A Computationally Efficient Framework for Large-scale Distributed Fingerprint Matching

Atif Muhammad
School of Computer Science and Applied Mathematics
University of the Witwatersrand

Supervised by:
Prof. Turgay Çelik

A dissertation submitted for the degree of
Masters of Science
(Computer Sciences)

May 2017

UNIVERSITYOF THE WITWATERSRAND, JOHANNESBURG

School of Computer Science and Applied Mathematics

Declaration

2017

I, **Atif Muhammad**, Student number **447235,** am a student registered for MSc Computer Science in the School of Computer Science and Applied Mathematics.

This declaration applies to the MSc Dissertation

I hereby declare the following:

- I am aware that plagiarism (the use of someone else's work without their permission and/or without acknowledging the original source) is wrong.
- I confirm that the work submitted for assessment is my own unaided work except where I have explicitly indicated otherwise.
- I have followed the required conventions in referencing the thoughts and ideas of others.
- I understand that the University of the Witwatersrand may take disciplinary action against me if there is a belief that this is not my own unaided work or that I have failed to acknowledge the source of the ideas or words in my writing.

Signature: _____ Date: _01/06/2017_____

Abstract

Biometric features have been widely implemented to be utilized for forensic and civil applications. Amongst many different kinds of biometric characteristics, the fingerprint is globally accepted and remains the mostly used biometric characteristic by commercial and industrial societies due to its easy acquisition, uniqueness, stability and reliability.

There are currently various effective solutions available, however the fingerprint identification is still not considered a fully solved problem mainly due to accuracy and computational time requirements. Although many of the fingerprint recognition systems based on minutiae provide good accuracy, the systems with very large databases require fast and real time comparison of fingerprints, they often either fail to meet the high performance speed requirements or compromise the accuracy.

For fingerprint matching that involves databases containing millions of fingerprints, real time identification can only be obtained through the implementation of optimal algorithms that may utilize the given hardware as robustly and efficiently as possible. There are currently no known distributed database and computing framework available that deal with real time solution for fingerprint recognition problem involving databases containing as many as sixty million fingerprints, the size which is close to the size of the South African population.

This research proposal intends to serve two main purposes: 1) exploit and scale the best known minutiae matching algorithm for a minimum of sixty million fingerprints; and 2) design a framework for distributed database to deal with large fingerprint databases based on the results obtained in the former item.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	**Meaning**
HPC	High Performance Computing
GPGPU	General-Purpose Computing on Graphics Processing Unit
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
SIMD	Single Instruction Multiple Data
MCC	Minutia Cylinder-Code [4]
GUI	Graphical User Interface
KMPS	Thousand matches per second
FNMR	False Negative Match Rate
FPMR	False Positive Match Rate

Chapter 1

Introduction

This dissertation conducts a detailed study comparing some of the best fingerprint matching algorithms available. This assists in developing a real-time fingerprint identification framework involving a large number of fingerprints. The framework is planned to be used for interactive applications with real time fingerprint identification solution that promises to be scalable and at the same time ensure minimal or no accuracy is compromised. This chapter thoroughly discusses the background of fingerprint matching followed by the motivations and objectives behind this research project. The chapter concludes with a brief overview of the rest of the chapters.

1.1 Research Background

Biometric features were mostly used in the past for identification and verification purposes in forensic applications. However, for the last two decades it has been widely implemented to be utilized for civil applications such as access control, voters' authentication, immigration at border control, drivers' license applicants etc [34, 22]. Many different kinds of biometric characteristics (e.g. fingerprints, signature, iris, face, voice etc) have been intensely studied and analyzed in the past [37]. In comparison with most, the fingerprint is widely accepted and remains as one of the most used biometric characteristic by the commercial and industrial societies due to its easy acquisition, uniqueness, stability and reliability [48, 5]. Fingerprints are captured using contact-based scanners which require the placing of fingers against a glass-like surface. These scanners are often used by multiple people and thus, they are prone to finger sweats which raises health and hygiene related issues [32]. In order to overcome these risks, many contactless fingerprint capturing techniques have been introduced [33].

Fingerprint identification facilitates the replacement of the conventional authentication systems that are based on passwords, identity cards etc [64]. Fingerprints are graphical

flow-like ridges present on human fingers. Minutiae are the endpoints and bifurcations of fingerprint ridges. Every human being is known to have unique and immutable fingerprints, due to minutiae which remain unchanged over an individual's lifetime [40, 62]. This leads to a very discriminative classification of fingerprints [62]. The main minutiae points in a fingerprint include ridge ending, bifurcation and short ridge as shown in Figure 1.1 [29, 24]. Minutiae and the pattern of ridges are very important when analysing fingerprints as the uniqueness of a given fingerprint is based on them.

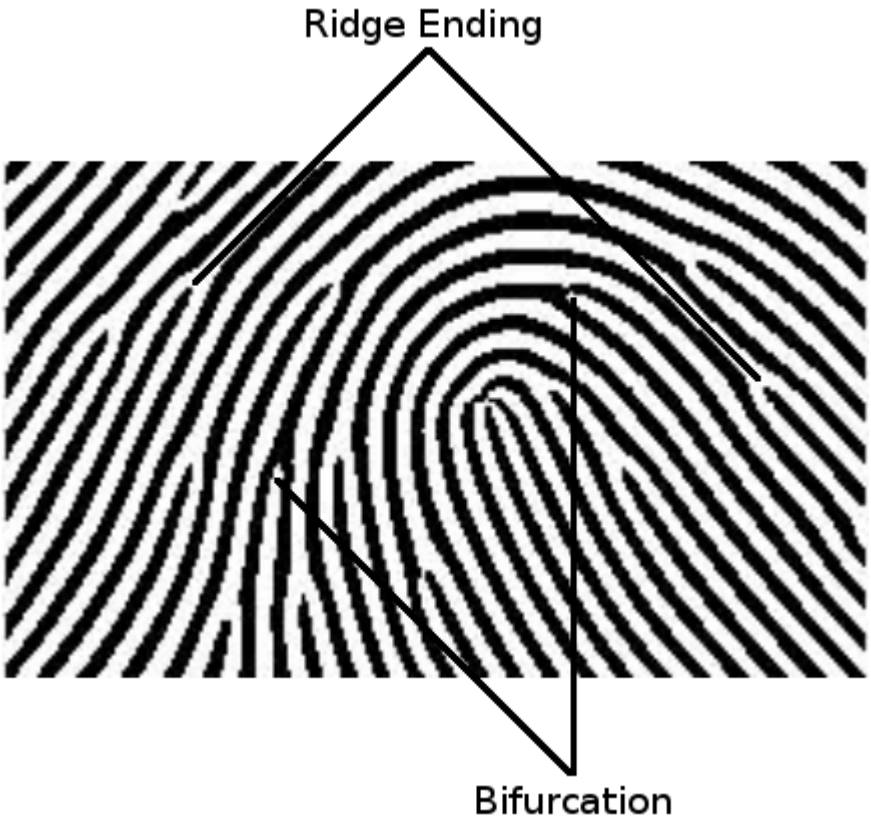


Figure 1.1: An image of a fingerprint with some of the important ridge characteristics.

These days many fingerprint identification systems are based on minutiae matching. Thus, there has been ongoing progress to improve the performance in terms of accuracy and processing time. In the past decade, scientists have made a significant amount of progress in both fingerprint recognition algorithms and computational processors [40, 48]. There are currently various effective solutions available, yet the fingerprint identification is still not considered a fully solved problem. This is mainly due to accuracy and computational time requirements [40, 4]. Although many of the minutiae based fingerprint recognition systems provide good accuracy. There are systems with very large databases which require fast and real time comparisons of fingerprints. But owing to the size of the database, they often

either fail to meet the high performance speed requirements or compromise on accuracy [62].

When dealing with scenarios that involve an arbitrarily large sized database where computational time is crucial, High Performance Computing (HPC) is one of the tools that promise an optimal solution by executing many processes concurrently in a reasonable amount of time [57, 44]. Over the last decade, HPC has proven to be successfully implemented in image comparisons [16, 44] and pattern recognition problems [54, 13, 56, 44]. For fingerprint matching that involves databases containing millions of fingerprints, real time identification can only be obtained through the implementation of optimal algorithms that may utilize the given hardware as robustly and efficiently as possible. Currently, to our knowledge, there is no known distributed framework available that has the capability of dealing with a database containing as much as sixty million fingerprints, which is able to solve the fingerprint identification problem in real time.

1.2 Research Motivation

The main motivation behind designing and implementing a distributed framework for fingerprint matching are as follows.

- **Real time matching system**: The recent increase in demand for fingerprint recognition systems require real time solution with large databases. This will help banks and border crossing locations to minimize frauds, and produce accurate identification to avoid administrative delays. This provides ease to the people that fail to produce conventional identification documents, in real time.

- **Accuracy and scalability**: An effective way to provide real time fingerprint matching for large databases is to implement parallel execution of computation using CPUs and GPUs in a distributed system. There is very little research available on how a distributed HPC system for fingerprint recognition with large databases may impact the accuracy of the current best-known fingerprint matching algorithms. The framework has to be thoroughly studied for analysis in order to determine whether it can be efficiently scaled to support larger databases.

1.3 Problem Statement and Research Objectives

In this era where many effective solutions are available for a fingerprint matching system, the processing time for large databases still remain significantly high. The fingerprint matching

problem is comprised of two main sub-problems: verification and identification. Verification is the process of determining whether two fingerprints namely S_i and S_j belong to the same person. This process requires one to one comparison and whereby computation time is usually not a problem. Identification is the process of finding a matching fingerprint in a database in order to identify its owner [44]. This involves one-to-many comparisons since a fingerprint database (F) is a set of N fingerprints, i.e, $F = \{F_1, F_2, \ldots, F_N\}$. Thus, for a given (query) fingerprint F_q has to be compared with N fingerprints to find the fingerprint that provides the highest matching score against the given fingerprint F_q [44]. The identification problem can be described as the verification process once for every fingerprint in the database. Thus the major difference between verification and identification is the complexity order [44].

Most of the fingerprint matching algorithms are implemented to achieve higher accuracy and with very little focus on computation time. The issue of the processing time becomes highly critical as the size of the fingerprint database increases. Although, some algorithms do take computation time into consideration, however, working with large data sets remain time consuming [44].

The problem is centered around designing a distributed system for parallelization of existing best known matching algorithms which provide real-time identification in a *Big Data* environment with negligible accuracy degradation. Furthermore, the systematic scalability of the distributed system is crucially important while maintaining the accuracy.

The major research objective is to design and implement an efficient and scalable distributed system that allow fingerprint matching to be performed in real-time or almost near real-time. Furthermore, allowing us to analyse how the current best known algorithms react when implemented in a parallelized large database environment.

1.4 Dissertation Organization

The remainder of this document is structured as follows. Chapter 2 presents the literature survey on fingerprint matching algorithms, synthetic fingerprint creation, and parallelization of some fingerprint matching algorithms on CPU and GPU. Chapter 3 outlines the methodology that we intend to follow in order to answer our research question. Chapter 4 presents the achieved results. Finally, Chapter 5 summarises this research project followed by a future work section.

Chapter 2

Literature Review

The previous chapter outlined the main problem and techniques that surround fingerprint recognition. This chapter investigates, identifies and presents a review of existing literature techniques and algorithms used to solve the components related to fingerprint recognition.

It begins with a basic background of fingerprint recognition which is provided in Section 2.1 in order to gain a better understanding of the concepts. Furthermore, a discussion of the importance of fingerprint recognition together with how fingerprints' distinguishable characteristics allow for each fingerprint to be unique.

Section 2.2 provides an overview of some of the general methods used in literature in order to successfully perform fingerprint recognition, those being **texture** and **minutiae**. The analysis from past research has shown several various feature extraction methods which were considered for correctly identifying and extracting minutiae features.

Section 2.3 deals with the use of the fingerprint database and the necessity of developing a synthetic fingerprint database generator. Section 2.4 discusses the two main feature extraction methods which ensure an acceptable accuracy in determining a set of fingerprints as unique, namely **binarisation approach** and **ridge following**. Those mentioned methods will be discussed further in detail on how they differ in comparison to other methods along with their apparent advantages and disadvantages.

For comprehensive data analysis and increasing computational power, large scale computation, Section 2.5 will introduce High Performance Computing (HPC). Section 2.6 introduces the general purpose Graphical Processing Unit (GPU) which have taken over CPUs. GPUs offer performance boosts which help greatly in parallelism. Section 2.7 will discuss the differences and benefits of constant and texture memory. Finally, insight into how mobile fingerprint verification works in mobile devices will be introduced in Section 2.8. It will be shown that mobiles using GPUs for fingerprint feature extraction tends to be better and faster.

2.1 Fingerprint Recognition Background

Fingerprint recognition has been widely accepted due to its reliability and convenience [64]. Over the last decade, there has been a significant amount of research which has been conducted in the area of fingerprint recognition. Fingerprint recognition mainly involves matching process which is to compare two fingerprints in order to establish a match between the fingerprints. Many different approaches have been suggested to provide effective fingerprint matching solutions.

Given two fingerprints, the basic process of finding whether the fingerprints match (i.e they belong to the same person) is to extract important features from the fingerprints and compute the correlation between them. These important features refer to the patterns on the skin. Those being, the characteristics of ridges, and minutia points, which make each pattern unique and distinguishable.

The level in which these fingerprints correlate, determines whether the fingerprints are indeed the same. Depending on the chosen technique, the fingerprint pair in question may still be found to be the same. However, the fingerprints may vary in terms of size, orientation and amount of information available. This situation is very probable in the case when the enrolled fingerprint is taken in an environment which might be different when the query fingerprint is captured [64].

2.2 Recognition Methods

There are three major types of fingerprint recognition methods depending on the information extracted from fingerprint images, namely texture-based, minutiae-based and hybrid methods which utilize both texture and minutiae information.

2.2.1 Texture-based Methods

These methods involve the extraction of features, rather than characteristics of minutiae. These features from the fingerprint may include local orientation, ridge shape and texture information. These methods allow major discriminatory information to be extracted with high accuracy and reduce processing time compared to minutiae-based feature vector extractions. Although minutiae-based methods are considered to be more popular, texture-based methods are specifically useful when the image quality is significantly low, thus reducing the credibility of information being extracted from the image [63]. Basic texture-based methods often fail to provide very high accuracy in the cases when fingerprints may be variant to transformations. Recently, it is noted that more texture-based methods combining local

and global fingerprint structures achieve acceptable accuracy and provide solutions that are invariant to transformations [63].

2.2.2 Minutiae-based Methods

These methods have gained more popularity over other methods and have been widely accepted in the last decade. They work by extracting feature vectors from the fingerprints and storing important information as mapping points in the multidimensional plane. These points may comprise of several characteristics such as the minutiae coordinates (location), orientations, type etc [65, 26, 38, 23, 59, 51, 63]. Previously, the minutiae based methods involved performing a global fingerprint alignment. This technique often lead to higher computation time required to match fingerprints as well as being less effective due to providing inadequate robustness against deformations. In order to address the major shortcomings of minutiae-based algorithms, the local minutiae matching based technique was introduced [4]. This technique made significant contributions towards solving fingerprint distortion and high computation time of fingerprint matching [4]. The local minutiae matching technique uses structures that are immune to global transformations such as translation, rotation etc. Furthermore, it allows a fingerprint matching process to reject a match at an early stage in the case where the two fingerprints in question are completely different [4].

In the work proposed in [26], the authors present a fingerprint matching technique which is one of the first approaches that successfully derived the relationship between minutia and its neighbouring minutia for invariant distances and angles [4]. In this method, each minutia is represented with a feature vector which is also dependent on its neighbours. Those feature vectors in both fingerprints are compared in pairs with the assumption that the majority of the corresponding pairs, in terms of relative angle and position, represent the same minutia. This method ignores the translation and rotation problem [44].

In [9] the authors mainly focused on coping with errors even if the original fingerprint has an altered shape of characteristics. A local matching technique is used to compute and discretize every minutia with a fixed radius R which is then compared with the minutia of other fingerprints. Once enough similarities have been found, it then modifies the radius to cater for altered shape of characteristics problems [44].

Recently, Minutia Cylinder Code (MCC) was introduced which involves the use of local and global fingerprint information for recognition [4, 44, 17]. It defines every minutia m by parameters (x, y, θ), where (x, y) denote its location coordinates and θ is its orientation [28, 62]. Figures 2.1a and 2.1b show a sample of a captured fingerprint and detection of minutiae points respectively. The main reason it does not consider minutia type is that the

feature extractors might make an incorrect classification for minutia type which could deem the method to be unreliable.

MCC creates a 3-dimensional (3D) cylinder for each minutia m, which stores and represents the spatial and directional relationship between the minutia and its neighbourhood. The cylinder's height is 2π with a fixed radius, R, which makes all the cylinders have the same size (base and height). Furthermore, every cylinder is divided and discretized into N_D sections and N_S cells where each cell holds a value that represents the cell's position as well as the corresponding position and direction of neighbouring minutiae. Based on this value, a cell can be classified as either valid or invalid, thus allowing only valid cells to be included for the computational process of matching [44]. The final process is to compare every cylinder, cell by cell of both fingerprints by accumulating the euclidean and hamming distance to obtain the overall score of the match. This process continues for every other fingerprint in the database, and then the corresponding fingerprint with the best score is recommended to be the successful match. MCC is shown to be a state-of-the-art fingerprint recognition method which provides efficient fingerprint matching with high accuracy.

2.2.3 Hybrid Methods

A minutiae- and texture-based algorithm was presented in [15]. It uses the combination of textual and minutiae-based descriptors for obtaining orientation and frequency of a minutia and representing a relationship between the minutiae and its neighbourhood. It then uses the greedy matching algorithm based on alignment [58, 15] to derive the similarities between the minutiae.

In [53], authors proposed a method involving statistical features extracted from the fingerprint images. The features include

- *entropy* - calculated using the intensity histogram of the fingerprint image;
- *correlation* - computed based on a 2D median filter of the image;
- *energy* - obtained using 5-level wavelet decomposition.

The proposed method focused highly on accuracy and as a result no processing time was reported.

[63] used global minutia and variant moments to demonstrate high performance matching between fingerprints. In [59], it is demonstrated a matching algorithm that made use of the similarity of local structures involving neighbouring minutiae. In [38], the author exhibited the reliability of fingerprint recognition by changing the size of matching box based on the distance from corresponding minutiae.

(a) (b)

Figure 2.1: (a) shows an original scanned fingerprint. (b) shows detected minutiae points from the original fingerprint directed towards its orientation [63].

2.3 Fingerprint Database

In order to verify and validate the performance and scalability of any of the above algorithms, a database containing a sufficient amount of fingerprints is required. Usually, collecting large amounts of fingerprints can be very time consuming and a rather expensive activity. It is also challenging to find large fingerprint databases to be publicly available due to security concerns associated with fingerprints. Hence, this concern requires the need to generate a large number of fingerprints which can only be made possible through the use of synthetic fingerprints. A fingerprint database which can be effectively used to support fingerprint identification problem can be collected using:

2.3.1 NIST DB4 and DB14

The National Institute of Technology (NSIT) provides databases DB4 [60, 17] and DB14 [61, 17] with 2000 and 27000 of rolled fingerprint pairs respectively. The average number of minutiae present in DB4 and DB14 is 135.84 and 206.9, respectively [17]. These captured fingerprint databases can be beneficial in detecting whether the large scale distributed system can deal with captured fingerprints. Although, these databases hold various types of fingerprints, the amount of fingerprints available on the databased are not enough for the large scale fingerprint identification problem that we aim to address. Since, this research

involves working with large datasets, the quantity of the dataset is more important than the quality.

2.3.2 Synthetic Fingerprint Database Generations (SFinGE)

Synthetic Fingerprint Database Generations (SFinGE) [6] allows for realistic fingerprint images to be generated. This process randomly generates fingerprint minutiae, when given some input parameters. Based on the minutiae, SFinGE derives a complete fingerprint image that appears identical to a natural fingerprint. In order to create a natural-like fingerprint, SFinGE first defines the global shape of the fingerprint, followed by generating a consistent directional map to the shape. Hereafter, a density map is generated, and finally a ridge-line pattern is generated in order to fabricate minutiae at random positions. In order to make the fingerprint appear more natural, SFinGE performs a variation of ridge average thickness, distortions, noise rendering and global translation/rotation to the generated fingerprint [6].

Some experiments were conducted on SFinGE where 90 fingerprint experts were asked to identify artificial fingerprints generated by SFinGE. Only 23% of the generated fingerprints were identified to be artificial [6]. SFinGE has been proven to be very useful for performance evaluation, learning and testing fingerprint based systems [6].

2.3.3 Anguli: Synthetic Fingerprint Generator

Anguli is an open-source application written in C++. It is an implementation of *SFinGE* based on the techniques proposed in [6]. Its main purpose is to allow bulk synthetic fingerprints to be generated with ease. Figures 2.2a and 2.2b are the examples of the fingerprints generated using *Anguli*. In the figures it is first shown a random template of a fingerprint (2.2a) which is followed by a distorted image of the fingerprint (2.2b) by adding noises and a scratch to the original fingerprint template generated. Since bulk image generators often tend to be time consuming, *Anguli* allows to generate bulk fingerprints in parallel using multiple cores. It claims to produce a million fingerprints in less than four days using 8 physical processing cores running at 2 GHz clock speed. *Anguli* is compatible with major operating systems such as Windows and Linux. The friendly GUI allows any number of fingerprints to be generated with different configurations. Some of its major features include:

- Generation of multiple impressions from a generated fingerprint;
- Saving of meta data;
- Setting a constant amount of fingerprints to be stored per directory;

- Percentage of distortion added such as noise and scratch;
- Range of pixels by which the fingerprints are transformed.

(a) (b)

Figure 2.2: (a) A synthetic fingerprint generated using *Anguli*. (b) Distortion added by *Anguli* to the synthetic fingerprint template (a).

2.4 Feature Extraction

Fingerprints are known to be unique mainly due to the distinguishable set of features they posses [49]. It is therefore important for these features to be extracted robustly in order to efficiently recognise fingerprints. Fingerprints are also prone to noise and distortions, thus recognition systems are becoming more relied on feature extraction methods [55]. The noise and distortion in the fingerprint image may occur due to variations in skin and impression conditions such as scars, humidity, dirt, and nonuniform contact with the fingerprint capture device [1]. Figure 2.3 provides visuals of the example of some of different quality of captured fingerprint images.

There are various feature extraction methods proposed in the past which provide acceptable accuracy. While some of the feature extraction methods are based on a neural network approach [36, 35], most of the feature extraction methods can be typically divided into two main categories, namely binarised image approach and ridge following [10].

2.4.1 Binarisation

The binarisation method usually involves converting images into binary images. Furthermore, a thinning process is applied to the resultant binary image which allows the width

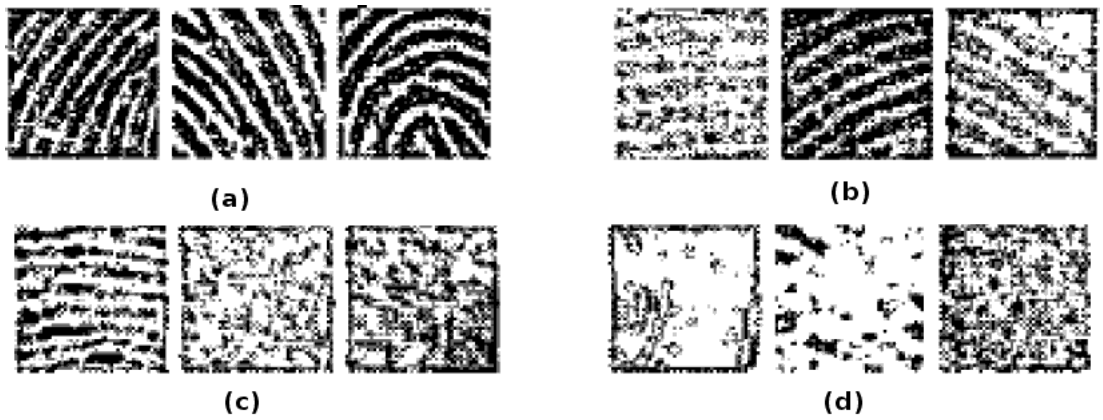


Figure 2.3: Example of different quality of captured fingerprints adopted from [8]. The fingerprint images are ordered from "good" to "bad" quality where (a) being "very good" and (d) represents "very bad" quality images.

of the ridge lines to be minimised to a width of one pixel. The minutia detection process then begins locating minutia, based on the amount of neighbouring pixels [39]. Figure 2.4 illustrates the process being implemented on a gray scale fingerprint images with resultant images.

The main disadvantage of the binarisation method is that it may lose a significant amount of information from the original image. Since binarised images usually follow a thinning procedure to the edges, this could present false orientations of minutiae in the fingerprint images, due to the sharp turns resulting from the thinning process.

Furthermore, the binarisation and thinning processes tend to be computationally expensive which may be critical in live recognition systems. Some of these shortcomings are overcome by normalising and enhancing the images before applying the binarisation process. However, a significant accuracy degradation still remains an issue [50].

2.4.2 Ridge Following

The ridge line following method involves working with gray level fingerprint images in order to successfully detect minutiae [10]. The basic idea of this approach is to follow the ridge lines until the line ends or splits. This method allows minutiae to be detected directly from the gray-level images. To ensure no ridge line is processed more than once, when a ridge line has been visited, it will be marked. This method was first proposed in [39] which demonstrated the efficiency and superiority of the method compared to the binarisation process. Further improvements to this method were later shown in [27].

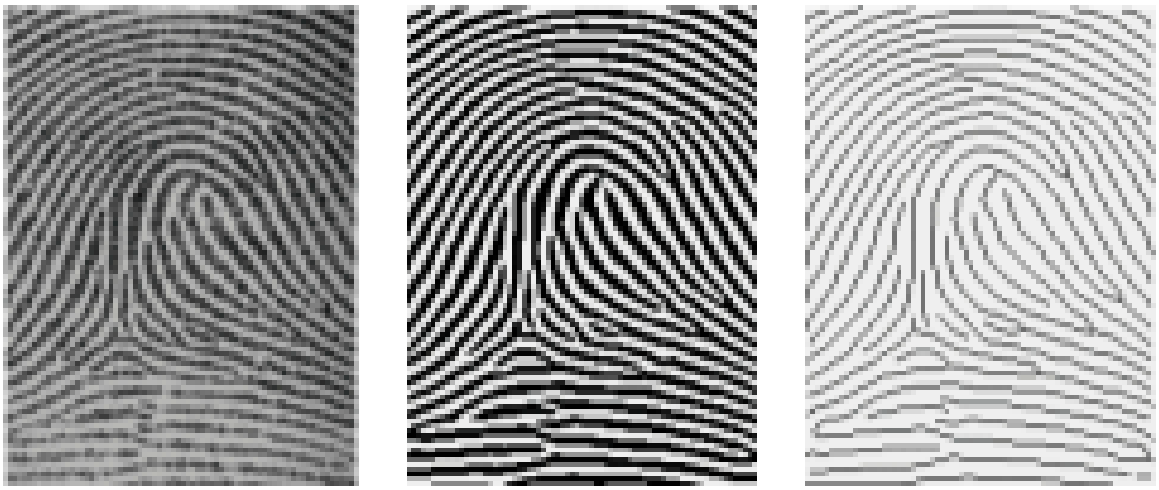


Figure 2.4: Sequence of binarisation process followed by thinning of image [39].

2.5 High Performance Computing

The advancement in research has shifted ahead the capacity of a single-core processor to a large number of multi-core processors for comprehensive data-analysis, multi-level simulations, and large-scale computation due to the increased complexity. Therefore, the requirements for additional computational horse power have tremendously increased [3].

HPC is the practice of utilizing computer resources for aggregating computing power and gaining maximum efficiency. The need for more processing power is met with closely integrated computer systems that work in a parallel environment with many multi-core processors, huge amount of storage and low latency interconnects. It is estimated that the use of HPC will grow greatly in the near future [3].

The increasing demand of deployment of computer cluster/parallel computing has exposed the need to exploit the area of HPC. A computer cluster is a type of parallel or distributed system, consisting of a collection of interconnected stand-alone computers which work cooperatively together as a single integrated computing resource [46]. Computer clusters improve performance of a system and are more cost effective than single computers. There has been research conducted in the area of HPC, that are related to enhancing performance in computer cluster using different tools and algorithms.

Most of the clusters which have a large number of compute nodes are mainly experimental in nature. These days, GPU clusters deployed for production purposes are not very rare anymore. It has been established that in order to maintain a balanced system, especially when GPUs are heavily relied on to perform intensive calculations, other components of the cluster such as memory, bus speed and network throughput need to be kept consistent and matched with the GPU expected performance [30]. Thus, when designing a

cluster architecture, considering all the performance issues is very significant in the HPC environment.

Recently, we notice a trend where HPC is increasingly being utilised in the area of fingerprint recognition problem. In [5, 45, 18], it is shown that the use of HPC techniques produce significant increases in performance when used for fingerprint identification compared to the conventional methods. Hence, in order to achieve better efficiency for large scale fingerprint identification problem, the practices of HPC need to be considered.

2.6 General Purpose Graphical Processing Unit and CUDA

2.6.1 GPU

The field of High Performance Computing was largely dominated by CPUs until the rise of an existing technology, newly applied to HPC, which is known as Graphical Processing Unit (GPU). Single Instruction Multiple Data (SIMD) architecture is used in GPU devices to introduce parallelism. GPUs are capable of performing up to trillions of computations per second. Such performance is achieved through the data locality and parallelism of the GPUs by dividing a given task among multiple processing units and solving the divided parts simultaneously. Since the introduction of the first GPU, the GPUs have greatly improved in performance, but the power consumption has also increased significantly [20]. Although GPUs are known to be power hungry devices, they provide very good performance per watt ratio compared to CPUs. In order to achieve real time performance in fingerprint identification with large datasets, GPU devices promise to be on of the best candidates, due to its introduction of massive parallelism which reduces the computation time significantly.

While GPUs offer performance boost with massive parallelism, there are also some limitations associated with it which needs to be adhered to when utilising a GPU.

2.6.1.1 Branch Divergence

It is not unusual for an algorithm to have conditional statements such as **if-then-else**, **switch** or **while** loop. When these conditional statements are implemented they create a possibility of thread divergence. Thread divergence could cause great performance degradation therefore it is crucial to understand thread divergence in order to avoid a negative impact on the performance.

While CPU manufacturers have invested an immense amount of effort into predicting branch divergence to minimise the impacts of thread divergence, the impacts of branch divergence are quite apparent on GPUs.

Algorithm 1 Branch Divergence on GPU

```
1: if (threadIdx.x & 1) then
2:     Path(A);
3: else
4:     Path(B);
5: end if
```

A typical GPU warp consists of 32 threads. Branch divergence takes place when different threads in a warp embarks on executing different paths. The different paths are traversed one at a time until all have been visited. This causes different paths to be executed in a serialised manner, thus defeating the purpose of parallelism. Algorithm 1 shows a basic example of an algorithm that could cause branch divergence on a GPU. In this example, threadIdx.x represents the id of the thread which ranges from 0 to 31 (32 threads). The algorithm executes Path(A) for an odd and Path(B) for an even number of the thread id. Figure 2.5 illustrates a graphical presentation of branch divergence in Algorithm 1. It can be demonstrated from the example that either of the execution paths take half of the number of threads. Since the threads in a warp cannot execute different paths concurrently, only the threads which are part of the same execution path may run concurrently. Thus, the total run time would be the total time for the execution of all the paths. Based on this, it is noted that the example in Algorithm 1 would theoretically face about 50% performance loss.

An alternative example shown in Algorithm 2 avoids branch divergence in Algorithm 1. In this example, Path(A) is executed for all the threads whose id is less than 32. The algorithm still creates different execution paths but are distributed over different warps/blocks compared to Algorithm 1 where the different execution paths belong to the same warp.

Algorithm 2 Minimising Branch Divergence on GPU

```
1: if (threadIdx.x & 32) then
2:     Path(A);
3: else
4:     Path(B);
5: end if
```

2.6.2 Memory Divergence

Memory Divergence is considered to be another bottleneck that could cause major performance degradation in applications which are memory-intensive by nature. It is believed that memory divergence has the biggest impact on performance compared to other bottlenecks. Once again, the memory divergence is typically associated to a group of threads

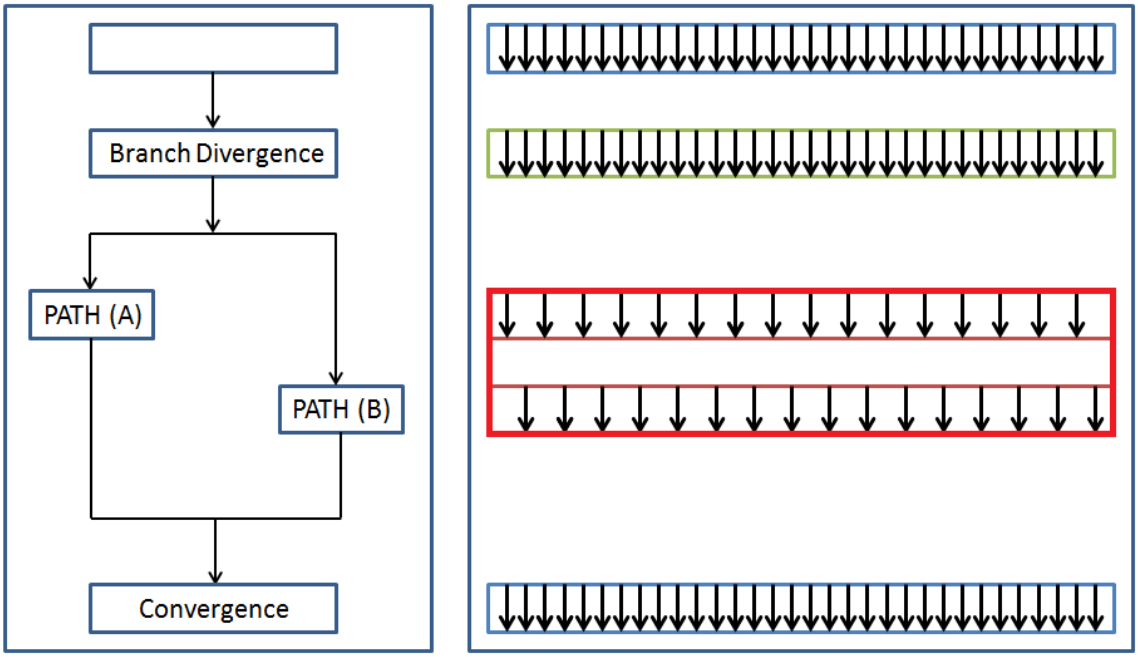


Figure 2.5: Illustration of branch divergence on a GPU based on Algorithm 1 demonstrating the flow of the algorithm (left) and it's associated behaviour in terms of the distribution of threads (right).

(warp) where each thread accesses data in the global memory which is not cache-line aligned, thus having uncoalesced memory accesses. A warp size of 32 threads could have up to 32 individual cache accesses, resulting in an enormous high latency.

2.6.3 CUDA

When discussing GPUs, CUDA is often mentioned. CUDA is an architecture that provides great performance increase by utilizing the GPUs. The platform provides an API that enables developers to use GPU for general computing. CUDA has become dramatically important to researchers and the number of institutes teaching CUDA is increasing. CUDA can be used to overlap complete execution and I/O of GPUs [7].

When performing parallelization using libraries, parallel programs are prone to errors and bugs that degrade performance. Research suggests that having a bug while executing a CPU program with a couple of threads may have very low probability of occurrence. The same will have a high probability of occurring in a program with a massive number of threads. At the same time, debugging and testing tools are also not widely available in an extreme parallel environment [2].

2.6.4 Global Memory

Memory is the predominant aspect of any computing device as memory is present in almost every device that performs some sort of processing. In particular, memory allows the system or device to operate and perform the tasks as required. Global memory is the primary memory that the GPU operates on and it's a virtual address space that can be mapped to the memory on the device, and this can be thought of as memory that is accessible by every component of the GPU, and is usually dynamic [52].

Global memory plays a huge role on the performance of the GPU, which results in the overall performance of the system being affected, especially when parallelism is a huge concern. However, CUDA allows for more efficient parallelism through assisting programs to decide on the most effective way to decompose a single, usually huge, problem across threads using multithreading. This will ultimately allow threads to share and communicate resources to solve problems. Ultimately, this is exactly where global memory become part of the problem solving, as the processing done has to be stored on some mechanism.

With regards to performance, a GPUs primary concern is to manage global memory latency without affecting the performance of the application. In these devices, threading is a big part of ensuring that the device reaches the required performance standard. A solution to this may be to create enough threads to occupy the GPU while the other threads are waiting to access the global memory and resides in DRAM. This could mean that the number of threads required will depend on the percentage of global memory access.

Another performance concern could be the limit of throughput due to global memory bandwidth. This could be problematic as resources are usually limited and by reducing the number of resources from a process to provide another process with more resources could ultimately reduce the entire performance. Concurrently, limiting the number of threads running simultaneously could result in pressure on global memory's bandwidth. Alleviating the pressure involves using additional registers and shared memory to reuse data [52].

Shared memory is expected to be a low-latency memory, which is similar to L1 cache. Shared memory can therefore, provide for high-performance communication and data sharing among the threads of a thread block [42]. The data in shared memory can be shared among all threads in a thread block, enabling interthread data reuse. Shared memory is also much faster than local and global memory, because it is on a chip. The two-way arrows in Figure 2.6 indicate read/write capability where each thread can [31]:

- Read/write per thread registers;
- Read/write per thread local memory;

- Read/write per block shared memory;
- Read/write per grid global memory;
- Read/only per grid constant memory.

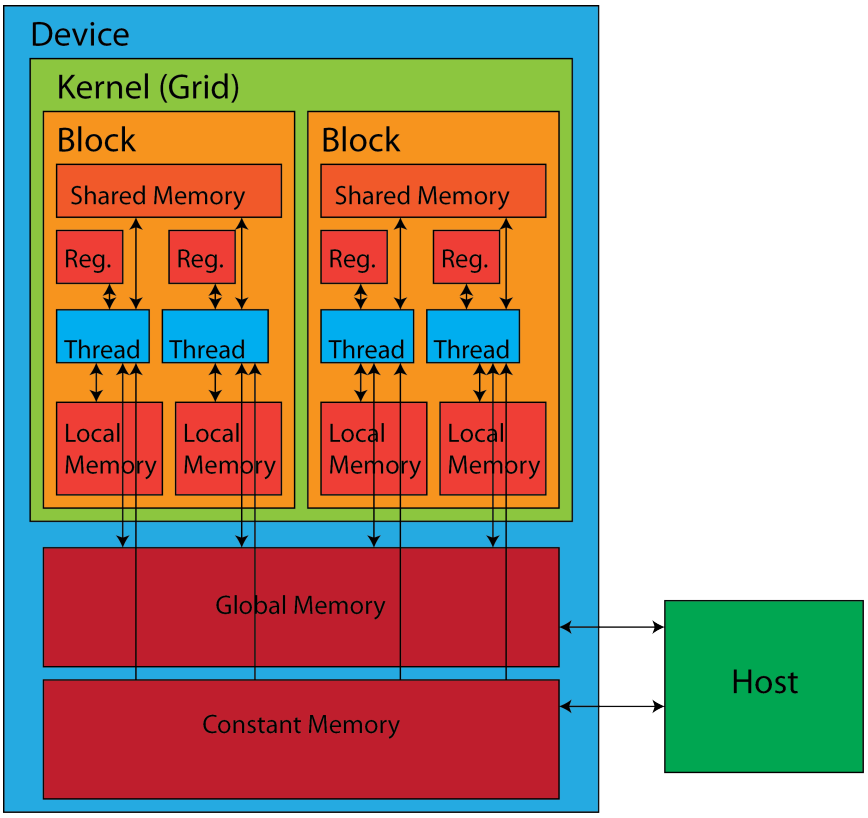


Figure 2.6: Global memory architecture [12].

2.7 Constant and Texture Memory

In some cases, we may have data that doesn't require any alterations, or the data is fixed. This means there has to be a place to store constant data. Hence the need for constant memory, this is used to store data that doesn't change over a course of a kernel execution. The use of constant memory as opposed to global memory will assist with the memory bandwidth being reduced.

Besides reducing memory bandwidth, constant memory improves the speed of the CUDA code for multiple reasons. Such reasons include constant memory that is widely known to be cached which reduces the number of transactions of retrieval, and it is important to note that reading threads that belong to the same address from cached memory is excessively

faster than the traditional read. Constant memory resides in the memory of the device and has a constant cache where all the caches are stored [52].

The downside of using constant memory is that the size of the cache is very small and limited (usually 64KB), which means that the system has to perform operations extremely fast in order to use and empty the cache as fast as possible. To avoid this issue, we could copy the data needed to constant memory, followed by reading the data into cache. Once the processing has completed, the data stored in constant memory could be removed.

Texture memory as constant memory, forms part of the read-only memory family. The purpose of texture memory is to reduce the amount of time taken to execute a particular operation, depending on the access pattern used. Texture memory is similar to constant memory in the sense that both are on a chip and cached. This is extremely useful and efficient as the number of memory requests to the DRAM off-chip will be reduced. The main principle behind texture memory is that threads are more likely to read from memory addresses that are located in a location that is in the near address domain of the adjacent threads. Figure 2.7 demonstrates the access of memory by threads.

To achieve the best performance, the thread will have to read texture addresses that are close to one another. This will significantly reduce the number of transactions made, thus increasing performance. The benefit of using texture memory as opposed to global memory is that performance should increase, since texture memory accesses neighboring locations in memory, thus reducing access time considering the memory locations are adjacent to one another.

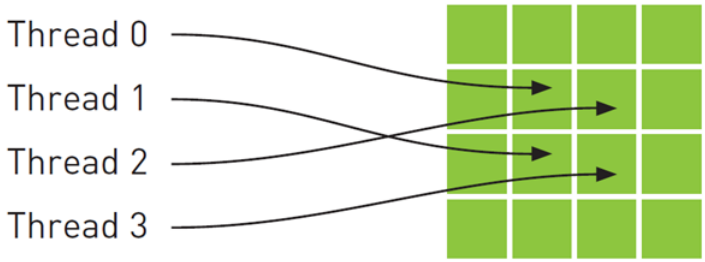


Figure 2.7: A thread is likely to read from the nearby addresses that nearby threads read [43].

The downside of using texture memory is that it is read-only which limits the use of data stored in texture memory. As opposed to global memory, if coalesced, the read is significantly faster and can be written and read consistently, unlike the read of texture memory which could be slower in certain circumstances, especially when the memory locations are

not adjacent and if the global memory is not coalesced, and texture memory is read-only which limits the operations performed on the data stored in texture memory.

2.7.1 Central Processing Unit

2.7.1.1 Streaming SIMD Extensions Instructions

Streaming SIMD Extensions (SSE) has a set of eight registers each consisting of 128 bits. Once data is loaded into the registers, the instructions are carried out directly and in most cases the instructions are performed simultaneously. SSE instructions is one of the multiple methods used to promote SIMD (Single Instruction and Multiple Data) parallelism in common types of processors [25].

Single Instruction and Multiple Data stream is a class of Parallel computer [21]. This describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. Thus, promotes parallelism. SIMD is particularly applicable to common tasks like adjusting the contrast in a digital image or adjusting the volume of digital audio [21].

SIMD is a class of Parallel computers, whereby different sets of data are executed in parallel. SIMD models are used for solving problems which have regular structures. The advantage of SIMD is that it uses far less amount of hardware control, which results in the performance being boosted and the level of parallelism in a SIMD system is typically much higher. The reason why SIMD computers require less hardware is because they only have one control unit [21]. Another advantage is that SIMD computers require less memory since just one copy of the program needs to be stored. Figure 2.8 depicts how the SIMD computational model is implemented in general.

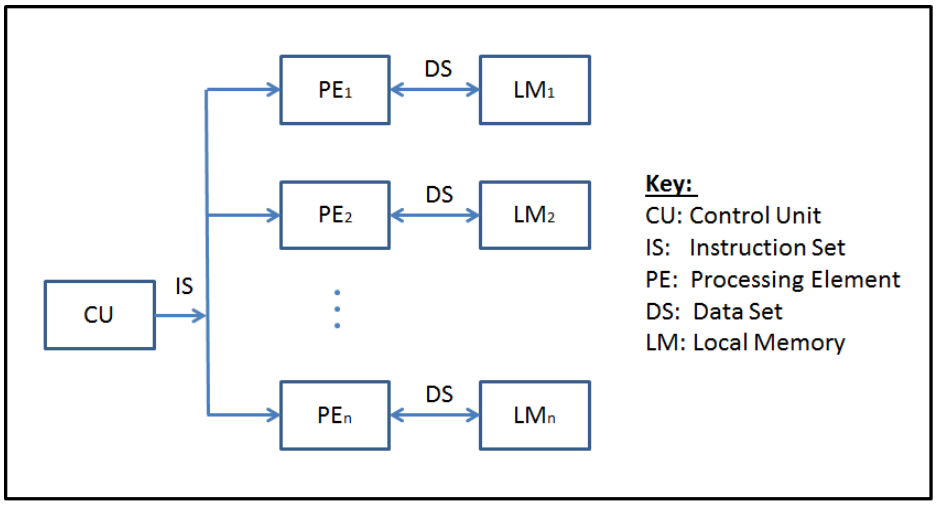


Figure 2.8: Implementation of the SIMD computational model [21].

In this organisation, multiple processing elements work under the control of a single control unit. It has one instruction and multiple data streams. All the processing elements of this organization receive the same instruction broadcast from the CU. Main memory can also be divided into modules for generating multiple data streams acting as a distributed memory as shown in the diagram. Therefore, all the processing elements simultaneously execute the same instruction and are said to be 'lock-stepped' together. Each processor takes the data from its own memory and hence it has on distinct data streams. (Some systems also provide a shared global memory for communications.) Every processor must be allowed to complete its instruction before the next instruction is taken for execution. Thus, the execution of instructions is synchronous. As SSE instructions are specific and in some cases unique to a processor, it would be beneficial to look up the version support for a specific processor. In general, SSE code will run faster, if run on the supported processors, and thus does not depend on so many factors.

The eight 128 bit registers were first implemented for single-precision computations and processing, typically for float data types. SSE2, which extends on the SSE instructions mainly to fully replace MMX, registers can make use of any of the primitive data types. The eight registers can be visualized as in Figure 2.9.

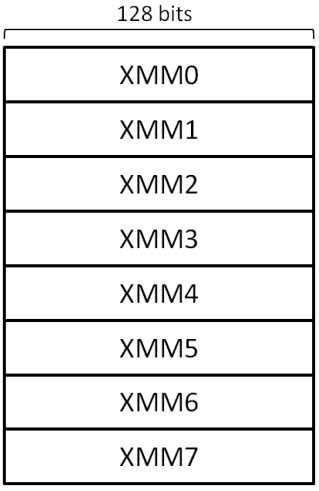


Figure 2.9: Eight XMM 128-bit registers.

2.8 Fingerprint Verification on Mobile Device

As many latest mobile phones are launched with an integrated fingerprint sensor, online applications that require authentication are being customized to allow fingerprint enrolling and identity verification using mobile devices. The change in trend makes it important for mobile application developers to implement power-efficient and real-time fingerprint

recognition methods to be performed on mobile devices. In [48], fingerprint verification using mobile devices is presented. Furthermore, it is shown that using mobile GPU for fingerprint feature-extraction tends to not only be fast, but also more power efficient than using mobile CPU [48].

2.9 Conclusion

This chapter introduced the various concepts necessary to understand fingerprint verification, and the various methods used for fingerprint recognition from past literature. Those methods being, texture and minutiae recognition which improve reliability and convenience. Owing to the extraction of unreliable features caused by issues such as noise and distortion, feature extraction methods such as binarisation and ridge following are used to combat such issues, whereby ridge following is a more superior approach. Ultimately, although minutiae based extraction may be more popular, texture based extraction provides a higher accuracy and reduces processing time.

Subjects such as High Performance Computation is a very important field for optimisation and for working with large datasets which require fast computation and a powerful tool capable of processing a large number of data. Without the use of these powerful tools, fingerprint recognition would not be an efficient process for feature extraction. However in the field of HPC, CPUs have been slowly replaced by GPUs, which are more capable of performing up to trillions of computations per second. However, due to their power hungry nature, and other bottleneck issues such as performance degradation there are limitations associated with it which needs to be adhered to. Those bottlenecks being, branch divergence, memory divergence, and global memory. Where branch divergence minimizes the impacts of thread divergence. GPUs are often associated with the CUDA architecture, which provides great performance increase when utilising a GPU. Interestingly, the power of GPUs can be used in applications such as fingerprint recognition on mobile devices to check for identity verification.

Becoming familiarised with the literature from the past has helped to develop the methodology implementation required to solve the objectives to the problem. The methodology will be stated in the following chapter.

Chapter 3

Methodology

3.1 Introduction

The previous chapter presented various methods for fingerprint recognition and their applications. Having investigated this, this chapter presents and discusses the main goals of the proposed research, along with the methods that have been collected through the literature review process. Using the data collected and an analysis of the data, we can answer the research questions.

This chapter explains and formally states the details of the research methodology. Section 3.2 presents the developmental approach to solving the large scale fingerprint recognition problem. Followed by a motivation to the chosen MCC recognition method being the most flexible and scalable approach, in Section 3.3. Section 3.4 describes the process of collecting data and the generation of each fingerprint data using the open-source project *Anguli*. Section 3.5 then describes the feature extraction process from the data. Once all the data in the database have been understood and processed for easy access, Section 3.6 begins the MCC implementation, and finally Section 3.7 provides and describes the layout of the proposed distributed fingerprint matching framework. This optimised framework is tested to check for accuracy and to analyse its performance.

3.2 Developmental Approach

The ultimate goal of this research is to develop a framework that fulfils the requirement of large scale fingerprint recognition problem. There are a number of dependencies that are crucial for the successful development of this framework. In order to implement the fingerprint recognition framework, the following steps are conducted:

- Investigate the fingerprint recognition method that is flexible, scalable and can be integrated in the proposed framework;

- Implement a known fingerprint recognition method that promises good accuracy and high performance;
- Generation of fingerprint data;
- Extracting of features from the data;
- Setting up and configuring database for easy access to bulk fingerprint data for analysis;
- Integrate the fingerprint recognition system in the framework and optimise it in the current environment;
- Test the framework for accuracy and analyse performance.

These steps are further explained in the next sections.

3.3 Recognition Method

Currently, there are many state-of-the-art fingerprint recognition solutions available [41]. Recently, many of the proposed methods are minutiae based which are often related and similar to previously suggested minutiae based methods. The reason for minutiae based approach to be popular is known to be due to on going intensive amount of research conducted on this approach. While many methods promise good accuracy, only few actually are known to provide good overall accuracy while ensuring high performance with scalability for a large-scale fingerprint recognition system. Amongst them, MCC [4, 44, 17] is known to be one of the methods that is recently introduced and provides fingerprint recognition invariant to rotation and translation. It also promises scalability with real time matching performance. The flexibility provided by this method allowed us to further explore this state-of-the-art technique. We decided to use MCC for this research project because it is alignment-free and computationally light which fits our criteria perfectly. The full description of MCC is given in section 3.6.

3.4 Data Generation

Due to the nature of the project, massive amounts of synthetic fingerprints were required. Based on the literature, we find that SFinGe [6] method may be used to satisfy fingerprint data requirement. Thus, we initially planned to implement the method as no algorithm is provided with the method proposed. It was further discovered that an open-source project

```
<Anguli>
    <FingerprintGen>
        <NumScratches high="3" low="0"/>
        <NoiseLevel high="8" low="0"/>
        <NumThreads value="1"/>
        <OutDir value="/path/to/directory/"/>
        <ImprPerFinger value="1"/>
        <customSeed value="1" seedValue="200087"/>
        <SaveMetaInfo value="1"/>
        <ImageType value="png"/>
        <FingerPerDir value="1000"/>
        <StartFinger value="1"/>
        <ClassDistr value="0"/>
        <NumFingerprints value="200000"/>
    </FingerprintGen>
</Anguli>
```

Figure 3.1: An example of one of the configuration file used to generate a group of 200,000 fingerprint data

Anguli as described in Section 2.3.3, is readily available for use which is based on [6]. Thus *Anguli* was the best choice as it allows customisations to be made to its source code.

Anguli allows various different configurations to be used to generate fingerprint data. These configurations were used extensively in order to ensure a variety of data is available to be used for effective analysis.

The data is initially generated and distributed across many directories on multiple hard drives. Each main directory includes 200 sub-directories where each sub-directory holds 1000 generated fingerprint data. This way, 200,000 total generated synthetic fingerprint data is stored in a main directory. The process is repeated 300 times, enabling 60 million fingerprint data to be generated in 300 main directories. For every group of 200,000 fingerprint data, the same configuration parameters are used except for the seed value. Table 3.1 presents the main parameters available with its description. Figure 3.1 shows an example of one of the configuration file used to generate a group of 200,000 fingerprint data.

Anguli claims to be capable of generating about one million fingerprints taking close to 4 days when using 8 physical cores. Data generation in our environment resulted in similar performance. Although, the *Anguli* source code could possibly have room for optimisation, this was overlooked for now since fingerprint data generation for this project is a once off task, and time spent optimising was not necessarily justified. In order to minimise the number of days required to generate all 60 million fingerprint data, a computer cluster with more than 40 nodes had to be configured and utilised to speed up the data generation process.

Parameter	Input	Description
NumFingerprints	*number*	Number of fingerprints to be generated.
FingerPerDir	*number*	Number of fingerprints per directory. Used only for generating fingerprints. Default value is 1000.
ClassDistr	*distribution*	Fingerprint class Distribution. Class distributions are natural, arch, arch, right loop, left loop, double loop and whirl loop. Default distribution is natural.
ImpPerFinger	*number*	Number of impressions per fingerprints. Can be used for generating fingerprints and impressions. Default value is 0.
SaveMetaInfo	*None*	Enables saving of meta information, like class of fingerprint, in text file of corresponding finger. Default is disabled.
numThreads	*number*	Number of threads to be created for generating fingerprints and impressions. Can be used for generating fingerprints and impressions. Default value is 1.
NumScratches	*number* *number*	Minimum and maximum number of scratches to be added to impressions. Default value is 0.
NoiseLevel	*number* *number*	Minimum and maximum number of noise levels[0, 8] to be applied to impressions. Default value is 0.

Table 3.1: *Anguli* configuration parameters with descriptions.

3.5 Feature Extraction

Based on the literature survey above, there have been many suggested methods that can be used to extract features. Extracting features from each fingerprint from the bulk dataset could take an enormous amount of computation time. Since the aim of this research project is mainly centered towards performance of fingerprint matching, it was deemed best to include feature extraction process within the data generation process. This strategy promised to avoid delays caused by re-accessing of data for feature extraction after the data is generated and stored. Additionally, it also ensured accuracy as the main features could easily be extracted while being generated thus not requiring additional methods to recognise and then extract features which is also prone to some accuracy degradation. *Anguli* source code had to be modified and recompiled to achieve this.

The process of extracting features combined with data generation increased the total processing time of the process mentioned in Section 3.4 causing more work to be assigned to each processing core of the computer cluster. The extracted features were stored in a uniquely created file in the sub-directory as the original generated fingerprint, for easy future access.

Since it was decided to use a minutiae based method for fingerprint recognition, while considering extracting important information from the fingerprints, only those features were

$$
\begin{array}{|l|}
\hline
w \\
h \\
n \\
x_1, y_1, \theta_1 \\
x_2, y_2, \theta_2 \\
\vdots \\
x_n, y_n, \theta_n \\
\hline
\end{array}
$$

Figure 3.2: An example of a text file containing data of extracted minutiae from a fingerprint.

extracted that were required for a minutiae based fingerprint matching method. In this case, we extracted minutiae points (termination and bifurcation) and the orientation (angle) of the points from the generated fingerprints. Figure 3.2 demonstrates the information stored related to extracted minutiae points where the first line represents the width of the fingerprint image; similarly, the second line indicates the height of the fingerprint image; the value n in the third line reports the number of minutiae points extracted from the fingerprint; and the third line is followed by n lines where each line represents the coordinates (x, y) of the minutiae point in the fingerprint followed by its orientation θ (in degrees).

3.6 MCC Implementation

Implementation of methods can be often time consuming and very complicated mainly due to the fact that often proposed methods are not provided with the complete details which makes it difficult to achieve the ideal results as claimed in the literature. In order to ensure perfect MCC implementation, it was critical to follow the important details and use the same parameters as used in [4]. Thus, before MCC could be implemented, the first task was to get the necessary fingerprint minutiae data in the similar format as required by the technique.

Since MCC is purely minutiae based fingerprint recognition method, its approach is to use every minutia found in the fingerprint to create a unique spatial directional relationship between the minutia itself and the neighbouring minutiae located within its fixed radius. Let $M = \{m_1, m_2, ..., m_n\}$ be the set of minutiae of a fingerprint template M where m_i is defined by the parameters $(x_{m_i}, y_{m_i}, \theta_{m_i})$. The parameters x_{m_i}, y_{m_i} represent the location coordinates of the minutia m_i in the fingerprint template whereas $\theta_{m_i} \in [0, 2\pi]$ denote the orientation of the minutia. Using this information, for every minutia m_i, a fixed size unique structure is formed by creating a (3D) cylinder. The height of this cylinder is 2π and radius

is predefined by R. The cylinder of a given minutia m_i holds information related to the neighbouring minutiae of m_i which attributes to the uniqueness of the cylinder. The details of how cylinders are created are discussed in the following subsection.

Based on the minutia parameters defined above, the extracted feature template as shown in Figure 3.2 did not have to be modified much in order to obtain the desired minutiae template format for our MCC implementation. Since the generated fingerprint contained fixed width and height, it was not necessary to keep the dimension of the fingerprint image in every template. Furthermore, for the sake of consistency and repeating the method with the same details as proposed in [4], the θ value in Figure 3.2 was transformed from degrees to radians.

3.6.1 Cylinder Creation

One of the major steps in MCC is to create a descriptor for each minutia. The cylinder is centered at the minutia and represents the spatial and directional relationship between the minutia and its neighbouring minutiae located within a fixed radius R. This descriptor is created using a linearised cylinder whose dimensions are related to the directional and spatial information and its volume contains the weighted spatial and angular distances. This results in a fixed length descriptor which is invariant to rotations and translations and may survive against skin distortions. Furthermore, computing similarities between these cylinders from different fingerprints can be significantly fast. We use the proposed method in [4] to compute these 3D cylinders for our dataset. The method briefly explains how the cylinders can be computed. However, some of the steps mentioned are generalised with little information. The details of major steps to create cylinders are left for the implementers to figure out. As it can be understood the reason for not providing precise details of the steps could be to maintain focus on the overall method presented.

Let $M = \{m_1, m_2, ..., m_n\}$ be the set of all the minutiae found in a fingerprint M. The number of cells in a cylinder is defined by $N_c = N_s \times N_s \times N_d$ where N_s and N_d are predefined parameters representing the cylinder's base and height respectively. For each minutia m, Let $c_m = \{c_1, c_2, ..., c_{N_c}\}$ be the representation of the cylinder's cells of a given minutia m where $c_{m,h} \in \{0, 1\}$ and $h \in [1, N_c]$. For simplicity, cells of the cylinder can be represented in 3D by 3 indices, namely i, j, k where $i, j \in [1, N_s]$ denote the base of the cylinder and $k \in [1, N_d]$ denotes the height of the cylinder. Thus, h can then be represented using a tuple (i, j, k) as $h = N_d((N_s \times (i-1)) + (j-1)) + k$. Figure 3.3 provides a visualisation of a cylinder. Furthermore, for each cell, the location $L^m_{i,j}$ of the center of the cell is calculated, protruded on the base of the cylinder where

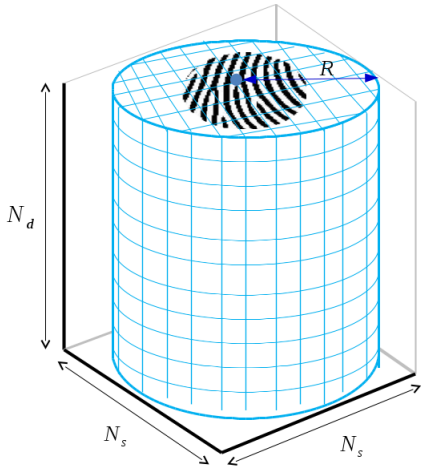


Figure 3.3: An example of a 3D cylinder with radius R and height 2π. The cells of the cylinders are divided into $N_s \times N_s \times N_d$ sections.

$$L_{i,j}^m = \begin{bmatrix} x_m \\ y_m \end{bmatrix} + \frac{2R}{N_s} \cdot \begin{bmatrix} cos(\theta_m) & sin(\theta_m) \\ -sin(\theta_m) & cos(\theta_m) \end{bmatrix} \cdot \begin{bmatrix} i - \frac{N_s+1}{2} \\ j - \frac{N_s+1}{2} \end{bmatrix} [4] \tag{3.1}$$

As discussed in Section 3.6, for every cell $c_{m,h}$ in a cylinder, a bit value is computed which represents the spatial and directional contribution between the minutia m and the minutiae found in the neighbourhood $\mathcal{N}_{L_{i,j}^m}$ of the cylinder's cell $\forall m_s \in M, m_s \neq m$ and $D_s(m_s, L_{i,j}^m) \leq 3\sigma_S$. $D_s(a,b)$ is the euclidean distance between a and b and $3\sigma_S$ is the predefined radius of the neighbourhood. The value for $c_{m,h}$ is then computed as [4]

$$c_{m,h} = Z\left(\Sigma_{m_s \in \mathcal{N}_{L_{i,j}^m}} \left(G_S\Big(D_s(m_s, L_{i,j}^m)\Big) \cdot G_D\Big(dF(d\varphi_k, dF(\theta_m, \theta_{m_s}))\Big)\right)\right) \tag{3.2}$$

where

$$G_S(y) = \frac{1}{\sigma_S\sqrt{2\pi}} e^{(-\frac{y^2}{2\sigma_S^2})} \tag{3.3}$$

is the Gaussian distribution of euclidean distances with the neighbourhood minutiae m_s and

$$G_D(\Gamma) = \frac{1}{\sigma_D\sqrt{2\pi}} \int_{\Gamma-\frac{\pi}{N_d}}^{\Gamma+\frac{\pi}{N_d}} e^{(-\frac{y^2}{2\sigma_D{}^2})} dy \tag{3.4}$$

where $G_D(\Gamma)$ represents the Gaussian distribution under a specific area with the predefined standard deviation σ_D. $dF(\theta_a, \theta_b)$ presents the directional difference between between θ_a and θ_b whereas $d\varphi_k$ is the angle at height k defined to be

$$d\varphi_k = -\pi + (k - 0.5) \cdot \frac{2\pi}{N_d} \tag{3.5}$$

The function $Z(w)$ is the logistic function (defined below) which takes the accumulated spatial and directional contribution for a minutia and produces a value between 0 and 1.

$$Z(w) = \frac{1}{1 + e^{-\tau(w-\mu)}} \tag{3.6}$$

where μ and τ are part of the predefined parameters. Furthermore, the value of $Z(w)$ is converted to a bit value with

$$Z(w) = \begin{cases} 1, & \frac{1}{1+e^{-\tau(w-\mu)}} \geq \mu \\ 0, & otherwise \end{cases} \tag{3.7}$$

Additionally, the number of cylinder cells can be reduced by discarding the (invalid) cells that do not produce enough information. This is achieved by creating a convex hull [47] using all the minutiae in M and validating whether a cell falls within the convex hull. The validation is defined as

$$c_{m_i,h} = \xi(L_{i,j}^{m_i}) = \begin{cases} valid & L_{i,j}^{m_i} \in C_H(M, \Omega) \\ invalid & otherwise \end{cases} \tag{3.8}$$

where $C_H(M, \Omega)$ is a convex hull of all the minutiae M which is expanded by Ω pixels [4]. The convex hull has to be expanded precisely in order to ensure the cylinder's cells are perfectly validated. We deduce that using a centroid point of M, the convex hull can be efficiently expanded.

Let $M = \{m_i\}_{i=1}^{n}$ where $m_i = [x_{m_i} \quad y_{m_i}] \in \mathcal{R}^2$ with x_{m_i}, y_{m_i} representing the location coordinates of minutia. The centroid of the minutiae points can be calculated as

$$C_M = \frac{1}{n} \sum_{i=1}^{n} m_i \tag{3.9}$$

where $C_M = [C_{M,x} \quad C_{M,y}]$ is the centroid of the set M. Now let $M_v = \{v_1, v_2, \cdots, v_n\}$ be the vertices of the original convex hull of points M where $v_i = [v_{i,x} \quad v_{i,y}] \in M$. $\forall i \in \{1, \cdots, n\}$ the new expanded vertex $\mathbf{v_i}$ can be computed as

$$\mathbf{v_i} \leftarrow \alpha \cdot (\mathbf{v_i} - \mathbf{C_M}) + \mathbf{C_M} \tag{3.10}$$

where α is the scalar that is calculated using

$$\alpha = 1 + \frac{\Omega}{\sqrt{v_{i,x}^2 + v_{i,y}^2}} \tag{3.11}$$

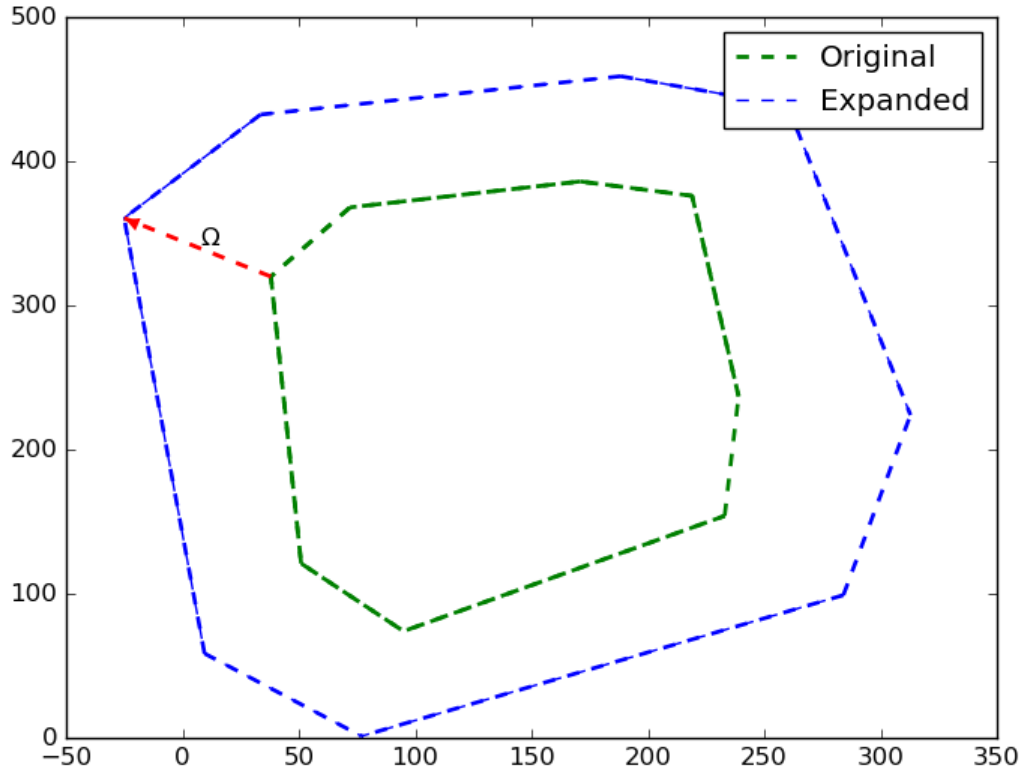


Figure 3.4: An example of an enlarged convex hull where the corners represent convex hull vertexes and $\Omega = 75$

where Ω is the given number of pixels that the original convex hull is expanded by. Figure 3.4 demonstrates a sample convex hull together with an enlarged convex hull calculated using the above method with $\Omega = 75$. The corners represent the vertices in the dotted line convex hulls.

We decided to pre-compute these cylinders for all the minutiae found in the fingerprints from our entire dataset using our cluster computing facility. The creation of cylinders revealed to be the slowest task to date. After our intensive investigation and considering the steps taken in [5] to reduce the number of cells in a cylinder, we decided not to take cells' validity into consideration as it was noted that ignoring the validity of cells provides almost no performance degradation when working with standard fingerprint acquisition sensors. These cylinders when successfully computed, were stored in a template form for each fingerprint separately. Figure 3.5 illustrates the template of the cylinders in which the lines followed by n denote the cylinder values $c_i = [c_{i,1}, c_{i,2}, \ldots, c_{i,N_c}] \in \{0,1\}^{N_c}$ where $\mathbf{c}_i$ is the bitwise cylinder representation of minutia m_i. The parameters used in creating these

Table 3.2: Parameter values for creating cylinders [4]

Parameter	Description	Value
R	Cylinder Radius	75
N_s	Number of cells along the diameter	8
N_d	Number of cylinder sections	5
σ_S	Standard deviation	6
σ_D	Standard deviation	0.436
Ω	pixels by which convex hull is expanded by	75
μ, τ	Sigmoid parameters	0.005,400
min_{VC}	Minimum number of valid cells for a cylinder to be valid	20%
min_M	Minimum number of minutiae for the cylinder to be valid	1

cylinders are provided in Table 3.2 which are defined in [4]. The reason of creating all the cylinders once off and storing them, is to avoid creation of cylinders for every fingerprint in the dataset during fingerprint matching process. Computing of cylinders every time during fingerprint identification process leads to a computationally demanding task which could severely impact the fingerprint identification time.

3.6.2 Database

During the phase of cylinder creation for the entire dataset of fingerprints, we began storing fingerprint's minutiae information in a PostgreSQL database. We choose PostgreSQL for its known scalability and strict data integrity capabilities. This was to ensure organised collection and storage of data which could also be utilised in future. The database contained 300 tables with each table storing 200,000 fingerprint data. The division of the data was conducted to allow faster and easy access to smaller chunk of data without the need to go through the entire data. It also allowed the data to be distributed over several systems

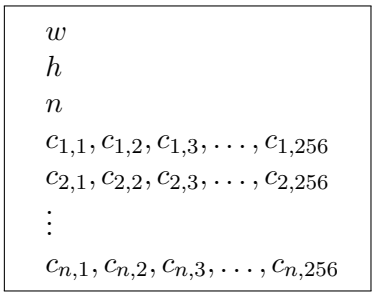


Figure 3.5: Cylinder template.

easily, thus providing scalability. Table 3.3 provides the structure of the tables. As it can be noted that there are four 64-bit *cylinderValue* fields which represent the 256 bit cylinder value of a minutia divided into chunks of four 64-bit values. The *popCount* value reflects the total number of 1's found in all the *cylinderValue* fields. This value was pre-computed to avoid repeated computation of cylinder's population count during fingerprint recognition process.

3.6.3 Fingerprint Matching and Recognition

In order to find similarities between two fingerprints using MCC, according to [4], the matchable cylinders of the fingerprints are compared pairwise to derive a local similarity score. The local similarity score determines the likelihood of how similar the cylinders are. Two cylinders, $\mathbf{c_i}$ and $\mathbf{c_j}$ are matchable if the directional difference $dF(\theta_i, \theta_j)$ between their minutiae (θ_i and θ_j) is less than the predefined threshold value δ_θ. Hamming distance can then be used to find similarities between two cylinders. Subsequently, the similarity score between the matchable cylinders $\mathbf{c_i}$ and $\mathbf{c_j}$ can be computed as

$$H_{i,j} = 1 - \frac{\|\mathbf{c_i} \oplus \mathbf{c_j}\|}{\|\mathbf{c_i}\| + \|\mathbf{c_j}\|} \tag{3.12}$$

where $\mathbf{c_i} \oplus \mathbf{c_j}$ is the exclusive or (XOR) between bit-vectors $\mathbf{c_i}$ and $\mathbf{c_j}$, $\|\cdot\|$ is the norm of bit-vector. The norm of bit-vector can be computed by calculating the square root of the Hamming weight of the bit-vector. The Hamming weight of a bit-vector is simply the number of non-zero values.

After the local similarities between all the cylinders' of two fingerprints are computed pairwise, the highest np similarity scores are selected to calculate the global (final) score between the fingerprints. np defines the number of cylinder pairs $(\mathbf{c_i}, \mathbf{c_j})$ that are taken for consideration to calculate the global score. The value of np is not usually constant and depends on the number of minutiae available in two fingerprints [4]. However, we define this value to be constant because the maximum np value that can be attained is

Table 3.3: Database structure

Field	Description	datatype
fId	Unique identity assigned to a fingerprint	*integer*
minutiaId	Unique identity assigned to a minutia of a fingerprint	*integer*
angle	The orientation of the minutia	*double*
popCount	The number of 1's in the minutia's cylinder	*integer*
cylinderValue1	1 - 64 bits of the minutia's cylinder	*bigint*
cylinderValue2	65 - 128 bits of the minutia's cylinder	*bigint*
cylinderValue3	129 - 192 bits of the minutia's cylinder	*bigint*
cylinderValue4	193 - 256 bits of the minutia's cylinder	*bigint*

12 in [5] whereas in our dataset, we don't have any fingerprint with less than 12 minutiae. Subsequently, for the identification problem, the global score is computed between the *query* and every fingerprint in the database. The fingerprint (in the database) with the highest global score against the *query* fingerprint is recommended to be the identical fingerprint, given that the global score is above certain threshold (α). Checking of the best global score against the threshold is necessary to avoid cases where the *query* fingerprint does not exist in the database, which could produce false positive results.

3.6.4 Baseline CPU Implementation and Optimisation

In order to investigate the MCC method and identify gaps, we begin with the fingerprint identification implementation by following the MCC baseline implementation and optimisations suggested in [5]. Our baseline implementation shown in Algorithm 3 is derived from [5]. The algorithm expects a template of a query fingerprint (F_q) as well as the templates of N fingerprints F (dataset) to which the query fingerprint is expected to be matched and identified against. The algorithm compares every minutia in the query fingerprint pairwise with all the minutiae in the dataset fingerprints (F). When two minutiae a and b are compared, their similarity score is only considered after their directional difference $dF(\theta_a, \theta_b)$ is validated to be below the predefined threshold δ_θ. Once the similarity score for all the minutiae of a fingerprint F_i is computed pairwise with the minutiae of query fingerprint, the scores are sorted in descending order. After which the first np scores are selected and their average is computed. This average (score) is then regarded as the global score between F_q and F_i. This process is repeated for the rest of the fingerprints in F after which the fingerprint with the highest global score is recommended to be the most identical fingerprint to F_q, given that the score is above the predefined threshold value α.

We then apply the following optimisations to the baseline sequential implementation:

- Counting-sort [11] promises to significantly speed up performance compared to quick/merge sort for sorting the local similarity scores. The speed up in performance is the result of reduction of the computational complexity from $\mathcal{O}(n \log(n))$ to $\mathcal{O}(n + t)$, where t represents the number of values into which all the local similarity scores are quantised;

- Instead of computing the norms of cylinders at every comparison, the norms are precomputed to reduce the computation time;

- Utilisation of SSE instructions on 128 bit registers reduces the number of SSE instructions required to compute XOR of the cylinders. Additionally, *popcnt* instruction is used to calculate the number of non-zero values in a bit-vector;

- A pre-computed look-up table is used to compute the square roots;
- Reducing floating-point operations by storing non-integer values as integers using fixed-point arithmetic.

Algorithm 3 Baseline MCC Implementation

Input:
- F_q, Query Fingerprint
- N Fingerprints, $F = \{F_0, F_1, ...F_{N-1}\}$

Output:
- k, index of the most identical fingerprint in DB with $H_{k,q} \geq \alpha$
- $\{\emptyset\}$, if no identical fingerprint found.

$highestScore = 0$;
$globalScores = new\ List()$;
$localScores = new\ List < List > ()$;
for $i = 0 \rightarrow N - 1$ **do**
  $localScores.Add(new\ List())$;
  for each minutia $M_a = (\mathbf{c_a}, \theta_a) \in F_q$ **do**
    for each minutia $M_b = (\mathbf{c_b}, \theta_b) \in F_i$ **do**
      if $dF(\theta_a, \theta_b) \leq \delta_\theta$ **then**
        $score = 1 - \frac{\|\mathbf{c_a} \oplus \mathbf{c_b}\|}{\|\mathbf{c_a}\| + \|\mathbf{c_b}\|}$;
        $localScores[i].Add(score)$;
      end if
    end for
  end for
  $S = Sort(localScores[i])$;
  $aggregateScore = \frac{Sum(S[0:n_p])}{n_p}$;
  $globalScores.Add(aggregateScore)$;
  if $aggregateScore > highestScore$ **then**
    $highestScore = aggregateScore$;
    $k = i$;
  end if
end for
if $highestScore \geq \alpha$ **then**
  $return\ k$;
end if
$return\ Null$;

After implementing the baseline algorithm (Algorithm 3) and the optimisations mentioned above, we successfully verify the results claimed by MCC [5] in terms of accuracy and time. Our computation time is slightly lower than the ones claimed in [5] but this can be justified by the fact that our CPU (harware specification is shown in Table 3.4) is not as powerful **as** the one used in [5].

3.6.5 Improvements

The above implementation is optimised and produces impressive results. However, we note that the pairwise comparison between the query and fingerprints in the database is the most computationally demanding task. Although, comparing every minutia (cylinder) pairwise ensures the best accuracy, there are large amount of comparisons whose scores are discarded and as such they do not influence the accuracy. Based on this observation, we deduce that a new approach needs to be introduced to the MCC method in order to minimize the number of comparisons made and achieve a reduced computation time without compromising the accuracy.

3.6.5.1 Clusterisation

Let $F = \{F_1, F_2, ..., F_N\}$ be the number of fingerprints in the dataset with $F_i = \{m_1, m_2, ..., m_{n_i}\}$ being the set of n_i arbitrary number of minutiae belonging to fingerprint F_i. Calculating the global score between two fingerprints (F_i and F_j) involve calculating the similarity scores pairwise between the fingerprints where only the best np ($np \leq n_i \times n_j$) scores are used to calculate the global similarity score. In other words, the np scores that are chosen to compute the global score, derive only from the np cylinders (for a specific fingerprint) which have the highest similarities with the cylinders of the query fingerprint. This encourages us to clusterise the cylinders of the fingerprints in the database based on the position of the bits in the bit-vectors (fingerprint's cylinders). Furthermore, the query cylinder may then be compared only against the cylinders of the cluster whose centroid (average) value has the highest similarity with the query cylinder. This approach promises to reduce the overall amount of comparisons previously made, thus bettering the identification time.

Method A

Our first approach of clusterisation is using *k-Means* clustering [19] where we distribute cylinders based on their Hamming distance against the cluster's centroids. Let $F = \{F_1, F_2, ..., F_N\}$ be the set of N fingerprints in the database and $F_i = \{c_{i,1}, c_{i,2}, ...c_{i,k_i}\}$ be the set of k_i arbitrary number of minutiae cylinders for fingerprint F_i where $c_{i,j} \in \{0,1\}^{256}$. Let $M_c = \{c_{1,1}, c_{1,2}, ..., c_{1,k_1}, ..., c_{N,1}, c_{N,2}, ..., c_{N,k_N}\}$ be the set of cylinders of all the fingerprints in F. For the ease of notations, let $M_c = \{c_1, c_2, ..., c_k\}$ where k is the total number of minutiae (cylinders) in all the fingerprints F i.e, $k = \sum_{i=1}^{N} k_i$. We define the set of the clusters to be $S = \{S_1, S_2, ..., S_{N_k}\}$ where N_k is the predefined number of clusters and $S_i = \{c_{i,1}, c_{i,2}, ...c_{i,n_i}\}$ with $c_{i,n_i} \in \{0,1\}^{256}$ and n_i be the arbitrary number of cylinders allocated to S_i. Then, the centroid of these clusters can be defined as $C = \{C_1, C_2, ..., C_{N_k}\}$ where $C_i \in \{0,1\}^{256}$. Initially, a random cylinder $\mathbf{c} \in M$ is allocated each centroid $\mathbf{C_i}$. Hence, every cylinder $\mathbf{c_p} \in M_c$, is allocated to a cluster in S using

$$S_i = \{\mathbf{c_p} : \frac{\|\mathbf{C_i} \oplus \mathbf{c_p}\|}{\|\mathbf{C_i}\| + \|\mathbf{c_p}\|} \leq \frac{\|\mathbf{C_l} \oplus \mathbf{c_p}\|}{\|\mathbf{C_l}\| + \|\mathbf{c_p}\|} \forall l, 1 \leq l \leq N_k\} \tag{3.13}$$

Once all the minutiae in M_c are assigned to their relevant sets, we update each centroids using

$$C_i(j) = \begin{cases} 1 & \sum_{k=1}^{n_i} S_{i,k}(j) \geq \frac{n_i}{2} \\ 0 & otherwise \end{cases} \tag{3.14}$$

Where n_i is the number of cylinders that were previously assigned to S_i and $j \in [1, 256]$. We repetitively assign the minutiae into the sets and update the centroids upon completion of assignments until the algorithm converges. The algorithm converges when the assignments no longer change.

Method B

In **Method A**, the centroid values of clusters S in C are prone to become biased towards some cylinders. This behaviour mainly occurs when the new assignments (or cylinders) have small differences in comparison to the centroid value of the cluster but gradually influence the centroid value as the number of assignments increases. Thus, the centroid values may not be good representation of the assignments which leads to some assignments being no longer represented by their respective clusters. Usually, the impact of this behaviour can be reduced by increasing the number of clusters which may allow the cylinders to be distributed more effectively. However, this may significantly increase the computation complexity as more clusters would need to be searched when performing identification process.

We introduce a unique method (in the context of MCC) which enables us to calculate the centroid values in C with a better central tendency and cluster the minutiae more effectively. The method is based on dividing the elements of minutia in contiguous equal sized chunks, and creating predefined number of blocks with overlapping parts of the neighbouring chunks. Let $M_c = \{c_1, c_2, ..., c_k\}$ be the set of k number of minutiae cylinders of all the fingerprints in the database and

$$N_B = (2 \times \frac{B_T}{B_S}) - 1 \tag{3.15}$$

be the total number of blocks into which each cylinder c_i is divided. B_T is the bit-vector size for every minutia cylinder c_i and B_S represents the predefined size (number of bits) for each block to be allocated. Now, every cylinder $c_i \in M_c$ is represented as B_i where $B_i = \{1, ..., N_B\}$, $B_{i,j} \in [0, 1]$ and $j \in [0, N_B - 1]$. We can then compute $B_{i,j}$ as

$$B_{i,j} = \frac{popCnt(c_i[k, k+1, ..., k+B_S-1])}{B_T}, k = \frac{j \times B_S}{2}, \tag{3.16}$$

where $popCnt(a)$ = total number of 1's present in the bit-vector a. Figure 3.6, illustrates a graphical representation of the overlapping blocks of a minutia cylinder. After computing

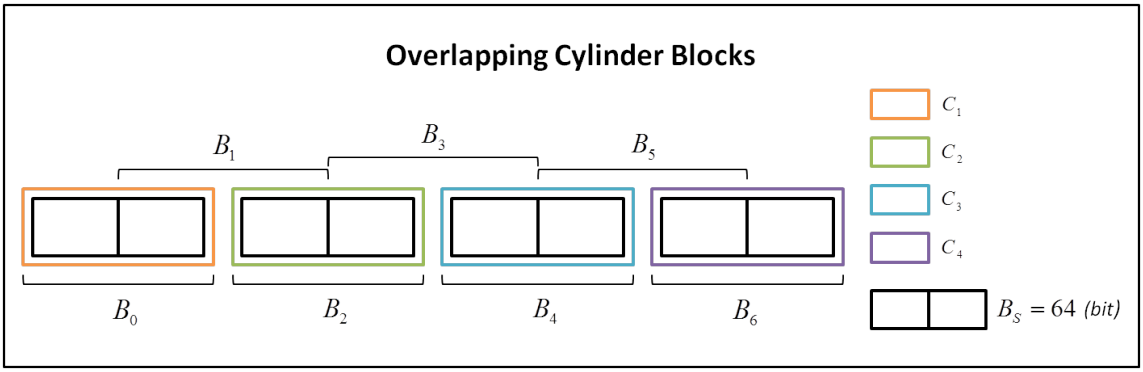


Figure 3.6: Overlapping block representation of a minutia cylinder of 256 bit-vector.

the values of overlapping blocks, we use *k-means* clustering on the cylinders' blocks. This time, we use *euclidean* distance to find the best similarity between the blocks and cluster centroids. The cluster centroids are computed by simply taking the average of the values of the blocks that fall under the respective cluster. This method promises better performance as each cluster is a better representation of the minutiae cylinders that it contains. Additionally, depending on the B_S value, the number of elements in each block may effectively be small resulting in less number of comparisons when comparing against the centroid value and thus, may lead to reduced computation time.

After combining the optimisations suggested previously with our clusterisation approach (method B), the resulting algorithm is shown in Algorithm 4, where N represents the total number of fingerprints in the database and PopCnt(.) calculates the population count in the bit-vector using *popcnt* instruction. indexOfClosestCentroidFor(.) returns the index of the cluster whose centroid has the least euclidean distance with the given minutia.

3.6.6 GPU Implementation

As discussed in the literature review above, general-purpose computing on graphics processing unit (GPGPU) have been very efficient in terms of reducing computation time of computationally intensive tasks. We decide to implement the proposed fingerprint matching method to validate as well as understand and observe the behaviour of the methods in a highly parallelised environment, which the GPU's offer. Our baseline GPU implementation is shown in Algorithms 5, 6 and 7. Algorithm 5 runs on the host (CPU) which processes the tasks to GPU and collects results. Algorithms 6 and 7 execute on the GPU kernel which is called and initialised by the host. The algorithms are responsible for calculating local similarity scores (CalculateSimilarityScore(.)) and then accumulating the local scores into global score (AccumulateScore(.)) respectively for every fingerprint in the database. $nC(.)$

Algorithm 4 MCC Clusterisation Implementation

Input:

- F_q, Query Fingerprint
- N_k Fingerprint Clusters $S = \{S_1, S_2, ..., S_{N_k}\}$ and their Centroids $C = \{C_1, C_2, ...C_{N-k}\}$

Output:

- k, Index of the most identical fingerprint in DB with $H_{k,q} \geq \alpha$
- $\{\emptyset\}$, if no identical fingerprint found.

$highestScore = 0$;
Set $bucket[i][j] = 0; \forall\ 0 \leq j \leq t\ and\ 0 \leq i < N$
for each minutia $M_a = (\mathbf{c_a}, \theta_a, \eta_a) \in F_q$ **do**
  $v = indexOfClosestCentroidFor(M_a)\ in\ C$;
  for each minutia $M_b = (\mathbf{c_b}, \theta_b, \eta_b, i_b) \in S_v$ **do**
    if $dF(\theta_a, \theta_b) \leq \delta_\theta$ **then**
      $L_s = \left\lceil t \cdot \frac{Lut_{sqrt}[PopCnt(\mathbf{c_a} \oplus \mathbf{c_b})]}{\eta_a + \eta_b} \right\rceil$
      $bucket[i_b][L_s] += 1$;
    end if
  end for
end for
for $i = 0 \rightarrow N - 1$ **do**
  $j = 0, h = n_p, sum = 0$;
  while $j \leq t\ and\ h \geq 0$ **do**
    $sum = sum + min(bucket[i][j], h) \cdot j$;
    $t = t - min(bucket[i][j], h)$;
    $j = j + 1$;
  end while
  $sum = sum + t \cdot h$;
  $aggregateScore = 1 - \frac{sum}{np \cdot w}$;
  if $aggregateScore > highestScore$ **then**
    $highestScore = aggregateScore$;
    $k = i$;
  end if
end for
if $highestScore \geq \alpha$ **then**
  $return\ k$;
end if
$return\ Null$;

represents the number of cylinders that are present in the (v^{th}) given cluster S_v. T_1 and T_2 represent the number of threads per block that are launched with the GPU kernel call where as N is the total number of fingerprints in the database. The information of the GPU device on which we perform fingerprint identification process is shown in Table 3.5.

Algorithm 5 GPU Clusterisation Implementation of MCC: CPU Execution

Input:

-F_q, Query Fingerprint

- N_k Fingerprint Clusters $S = \{S_1, S_2, ..., S_{N_k}\}$ and their Centroids $C = \{C_1, C_2, ...C_{N-k}\}$

Output:

- k, Index of the most identical fingerprint in DB with $H_{k,q} \geq \alpha$
- $\{\emptyset\}$, if no identical fingerprint found.

1: $highestScore = 0$;
2: Set $bucket[i][j] = 0; \forall\ 0 \leq j \leq t\ and\ 0 \leq i < N$
3: Copy C to GPU memory
4: **for each** minutia $M_a = (\mathbf{C_a}, \theta_a, \eta_a) \in F_q$ **do**
5: $v = indexOfClosestCentroidFor(M_a)\ in\ C$;
6: Copy v and M_a to GPU memory.
7: Call GPU Kernel CalculateSimilarityScore() with $\frac{nC(S_v)}{T_1}$ blocks and T_1 threads per block
8: **end for**
9: Call GPU Kernel AccumulateScore() with $\frac{N}{T_2}$ blocks and T_2 threads per block
10: Copy GlobalScores $\in \mathbb{N}^{N \times 1}$ from GPU memory to Host memory.
11: **for** $i = 0 \rightarrow N - 1$ **do**
12: $aggregateScore = GlobalScores[i]$;
13: **if** $aggregateScore > highestScore$ **then**
14: $highestScore = aggregateScore$;
15: $k = i$;
16: **end if**
17: **end for**
18: **if** $highestScore \geq \alpha$ **then**
19: $return\ k$;
20: **end if**
21: $return\ Null$;

Algorithm 6 GPU Clusterisation Implementation of MCC: GPU Execution CalculateSimilarityScore()

Input:

- $M_a = (\mathbf{c_a}, \theta_a, \eta_a)$, Query Fingerprint Minutia
- N_k Fingerprint Clusters $S = \{S_1, S_2, ..., S_{N_k}\}$
- v, Index of fingerprint cluster which needs to be scanned.

Output:

- Bucket $\in \mathbb{N}^{N \times w}$

1: $k = T_1 * blockIdx.x + threadIdx.x$;
2: $M_b = (\mathbf{c_b}, \theta_b, \eta_b, i_b) = S_v[k]$;
3: **if** $dF(\theta_a, \theta_b) \leq \delta_\theta$ **then**
4: $\quad L_s = \left\lfloor w \cdot \frac{Lut_{sqrt}[PopCnt(\mathbf{c_a} \oplus \mathbf{c_b})]}{\eta_a + \eta_b} \right\rfloor$
5: $\quad bucket[i_b][L_s] += 1$;
6: **end if**

Algorithm 7 GPU Clusterisation Implementation of MCC: GPU Execution AccumulateScore()

Input:

- Bucket $\in \mathbb{N}^{N \times w}$

Output:

- N Global Scores, $G_S = \{S_0, S_1, ..., S_{N-1}\}$

1: $k = T_1 * blockIdx.x + threadIdx.x$;
2: $j = 0, t = n_p, sum = 0$;
3: **while** $j < t$ *and* $h > 0$ **do**
4: $\quad sum = sum + min(bucket[k][j], h) \cdot j$;
5: $\quad t = t - min(bucket[k][j], h)$;
6: $\quad j = j + 1$;
7: **end while**
8: $sum = sum + t \cdot t$;
9: $G_S[k] = 1 - \frac{sum}{np \cdot t}$;

3.7 Framework

The layout of the proposed distributed fingerprint matching framework is shown in Figure 3.7. As illustrated, the framework is a comprehensive fingerprint management system with the following main capabilities

- Query a fingerprint;
- Enrollment of new fingerprints;
- Adding/removing of agents (computation nodes).

Master Node is responsible for coordination activities between the computation (child) nodes and to keep the child nodes updated with changes in the database. The child nodes are responsible for distributed computing and receive their assignments from the master node. *Gateway* is the core module of the master node which is a point of contact between the user and the framework. It is responsible for processing and routing the user's request into the appropriate module. Additionally, it also processes the output received from the modules and channels it back to the user. The message may contain the information about the particular request that the user is making followed by the source of the file that contains the piece of information in a structured manner for the given request. The gateway is responsible for extracting the information. The *Enrollment* process is briefly shown in Figure 3.8. The process involves extracting minutiae details from the fingerprint and converting it to our fingerprint template as shown in Figures 3.2 and 3.2. The template is then used to calculate the values required for our database structure (Figure 3.3). These values are sent to the child nodes in order to update their individual copies of fingerprint minutiae clusters to accommodate the new enrolled fingerprint. Since we use the clustering method for identification process as discussed in Section 3.6.5.1, every minutia from the new enrolled fingerprint must be assigned to the appropriate clusters using the clustering method (**B**) mentioned previously. Once all the minutiae are assigned to their respective clusters, the cluster centroids have to be updated to incorporate these new changes. Let m be the minutia that is newly assigned to a cluster. The cluster S_t on which the minutia is assigned, its centroid value is updated using:

$$C_t \leftarrow \frac{n_t \cdot C_t + C_m}{n_t + 1}, \tag{3.17}$$

where C_t represents the centroid value of the cluster that is being updated and C_m is the value that represents the minutia m. n_t is the number of assignments that exist in the cluster prior to this new assignment.

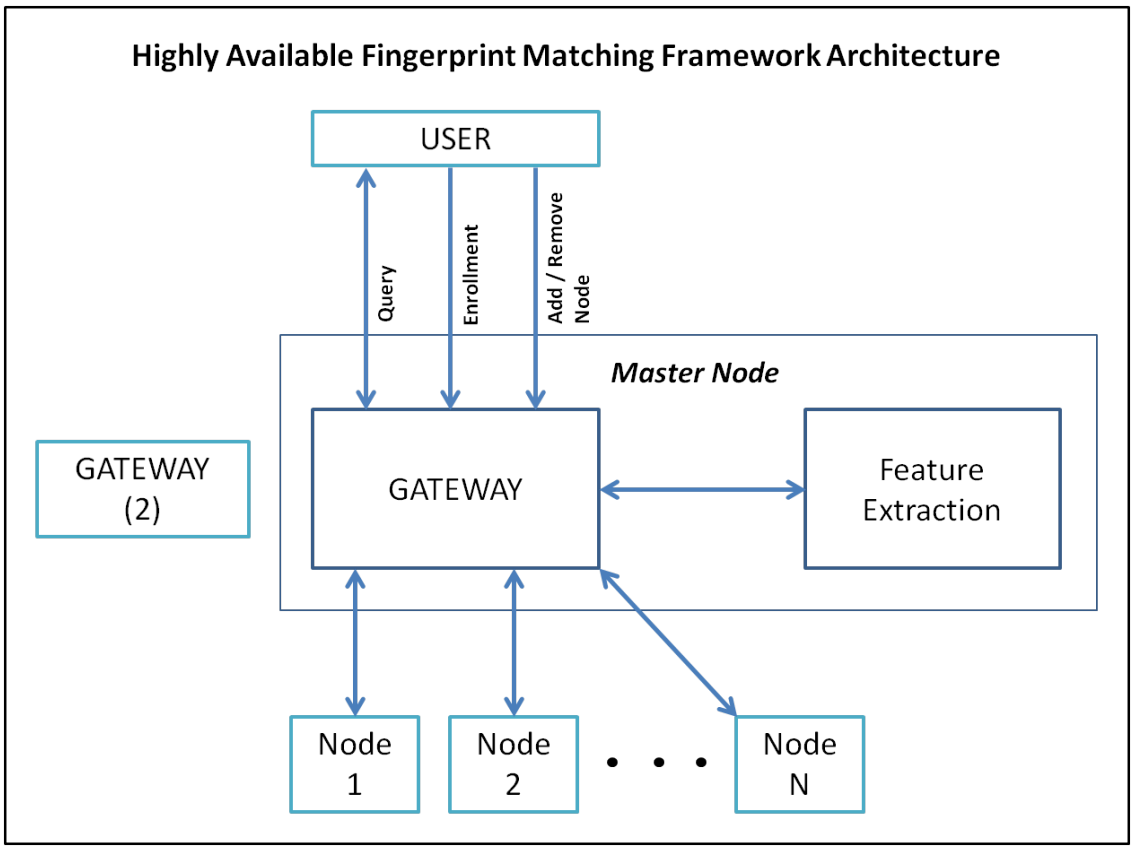


Figure 3.7: Distributed fingerprint matching framework

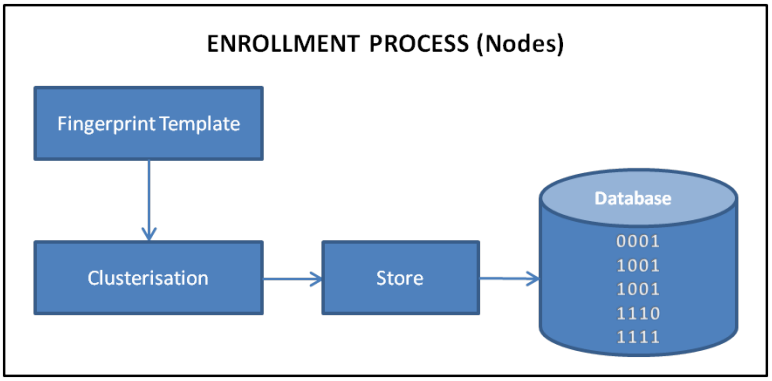


Figure 3.8: Fingerprint enrollment process in the distributed fingerprint matching framework.

Matching process refers to performing an identification on a query request received by the user against all the fingerprints in the database. The process is mainly done on the child nodes but the extraction of information from the query fingerprint and assigning of the task to child nodes is dealt by the master node. Each child node performs the identification on a portion of query minutiae set against the clusters and responds back to the master node. The response is the the score matrix that the child node has derived based on the similarity scores computed against the minutiae set which is allocated to the individual child node. The master node then accumulates the these scores by performing an addition to all the score matrices received by children nodes. This allows master node to determine the best match (if present) in the database for the given query fingerprint. The number of query minutia allocated to child nodes is dependent on the number of children nodes currently available in the pool. Thus, the reason for keeping a copy (of all the clusters that represent the entire database) in every child node is that the number of minutia allocated to each child to perform matching process may dynamically be adjusted when the pool size changes. Figure 3.9 illustrates the matching process conducted by child nodes.

Scaling is an important factor when dealing with distributed systems. The framework allows users to add or remove child nodes by sending an appropriate request with the details of the node that needs to be added/removed. In the case of addition, the master node, copies all the clusters (that represent the fingerprint database) to the new child node and includes the new node in the pool of child nodes for future enrollment and matching requests. Similarly, when a specific node is requested to be removed, the master node removes it from the pool of child nodes and incorporates this change when distributing the query minutia set for the future matching request.

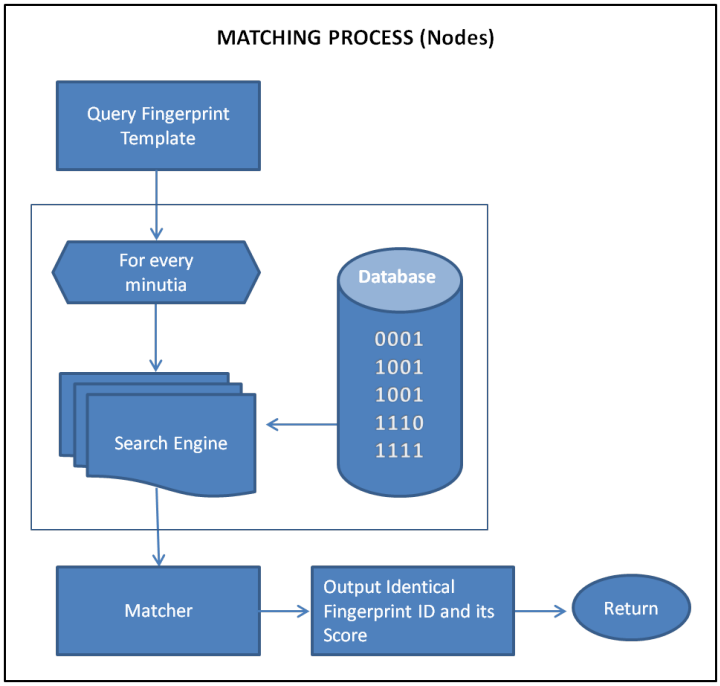


Figure 3.9: Fingerprint identification process in the distributed fingerprint matching framework.

Table 3.4: System specifications

Component	**Specification**
CPU	Intel i7 3770k
CPU Cache Memory	8 MB
RAM	32GB DDR3 1333MHz
Motherboard	Gigabyte - GA Z97X-UD3H
Hard Drive	80GB SSD
g++ - version	4.6.7

Table 3.5: GPU device information used for experiments.

Component	Specification
Device Name	Gtx Titan Black
Architecture	GK110 Kepler
Multiprocessors	15
Total CUDA Cores	2880
Maximum threads per multiprocessor	2048
Maximum threads per block	1024
Maximum registers per block	65536
Global memory	6 GB
Shared memory per block	48 KB
L2 cache size	1.5 MB
Constant memory	64 KB

3.8 Conclusion

This chapter described the research methodology in accomplishing the goal. In accomplishing the goal of reducing the overall number of comparisons made in the MCC implementation, clusterisation was introduced. This optimised approach proved to provide a better identification time, than compared to the baseline MCC implementation. From Section 3.6 the tests were conducted in order to prove which methodology implementation was better. The results conducted from the experiments are produced in the next chapter.

Chapter 4

Experiments and Results

This chapter reports configurations and an experimental environment, followed by an evaluation of the performance on both CPU and GPU (algorithms). Sections 4.1 Benchmark and 4.2 Performance provide comprehensive details of the experiments conducted on all the proposed algorithms and report their achieved performance. We then analyse and compare our results with other studies in Section 4.3 Discussion.

4.1 Benchmark

Our proposed algorithms are simply modifications of the different MCC algorithms defined in [5]. Therefore, in order to analyse the algorithms fairly and effectively, we define our benchmarks to be as similar as possible to the ones (in [5]). In [5] each test was conducted against the evaluation of the original MCC algorithm on the CPU and the GPU.

4.1.1 Dataset

We extract 250,000 fingerprints as our dataset for the experiments. These fingerprints are extracted from our main database of 60 million fingerprints as discussed in Section 3.6.2. The extracted fingerprints were randomly selected. The total number of cylinders (minutiae) for this dataset is 8,237,868 with 32.95 cylinders on average in a fingerprint template. Our query fingerprints consist of 100,000 fingerprints where 50,000 of these fingerprints are randomly picked from the above dataset of 250,000 fingerprints. The rest of the 50,000 query fingerprints are extracted from our main database while ensuring that none of these fingerprints belong to the previously extracted fingerprints. This is done to ensure that the accuracy of the proposed algorithms is tested with in-mate and non-mate fingerprints in the database. The average number of cylinders in the query fingerprints is 32.5.

4.1.2 Experiments

The following experiments are conducted to evaluate the proposed algorithms' performance in terms of time, accuracy and scalability.

- **Experiment A**: Performing 10 queries against the entire dataset of 250,000 fingerprints;
- **Experiment B**: Performing 10 queries against the different size of dataset starting from 10 fingerprints and scaling it logarithmically;
- **Experiment C**: Performing all 100,000 queries on the dataset starting from 50,000 fingerprints and incrementing it by 20,000 fingerprints until the dataset reaches the size of 250,000 fingerprints.

These experiments are motivated by the scalable benchmarks introduced and performed in [5] as they allow the fingerprint recognition algorithm to be thoroughly tested for their speed and accuracy. Each of the experiments above is conducted three times and an average of the results is reported as the achieved result.

4.1.3 Parameters

The parameters used for identification for all the experiments are provided in Table 4.1. All the experiments were conducted on a PC available at the time of this research. The specifications of the PC are listed in Table 3.4. The information of the GPU device on which we perform our GPU implementation of fingerprint identification is shown in Table 3.5. We use C++ for all our CPU based implementation and CUDA C for our GPU based implementation.

4.1.4 Algorithms

We perform our experiments on the different algorithms that we have previously discussed. These algorithms are summarised as follows.

- *CPU Baseline:* This refers to our implementation (Algorithm 3) of the baseline MCC algorithm on CPU which is reported in [5]. The reason for re-implementing this algorithm is to verify the results originally published in the literature. Additionally, this also provides an opportunity to have a better understanding of the possible shortcomings of the algorithm;
- *CPU Optimised*: This is an implementation of the optimised version of the MCC *Baseline* algorithm on the CPU which was originally introduced in [5] and discussed in Section 3.6.4;

Table 4.1: Parameter values used for matching.[4]

Parameter	Description	Value
n_c	Number of bits in each cylinder	255
N_k	Number of clusters (clustering method)	100
δ_θ	Maximum global rotation allowed between two minutia	$\frac{\pi}{6}$
t	Values for quantized local similarity scores	64
z	Values for quantized angles.	256
np	Parameter used to determine the number of best local similarity scores.	12
α	Threshold value at which the highest global score is acceptable	0.75
T_1, T_2	Threads per block in GPU implementation (Algorithm 5)	32, 64

- *CPU Clusterised*: Refers to the Algorithm 4 which involves the clustering method that was discussed in Section 3.6.5.1. We introduced this method with the intention of providing improvements to the previously optimised CPU implementation of MCC;
- *GPU Clusterised*: This is simply a GPU (parallel) implementation of the clustering method. The algorithm is discussed in detailed in Section 3.6.6.

The algorithms above are implemented in the same order as they are listed. Each algorithm is supposed to provide a better efficiency of its immediate previous implementation.

4.1.5 Evaluation

We evaluate the above mentioned algorithms for mainly their computation time and the accuracy obtained for each experiment. The computational time is recorded using a high resolution clock in milliseconds. For each algorithm, we begin recording the time from the moment a query is loaded, until the algorithm returns a unique identification key of the matched fingerprint (or *Null* in the case of no match found). As for accuracy, we determine the number of times each algorithm wrongly identifies a fingerprint in each of the experiments. In order to have a better understanding of the errors (wrong identifications), the errors are classified into the following categories:

- False Negative Match Rate (FNMR): Percentage of incorrect identifications for queries that are in the database.

- False Positive Match Rate(FPMR): Percentage of queries which are not in the database and are mistakenly identified.

4.2 Performance

Table 4.2 shows results obtained from executing **Experiment A** on all the sequential CPU implementations. These algorithms are discussed in detail above. The table reports the total execution time in milliseconds for the corresponding algorithms as well as the average amount of fingerprint comparisons for per unit of time in seconds. A total of 10 queries were used in this experiment along with the dataset consisting of 250,000 fingerprints. The table reports that the results of the *CPU Baseline* and *CPU Optimised* algorithms are in-line with the original implementation of those algorithms in [5]. This validates our implementation of the original MCC CPU based algorithms reported in [5]. Furthermore, the performance of the *CPU Clusterised* algorithm clearly shows that the clustering approach (Method B) that we introduced in Section 3.6.5.1, as anticipated, has reduced the execution time significantly. Consequently, this helped boost its average throughput to above 4.1 million matches per second. The throughput (KMPS) is calculated using

$$\text{throughput} = \frac{N \times N_q}{time} \tag{4.1}$$

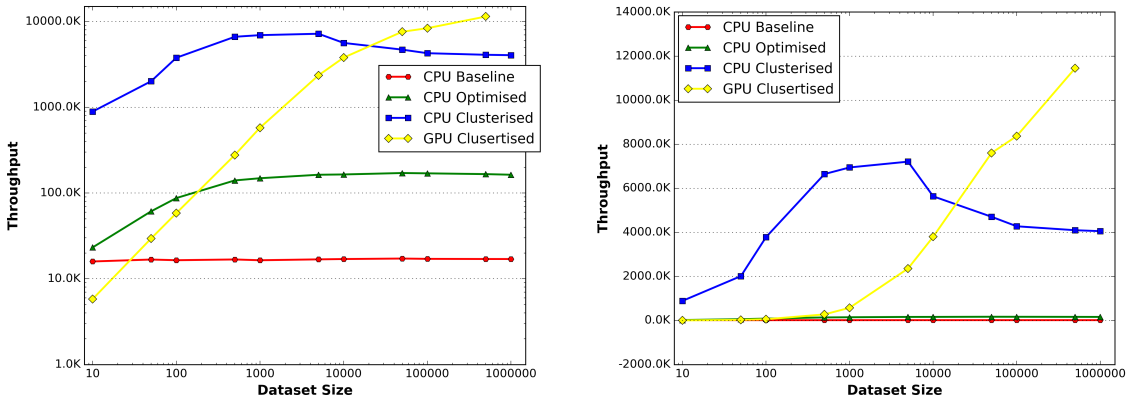
where N represents the total number of fingerprints in the dataset and N_q is the number of query fingerprints. *time* is the total time taken (in seconds) to execute the benchmark.

Figures 4.1(a) and 4.1(b) present the average throughput of all the fingerprint identification algorithms over different (increasing) dataset (**Experiment B**) with the total number of queries to be 10. Both figures illustrate the performance of the same experiment. However, in Figure 4.1(a) logarithmic scale is used for the throughput axis while in Figure 4.1(b), linear scale is used. The use of different axis scales is to show an in-depth view of the achieved results. It is evident from these results that the performance of the cluserting method is consistent as the dataset has increased logarithmically. This shows that the method is scalable. Some of the abnormalities are observed while executing small datasets, these observations are discussed in detail in the discussion section of this chapter.

Figures 4.2(a) and 4.2(b) present the execution time of **Experiment B** for the different size datasets. Once again, both the logarithmic and linear scale is used for the purpose of displaying the trends better in the same experiment. The results show that the time performance of the *CPU Baseline* algorithm gradually becomes exponential while the *CPU Clusterised* algorithm displays no scalability issues.

Table 4.2: Performance time of CPU implementations for ten queries on the complete dataset (**Experiment A**) and its associated throughput (KMPS: thousand matches per second)

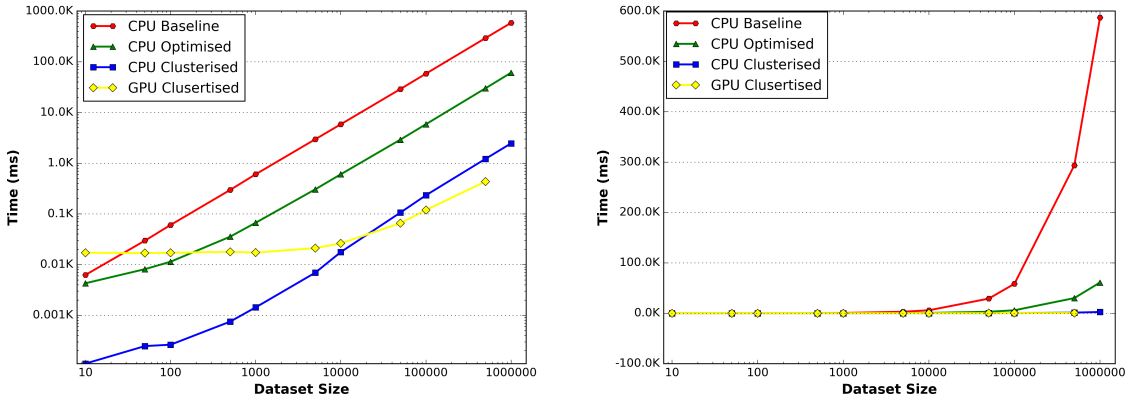
Algorithm	**Time (ms)**	**Throughput**
CPU Baseline	146711	17.04K
CPU Optimised	14747.5	169.52
CPU Clusterised	604.76	4133.88K



(a) Logarithmic scale used for both axis

(b) Linear scale used for the throughput axis

Figure 4.1: Results of **Experiment B**.



(a) Logarithmic scale used for the time axis

(b) Linear scale used for the time axis

Figure 4.2: Execution time (in milliseconds) for different size of dataset in **Experiment B**.

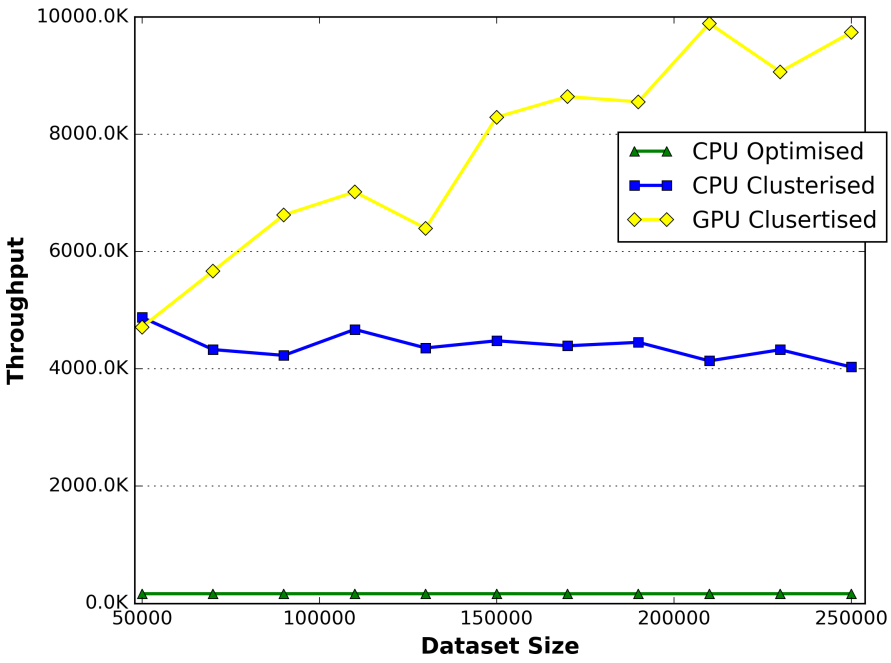
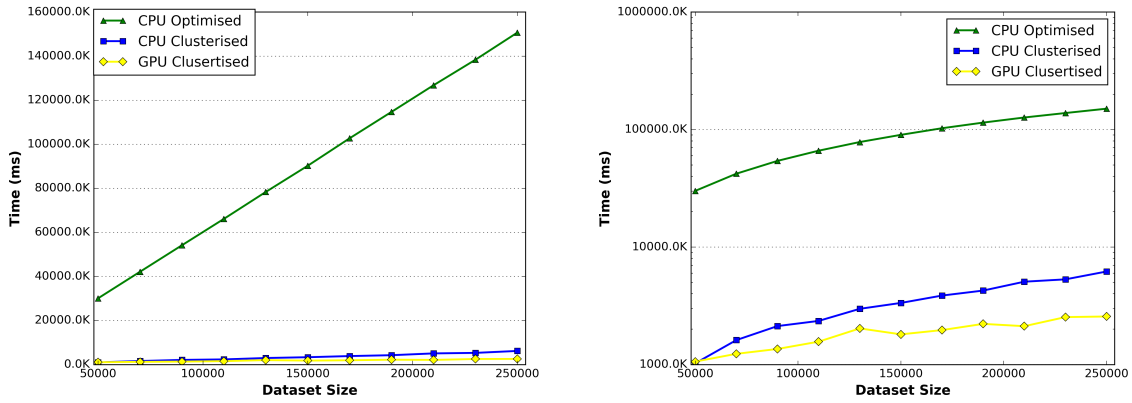


Figure 4.3: Results of **Experiment C**, displaying throughput (thousand matches per second) for the increasing size of datasets.

The results obtained above for the *CPU Clusterised* algorithm seem to be satisfactory. This allows us to proceed with more rigorous testing of the algorithm in order to test and verify any scalability issues. Hence, we conduct **Experiment C** which involves executing the fingerprint identification using 100,000 queries with half of the query fingerprints not included in the dataset of 250,000 fingerprints. This experiment is performed on all the discussed algorithms except for the *CPU Baseline* algorithm. The reason that we do not execute **Experiment C** on that particular algorithm is because the experiment would take weeks to finish. This could result in redundancy, especially when the *CPU Optimised* algorithm could instead be used to compare the proposed clustering method, since it is already established in [5] that the *CPU Optimised* is more efficient than the *CPU Baseline* algorithm.

Figure 4.3 depicts the results obtained from conducting **Experiment C**. From the results it is certainly noticeable that as the dataset increases linearly, the average throughput for the clustering method remains stable above 4 million fingerprint matches per unit of time in seconds. This together with the previous results show that the clustering method introduced in this study promises no scalability issues over large datasets. Furthermore, Figures 4.4(a) and 4.4(b) provide the execution time over varying dataset sizes for the same experiment. Recall, as previously mentioned, the time axis in both figures display a different



(a) Logarithmic scale used for the time axis

(b) Linear scale used for the time axis

Figure 4.4: Execution time (in milliseconds) for different size dataset in **Experiment C**.

Table 4.3: Performance time of all the implementations for 100,000 queries on the complete dataset (250,000 fingerprints) and its associated throughput (thousand matches per second).

Algorithm	**Time (seconds)**	**Throughput**
CPU Optimised	150,719.8	161.03K
CPU Clusterised	6,198.2	4,033.43K
GPU Clusterised	2,566.63	9,740.40K

scale. This is to showcase different point of views within the results obtained. From these figures, it is noticeable that the change in execution time is directly proportional to the dataset size for all the algorithms.

Finally, Table 4.3 presents the actual values of the results obtained when performing 100,000 queries on the entire dataset of 250,000 fingerprints using all the discussed algorithms (except for the *CPU Baseline* algorithm). The accuracy for all algorithms were carefully evaluated. **Experiment A** and **Experiment B** (involving only 10 queries) reported 0% for both FNMR and FPMR on all the algorithms. Meanwhile **Experiment C** reports 0.12% for FPMR and 0.03% for FNMR. The reported accuracy, FPMR and FNMR is achieved by all the algorithms on which the experiment was conducted. The accuracy obtained by these algorithms is consistent with the standard accuracy of the state-of-the-art fingerprint verification systems [14]

4.3 Discussion

After implementing the baseline algorithm (*CPU Baseline*) and the optimisations (*CPU Optimised*), we successfully verified the results claimed by MCC [5] in terms of accuracy and time. Our computation time as shown in Table 4.2, is slightly lower than the ones

claimed in [5]. This can be justified by the fact that our CPU (hardware specification is shown in Table 3.4) is not as powerful than the one used by author in [5].

As it was previously assumed, the computation time of fingerprint recognition is influenced by the size of the fingerprint dataset. This is now evident by the Figures 4.4(a) and 4.4(b) where the execution time is shown to be proportional to the size of the dataset.

From Figures 4.1(a) and 4.1(b), we note the sharp and abrupt changes in the throughput for the *CPU clusterised* algorithm. It can be seen, the throughput initially increases with the dataset and managed to provide over 7 million matches per second. When the dataset size reaches 10,000 fingerprints, it consistently provides throughput of just above 4 million matches per second. This behaviour is the result of the CPU cache memory being filled at its full capacity as the size of dataset increases above 10,000 fingerprints. Based on this observation, it can be safely assumed that if the CPU cache memory is increased, corresponding throughput can be increased proportional to the size of cache memory.

The results of the **Experiment B** and **Experiment C** for the proposed clustering method 3.6.5.1 look impressive as it successfully reduces the execution time by more than 24 times compared to the *CPU optimised* and more than 242 times in comparison to *CPU baseline* algorithm. Consecutively, the performance of the parallel execution (*GPU clusterised*) using a GPU shows a reduction of more than half of the execution time using the sequential (CPU) execution. The massive reductions in the execution times by the proposed clustering method 3.6.5.1 enabled us to increase the number of fingerprint matches in a given time. This is very critical when dealing with large fingerprint datasets.

Chapter 5

Conclusion and Future Work

This research focused on building a computationally efficient framework well suited for a large-scale distribution system, used for real-time fingerprint identification. The framework can be applied in various use-cases including forensics and civil applications, as fingerprint remains one of the most used biometric characteristics due to its uniqueness. The framework was developed after conducting a detailed study whereby various known fingerprint matching algorithms were considered. Thereafter, the best algorithm was chosen as a baseline algorithm for this study. We aimed to achieve optimal solution where high accuracy is maintained while ensuring real-time performance.

The aim was met when we introduced improvements (clustering method) to the existing best known fingerprint matching method and developed an optimal framework using these improvements. Recall that **Method B** produced the optimal solution with high accuracy and minimal computational complexity.

We proposed a new approach to the existing MCC fingerprint recognition method. Where we found that the overall number of comparisons were greatly reduced when the MCC implementation with our clustering method was used. We then performed experiments on different algorithms, to compare the efficiency of each algorithm. Those algorithms include, a *CPU Baseline*, a *CPU optimised*, a *CPU Clusterised*. In the order of the mentioned algorithms, each algorithm outperformed its previous counterpart.

Three experiments were conducted for each of the proposed algorithms, the findings presented in the previous chapter for each of the experiments for the implemented algorithms are as follows.

Experiment A's *CPU Baseline* and *CPU Optimised* results were in-line with the original implementation from the literature. While the *CPU Clusterised* algorithm has greatly improved on the former approaches, since the results have revealed a significant reduction in the execution time, and therefore boosted the average throughput rate. After running the algorithm, the results reveal a time of 146,711 ms and a throughput of 17.04K for the

CPU baseline. Indicating that the *CPU Baseline* is the least effective since it was simply a re-implementation of past published literature.

Experiment B produced satisfactory results for the CPU Clusterised algorithm. These satisfactory results led us to continue with more rigorous testing in **Experiment C**. **Experiment C**'s performance time for all the implementations for 100,000 queries on the complete dataset, consisting of 250,000 fingerprints and its associated throughput, has shown that the clustering approach has successfully and significantly reduced the execution time. Quite evident that there is an improvement compared to both *CPU Optimised* and *CPU Baseline* algorithms. It is noticeable that as the dataset size increases linearly, the average throughput for the clustering method on a sequential execution remains stable above 4 million fingerprint matches per unit of time in seconds. This together with the previous results, indicate yet again that the clustering method has the added advantage of promising no scalability issues over large datasets.

Overall, from all the results obtained from the various experiments, it is revealed that the CPU clustering algorithm provided the most efficiency. The results produced in this algorithm was exactly what we were working towards when building this large scaled fingerprint matching system. This system necessitates the need to build a good framework which can produce the greatest effect in reducing the execution time obtained in the experiments. It also attempts to produce optimal solutions that perform well and are accurate in the sense that there will be an increase to the number of fingerprint matches at a given time. This algorithm was able to identify the most number of fingerprints correctly in each experimental run, hence making this the best algorithm.

As future work, we plan to exploit and optimise the proposed clustering method for the GPU (parallel execution) as the current GPU algorithm is basic and provides very little performance increases. Furthermore, we also plan to study the aspects of reducing the size of dataset in the memory by utilising compression methods.

Bibliography

[1] Roli Bansal, Priti Sehgal, and Punam Bedi. Minutiae extraction from fingerprint images-a review. *arXiv preprint arXiv:1201.1422*, 2011.

[2] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of cuda programs. In *Third Workshop on Software Tools for MultiCore Systems*, 2008.

[3] Rich Brueckner. Nsf report: Universities show growing use of hpc, 2013.

[4] Raffaele Cappelli, Matteo Ferrara, and Davide Maltoni. Minutia cylinder-code: A new representation and matching technique for fingerprint recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(12):2128–2141, 2010.

[5] Raffaele Cappelli, Matteo Ferrara, and Davide Maltoni. Large-scale fingerprint identification on gpu. *Information Sciences*, 306:1–20, 2015.

[6] Raffaele Cappelli, Dario Maio, and Davide Maltoni. Synthetic fingerprint-database generation. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 3, pages 744–747. IEEE, 2002.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10):1370–1380, 2008.

[8] Tai Pang Chen, Xudong Jiang, and Wei-Yun Yau. Fingerprint image quality analysis. In *Image Processing, 2004. ICIP'04. 2004 International Conference on*, volume 2, pages 1253–1256. IEEE, 2004.

[9] Xinjian Chen, Jie Tian, and Xin Yang. A new algorithm for distorted fingerprints matching based on normalized fuzzy similarity measure. *Image Processing, IEEE Transactions on*, 15(3):767–776, 2006.

[10] Sharat Chikkerur, Chaohang Wu, and Venu Govindaraju. A systematic approach for feature extraction in fingerprint images. In *Biometric Authentication*, pages 344–350. Springer, 2004.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 8.2: Counting sort. *Introduction to Algorithms,*, pages 168–170, 2001.

[12] Cornell University. Memory architecture, 2016. [Online; accessed September 20, 2016].

[13] Amitava Datta and Subbiah Soundaralakshmi. Fast parallel algorithm for distance transform. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 33(4):429–434, 2003.

[14] Bernadette Dorizzi, Raffaele Cappelli, Matteo Ferrara, Dario Maio, Davide Maltoni, Nesma Houmani, Sonia Garcia-Salicetti, and Aurélien Mayoue. Fingerprint and on-line signature verification competitions at icb 2009. In *International Conference on Biometrics*, pages 725–732. Springer, 2009.

[15] Jianjiang Feng. Combining minutiae descriptors for fingerprint matching. *Pattern Recognition*, 41(1):342–352, 2008.

[16] Minglun Gong, Yilei Zhang, and Yee-Hong Yang. Near-real-time stereo matching with slanted surface modeling and sub-pixel accuracy. *Pattern Recognition*, 44(10):2701–2710, 2011.

[17] Pablo David Gutierrez, Miguel Lastra, Francisco Herrera, and Jose Manuel Benitez. A high performance fingerprint matching system for large databases based on gpu. *Information Forensics and Security, IEEE Transactions on*, 9(1):62–71, 2014.

[18] Pablo David Gutierrez, Miguel Lastra, Francisco Herrera, and Jose Manuel Benitez. A high performance fingerprint matching system for large databases based on gpu. *IEEE Transactions on Information Forensics and Security*, 9(1):62–71, 2014.

[19] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[20] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[21] Prabhudev S Irabashetti, B Gawali Anjali, and S Betkar Akshay. Architecture of parallel processing in computer organization. *American Journal of Computer Science and Engineering*, 1(2):12–17, 2014.

[22] Anil Jain, Lin Hong, and Ruud Bolle. On-line fingerprint verification. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(4):302–314, 1997.

[23] Anil Jain, Lin Hong, and Ruud Bolle. On-line fingerprint verification. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 19(4):302–314, 1997.

[24] Richa Jani and Nidhi Agrawal. A proposed framework for enhancing security in fingerprint and finger-vein multimodal biometric recognition. In *Machine Intelligence and Research Advancement (ICMIRA), 2013 International Conference on*, pages 440–444. IEEE, 2013.

[25] Hwancheol Jeong, Sunghoon Kim, Weonjong Lee, and Seok-Ho Myung. Performance of sse and avx instruction sets. *arXiv preprint arXiv:1211.0820*, 2012.

[26] Xudong Jiang and Wei-Yun Yau. Fingerprint minutiae matching based on the local and global structures. In *Pattern recognition, 2000. Proceedings. 15th international conference on*, volume 2, pages 1038–1041. IEEE, 2000.

[27] Xudong Jiang, Wei-Yun Yau, and Wee Ser. Detecting the fingerprint minutiae by adaptive tracing the gray-level ridge. *Pattern Recognition*, 34(5):999–1013, 2001.

[28] Manvjeet Kaur, Mukhwinder Singh, Akshay Girdhar, and Parvinder S Sandhu. Fingerprint verification system using minutiae extraction technique. *World Academy of Science, Engineering and Technology*, 46:497–502, 2008.

[29] HB Kekre, Sudeep Thepade, and Akshay Maloo. Eigenvectors of covariance matrix using row mean and column mean sequences for face recognition. *CSC-International Journal of Biometrics and Bioinformatics (IJBB)*, 4(2):42–50, 2010.

[30] Volodymyr V Kindratenko, Jeremy J Enos, Guochun Shi, Michael T Showerman, Galen W Arnold, John E Stone, James C Phillips, and Wen-mei Hwu. Gpu clusters for high-performance computing. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[31] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[32] Prasanna Krishnasamy, Serge Belongie, and David Kriegman. Wet fingerprint recognition: Challenges and opportunities. In *Biometrics (IJCB), 2011 International Joint Conference on*, pages 1–7. IEEE, 2011.

[33] Ajay Kumar and Cyril Kwong. Towards contactless, low-cost and accurate 3d fingerprint identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3438–3443, 2013.

[34] Henry C Lee. Re gaensslen eds. advances in fingerprint technology, 1991.

[35] M-T Leung, WE Engeler, and P Frank. Fingerprint image processing using neural networks. In *Computer and Communication Systems, 1990. IEEE TENCON'90., 1990 IEEE Region 10 Conference on*, pages 582–586. IEEE, 1990.

[36] WF Leung, SH Leung, WH Lau, and Andrew Luk. Fingerprint recognition using neural network. In *Neural Networks for Signal Processing [1991]., Proceedings of the 1991 IEEE Workshop*, pages 226–235. IEEE, 1991.

[37] Simon Liu and Mark Silverman. A practical guide to biometric security technology. *IT Professional*, 3(1):27–32, 2001.

[38] Xiping Luo, Jie Tian, and Yan Wu. A minutiae matching algorithm in fingerprint verification. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 4, pages 833–836. IEEE, 2000.

[39] Dario Maio and Davide Maltoni. Direct gray-scale minutiae detection in fingerprints. *IEEE transactions on pattern analysis and machine intelligence*, 19(1):27–40, 1997.

[40] Davide Maltoni, Dario Maio, Anil K Jain, and Salil Prabhakar. *Handbook of fingerprint recognition*. Springer Science & Business Media, 2009.

[41] Karthik Nandakumar. Fingerprint matching based on minutiae phase spectrum. In *Biometrics (ICB), 2012 5th IAPR International Conference on*, pages 216–221. IEEE, 2012.

[42] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[43] Nitin Gupta. Texture memory in cuda — what is texture memory in cuda programming, 2014. [Online; accessed September 20, 2016].

[44] Daniel Peralta, Isaac Triguero, R Sanchez-Reillo, Francisco Herrera, and José Manuel Benítez. Fast fingerprint identification for large databases. *Pattern Recognition*, 47(2):588–602, 2014.

[45] Daniel Peralta, Isaac Triguero, Raul Sanchez-Reillo, Francisco Herrera, and José Manuel Benítez. Fast fingerprint identification for large databases. *Pattern Recognition*, 47(2):588–602, 2014.

[46] Nikita D Prakhov, Alexander L Chernorudskiy, and Murat R Gainullin. Vsdocker: a tool for parallel high-throughput virtual screening using autodock on windows-based computer clusters. *Bioinformatics*, 26(10):1374–1375, 2010.

[47] Franco P Preparata and Michael Ian Shamos. Introduction. In *Computational Geometry*, pages 1–35. Springer, 1985.

[48] Zhi Qi, Wen Wen, Wei Meng, Ya Zhang, and Longxing Shi. An energy efficient opencl implementation of a fingerprint verification system on heterogeneous mobile device. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–8. IEEE, 2014.

[49] Uday Rajanna, Ali Erol, and George Bebis. A comparative study on feature extraction for fingerprint classification and performance improvements using rank-level fusion. *Pattern Analysis and Applications*, 13(3):263–272, 2010.

[50] Nalini K Ratha, Shaoyun Chen, and Anil K Jain. Adaptive flow orientation-based feature extraction in fingerprint images. *Pattern Recognition*, 28(11):1657–1672, 1995.

[51] Nalini K Ratha, Kalle Karu, Shaoyun Chen, and Anil K Jain. A real-time matching system for large fingerprint databases. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 18(8):799–813, 1996.

[52] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008.

[53] Manidipa Saha, Jyotismita Chaki, and Ranjan Parekh. Fingerprint recognition using texture features. *International Journal of Science and Research (IJSR)*, 2:12, 2013.

[54] Mark S Seidenberg and James L McClelland. A distributed, developmental model of word recognition and naming. *Psychological review*, 96(4):523, 1989.

[55] M Sepasian, W Balachandran, and C Mares. Image enhancement for fingerprint minutiae-based algorithms using clahe, standard deviation analysis and sliding neighborhood. In *Proceedings of the World congress on Engineering and Computer Science*, pages 22–24, 2008.

[56] Alexandros Stamatakis and Michael Ott. Exploiting fine-grained parallelism in the phylogenetic likelihood function with mpi, pthreads, and openmp: a performance study. In *Pattern Recognition in Bioinformatics*, pages 424–435. Springer, 2008.

[57] Harold S Stone. High-performance computer architecture. 1987.

[58] Marius Tico and Pauli Kuosmanen. Fingerprint matching using an orientation-based minutia descriptor. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(8):1009–1014, 2003.

[59] Abdul Wahab, SH Chin, and EC Tan. Novel approach to automated fingerprint recognition. *IEE Proceedings-Vision, Image and Signal Processing*, 145(3):160–166, 1998.

[60] CI Watson and CL Wilson. Nist special database 4. *Fingerprint Database, National Institute of Standards and Technology*, 17:77, 1992.

[61] Craig I Watson. Nist special database 14 mated fingerprint cards pairs 2 version 2. *CD-ROM and documentation*, 1993.

[62] Haiyun Xu, Raymond NJ Veldhuis, Asker M Bazen, Tom AM Kevenaar, Ton AHM Akkermans, and Berk Gokberk. Fingerprint verification using spectral minutiae representations. *Information Forensics and Security, IEEE Transactions on*, 4(3):397–409, 2009.

[63] JuCheng Yang, JinWook Shin, ByoungJun Min, JoonWhoan Lee, DongSun Park, and Sook Yoon. Fingerprint matching using global minutiae and invariant moments. In *Image and Signal Processing, 2008. CISP'08. Congress on*, volume 4, pages 599–602. IEEE, 2008.

[64] Jiawei Yuan and Shucheng Yu. Efficient privacy-preserving biometric identification in cloud computing. In *INFOCOM, 2013 Proceedings IEEE*, pages 2652–2660. IEEE, 2013.

[65] Weiwei Zhang and Yangsheng Wang. Core-based structure matching algorithm of fingerprint verification. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 1, pages 70–74. IEEE, 2002.