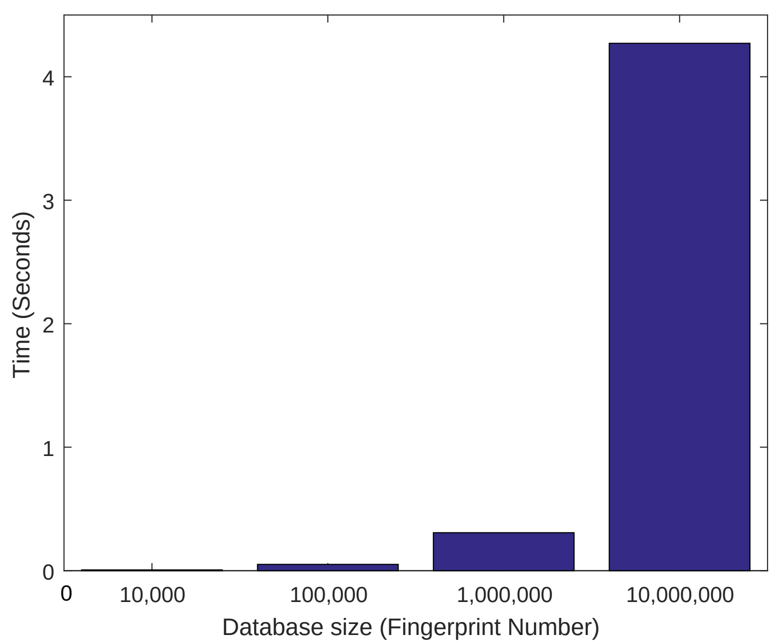


Figure 6.8: Result: Database Transfer Time

to note that the total sleeping time of threads is not the searching time of system as the threads may sleep at the same time.

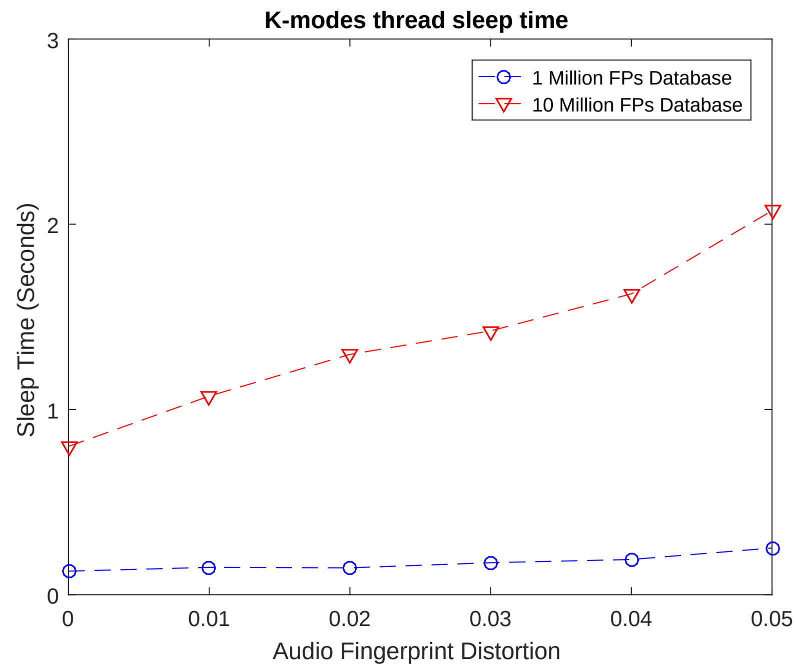


Figure 6.9: Result: CPU Sleep time

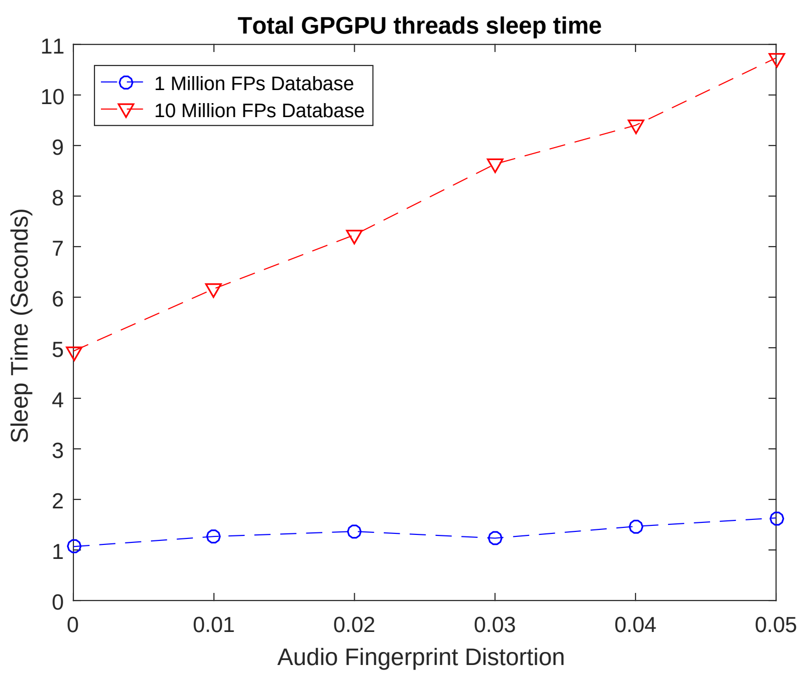


Figure 6.10: Result: GPGPU Sleep time

Until now, we have just discussed the accuracy of system caused by the error of LSH functions with distortion query. Figure 6.11 shows that the accuracy of hierarchy searching

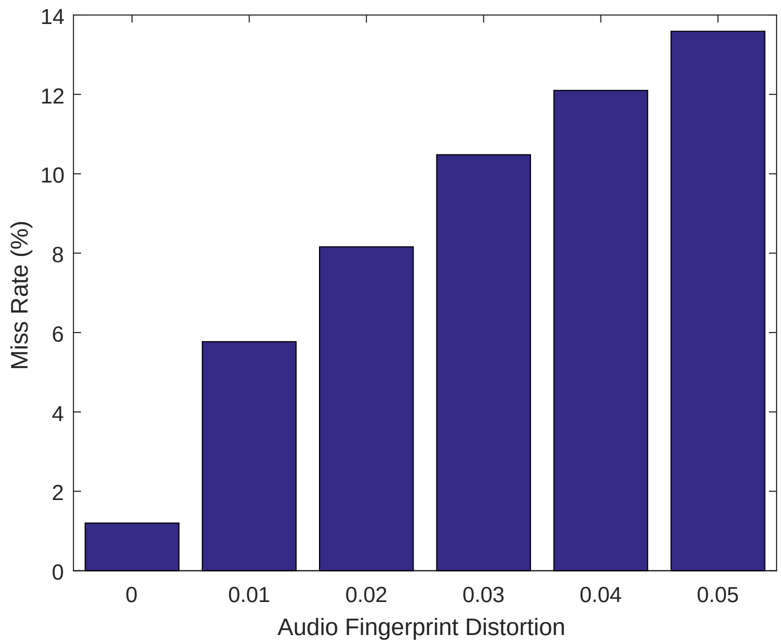


Figure 6.11: Result: Miss Ratio

system is also affected by level 1 because of the error of K-modes cluster. When the query is changed by error bits, the hamming distance from this to centroids is also changed, which leads to the wrong choice of cluster/GPGPU for continuing second level stage. We can see that the miss rate is proportional to the distortion ratio.

6.4 Comparison Results

The comparison with other parallel system using single device is already discussed in Table 6.6. We also implement other method which using the same Level 2 but different Level 1 from our method for evaluating strengths of K-modes than other cluster methods.

In Figure 6.12 and 6.13, we show the comparison of Accuracy and Searching Time for different methods of Level 1. ORIGINAL method is the method which uses only single GPGPU for searching. We use only 1 hash function for sensitive hashing in HASH method. HASHBITS method uses 16 hash function and divide them into 2 groups by hamming distance. The FKMODES is Fuzzy K-Modes which uses a fuzzy multi-cluster for every point to every centroid. We can see in term of accuracy, K-modes method can almost achieve the upper bound of the original method. And in Figure 6.13, the searching time of K-modes does not gap with other methods.

The standard LSH algorithm is good method for ε-NNS problem, but this algorithm does not work well with the particular forms of database. With the high distortion query, the standard LSH is easy to skip the good bucket by the error bits. In our method, when the current buckets have few data points or the distances are far from the query, our

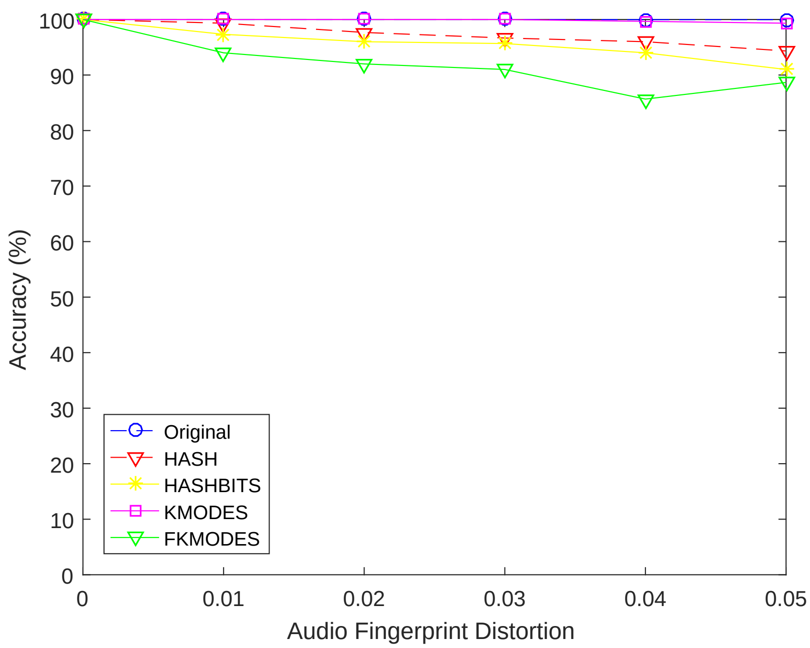


Figure 6.12: Result: Comparing The Accuracy when using different Level 1'method

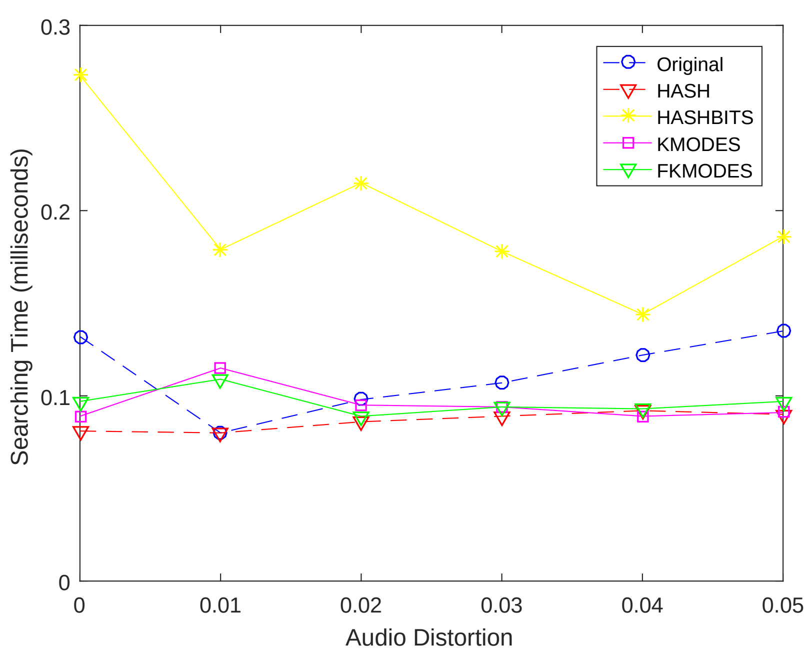


Figure 6.13: Result: Comparing The Searching Time when using different Level 1'method

algorithm tries to search in near buckets by changing the 1 bit of hash value. Our method is similar to using E8-lattice for finding the similar buckets but we show a simple way for indicating the nearest buckets for binary vectors such as HIFP2.0's audio fingerprint. The functions of standard LSH is deterministic, so when the source vector has high dimensions and the number of hash functions is small, then it is easy to lose the information. In our implementation, the fingerprint has 4096 bits, but the hash value has 20bits. So if we use standard LSH, we will lose information of 4076 bits. To avoid this problem, we divide the source vector into 126 sub-vector of 96 bits by overlapping 64 bits of every sub-vector. This helps to reduce the skipped bits of source vector, and it also increases the accuracy of the system by using the multiple chances of finding potential buckets by 126 times.

Compared with PLSH, our method can work on massively parallel system using multiple GPGPUs for taking advantages of power of graphics devices. Besides that, PLSH has threads on CPUs and handles the database files at the same time. Moreover, for the real big database like Tweets's, 1000 queries at the same time are not enough in real cases. Our experiments aim is to handle an extreme number of queries, so we test with 100000 queries in parallel.

Compared with the Bi-level Locality sensitive Hashing, our method uses K-modes instead of random projection tree. For the binary vector, RP-tree first tends to reduce the dimensions of vectors. On the contrary, in case of fingerprint, we will not reduce the dimensions of vectors to reduce the information loss.

Chapter 7

Conclusion and Future work

7.1 Conclusion

Hierarchy Searching on Massively Parallel with multi-GPGPUs can meet requirements of real database (1 millisecond per query for 10 million fingerprint database) when nowadays there are millions of contents of audios/videos uploaded to the Internet per day. Our method can search thousands of queries in parallel, it is suitable for a retrieval system using GPGPUs. Hierarchy structure helps the system work in supercomputer/PC cluster with many GPGPU devices. With the searching speed and database storing strategy, our method can be compatible with the real data of millions songs/tracks and the searching time can meet difficult requirements of real world's cases.

With taking advantages of total cores of GPGPUs in wholes, our method not only make a massively parallel in GPGPUs, but also take leverage on the capabilities of multi-task of CPU. CPU works as the cluster to distribute queries and control the kernel of all GPGPUs. With multiple threads on CPU, the thread of K-modes and threads of GPGPUs can work at same time. Technically, our method can achieve massively parallel for both CPU and GPGPUs.

7.2 Future work

7.2.1 Current Problems

1. HiFP2.0 of Yang is a good algorithm for extracting the fingerprint, especially with distorted waveform. But that is not a good algorithm for extracting the audio fingerprint for edited/cut audio, which makes HiFP2.0 very vulnerable to hack by shifting/cutting the original audio. We want to build our system that can handle with all cases of audio transformation for adapting to the real data [2].

2. The calculation of hash table is unique, for each database we have to calculate the hash table once. The problem is when we have a new song or fingerprint, we need to re-calculate the hash table again and load the data to the device again. It is a

big problem with LSH. In the real life, the database should be updated hourly. In addition, K-modes is also a static method for clustering and takes a long time for dividing a big database (9 days for 10 million fingerprints database). So, the cluster should be automatically changed when data is changed to reduce miss ratio [2, 21].

3. In addition, we already built a massive parallel system with multiple Cuda devices. When we increase the number of devices, the miss rate also increases. Miss rate is a measure of evaluating whether a query is sent to right cluster or not. It is greatly affected by the accuracy of the whole system.

7.2.2 Solutions for Future work

1. For easy attack of HiFP2.0 : There are several audio fingerprint extraction algorithms which optimize the memory of database and is based on the content base. We consider all of the advantages of these algorithms and propose a new method that is the most suitable with memory structure of parallel processing.

2. For the static database structure: It is very important to extend the LSH to a dynamic structure. The pointer for every bucket should have several fragmentation structures. It will help add or remove the fingerprints if needed. The cluster of each device should be updated automatically when its data is changed and it should interact with other devices. For example, when a device is full, it need to send the irrelevant fingerprints to other devices [23].

3. In term of miss rate: We can not avoid the miss occurs when our system has many devices. The only way is dealing with miss case when it already happens. We propose to use new structure for organizing the clusters by distance. It will be helpful when we detect a missing case, we should send it to near clusters by distance. To do that, we should choose a good measure for measuring the distance between clusters.

Bibliography

[1] Fan Yang, Yukinori Sato, Yiyu Tan and Yasushi Inoguchi. *Searching Acceleration for Audio Fingerprinting System. Joint Conference of Hokuriku Chapters of Electrical Societies, 2012*

[2] V. K. Jain K. Araki, Y. Sato, and Y. Inoguchi. *Performance evaluation of audio fingerprint generation using haar wavelet transform*. International Workshop on Nonlinear Circuits, Communication and Signal Processing, 2011

[3] Pedro Cano and Eloi Batlle. *A Review of Algorithms for Audio Fingerprinting. 9-11 Dec. 2002 - 0-7803-7713-3*

[4] Rafia Inam. *An Introduction to GPGPU Programming- CUDA Architecture.*

[5] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. *A Comprehensive Performance Comparison of CUDA and OpenCL. 13-16 Sept. 2011. 978-1-4577-1336-1*

[6] Alexandr Andoni, and Piotr Indyk. *Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. Oct. 2006. 0-7695-2720-5*

[7] Jaap Haitsma, and Ton Kalker. *A Highly Robust Audio Fingerprinting System. Journal of New Music Research, Vol. 32(2003), No. 2, p. 211-222*

[8] Cen, Wei, and Kehua Miao. *An improved algorithm for locality-sensitive hashing Computer Science & Education (ICCSE), 2015 10th International Conference on. IEEE, 2015.*

[9] Chang, and Edward Y. *Approximate High-Dimensional Indexing with Kernel. Foundations of Large-Scale Multimedia Information Management and Retrieval. Springer Berlin Heidelberg, 2011. 231-258.*

[10] Pan, Jia, Christian Lauterbach and Dinesh Manocha. *Efficient nearest-neighbor computation for GPU-based motion planning. Intelligent Robots, and Systems (IROS), 2010 IEEE/RSJ International Conference on. IEEE, 2010.*

[11] Jia Pan, and Dinesh Manocha. *Bi-level Locality Hashing of K-nearest Neighbor Computation. 1-5 April 2012. 978-1-4673-0042-1*

[12] Pan, Jia, and Dinesh Manocha. *Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. Proceedings of the 19th ACM SIGSPATIAL international conference on advances in geographic information systems. ACM, 2011.*

[13] Narayanan Sundaram, Aizana Turmukhametova, Nadathur Satish, Todd Mostak, Piotr Indyk, Samuel Madden, and Pradeep Dubey. *Stream Similarity Search over one Billion Tweets using Parallel Locality-Sensitive Hashing.Journal Proceedings of the VLDB Endowment Volume 6 Issue 14, September 2013 Pages 1930-1941*

[14] Garcia, Vincent, Eric Debreuve, and Michel Barlaud. *Fast k nearest neighbor search using GPU. Proceeding Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on. IEEE, 2008.*

[15] Tzortzis, Grigorios, and Aristidis Likas. *The global kernel k-means clustering algorithm. 2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence). IEEE, 2008.*

[16] Joshua Zhexue Huang *Clustering Categorical Data with k-Modes . Encyclopedia of Data Warehousing and Mining, Second Edition*

[17] Zhe Huang, and Michael K. Ng *A Fuzzy k-Modes Algorithm for Clustering Categorical Data. IEEE TRANSACTIONS ON FUZZY SYSTEMS, VOL. 7, NO. 4, AUGUST 1999*

[18] Sanjoy Dasgupta, and Yoav Freund. *Random projection trees and low dimensional manifolds. Proceeding STOC '08 Proceedings of the fortieth annual ACM symposium on Theory of computing. USA 2008. ISBN: 978-1-60558-047-0*

[19] Herv Jgou, Laurent Amsaleg, Cordelia Schmid, and Patrick Gros *Query-Adaptive Locality Sensitive Hashing . March 31 2008-April 4 2008. 978-1-4244-1483-3*

[20] Francois Fleuret *Fast Binary Feature Selection with Conditional Mutual Information . Journal The Journal of Machine Learning Research. Volume 5, 12/1/2004 Pages 1531-1555*

[21] Zhexue Huang. *Extensions to the K-Means Algorithm for Clustering Large Data Sets with Categorical Values . Data Mining and Knowledge Discovery, Volume 2, Issue 3, pp 283-304*

[22] Piotr Indyk, and Rajeev Motwani. *Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality . Proceeding STOC '98 Proceedings of the thirtieth annual ACM symposium on Theory of computing Pages 604-613 ISBN:0-89791-962-9*

[23] Anirban Dasgupta, Ravi Kumar, and Tams Sarls. *Fast Locality-Sensitive Hashing . Proceeding KDD '11 Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining Pages 1073-1081. ISBN: 978-1-4503-0813-7*

[24] Chua Hock-Chuan. *3D Graphics with OpenGL Basic Theory . http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz4BuBBAY8M 2016/6/22*

[25] NVIDIA corporation. *CUDA Parallel Computing Platform. http://www.nvidia.com/object/cuda_home_new.html 2016/6/22*

[26] NVIDIA. *Parallel Thread Execution ISA Version 4.3. http://docs.nvidia.com/cuda/parallel-thread-execution/#axzz4BuBBAY8M*

[27] J. S. Seo, Minho Jin, Sunil Lee, and Dalwon Jang *Audio fingerprinting based on normalized spectral subband centroids . 18-23 March 2005 ISSN :1520-6149 Print ISBN:0-7803-8874-7*

[28] NVIDIA. *How Content ID works. https://support.google.com/youtube/answer/2797370*

[29] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. *GPGPU PROCESSING IN CUDA ARCHITECTURE . 20 Feb 2012, arXiv:1202.4347v1*

[30] Jain, and Anil K. *Data clustering: 50 years beyond K-means. Pattern recognition letters 31.8 (2010): 651-666*

Appendix A

Extended K-modes for achieving the desired-size clusters

In this thesis, we also propose a new extension for K-modes for clustering the database in to sub-database with different size of memory. pseudocode:

Algorithm 6 Algorithm for Extended K-modes for achieving the desired-size clusters

Require: Database, K, limited-size of clusters
  Initialization:
  Sort the points by the delta of distance from nearest to farthest cluster;
  Sort the cluster by distance from nearest to farthest cluster for every points;
  For the points in sorted list: Assign the label by the sorted cluster if desired cluster is not full;
  Loop:
  while centroids change and not reach maximum loop number **do**
    Re-update the centroid cluster by mean vector;
    Sort the points by the delta of distance from nearest to current cluster;
    for For every points if delta not equal 0 **do**
      Sort the cluster by distance from nearest to farthest cluster;
      if desired cluster is not full **then**
        Assign new label for this point;
      end if
      if desired cluster is full **then**
        if Exist from target cluster prefer to join current point's cluster **then**
          Swap labels for these points ;
        end if
      end if
    end for
  end while
  return K Sub-database with limited-size of clusters

Figure A.1: Algorithm for Extended K-modes for achieving the desired-size clusters

Appendix B

Source Code: Searching on Single GPGPU (Level2)

```

#include "TestingFingerPrints.h"
#include "LSHQueue.h"


class Level2GPU
{
private:

  unsigned int   dev_fp_array_size, dev_ht_array_size;
  unsigned int   *dev_fp_array, *dev_ht_array;
  unsigned int* fp_testing, *dev_fp_testing;
  unsigned int qfp_array_size;
  unsigned int* dev_result_array;
  unsigned int* dev_test_array, *test_array;
  unsigned int* expected_id;

  //TestingFingerPrints *testingfingerprint;
  int DEVICE_ID;
public:
  unsigned int ht_array_size, fp_array_size, num_test;
  unsigned int *fp_array, *ht_array, *result_array;
  unsigned TOTAL_test, TOTAL_test_right;
  float TOTAL_SEARCHING_TIME, TOTAL_TRANFER_TIME, TOTAL_TRANFER_DATABASE_TIME,
    TOTAL_LOAD_DATABSE_TIME;

public:
  Level2GPU();
  ~Level2GPU();
  Level2GPU(int device_id);
  bool LoadData(char* database, char* hashtable);
  int TestOriginalMethod(unsigned int* query, unsigned int hbits, int id_test);
  int TestOriginalMethod_BruceForce(unsigned int* query, unsigned int, int id_test);
  void FastTestOriginalMethod( int hbits, int num_test );
```

```
  void FastTestOriginalMethod_BruceForce(int hbits, int num_test);
  void FastTestOriginalMethod();
  void FastTestOriginalMethod_BruceForce();
  void PrintInfomation();
  void CopyDataToDevice(int device_id);
  void SetTestingData(TestingFingerPrints *test_fingerprints);
  void SetTestingQueueData(LSHQueue *lshqueue);
  void CopyTestingDataTODevice();
  void LSH_Cuda_1_Thread();
  void CopyResultArrayandTestArrayToHost();
  void LSH_Cuda_Mutiple_Threads();
  void CalcTheAccuracyAndUpdate();
  //testing and more information
  void Test_CopyTestingDataTODevice();
  void Test_CopyDataToDevice(int device_id);
  void ShowCudaDevicesInfo();
  unsigned int test_hd(unsigned int i1[], unsigned int i2[], int num);
};
```

Listing B.1: Source code Level2GPU's header file

```
#include "Level2GPU.cuh"
#include <string.h>
#include <stdio.h>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <ctime>
#if __linux__
#include <sys/time.h>
#endif

__device__ const int hfunc_dev[] = { 53, 49, 45, 60, 2, 72, 14, 82, 62, 46, 35, 95,
    43, 50, 0, 77, 28, 88, 13, 10, 65, 54, 29, 93, 24, 74, 23, 90, 75, 58, 56, 21,
    15, 27, 68, 64, 33, 42, 94, 48, 9, 73, 5, 25, 19, 7, 69, 34, 89, 4 };
__device__ unsigned int hd_dev(unsigned int i1[], unsigned int i2[], int num)
{
  int i;
  unsigned int xor2;
  unsigned int hd = 0;
  for (i = 0; i < num; i++) {
    xor2 = i1[i] ^ i2[i];
    xor2 = (xor2 & 0x55555555) + ((xor2 >> 1) & 0x55555555);
    xor2 = (xor2 & 0x33333333) + ((xor2 >> 2) & 0x33333333);
    xor2 = (xor2 & 0x0F0F0F0F) + ((xor2 >> 4) & 0x0F0F0F0F);
    xor2 = (xor2 & 0x00FF00FF) + ((xor2 >> 8) & 0x00FF00FF);
    xor2 = (xor2 & 0x0000FFFF) + ((xor2 >> 16) & 0x0000FFFF);
    hd += xor2;
  }
  return hd;
}
unsigned int lsh(unsigned int *query, int hbits)
{
  unsigned int hash;
```

```
  int i;
  hash = 0;
  for (i = 0; i < hbits; i++) {
    hash <<= 1;
    if (hfunc[i] < 32) {
      hash |= (query[2] >> hfunc[i]) & 1;
    }
    else if (hfunc[i] < 64) {
      hash |= (query[1] >> (hfunc[i] - 32)) & 1;
    }
    else {
      hash |= (query[0] >> (hfunc[i] - 64)) & 1;
    }
  }
  return hash;
}
__device__ unsigned int lsh_dev(unsigned int *query, int hbits)
{
  unsigned int hash;
  int i;

  hash = 0;
  for (i = 0; i < hbits; i++) {
    hash <<= 1;
    if (hfunc_dev[i] < 32) {
      hash |= (query[2] >> hfunc_dev[i]) & 1;
    }
    else if (hfunc_dev[i] < 64) {
      hash |= (query[1] >> (hfunc_dev[i] - 32)) & 1;
    }
    else {
      hash |= (query[0] >> (hfunc_dev[i] - 64)) & 1;
    }
  }
  return hash;
}
Level2GPU::Level2GPU()
{
  TOTAL_test = TOTAL_test_right = 0;
}
#include <cuda.h>
#include <stdio.h>
Level2GPU::~Level2GPU()
{
  result_array = NULL;
  test_array = NULL;
}
Level2GPU::Level2GPU(int device_id)
{
  TOTAL_test = TOTAL_test_right = 0;
  fp_testing = NULL; dev_fp_testing = NULL;
  result_array = NULL; dev_result_array = NULL;
```

```
  test_array = NULL; dev_test_array = NULL;
  DEVICE_ID = device_id;
  //DEVICE_ID = 0;
  if (cudaSuccess != cudaSetDevice(DEVICE_ID)) printf("\n__fail to set cuda device__,
    id=%d , line=%d infile: Level2GPU.cu \n", DEVICE_ID, __LINE__);
  TOTAL_SEARCHING_TIME = TOTAL_TRANFER_TIME = TOTAL_TRANFER_DATABASE_TIME =
    TOTAL_LOAD_DATABSE_TIME = 0;
}

void Level2GPU::CopyDataToDevice(int device_id = -1)
{
  if (device_id != -1) DEVICE_ID = device_id;
  cudaError_t cudaStatus;
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start);
  if (cudaSuccess != cudaSetDevice(DEVICE_ID)) printf("\n__fail to set cuda device__,
    id=%d , line=%d infile:");
  cudaStatus = cudaMalloc((void**)&dev_fp_array, fp_array_size * sizeof(unsigned
    int));
  if (cudaStatus != cudaSuccess) {fprintf(stderr, "Level2GPU.cu CUDA alloc failed! at
    line %d\n", __LINE__);}
  cudaStatus = cudaMalloc((void**)&dev_ht_array, ht_array_size * sizeof(unsigned
    int));
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "Level2GPU.cu CUDA alloc failed!
    at line %d\n", __LINE__); }
  cudaStatus = cudaMemcpy(dev_fp_array, fp_array, fp_array_size * sizeof(int),
    cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "Level2GPU.cu CUDA copy to device
    failed! at line %d\n", __LINE__); }
  cudaStatus = cudaMemcpy(dev_ht_array, ht_array, ht_array_size * sizeof(int),
    cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "Level2GPU.cu CUDA copy to device
    failed! at line %d\n", __LINE__); }
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  TOTAL_TRANFER_DATABASE_TIME = 0;
  cudaEventElapsedTime(&TOTAL_TRANFER_DATABASE_TIME, start, stop);
}
void Level2GPU::SetTestingData(TestingFingerPrints *test_fingerprints)
{
  qfp_array_size = test_fingerprints->qfp_array_size;
  num_test = test_fingerprints->DATA_N;
  fp_testing = test_fingerprints->DATA;
}
void Level2GPU::CopyTestingDataTODevice()
{
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start);
```

```
  if (cudaSuccess != cudaSetDevice(DEVICE_ID)) printf("\n__fail to set cuda device__, id=%d , line=%d infile: Level2GPU.cu \n", DEVICE_ID, __LINE__);
  cudaError_t cudaStatus;
  cudaStatus = cudaMalloc((void**)&dev_fp_testing, qfp_array_size * sizeof(unsigned int));
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "CUDA alloc failed! at line %d\n", __LINE__); }
  cudaStatus = cudaMemcpy(dev_fp_testing, fp_testing, qfp_array_size * sizeof(int), cudaMemcpyHostToDevice);
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "CUDA copy to device failed! at line %d\n", __LINE__); }
  cudaStatus = cudaMalloc((void**)&dev_result_array, num_test * sizeof(unsigned int));
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "CUDA alloc failed! at line %d\n", __LINE__); }
  cudaStatus = cudaMalloc((void**)&dev_test_array, 100 * sizeof(unsigned int));
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "CUDA alloc failed! at line %d\n", __LINE__); }
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  float tim_ = 0;
  cudaEventElapsedTime(&tim_, start, stop);
  TOTAL_TRANFER_TIME += tim_;
}
__global__ void kernel_method1_mutiple_songs(unsigned int* qfp_array_dev, unsigned int* fp_array_dev, unsigned int* ht_array_dev, unsigned int* result_array_dev,
  unsigned int qfp_array_size, unsigned int fp_array_size, unsigned int ht_array_size, unsigned int num_test, unsigned int hbits, unsigned int * test_array_dev)
{
  num_right_dev = 0;
  unsigned int qidx, query[128], hash, addr, num, tmp_hd, min_hd, mid, frame[3], fpdat[128], i ;
  int i_t = threadIdx.x;
  {
    memcpy(query, &qfp_array_dev[i_t * 128], 512);
    mid = 0xFFFFFFFF;
    min_hd = 0xFFFFFFFF;
    for (qidx = 0; qidx < 126; qidx++)
    {
      hash = lsh_dev(&query[qidx], hbits);
      if (hash == 0)
      {
        addr = 1 << hbits;
        memcpy(&num, &ht_array_dev[0], 4);
        num -= addr;
        num++;
      }
      else
      {
        memcpy(&addr, &ht_array_dev[hash - 1], 4);
        memcpy(&num, &ht_array_dev[hash], 4);
        num -= addr;
```

```
        addr++;
      }
      unsigned int addr2 = addr;

      for (i = 0; i < num; i++)
      {
        memcpy(&addr, &ht_array_dev[addr2 + i], 4);
        if ((addr & 0x7F) != qidx)
          continue;
        memcpy(frame, &fp_array_dev[addr], 12);

        if (hd_dev(frame, &query[qidx], 3) <= 24)
        {
          memcpy(fpdat, &fp_array_dev[addr / 128 * 128], 512);
          tmp_hd = hd_dev(fpdat, query, 128);
          if ((tmp_hd <= 1024) && (tmp_hd < min_hd))
          {
            min_hd = tmp_hd;
            mid = addr >> 7;
          }
        }
      }
      if (mid != 0xFFFFFFFF)
      {
        break;
      }
    }
    result_array_dev[i_t] = mid;
  }
}
void Level2GPU::LSH_Cuda_Mutiple_Threads()
{
  cudaEvent_t start, stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);
  cudaEventRecord(start);
  kernel_method1_mutiple_songs << <1, num_test >> >(dev_fp_testing, dev_fp_array,
    dev_ht_array, dev_result_array,
    qfp_array_size, fp_array_size, ht_array_size,
    num_test, 20, dev_test_array);
  cudaError_t cudaStatus = cudaDeviceSynchronize();
  if (cudaStatus != cudaSuccess) { fprintf(stderr, "Level2GPU.cu
    cudaDeviceSynchronize failed! at line %d\n", __LINE__); }
  cudaEventRecord(stop);
  cudaEventSynchronize(stop);
  float tim_ = 0;
  cudaEventElapsedTime(&tim_, start, stop);
  TOTAL_SEARCHING_TIME += tim_;
}
```

Listing B.2: Source code Level2GPU's code file

Appendix C

Source Code: Searching Management (Level1)

```
#pragma once
#include "Level2.h"
#include "TestingFingerPrints.h"
#include "KmodesModel.h"
#include "Kmodes.h"
#ifdef __linux__
#include <pthread.h>
#else
#include <thread>
#endif
class LSHSystemManager
{
private:
  int NUM_DEVICES;
  Level2 **LEVEL2;
  TestingFingerPrints* TESTINGFINGERPRINTS;
  Kmodes* KMODES;
  bool end_of_test;
  char* kernel_test_name;
  bool is_show_log;
#ifdef __linux__
  pthread_t ** THREADS;
  pthread_t  *MAIN_THREAD;
#else
  std::thread ** THREADS;
  std::thread * MAIN_THREAD;
#endif
public:
  unsigned int TOTAL_TEST, TOTAL_TEST_RIGHT, TOTAL_MISS;
  float TOTAL_LOAD_DATABSE_TIME,TOTAL_SEARCHING_TIME, TOTAL_TRANFER_TIME,
    TOTAL_TRANFER_DATABASE_TIME;
  float TOTAL_THREAD_SLEEP_TIME, TOTAL_GPU_THREAD_SLEEP_TIME,
    TOTAL_SEARCHING_TIME_REAL;
public:
```

```
  LSHSystemManager(char* kmode_file, int numdevices = -1, bool show_log = true );
  ~LSHSystemManager();
  void SetTetingPrints(TestingFingerPrints* t);
  void Start();
  void Task(int id);
  void MainTask();
  void TotalCalcTheAccuracy();
  void WriteResult(char* file_name, char* kmode_file, char* test_file );
  //test
  void savetest(char* t);

};
```

Listing C.1: Source code LSHSystemManager's header file

```
#include "LSHSystemManager.h"
#include "LSHQueue.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __linux__
#include <unistd.h>
#else
#endif
LSHSystemManager * STATIC_LSHSystemManager;
LSHSystemManager::LSHSystemManager(char* kmode_file, int numdevices,bool show_log)
{
  is_show_log = show_log;
  TOTAL_THREAD_SLEEP_TIME = TOTAL_GPU_THREAD_SLEEP_TIME = TOTAL_SEARCHING_TIME_REAL=0;
  TOTAL_LOAD_DATABSE_TIME=TOTAL_SEARCHING_TIME = TOTAL_TRANFER_TIME =
    TOTAL_TRANFER_DATABASE_TIME = 0;
  KMODES = new Kmodes();
  KMODES->LoadInfo(kmode_file);
  if (numdevices == -1) numdevices = KMODES->K;
  end_of_test = false;
  STATIC_LSHSystemManager = this;
  NUM_DEVICES = numdevices;
  LEVEL2 = (Level2 **)malloc(NUM_DEVICES*sizeof(Level2 *));
  for (size_t i = 0; i < NUM_DEVICES; i++)
  {
    LEVEL2[i] = new Level2(i, 1000,
      KMODES->FILE_CLUSTER_NAME[i],
      KMODES->FILE_CLUSTER_HASHTABLE_NAME[i]);
  }
  TOTAL_MISS = 0;
  kernel_test_name = "none";
}
void TSleepFor(int microseconds_)
{
#ifdef __linux__
  usleep(microseconds_);
#else
  std::this_thread::sleep_for(std::chrono::microseconds(microseconds_));
```

```
#endif
}

void woker(int id)
{
  STATIC_LSHSystemManager->Task(id);
}
void woker3()
{
  STATIC_LSHSystemManager->MainTask();
}
void *woker3_linux(void *arg)
{
  STATIC_LSHSystemManager->MainTask();
  return NULL;
}
void *woker2(void *arg)
{
  int id = *((int *)arg);
  STATIC_LSHSystemManager->Task(id);
  return NULL;
}
void LSHSystemManager::Start_TestAccuracy_CPU_BruceForce()
{
  unsigned int *vector = (unsigned int*)malloc(VECTOR_LENGTH_INT * sizeof(unsigned
    int));
  int device_id = -1;
  int num_right = 0;
  for (int i = TESTINGFINGERPRINTS->DATA_N-1; i >=0; i--)
  {
    memcpy(vector, &TESTINGFINGERPRINTS->DATA[128 * i], VECTOR_LENGTH_BYTE);
    device_id = KMODES->FindCluster(vector);
    if (device_id != KMODES->FINGERPRINTS_INDEX[i] / KMODES->DATA_N)
      TOTAL_MISS++;
    int id = LEVEL2[device_id]->LSHDEVICE->TestOriginalMethod(vector, 20, i);
    if (id == KMODES->FINGERPRINTS_INDEX[i] % KMODES->DATA_N) num_right++;
    printf(" %d ", id);
  }
  printf("\n\n TOTAL TEST BRUCE FORCE: %d \n", num_right);
}
void LSHSystemManager::Start()
{
  clock_t t1, t2;
  t1 = clock();
#ifdef __linux__
  THREADS = (pthread_t**)malloc(NUM_DEVICES*sizeof(pthread_t*));
  MAIN_THREAD = new pthread_t();
  for (size_t i = 0; i < NUM_DEVICES; i++)
    pthread_create(THREADS[i],NULL, woker2,
      (void*)(new int(i)));
  pthread_create(MAIN_THREAD, NULL, woker3_linux, NULL);
  for (size_t i = 0; i < NUM_DEVICES; i++)
```

```
    (void)pthread_join(*THREADS[i], NULL);
  (void)pthread_join(*MAIN_THREAD, NULL);
#else
  THREADS = (std::thread**)malloc(NUM_DEVICES*sizeof(std::thread*));
  for (size_t i = 0; i < NUM_DEVICES; i++)
    THREADS[i] = new std::thread(woker, i);
  MAIN_THREAD = new std::thread(woker3);
  for (size_t i = 0; i < NUM_DEVICES; i++)
    THREADS[i]->join();
  MAIN_THREAD->join();
#endif
  printf("\n__FINISH__\n");
  t2 = clock();
  TOTAL_SEARCHING_TIME_REAL = 1000 * ((float)t2 - (float)t1) / CLOCKS_PER_SEC;
}
void LSHSystemManager::Task(int id)
{
  while (true)
  {
    TSleepFor(10000);
    TOTAL_GPU_THREAD_SLEEP_TIME += 10;
    if (LEVEL2[id]->MUTEX_IS_FULL_QUEUE)
    {
      //printf("__catch__");
      while (LEVEL2[id]->MUTEX_IS_KERNEL_RUNNING)
      {
        TOTAL_GPU_THREAD_SLEEP_TIME += 0.01;
        TSleepFor(10);
      }
      LEVEL2[id]->CopyDataFromQueueToLSHLEvel2();
      LEVEL2[id]->StartTestDATA(); kernel_test_name = "StartTestDATA_GPU";
      LEVEL2[id]->CalcTheAccuracyAndUpdate();
      LEVEL2[id]->QUEUE->Clear();
      LEVEL2[id]->MUTEX_IS_FULL_QUEUE = false;
      if (is_show_log)
      {
        printf("\n result for divice %d:", id);
        LEVEL2[id]->PrintResultArray();
      }
    }
    else if (end_of_test)
    {
      while (LEVEL2[id]->MUTEX_IS_KERNEL_RUNNING)
      {
        TSleepFor(100);
        TOTAL_GPU_THREAD_SLEEP_TIME += 0.1;
      }
      LEVEL2[id]->CopyDataFromQueueToLSHLEvel2();
      LEVEL2[id]->StartTestDATA(); kernel_test_name = "StartTestDATA_GPU";
      LEVEL2[id]->CalcTheAccuracyAndUpdate();
      LEVEL2[id]->QUEUE->Clear();
      LEVEL2[id]->MUTEX_IS_FULL_QUEUE = false;
```

```
      if (is_show_log)
      {
        printf("\n result for divice %d:", id);
        LEVEL2[id]->PrintResultArray();
      }
      break;
    }
  }
}
void LSHSystemManager::MainTask()
{
  unsigned int *vector = (unsigned int*)malloc(VECTOR_LENGTH_INT * sizeof(unsigned int));
  int device_id = -1;
  for (size_t i = 0; i < TESTINGFINGERPRINTS->DATA_N; i++)
  {
    memcpy(vector, &TESTINGFINGERPRINTS->DATA[128 * i], VECTOR_LENGTH_BYTE);
    device_id = KMODES->FindCluster(vector);

    if (device_id != KMODES->FINGERPRINTS_INDEX[i] / KMODES->DATA_N)
      TOTAL_MISS++;

    while (LEVEL2[device_id]->MUTEX_IS_FULL_QUEUE)
    {
      TSleepFor(10);
      TOTAL_THREAD_SLEEP_TIME += 0.01;
    }
    //printf("_%d_", i);
    LEVEL2[device_id]->AddVectorOnly(vector, KMODES->FINGERPRINTS_INDEX[i] % KMODES->DATA_N);
  }
  end_of_test = true;
}
void LSHSystemManager::TotalCalcTheAccuracy()
{
  TOTAL_TEST = TOTAL_TEST_RIGHT  = 0;
  TOTAL_LOAD_DATABSE_TIME = TOTAL_SEARCHING_TIME = TOTAL_TRANFER_TIME = TOTAL_TRANFER_DATABASE_TIME = 0;
  for (size_t i = 0; i < NUM_DEVICES; i++)
  {
    TOTAL_TEST += LEVEL2[i]->LSHDEVICE->TOTAL_test;
    TOTAL_TEST_RIGHT += LEVEL2[i]->LSHDEVICE->TOTAL_test_right;
    TOTAL_LOAD_DATABSE_TIME += LEVEL2[i]->LSHDEVICE->TOTAL_LOAD_DATABSE_TIME;
    TOTAL_TRANFER_DATABASE_TIME += LEVEL2[i]->LSHDEVICE->TOTAL_TRANFER_DATABASE_TIME;
    TOTAL_SEARCHING_TIME += LEVEL2[i]->LSHDEVICE->TOTAL_SEARCHING_TIME;
    TOTAL_TRANFER_TIME += LEVEL2[i]->LSHDEVICE->TOTAL_TRANFER_TIME;
  }
  printf("\n RESULT: TOTAL TEST: %d , TOTAL_RIGHT: %d, TOTAL_MISS: %d\n", TOTAL_TEST, TOTAL_TEST_RIGHT, TOTAL_MISS);
}
```

Listing C.2: Source code LSHSystemManager's code file

Appendix D

Source Code: Hash Table Generation

```
import os
import sys
import struct
hfunc = [53, 49, 45, 60, 2, 72, 14, 82, 62, 46, 35, 95, 43, 50, 0, 77, 28, 88, 13,
    10, 65, 54, 29, 93]
if len(sys.argv) != 3:
  print "Usage: ./gen-ht.py FPDB-file HT-file";sys.exit()
ht = []
for i in range(2**len(hfunc)):
  ht.append([])
n = os.path.getsize(sys.argv[1]) / 512
f = open(sys.argv[1], "rb")
for i in range(n):
  fp = f.read(512)
  sfp = []
  for j in range(128):
    sfp.append((ord(fp[4*j+3]) << 24) | (ord(fp[4*j+2]) << 16) | (ord(fp[4*j+1]) <<
    8) | ord(fp[4*j]))
  for j in range(126):
    frame = (sfp[j] << 64) | (sfp[j+1] << 32) | sfp[j+2]
    hash = 0
    for e in hfunc:
      hash <<= 1
      hash |= (frame >> e) & 1
    ht[hash].append(128 * i + j)
f.close()
f = open(sys.argv[2], "wb")
addr = 2 ** len(hfunc) - 1
for h in ht:
  addr += len(h)
  f.write(struct.pack('I', addr))
for h in ht:
  for addr in h:
    f.write(struct.pack('I', addr))
f.close()
```

Listing D.1: Hash Table Generation [1]

Appendix E

Source Code: Main Appication

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdlib.h>
#include <stdio.h>
#include "LSHSystemManager.h"
int main(int argc, char *argv[])
{
  if (argc == 4)
  {
    char* name_kmode_model = argv[1];
    char* name_testing_fingerprints = argv[2];
    char* name_out_put = argv[3];
    TestingFingerPrints testf;
    testf.LoadData (name_testing_fingerprints);
    LSHSystemManager* LSHSYSTEMMAMAGER2 = new
    LSHSystemManager(name_kmode_model,-1,false);
    LSHSYSTEMMAMAGER2->SetTetingPrints(&testf);
    LSHSYSTEMMAMAGER2->Start();
    LSHSYSTEMMAMAGER2->TotalCalcTheAccuracy();
    LSHSYSTEMMAMAGER2->WriteResult(name_out_put, name_kmode_model,
    name_testing_fingerprints);
  }
  else printf("\nUse <name_kmode_model> <name_testing_fingerprints>
    <name_out_put>\n;");
  return 1;
}
```

Listing E.1: Main Appication File