STUDIES TO IMPROVE THE PROCESS OF DECODING RATELESS ERASURE CODE WITH HIGHLY-PARALLEL GPU ARCHITECTURE

By

CHONG SIN RAN

A dissertation submitted to the Department of Electrical and Electronic Engineering,
Lee Kong Chian Faculty of Engineering and Science,
Universiti Tunku Abdul Rahman,
in partial fulfilment of the requirements for the degree of
Master of Engineering Science
April 2018

ABSTRACT

STUDIES TO IMPROVE THE PROCESS OF DECODING RATELESS ERASURE CODE WITH HIGHLY-PARALLEL GPU ARCHITECTURE

CHONG SIN RAN

Rateless Erasure Code (REC) is a type of forward error correcting code for erasure channel. Such code is often used to improve networking throughput performance. While the backbone of the REC is made up of linear equations, Gaussian elimination (GE) with the entry complexity of $O(k^3)$ is the general solver / decoder for REC. Thus, the decoding phase of REC is the performance bottleneck. Even with our current central processing unit (CPU) technology that can easily reach the processing speed of 4GHz, solving thousands of k linear equations using Gaussian elimination in a CPU is still a time-consuming process. In response, this thesis will show how the state-of-the-art graphic processing unit (GPU) can replace the predominant CPU in decoding REC. Furthermore, by utilising parallel processing technology embedded in the GPU, our study will show that the decoding of REC riding on the state-of-the-art of the GPU, are capable of performing significantly better than CPU during the REC's decoding under certain circumstances. Apart from the typical GE decoding, we also propose a new decoding algorithm, namely Gaussian elimination method with matrix multiplication (GEMM) that comes with two degrees of parallelisation in the GPU. In the first degree of parallelisation, the GEMM will show its ability of decoding REC of one file 2x faster than GE that is computed in CPU, while the second degree of parallelisation of GEMM will prove the idea of decoding thousands of distinct files at once can perform more than 10x faster compared to a CPU decoding thousands of distinct files.

ACKNOWLEDGEMENT

I would like to express my sincere appreciations to my supervisor Dr. Lai An Chow, co-supervisor Dr. Tay Yong Haur and mentor Dr. Chong Zan Kai for their precious guidance throughout my Master’s study. Thank you for their invaluable knowledge and willingness to steer me in the right direction whenever I needed it. Besides, I would also like to thank those who have been part of my master study and also my friends as well as colleagues.

Finally, special thanks to my family members for their unfailing support and continuous encouragement throughout my years of study.

DECLARATION

I hereby declare that the dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.

Name ______________________________

Date _______________________________

APPROVAL SHEET

This dissertation/thesis entitled “**STUDIES TO IMPROVE THE PROCESS OF DECODING RATELESS ERASURE CODE WITH HIGHLY-PARALLEL GPU ARCHITECTURE”** was prepared by CHONG SIN RAN and submitted as partial fulfilment of the requirements for the degree of Master of Engineering Science at Universiti Tunku Abdul Rahman.

Approved by:

____

(Dr. Lai An Chow) Date:…………………..
Supervisor
Department of Electrical and Electronic Engineering
Faculty of Engineering & Science
Universiti Tunku Abdul Rahman

____

(Dr. Tay Yong Haur) Date:…………………..
Co-supervisor
Department of Internet Engineering and Computer Science
Faculty of Engineering & Science
Universiti Tunku Abdul Rahman

.

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

BEC	Binary erasure channel
BP	Belief Propagation
BPGE	Belief Propagation Gaussian Elimination
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
GE	Gaussian elimination
GEMM	Gaussian elimination with matrix multiplication
GPU	Graphical Processing Unit
IDGE	Inactivation Decoding Gaussian Elimination
IoT	Internet of Things
LDPC	Low-Density-Parity-Check Code
MTU	Maximum Transmission Unit
PCD	Probability of complete decoding
TCP	Transmission control protocol
WSN	Wireless Sensor Network

k	Amount of message symbols
n	Numbers of received packets
l	Encoded symbol length
ϵ	Overhead constant

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The communication network is a fast-growing technology that bridges the information in physical and cyber domains. Generally, most of the internet-connected devices are using the wireless channel and they are susceptible to the environmental interference. Furthermore, the ever-growing number of internet-connected devices continuously generate traffic that forms exponential and can potentially increase the frequencies of packet losses; hence, significantly decreasing the Internet throughput.

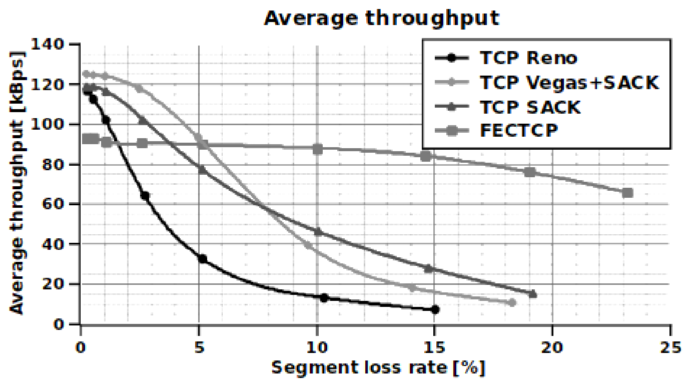


Figure 1.1: Average throughput performance of various TCP

Figure 1.1 demonstrate the effects of packet loss towards the throughput of various implementation of TCP. Other than FECTCP (Alqahtani, et al., 2016) which implements a TCP-like transport protocol by using Rateless Erasure Code, the other TCP variants show their high sensitivity towards packet loss. In other words, REC can play an important role in solving the packet loss problem in TCP/IP by encoding a message of k symbols into theoretically an infinite number of encoded symbols. In general, rateless erasure code (REC) is a type of error correcting code for the communication system that promises to achieve efficient erasure mitigation over lossy transmission channels and to improve bandwidth utilization of our communication networks. If the channel is lossy, the message is still recoverable given that sufficient encoded symbols (i.e. $n \geq k$) are received.

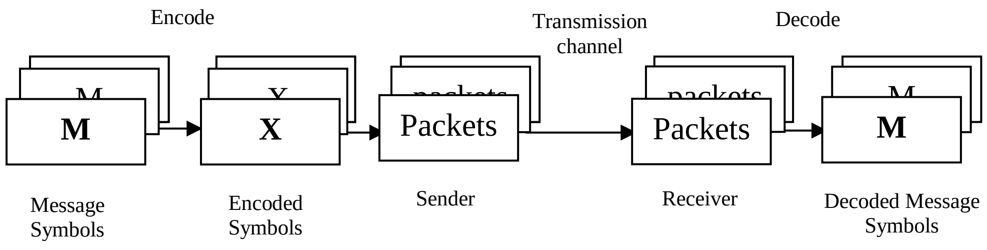


Figure 1.2: Transmission flow diagram

Figure 1.2 demonstrates briefly the transmission process of the communication network. Before a message is sent, it will first be subdivided into k amount of message symbols (**M**), then **M** will be encoded into several encoded symbols (**X**) for error handling mechanism according to the standards set by the particular protocol, and forms packets that are ready to be sent. A packet in a networking is often called a datagram, and it is a self-contained independent data carrier that carries the partial knowledge of the encoded information as well as the address of its sender and receiver etc. After all the relevant packets are received at the destination, the original message will be reconstructed according to that protocol's specific decoding process.

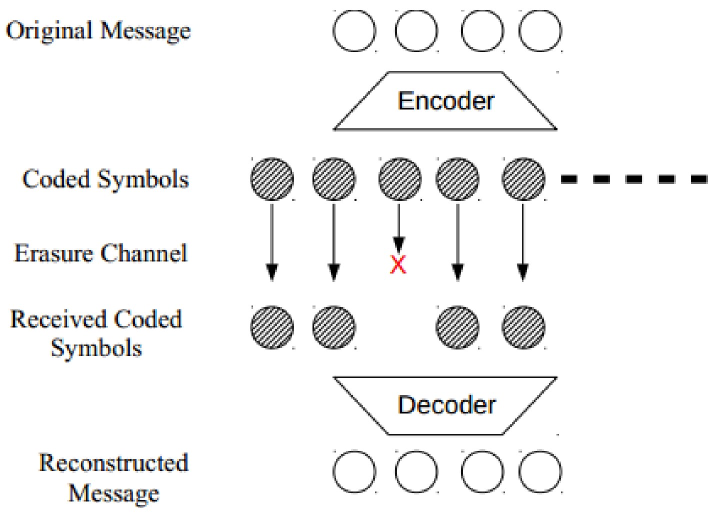


Figure 1.3: The encoding and decoding of rateless erasure code

Figure 1.3 demonstrates briefly the encoding and decoding process of REC. First, k amount of message symbols from the original message will arbitrarily generate a theoretically limitless number of encoded packets, and any subset of the n encoded packets that is slightly more than k, can be used to recover the original message at the receiver side. Such transmission allows the packets to be decode regardless of its sequences and subsequently minimise the needs for retransmission. However, one of the aspects that affect the applicability of REC is the high computational resources required by the decoding of REC.

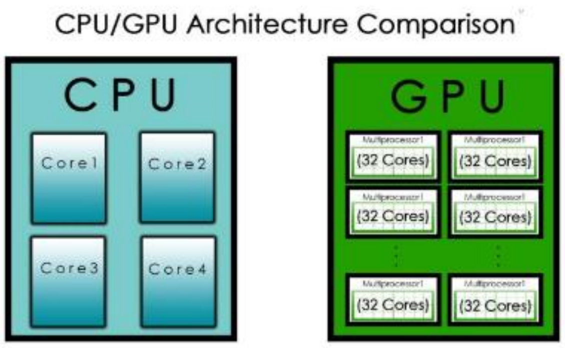


Figure 1.4: Comparison in Between CPU and GPU.

The gain from the REC is therefore paid for by the price of high computational resources in the decoding process (Chong, et al., 2016). To address this issue, we propose to speed up the decoding process using graphical processing unit (GPU) – a computer peripheral that is capable of processing massive amount of data in parallel. Generally, GPU consists of thousands of cores as compared to CPU but the former runs at a relatively lower clock speed (See Figure 1.4). Such computational architecture advancement challenges the proper design of parallel algorithms in utilising the full potential of GPU to reach the theoretical speed limit.

This chapter introduces the relationship between Gaussian Elimination and REC's impact to our networking system. It also explains the contribution of this project to the coding theory field particularly on enhancing Gaussian Elimination performance speed by using GPU.

In this thesis, we will briefly explain how to maximize the performance of REC by utilizing the state of art of GPU. GPU consists of thousands of computing cores that it run independently, each of GPU's core has lesser complexity than the core in CPU. Furthermore, GPU's operation requires certain computational overheads, hence we will study and analyze whether the GPU can help in improving the decoding performance of REC. Other than that, one more reason to study on GPU is due to its relatively low cost compared to mainstream CPUs. For example by using the information that we have as the reference, the price of **GPU-NVIDIA QUADRO K620 WORKSTATION GRAPHICS CARD** is retailed at RM900 (July, 2017) while

CPU-HP Xeon E3 Z240 Tower Workstation (HP-V1Z90PA) is retailing at a higher cost of RM4500 (July, 2017).

1.1 Background

Data transmission is a process of transmitting certain numbers of relevant packets from sender to receiver and this transmission is mediated by a communication protocol. TCP/IP is one of the most widely used communication protocol to ensure the reliability and the quality of packets during data transmission.

In general, before a message file is being transmitted, it will be encoded and subdivide into packets by following the TCP/IP standards. These packets will then travel individually to the receiver, and TCP/IP will "remounts" the packets in order to assemble the packets back into the original file, If for instance a packet is lost on the receiving side for reasons such as bit error, timeout, packets drop on network congestion and even wrong packets sequence, TCP/IP at the receiving side will feedback a signal to the sender, asking it to re-send the particular missing packets until all the packets have reached the destination.

However, due to the properties of TCP/IP that requires an acknowledgment for every received or lost packets, the transmission process would be very inefficient and can easily clutter the network. Furthermore, our communication system is being increasingly used for wireless communication, where a slight error could possibly lead to significant throughput degradation. For a long time, many researchers have claimed that TCP is inefficient in the high-speed internet. (Salyers, et al., 2008), (Kim & Lee, 2004).

1.1.1 Impact of Rateless Erasure Code to Network

In the past decades, REC has been proposed as the solution for such issue, with the properties mentioned previously, where "*A REC encoded file can be retrieved from any subsets of the encoded symbols disregarding of it sequences*".

The mechanism in TCP that requires acknowledgement for every lost and received packets to ensures data transmission reliability, is therefore less important in REC as data transmission by using REC doesn't require acknowledgement for every received or lost packets, and the only important thing in REC is that the receiver side received sufficient n $(n > k)$ amount of encoded symbol for complete decoding.

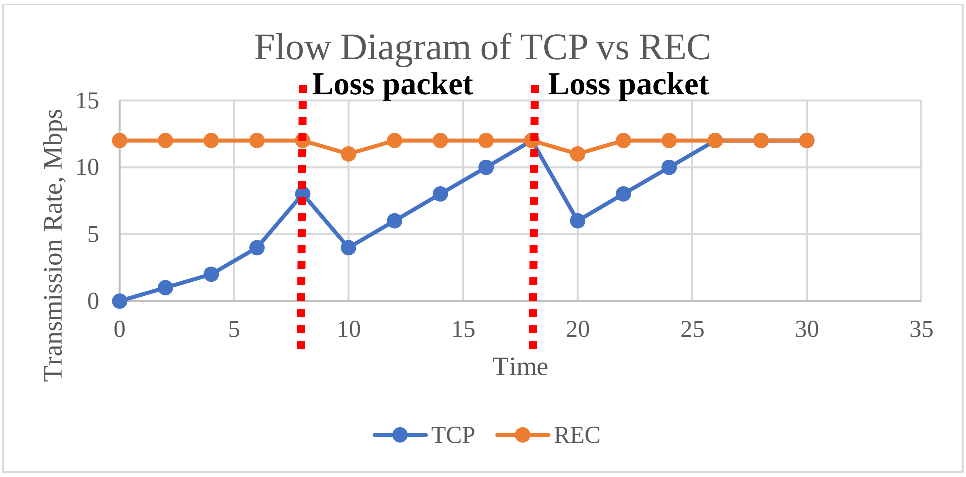


Figure 1.5: TCP vs REC Flow Diagram

Figure 1.5 demonstrates the transmission rate of TCP and REC during packets loss. For TCP, whenever a packet is sent successfully (sender receives an acknowledgement of the received packet), the transmission rate of TCP will increase exponentially until a packet loss is detected (sender received a failed to transmit signal or fail to receive acknowledgement signal). At this point, the transmission rate will be halved, and the process will continue until all the packets are delivered successfully (Mathis, et al., 1997). While for REC, the transmission rate is at its near maximum rate from the beginning until the end of the transmission regardless of packet loss (Yuan, et al., 2010). Since REC can transmit the packets at near the maximum transmission rate at all time, more bandwidth is utilised in transmitting the packets instead of wasting them on the acknowledgment mechanism of lost and received packets.

1.1.2 Rateless Erasure Code Riding on GPU

Ever since REC appears a few decades ago, many variants of REC have been proposed, i.e. Random code (Chong, et al., 2015), Lt code (Luby, 2002), and Raptor code (Shokrollahi, 2006) etc, in order to solve the REC's common issues such as the decoding speed or overhead problems.

All RECs are usually viewed as the linear codes over Galois fields, and are built on top of the linear algebra system that generally makes them optimally decodable using an algorithm namely Gaussian elimination (GE) (Anghel, et al., 2011). In contrast, such method of decoding linear system would be very computationally intensive. To address this issue, the most promising solution seems to be the implementation of parallel processing using GPU.

The exceptional GPU computing power is very attractive to general-purpose system development. However, the critical challenge during coding for GPU is the smaller degree of parallelization in the REC's decoding process. GE requires the decoding of each step to start only after the decoding of the previous step is finished. This implies that the parallel decoding process will be limited by GE's independencies.

Since GPU requires a certain large amount of threads to reach peak performance, a lesser parallelization degree will limit the performance gain by parallel processing. In addition, this research will be testing on the parallelization granularity of GPU-based decoding schemes and their performance for the different granularity setup. The details of the parallelization schemes will be presents in this thesis later.

Recently, Applications that harness the massive parallelism of GPU to speed up computational task have become increasingly common. In this research, we propose a new parallel decoding algorithm namely the Gaussian Elimination with Matrix Multiplication (GEMM) as matrix multiplication is known to be highly parallelizable. The goal of this study is to research on how to effectively offload parallel computations to the graphics card, and analyses the impact of GPU toward REC.

1.2 Research Problem

In previous research papers (Anghel, et al., 2010), (Chong, et al., 2015), (Chong, et al., 2016), it was shown that REC can lead to more scalable and robust protocols with better utilization of the available bandwidth at poor network condition.

However, to date, we have not observed any commercial application or protocol taking advantage of the power of REC. We believe that the main cause of this observation is due to the high complexity of the decoding algorithm- Gaussian elimination, employed in REC. Currently, Gaussian elimination is always the main component in REC's decoding schemes, as such it is crucial to speed up the performance of Gaussian Elimination by utilising modern state-of-the-art of computer accelerator – GPU. In this dissertation, the research problem is defined as the following:

"How to design a scalable and robust Rateless Erasure Code decoding algorithm that can utilise the resources in GPU"

1.3 Objective

Associate with the research problem, the objectives in this research will be:

- To study the state-of-art of REC that uses GE as the core in the decoding process.
- To propose a new parallel algorithm to speed up GE components with REC constraints
- To evaluate the scalability of the proposed algorithm.
- To analyse the proposed algorithm experimentally.

1.4 Outline

In this chapter, the issue on the traditional communication network are discussed, and the deployment of REC to improve the performance of the current communication network. Nonetheless, REC is not widely deployed due to the high complexity of encoding and decoding processes.

In Chapter 2, a few REC will be review to highlight the research problem. Since most of the REC are linear codes, they are decodable by many different mathematical approaches. Their advantages and disadvantages will be discussed from different aspects.

Chapter 3 introduces the core decoding method for REC, i.e., Gaussian Elimination, whose time complexity is $O(k^3)$. Basically, the chapter will cover the state of art of Gaussian elimination and propose a new parallel algorithm called GEMM to improve the performance of REC with any input k, using re-dimensioning techniques. Such parallel algorithms will be implemented and compared on GPU platforms. Furthermore, due to the inefficiency of parallelisation in Gaussian Elimination, this research proposes GEMM with two degrees of parallelisation. In short, the first-degree parallelisation will be performed on single file, and we will see significant improvement on the GPU decoding performance compared to CPU while the second-degree parallelisation of GEMM will be performed on multiple files, e.g., we propose to decode 1000 files in parallel while each file will be parallel processed at the same time as well.

Chapter 4 will demonstrate the result and comparison of convention Gaussian elimination (base case) with our proposed double degree parallelised GEMM, where GPU resources can be potentially exploited. In this chapter, analysis according to the experimental result will be done to prove the workability of our proposed algorithm, we will also discuss the performance of the proposed algorithm from a different aspect with different parameters to show its scalability. Finally, we will draw a conclusion and discuss the potential future work in Chapter 5.

CHAPTER 2

LITERATURE REVIEW

Rateless erasure codes (also called the Fountain codes) are a family of error correcting codes where the rate of transmitting coded packet can be adjusted on the fly. Such an approach is termed Digital Fountain (DF), as the transmitter is used as a fountain that emits coded packets that are continuously sent until the receiver has received the number of packets required for 100% probability of complete decoding (PCD) (Lu, et al., 2012). However, the deployment of rateless erasure code is limited, primarily due to the added computational complexity associated with linear coding-based encoding and decoding.

2.1 Rateless Erasure Code Variant

Variant of REC comes in as improvement to fit into different situation, e.g., the REC that has lower overheads will be used in lossy situation (deep space communication, long-distance communication, etc.) where a packet suffers high loss rate during transmission (Ren, et al., 2014); REC with high encoding and decoding speed performance are more suitable for daily communications (wireless communication etc.) with lower packet loss rate (Assefa, et al., 2016). In general, all the REC's are encoded in a way where:

$$G_{\infty\times k} \times M_{k\times l} = X_{\infty\times l} \qquad (2.1)$$

$G =$ Generated information for encoding

$M =$ Message symbols

$X =$ Encoded symbols

$k =$ Numbers of message symbols

$l =$ Length of one message symbols

The method of generating G determines the properties of the particular REC; different REC will possess different G for the encoding process, as long as the G can still be generated according to the particular REC's standard, the message can be theoretically encoded into an infinite number of encoded symbols.

Then the encoded symbols will be augmented together with its generating information to form packets in the form of $G_y|X_y$ $_{,y=0,1,2,3\ldots\infty}$, These packets will be transmitted to the receiver side for decoding since the packet $G|X$ are generated with linear algebra, they are basically decodable using Gaussian elimination (Bioglio, et al., 2009), as long as sufficient n numbers of packets are received. The equations will be explained in detail in Chapter 3. In this section, we will briefly introduce and review several existing REC.

2.1.1 LT Code

LT codes (Luby, 2002), are the first practical realization of the digital fountain approach, also called universal erasure codes. The main advantages of LT codes are:

1. The number of packets that can be generated from the message file is potentially infinite, or researchers call it on-the-fly (the encoded packets will be generated whenever it is needed).
2. Low complexity for both encoding and decoding processes (fast).

According to the linear equation for REC where $G \times M = X$, LT code generates G by using fine tune random degree distribution (Cheong & Fan, 2016), (Luby, 2002), i.e., the ideal Soliton distribution and the robust Soliton distribution for the optimal encoding and decoding performance (Zhu, et al., 2008).

2.1.1.1 Encoding

For the encoding of LT code (Luby, 2002):

1. Divide the message into equal length l bits, resulting in k numbers of messages symbols as shown; one row in the matrix will represent one message symbol.

$$M = 01000100111000100100 \rightarrow \overbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}_{k\times l}}^{l} \left.\vphantom{\begin{matrix}0\\0\\0\\0\end{matrix}}\right\} k$$

2. Randomly choose the degree (d) from fined tuned degree distribution for generating the G, e.g., let $d = 3$ (A row in G will be randomly allocated with a maximum of 3 ‘1’s)

$$G = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 1 \end{pmatrix}_{\infty \times k}$$

3. Matrix multiply G and M to form X, and the corresponding G will be augmented with X to form $G|X$ packet for transmission.

$$G \times M = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

$$G|X = \left(\begin{array}{cccc|ccccc} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ \hline 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right) \begin{array}{l} \\ \\ \text{Certain packet} \\ \text{Certain packet} \\ \\ \\ \\ \\ \end{array}$$

By encoding the file in LT code method, the encoded packets are either completely certain (only a ‘1’ in G) or uncertain (more than a ‘1’ in G). By using the method called belief propagation(BP) which can be only used in LT code decoding, the certain encoded packet will be used to eliminate all the uncertain packets back to certain packets during the decoding.

2.1.1.2 Decoding

The fastest way to decode LT code packets is to use the propagation method stated previously namely belief propagation (BP) (Chen, et al., 2013). As already mentioned, the packets consist of certain and uncertain packets. With this condition, the BP decoding will be demonstrated as shown:

1. Find any one row that contains certain packet. (only a ‘1’ in G)

$$G|X = \left(\begin{array}{cccc|ccccc} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right) \begin{array}{l} \\ \text{Certain packet} \\ \\ \\ \\ \\ \\ \\ \end{array}$$

2. Propagate one of the certain packets in step one to all the related uncertain packets (uncertain packets that contain ‘1’ in the same column with the certain packet), by adding them with the value of the “certain packet”.

$$G|X = \left(\begin{array}{cccc|ccccc} 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right)$$

3. Iterate the first two step until all uncertain packets are eliminated (left side of matrix becomes identity matrix), and the X on the right will convert back into M.

Identity matrix

$$G|X = \left(\begin{array}{cccc|ccccc} 1&0&0&0&0&1&0&0&0\\ 0&1&0&0&1&0&0&1&1\\ 0&0&1&0&1&0&0&0&1\\ 0&0&0&1&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0\\ \vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots\\ 0&0&0&0&0&0&0&0&0 \end{array}\right)$$

Decoded message

According to LT code (Luby, 2002), the LT codes overheads is calculated based on the soliton distribution; the overheads required in LT code will decrease as the k increase.

2.1.2 Raptor Code

The Raptor code is an extension of LT code (Shokrollahi, 2006), whereby a pre-coding stage (usually low-density-parity code (LDPC) code, a simple error correcting code with parity check) is used to extend the message symbols.

2.1.2.1 Encoding

The encoding of Raptor code is as shown:

1. Encode the message symbols by using LDPC code to get the optimal numbers of encoded symbols for degree distribution in the next step.
2. Then these pre-coded symbols will be encoded with LT encoding method in the previous section.

2.1.2.2 Decoding

After that, the received raptor code's packets will be decoded using BP which applies the same method as that for the decoding of LT code.

In general, the pre-coding stage that extends the numbers of message symbols are used to reach the optimal message number that is suitable for degree distribution in LT code, because if the numbers of message symbols are fewer than a certain value, it will not have an optimal degree distribution, and also have a higher chance to fail during decoding (Li, et al., 2014).

2.1.3 Random Code

Random code is another variant of REC that has lower overheads, compared to LT and Raptor codes that have optimally $\varepsilon = 0.03k$ overheads only at large k, Random code only needs $\varepsilon = 10$ overheads for 99.99% PCD. (Chong, et al., 2015), (Chong, et al., 2016)

2.1.3.1 Encoding

Just like standard REC encoding, Random code follows the general encoding where:

$$G \times M = X.$$

The encoding of Random code is demonstrated below:

1. Divide the message into equal length l bits which results in k numbers of messages symbols as shown. One row in the matrix will represent one message symbol.

$$M = 01000100111000100100 \rightarrow \overbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}}^{l}_{k \times l} \; k$$

2. Randomly generate the G. Unlike LT code, the Random code can generate its G without following the degree distribution. The 0 to 1 ratio in the whole generated matrix should be 1:1.

$$G = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 1 \end{pmatrix}_{\infty \times k}$$

3. Matrix multiply G and M to form X, and the corresponding G will be augmented with X to form $G|X$ packet for transmission.

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

$$G|X = \left(\begin{array}{cccc|ccccc} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \end{array}\right)$$

By encoding the message using the random code way, every encoded packet is uncertain, where all the encoded packets are made up of an average $k/2$ number of messages symbols by probability.

2.1.3.2 Decoding

In this case, when GE is used for the decoding, the n should be $k + 10$ for 99.99% PCD according to Kolchin's theorem (Chong, et al., 2015), and the decoding process is shown below:

1. When n numbers of packets in the form of $G|X$ are received, GE decoding commence.
2. GE in general convert $G|X \rightarrow I|M$ during decoding. The detail process of GE will be discussed in the next chapter.

Since the decoding algorithm for Random code is GE that has an entry complexity of $O(k^3)$, the decoding process is generally slower than LT and Raptor Code that uses fast decoding algorithm namely belief propagation (BP).

2.2 REC Applications

REC is a technique of applying linear algebra to all sorts of digital communication; it includes the data transmission in a lossy environment such as deep space communication (Ren, et al., 2014) and wireless communications (Kim & Lee, 2004) that we had mentioned earlier. Other than that, the REC often found in the real-life applications are the wireless sensor network (Hagedorn, et al., 2008) and distributed data storage (Anghel, et al., 2011).

2.2.1 REC in Wireless Sensor Network

Over the years, the application of REC in wireless sensor network (WSN) has always been a popular topic (Hagedorn, et al., 2008), whether in reducing power consumption where managing power consumption of thousand sensors can be very tedious and impractical and also environment of sensor where transmission is susceptible to interference. Numerical results from the paper (Jamshid, et al., 2011), show that the implementation of Raptor coded in their WSN network model is more energy efficient and robust than those normal un-coded WSN.

2.2.2 REC in Distributed Data Storage/ Cloud storage

Another interesting application of REC will be the distributed storage system or cloud storage (Anghel, et al., 2011). Recently, the ever-increasing amount of data generated from our daily internet usage are the main reason why servers (often simple commodity devices/machines) suffers from frequent hardware failures (Kevin, 2015), and the most typical method used to solve such issue is by replication, where a set of data will be duplicated and stored into 3 distinct storage systems, even when one side of data is corrupted or potentially gone missing, the same data from another storage is able to cater the corrupted or missing part (Julia & Thinn, 2011).

However, due to the poor 33% inefficiency of such replication method, researchers at Facebook, Microsoft, and Qualcomm etc. implemented the REC for the use in their distributed storage systems (Kevin, 2015). Such approach potentially reduces 60% of the storage space overhead, with the properties of REC that able to recover to some extent data that was corrupted or missing.

2.3 Bottlenecks in Rateless Erasure Code

In general, the development of REC usually faces common issues such as:

- Overhead
- Performance Speed

And these issues are the reasons why REC consist of many others variations. e.g., Raptor code, Lt code, and Random code.

2.3.1 Overhead

The overhead of rateless erasure codes such as LT code and Raptor code is only asymptotically optimal (Yeqing, et al., 2013), e.g., in real-time applications. Where the input k is small, the overhead could become larger than 10%. On the other hand, some rateless codes such as Random code can maintain its small overhead even for small values of k, at the cost of increasing its computational decoding complexity. Trade-off between overhead and complexity is the key point in the consideration of design phase of a rateless erasure scheme (Li, et al., 2014).

As mentioned previously, when k message symbols are encoded with REC, the symbols will be granted the ability to be sent out in random order and also have certain immunity towards packet loss. However, in order to successfully decode the received

REC packets, n (slightly more than k) numbers of packets are required for a successful complete decoding (a complete decoding indicates the matrices form of the packets are able to reach full rank, else more overheads packets have to be received) which is indicated by:

$$n = k + \varepsilon \tag{2.2}$$

where

n = number of received packets

k = amount of messages symbols

ε = overheads

In general, n is usually slightly larger than the k value, and different REC will have different n for a high probability of complete decoding (PCD). Packets received in matrix form will reach full rank at 99.99% when n numbers of packets are received.

In the research on Random code's PCD (Chong, et al., 2015), it is shown that:

Table 2.1: The PCD for Random code of k= 10.

n	PCD	n	PCD
k	28.66%	$k+7$	99.22%
$k+1$	57.76%	$k+8$	99.45%
$k+2$	78.01%	$k+9$	99.81%
$k+3$	89.02%	$k+10$	99.9996%
$k+4$	93.88%	$k+11$	99.99999%

$k + 5$	96.91%	$k + 12$	99.999999%
$k + 6$	98.45%	$k + 13$	99.99999999%

Results in Table 2.1 show the PCD of a variant of REC namely Random code, where such code has the capability to reach a 99.99% PCD whenever $n = k + 10$ number of packets are received.

Over the years several methods were proposed to reduce the overhead in REC, whereby sender can optimally adjust its Galois field (Hu, et al., 2012), e.g., $\mathrm{GF}(2)$ Raptor code might consist of $\varepsilon = 0.03k$ at $k > 2000$ for 99.99% PCD, but implementation of $\mathrm{GF}(256)$ Raptor code (RaptorQ) will only require $\varepsilon = 1$ for 99.99% PCD. However, increasing the Galois field "degree" also signified the increases in decoding complexity because, from the perspectives of computer architecture (Lidl & Niederreiter, 1997), Galois field indicates a new set of self-defined operations, For example,

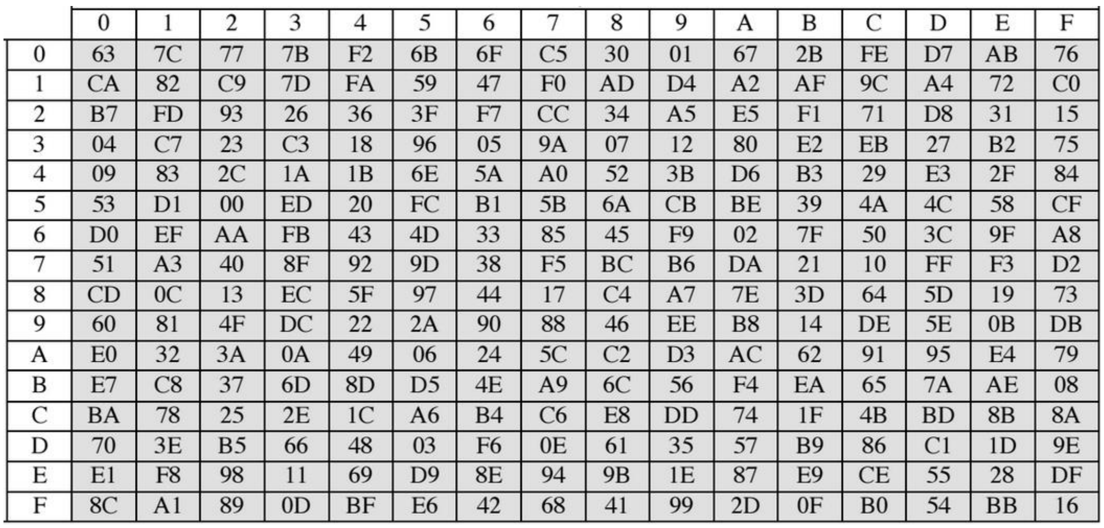
Table 2.2: $\mathbf{GF(2)}$ addition table

+	0	1
0	0	1
1	1	0

$GF2\ addition\ table$

in $\mathrm{GF}(2)$, we are able to utilise computation of XOR operation to replace the $\mathrm{GF}(2)$ addition because they are the same. e.g. $1 + 1$ in $\mathrm{GF}(2)$ is 0 while 1 XOR with 1 also zero.

Table 2.3: $\mathrm{GF}(256)$ addition table



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

But for $\mathrm{GF}(256)$, there are no computation operation to replace $\mathrm{GF}(256)$ addition

addition as shown in Table 2.3. In this case a self-defined library would be needed. In another word, the time used in reading the value from the self-defined library in this $\mathrm{GF}(256)$ addition will be far slower than $\mathrm{GF}(2)$ addition that can be replaced by XOR operation, (Mladenov, et al., 2012). Hence in this thesis, the REC will be constructed under $\mathrm{GF}(2)$ for optimized performance purpose.

Table 2.3: $\mathrm{GF}(256)$ addition table

2.3.2 Performance Speed

Apart from overheads, the performance speed is also one of the issues arising from the implementation of REC (Bioglio, et al., 2009). Note that the performing speed here is not referring to the throughput performance but the speed of encoding and decoding, specifically the delay of the transmission due to the decoding of the packets. It is a common issue for implementation of rateless erasure code. While this is compared with the classical communication network systems, a packet is usually directly "decodable", which means that packets sent will not require any further work or require the least effort to be read or decoded. The packet only need to be received in correct order and not lost for reliability, e.g. the TCP/IP mechanism that prioritize acknowledgement for all lost and received packets. (Salyers, et al., 2008)

Conversely, when a message is implemented with REC, the need of such acknowledgment for all lost and received packets is minimized, which means that more bandwidth can be utilized for transmission instead of being wasted on acknowledging the lost and received packets (Chong, et al., 2016). In exchange, extra work has to be done on the encoding and decoding phase. Especially at the decoding phase, to decode and reconstruct the REC's received packets back into original message, it requires a high complexity decoding algorithm namely Gaussian elimination (GE) (Chong, et al., 2016).

For being the most popular variants of REC, LT code (Luby, 2002) and Raptor code (Shokrollahi, 2006) advocate the idea of linear decoding speed; Instead of decoding using GE, LT and Raptor code can decode with their own decoding algorithm namely Belief Propagation (BP) with a low entry complexity of near $O(k)$ while the complexity will rise to $O(k^3)$ if they are decoded by using GE (Chen, et al., 2013). However, BP had a relatively lower PCD compared to GE which will be explained in the later section.

2.4 Gaussian Elimination

Gaussian elimination (GE) is a method widely used in many applications, it is implemented in application such as the wireless sensor network (Rossi, et al., 2010), linear coding (Li, et al., 2014), network coding (Hagedorn, et al., 2008) , and even encryption as well as scheduling algorithm.

In this research, where GE is used for the decoding algorithm for REC, study has shown that the complexity of GE is $O(k^3)$ (Bioglio, et al., 2009), which means that the time to decode a k size matrix would have increased exponentially as k increases as shown in the graph.

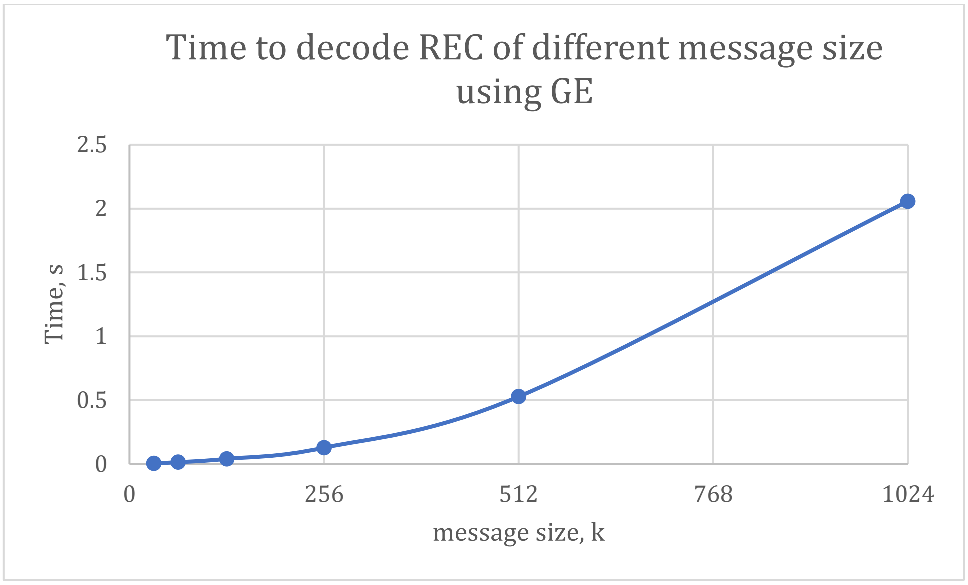


Figure 2.1: Gaussian elimination decoding time, s vs. message size, k

2.5 Belief Propagation

Belief Propagation (BP) is proposed while the decoding of REC is slow due to GE, researchers had come out with other variation of REC such as the LT code and Raptor code that allows for faster-decoding speed. This two codes have their own unique decoding method namely BP that has a decoding complexity of $O(k \ln k)$ and approximately $O(k)$ respectively, this also indicates that they can perform faster than GE in term of lesser complexity. (Shokrollahi & Luby, 2011) The steps of BP are explained in details in section 2.1.1.2 and 2.1.2.2.

Although decoding using BP is fast, it is found that decoding using GE has an advantage in terms of successful decoding rate. When the received packets are decodable using GE, it is not necessarily decodable using the BP (Hu, et al., 2012), (Bioglio, et al., 2009).

2.6 Inactivation decoding Gaussian Elimination

Inactive decoding Gaussian elimination (IDGE) also known as the Belief propagation Gaussian elimination (BPGE), is one of the improvised decoding methods used particularly for the Raptor and LT code (Hu, et al., 2012) & (Mladenov, et al., 2012), this method imposes a higher decoding complexity compared to BP algorithm while less complex than Gaussian elimination. At the same time, IDGE is capable of having a high PCD like Gaussian elimination when it is compared to BP decoding.

This method combines the decoding method of belief propagation (BP) in LT code and the decoding with Gaussian elimination and is denoted by BP-GE or IDGE (M & S, 2006). In this case, several steps are needed to be performed when the packets are received.

First, all the received packets will be processed with Belief propagation that converts most of the uncertain packets by using the certain packet. In this case, there will be chances that some uncertain packets are leftover, this forms a new entry of packets that are not decodable using BP since all the certain packets are used. Then these remaining new entries of uncertain packet will be processed by Gaussian elimination for a complete decoding.

The advantage of this method will be, since most of the packets are already decoded by BP, the leftover packets that are not decodable using BP even if it is linearly independent will be relatively smaller in k size when it is processed by Gaussian elimination, which will dynamically reduce the decoding time as GE is sensitive to the

k size with the entry complexity of $O(k^3)$. However, such a method is uniquely applicable only for Raptor and LT code, and there is no other REC variant can use this method. Other than that, by profiling IDGE, it is seen that more than 90% of the decoding time is still consumed at the GE part (Yeqing, et al., 2013), hence enhancing the performance of GE is an essential thing to do to improve the decoding speed of REC.

2.7 Parallel Processing for Rateless Erasure Code

Due to the fact that GE is the optimal solution for linear independent matrices compare to other solvers (Bioglio, et al., 2009), (Hu, et al., 2012), many researchers have come out with several ways to speed up GE, and the most promising one appears to be the parallel processing of GE, (Hu, et al., 2013), (Chong, et al., 2016)

On the other hand, GPU is by far the most popular device for parallel processing. In the study of Raptor GF(2) (Hu, et al., 2012), it is shown that the implementation of GE into GPU outperforms the other decoding method (BP and IDGE) in terms of parallelization, which concludes that GE is by far the most suitable decoding method for parallelization (Hu, et al., 2013). Most importantly, the workspace of GPU is independent of CPU in executing a task, this means that when GPU is performing a task, CPU is able to handle another task at the same time.

Recently, more and more applications traditionally run on the CPU are being re-implemented to run on the GPU. A decade ago, when Nvidia offered programming interfaces such as CUDA (CUDA, 2017) for making parallel processing accessible to all programmers, it has removed the limitation of GPU that was initially designed for computer graphics. In this thesis, CUDA will be the main platform for the parallelisation process.

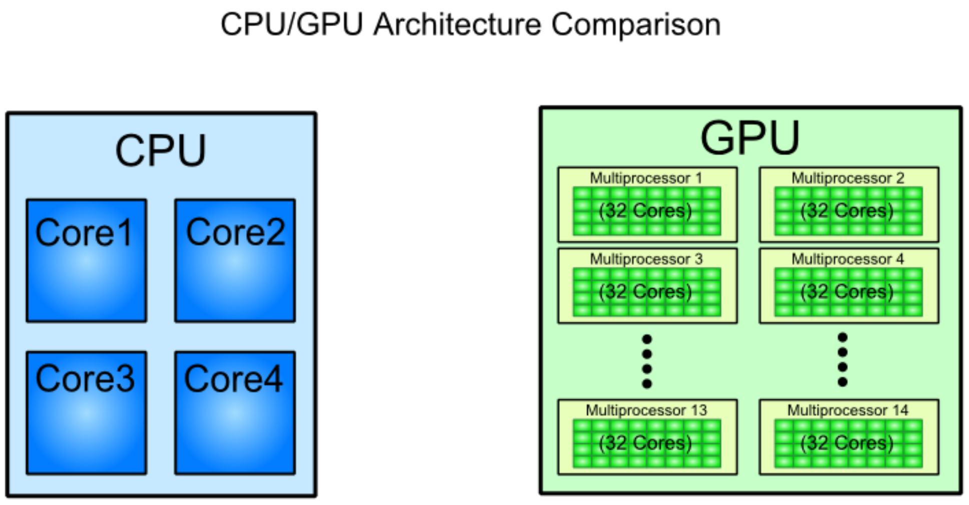


Figure 2.2: CPU vs. GPU architecture comparison

GPUs are a multithreaded stream processor that usually contain thousands of cores more than a CPU. In general, the parallelisation in CUDA is composed of two parts:

- Host (CPU) code that makes kernel calls,
- Device (GPU) code that actually implements the kernel.

The host is generally made up of serial C++ program, and device is where we perform parallel processing to harness the resources of the GPU. The fundamental of GPU is the streaming multiprocessors (SMs); Each SM will consist of a few blocks to hundreds of blocks depending on the architectures of the GPU and each block will contain 32 threads that can simultaneously execute the same instruction. The kernel that mentioned previously is executed by these threads on the GPU.

On the other hand, GPU still need to undergo a scheduling process before parallel processing, and the main scheduling unit in CUDA is a warp, which is made up of a group of 32 threads from the same block, and execution of an arithmetic instruction for the whole warp takes 4 clock cycles. The number of these warps is important in tolerating global memory access latency that will discuss later.

2.8 Relationship of k and l

In the research of REC, k that genuinely indicates the number of message symbols and l that indicates the length of the message symbols, are the key elements to develop a REC.

The k is generally calculated by dividing the stream of binary message with a self-defined l.

$$\underset{20\mathbf{bits}}{M = 01000100111000100100} \rightarrow \overbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}_{k\times l}}^{l=5} \left.\vphantom{\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}}\right\} k=4$$

In our case of study, maximum transmission unit (MTU) will be the standard that used to determine the l. Since REC are implemented to utilize the bandwidth of data transmission, throughput degradation will happen when a non-efficient l is used.

In the paper of MTU (Molnar, et al., 2014) & (Guo, et al., 2016), it is stated that the most efficient l for our current transmission will range from 400-500 bytes, and also 1000-1500 bytes maximum. Furthermore, based on future MTU (Shaneel & Paula-Rayond, 2013), when the up/downlink of transmission that reaches the speed of >1Gbps are generalised, the new MTU will increases to $l = 9$kb instead. This means that, during the transmission, more information can be transmitted efficiently using this length of l. Hence in our study will be using$1kb \leq l \leq 9kb$.

The parameter value of k should be $k \leq 512$, due to the fact that, according to the study of networking traffic (Brownlee & Claffy, 2002), the transmissions that are lesser than few hundred kilobytes appear to be the main contributor to 80% of networking traffics. Furthermore, in Chapter 3.3, an idea of decoding REC's large file ($k > 512$) will be showed in a nearly linear speed by mathematics and experimental evidence.

CHAPTER 3

GAUSSIAN ELIMINATION IN RATELESS ERASURE CODE

In this section, details of GE will be illustrated, followed by a few new proposals to improve the performance of REC. Note that all the mathematical operation in this thesis will be in GF(2). e.g. $0 + 0 = 0, 0 + 1 = 1, 1 + 1 = 0$.

3.1 Encoding

The general encoding method of most of the REC is built on top of the linear algebra system in the form of eq. 2.1 where:

$$G_{\infty \times k} \times M_{k \times l} = X_{\infty \times k}$$

As for random code, the message will generally divide equally into k amounts of symbols that contain l bits each. In this case, the value of l and k are essential for the development of better performance REC. In the encoding process, k number of equally divided message symbols M will be encoded by matrix multiplying M with the randomly generated G.

3.2 Decoding

In Chapter 1.1, we have mentioned that encoded symbols will be generated and sent in the form of packets of $G|X$, but for the decoding part, not all symbols are needed before the REC can be decoded. In fact, only n (slightly more than k) numbers of packets are required for a 99.99% of PCD.

3.2.1 Probability of Complete Decoding (PCD)

The overhead in the random code, ε, is equal to 10 and the total received packets, n, is denoted by eq. 2.2 where:

$$n = k + \varepsilon$$

$$n = k + 10\ (for\ random\ code)$$

Every time when n number of packets are received, the PCD of random code decoding is guaranteed to be 99.99% according to Kolchin's theorem (Chong, et al., 2015), which means that matrices formed from the n numbers of received packets will be a linearly independent with probability of 99.99%.

Example 3.2.1.a:

At $n = k + 10$, $k = 3$, $l = 1$

let $M_{3\times1} = \begin{pmatrix} M_{1,1} \\ M_{2,1} \\ M_{3,1} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $G_{13\times3} = \begin{pmatrix} G_{1,1} & G_{1,2} & G_{1,3} \\ G_{2,1} & G_{2,2} & G_{2,3} \\ G_{3,1} & G_{3,3} & G_{3,4} \\ G_{4,1} & G_{4,2} & G_{4,3} \\ G_{5,1} & G_{5,2} & G_{5,3} \\ G_{6,1} & G_{6,2} & G_{6,3} \\ G_{7,1} & G_{7,2} & G_{7,3} \\ G_{8,1} & G_{8,2} & G_{8,3} \\ G_{9,1} & G_{9,2} & G_{9,3} \\ G_{10,1} & G_{10,2} & G_{10,3} \\ G_{11,1} & G_{11,2} & G_{11,3} \\ G_{12,1} & G_{12,2} & G_{12,3} \\ G_{13,1} & G_{13,2} & G_{13,3} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

Encoding:

$$G \times M = \begin{pmatrix} X_{1,1} \\ X_{2,1} \\ X_{3,1} \\ X_{4,1} \\ X_{5,1} \\ X_{6,1} \\ X_{7,1} \\ X_{8,1} \\ X_{9,1} \\ X_{10,1} \\ X_{11,1} \\ X_{12,1} \\ X_{13,1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

When it is send as a packet it will forms a matrix of:

$$G|X = \left(\begin{array}{ccc|c} G_{1,1} & G_{1,2} & G_{1,3} & X_{1,1} \\ G_{2,1} & G_{2,2} & G_{2,3} & X_{2,1} \\ G_{3,1} & G_{3,3} & G_{3,4} & X_{3,1} \\ G_{4,1} & G_{4,2} & G_{4,3} & X_{4,1} \\ G_{5,1} & G_{5,2} & G_{5,3} & X_{5,1} \\ G_{6,1} & G_{6,2} & G_{6,3} & X_{6,1} \\ G_{7,1} & G_{7,2} & G_{7,3} & X_{7,1} \\ G_{8,1} & G_{8,2} & G_{8,3} & X_{8,1} \\ G_{9,1} & G_{9,2} & G_{9,3} & X_{9,1} \\ G_{10,1} & G_{10,2} & G_{10,3} & X_{10,1} \\ G_{11,1} & G_{11,2} & G_{11,3} & X_{11,1} \\ G_{12,1} & G_{12,2} & G_{12,3} & X_{12,1} \\ G_{13,1} & G_{13,2} & G_{13,3} & X_{13,1} \end{array}\right) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & \mathbf{1} \\ 1 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 1 & \mathbf{1} \\ 1 & 0 & 1 & \mathbf{0} \\ 0 & 1 & 1 & \mathbf{0} \\ 1 & 1 & 1 & \mathbf{1} \\ 1 & 0 & 0 & \mathbf{1} \\ 1 & 0 & 0 & \mathbf{1} \\ 0 & 1 & 0 & \mathbf{1} \\ 0 & 1 & 1 & \mathbf{0} \\ 1 & 0 & 0 & \mathbf{1} \\ 1 & 0 & 1 & \mathbf{0} \\ 0 & 0 & 1 & \mathbf{1} \end{array}\right)$$

Or in linear algebra equivalent, one packet is equivalent to one equation as shown:

$$1M_{1,1} + 0M_{2,1} + 0M_{3,1} = X_1 = 1$$

$$1M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_2 = 0$$

$$0M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_3 = 1$$

$$1M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_4 = 0$$

$$0M_{1,1} + 1M_{2,1} + 1M_{3,1} = X_5 = 0$$

$$1M_{1,1} + 1M_{2,1} + 1M_{3,1} = X_6 = 1$$

$$1M_{1,1} + 0M_{2,1} + 0M_{3,1} = X_7 = 1$$

$$1M_{1,1} + 0M_{2,1} + 0M_{3,1} = X_8 = 1$$

$$0M_{1,1} + 1M_{2,1} + 0M_{3,1} = X_9 = 1$$

$$0M_{1,1} + 1M_{2,1} + 1M_{3,1} = X_{10} = 0$$

$$1M_{1,1} + 0M_{2,1} + 0M_{3,1} = X_{11} = 1$$

$$1M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_{12} = 0$$

$$0M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_{13} = 1$$

In example 3.2.1.a, by solving all these packets using the simple substitution method, $M_{1,1} = 1$,$M_{2,1} = 1$, $M_{3,1} = 1$ and the original message is retrieved.

In the next example, we will prove the importance of overheads, for instance, we will eliminate all the overheads in example 3.2.1.a to conduct the next example.

Example 3.2.1.b:

At $n = k, k = 3, l = 1$

Let $M_{3\times 1} = \begin{pmatrix} M_{1,1} \\ M_{2,1} \\ M_{3,1} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, $G_{3\times 3} = \begin{pmatrix} G_{1,1} & G_{1,2} & G_{1,3} \\ G_{2,1} & G_{2,2} & G_{2,3} \\ G_{3,1} & G_{3,1} & G_{3,1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$

Encoding:

$$G \times M = X = \begin{pmatrix} X_{1,1} \\ X_{2,1} \\ X_{3,1} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

After receiving G and X from the sender the receiver can form G|X as following:

$$G|X = \left(\begin{array}{ccc|c} G_{1,1} & G_{1,2} & G_{1,3} & X_{1,1} \\ G_{2,1} & G_{2,2} & G_{2,3} & X_{2,1} \\ G_{3,1} & G_{3,1} & G_{3,1} & X_{3,1} \end{array}\right) = \left(\begin{array}{ccc|c} 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{array}\right)$$

Or in linear algebra equivalent:

$$1M_{1,1} + 0M_{2,1} + 0M_{3,1} = X_1 = 1$$

$$1M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_2 = 0$$

$$0M_{1,1} + 0M_{2,1} + 1M_{3,1} = X_3 = 1$$

By performing typical substitution method assuming all these packets are received, such equation cannot be solved completely because the 3 equation in the example are not linearly independent ($G_{1,2}M_{2,1} = 0, G_{2,2}M_{2,1} = 0, G_{3,2}M_{2,1} = 0$, and is a sign that the equation carries no information on $M_{2,1}$), hence more information (packets) have to be received in order to complete the linear solving process.

For the PCD in this $n = k$ case, the PCD will be at 26.66% according to studies in Kolchin's theorem shown in Chapter 2.3. e.g. if there are $n = k$ amount of received packet, the matrix that forms from the received packets will have a chance of 26.66% to be linearly independent (solvable). When more packets n are received, PCD increases where the matrix that forms from the n received packets will have a higher chance being linearly independent. In accomplice with Random code and Kolchin's theorem, the overall parameters set in this thesis will be:

$$n = k + 10$$

$$k < 512$$

$$1\text{kb} < \text{l} < 9\text{kb}$$

3.2.2 Decoding Using Gaussian Elimination

With all the essential parameters such as $n = k + 10$, $k < 512$, and $1\text{kb} < \text{l} < 9\text{kb}$, the decoding method using Gaussian elimination can be proceeded further. In Example 2, the method of substitution can be used to solve $k = 3$ linear equations, but when it comes to $k > 4$, a systematic method such as Gaussian elimination is needed to solve the issue. The Gaussian elimination consist of 3 major steps:

1. Searching for pivot;
2. Swap row;
3. XOR row operations

These 3 steps need to iterate k amount of time until all the packets are completely decoded, and first step have to be done before second step can start; while third step can only start after second step ends its operation.

Example 3.2.2.a:

The packets that consist of information in the form of $G|X$, will be combined whereby the first row of the matrix is formed by the first received packets and goes on until the nth packet is received and form the last line of the $n \times k$ matrix as shown:

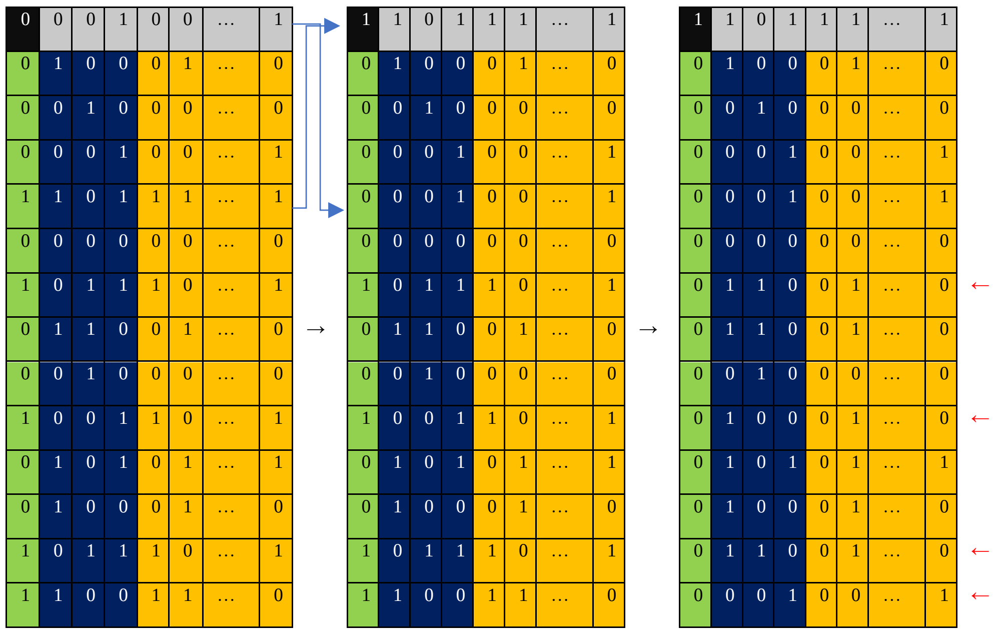
$k = 4, l = 8192, n = k + 10$

$$G|X = \left.\begin{array}{cccc|cccc}
\multicolumn{4}{c|}{\overbrace{\hphantom{0\quad 0\quad 0\quad 1}}^{k}} & \multicolumn{4}{c}{\overbrace{\hphantom{0\quad 0\quad \ldots\quad 1}}^{l}} \\
0 & 0 & 0 & 1 & 0 & 0 & \ldots & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & \ldots & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & \ldots & 1 \\
1 & 1 & 0 & 1 & 1 & 1 & \ldots & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & \ldots & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & \ldots & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & \ldots & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & \ldots & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & \ldots & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & \ldots & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & \ldots & 1 \\
1 & 1 & 0 & 0 & 1 & 1 & \ldots & 0
\end{array}\right\} n$$

🟦	Generated Matrix, *G*
🟨	Encoded Matrix, *E*
⬛	Pivot Point
▨	Pivot Row
🟩	Pivot Column
←	Pivot Column contains "1"

Step 1: Pivot Step 2: Swap Row Step 3: XOR

First iteration:



Step 1: Pivot

0	0	0	1	0	0	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
1	1	0	1	1	1	…	1
0	0	0	0	0	0	…	0
1	0	1	1	1	0	…	1
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
1	0	0	1	1	0	…	1
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
1	0	1	1	1	0	…	1
1	1	0	0	1	1	…	0

Step 2: Swap Row

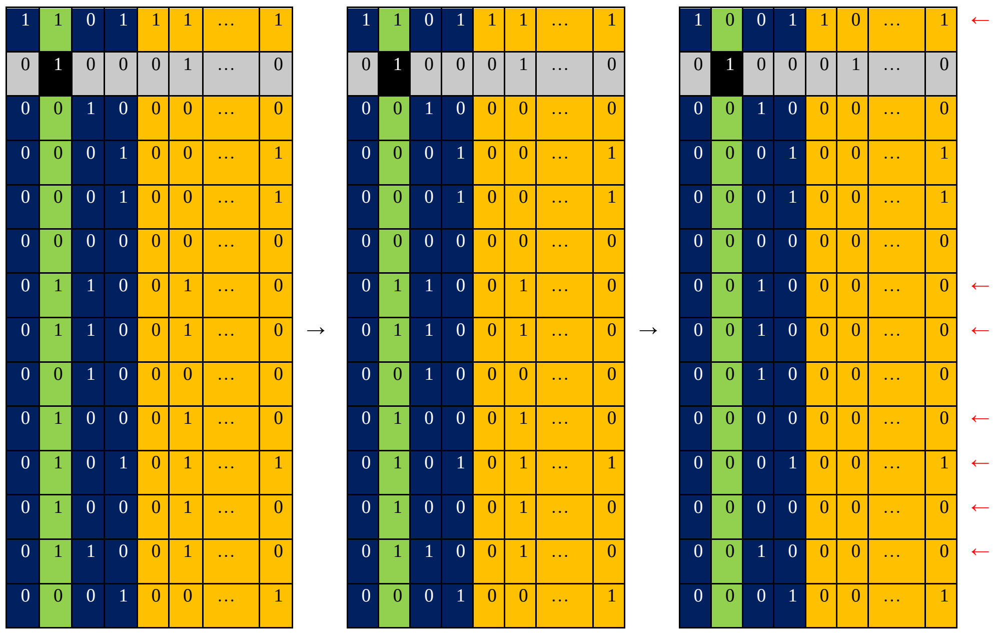
1	1	0	1	1	1	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
0	0	0	1	0	0	…	1
0	0	0	0	0	0	…	0
1	0	1	1	1	0	…	1
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
1	0	0	1	1	0	…	1
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
1	0	1	1	1	0	…	1
1	1	0	0	1	1	…	0

Step 3: XOR

1	1	0	1	1	1	…	1	
0	1	0	0	0	1	…	0	
0	0	1	0	0	0	…	0	
0	0	0	1	0	0	…	1	
0	0	0	1	0	0	…	1	
0	0	0	0	0	0	…	0	
0	1	1	0	0	1	…	0	←
0	1	1	0	0	1	…	0	
0	0	1	0	0	0	…	0	
0	1	0	0	0	1	…	0	←
0	1	0	1	0	1	…	1	
0	1	0	0	0	1	…	0	
0	1	1	0	0	1	…	0	←
0	0	0	1	0	0	…	1	←

Second iteration:

Step 1: Pivot Step 2: Swap Row Step 3: XOR



Step 1: Pivot

1	1	0	1	1	1	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
0	0	0	1	0	0	…	1
0	0	0	0	0	0	…	0
0	1	1	0	0	1	…	0
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
0	1	0	0	0	1	…	0
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
0	1	1	0	0	1	…	0
0	0	0	1	0	0	…	1

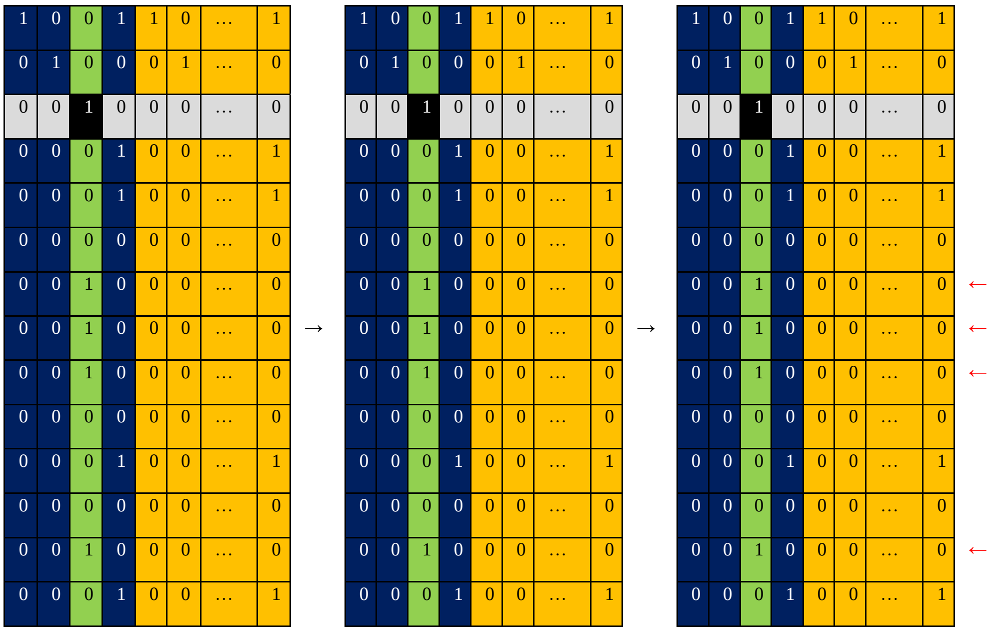
Step 2: Swap Row

1	1	0	1	1	1	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
0	0	0	1	0	0	…	1
0	0	0	0	0	0	…	0
0	1	1	0	0	1	…	0
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
0	1	0	0	0	1	…	0
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
0	1	1	0	0	1	…	0
0	0	0	1	0	0	…	1

Step 3: XOR

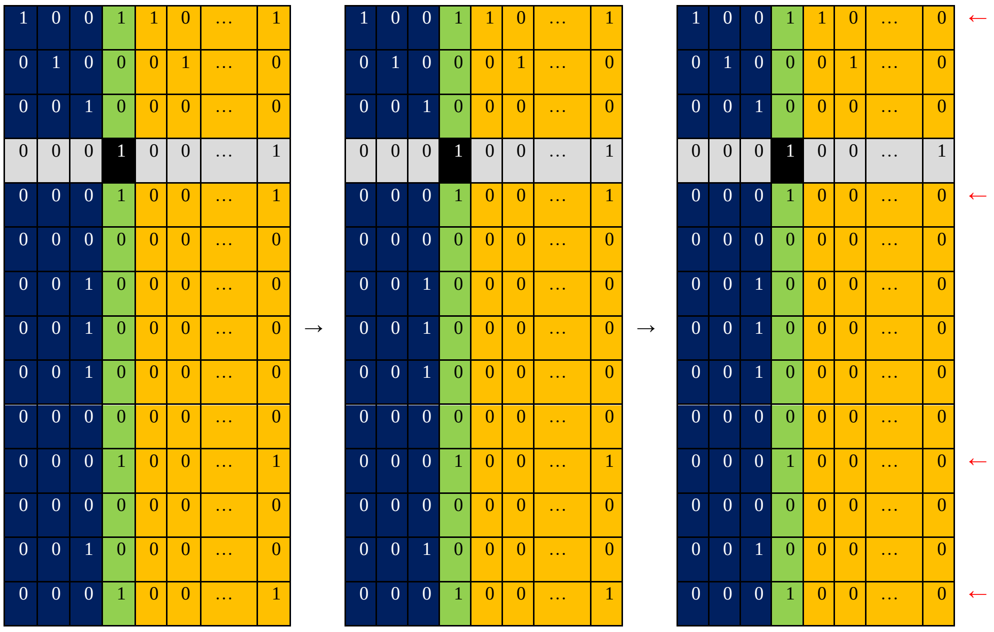
1	0	0	1	1	0	…	1	←
0	1	0	0	0	1	…	0	
0	0	1	0	0	0	…	0	
0	0	0	1	0	0	…	1	
0	0	0	1	0	0	…	1	
0	0	0	0	0	0	…	0	
0	0	1	0	0	0	…	0	←
0	0	1	0	0	0	…	0	←
0	0	1	0	0	0	…	0	
0	0	0	0	0	0	…	0	←
0	0	0	1	0	0	…	1	←
0	0	0	0	0	0	…	0	←
0	0	1	0	0	0	…	0	←
0	0	0	1	0	0	…	1	

Third iteration:



$$\begin{bmatrix} 1&0&0&1&1&0&\dots&1\\ 0&1&0&0&0&1&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1&0&0&1&1&0&\dots&1\\ 0&1&0&0&0&1&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1 \end{bmatrix} \rightarrow \begin{array}{cccccccc|l} 1&0&0&1&1&0&\dots&1&\\ 0&1&0&0&0&1&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\\ 0&0&0&1&0&0&\dots&1&\\ 0&0&0&1&0&0&\dots&1&\\ 0&0&0&0&0&0&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\leftarrow\\ 0&0&1&0&0&0&\dots&0&\leftarrow\\ 0&0&1&0&0&0&\dots&0&\leftarrow\\ 0&0&0&0&0&0&\dots&0&\\ 0&0&0&1&0&0&\dots&1&\\ 0&0&0&0&0&0&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\leftarrow\\ 0&0&0&1&0&0&\dots&1& \end{array}$$

Fourth (last) iteration:



$$\begin{bmatrix} 1&0&0&1&1&0&\dots&1\\ 0&1&0&0&0&1&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1 \end{bmatrix} \rightarrow \begin{bmatrix} 1&0&0&1&1&0&\dots&1\\ 0&1&0&0&0&1&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1\\ 0&0&0&0&0&0&\dots&0\\ 0&0&1&0&0&0&\dots&0\\ 0&0&0&1&0&0&\dots&1 \end{bmatrix} \rightarrow \begin{array}{cccccccc|l} 1&0&0&1&1&0&\dots&0&\leftarrow\\ 0&1&0&0&0&1&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\\ 0&0&0&1&0&0&\dots&1&\\ 0&0&0&1&0&0&\dots&0&\leftarrow\\ 0&0&0&0&0&0&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\\ 0&0&0&0&0&0&\dots&0&\\ 0&0&0&1&0&0&\dots&0&\leftarrow\\ 0&0&0&0&0&0&\dots&0&\\ 0&0&1&0&0&0&\dots&0&\\ 0&0&0&1&0&0&\dots&0&\leftarrow \end{array}$$

During the first iteration, pivoting point will be allocated at first row first column, then the first packet that contain “1” in the pivoting column will be searched and swap to the pivoting row, and then the other packets that contain “1” in the pivoting column will be eliminated by XOR operating them with the bitwise value $(G|X)$ of the pivoting row (highlighted in grey). In all the k iterations, different pivoting point ($pivot = 1,2,3 \dots k$) will be used in ascending order systematically to repeat the 3 steps for the completion of the GE process.

In the old days, even if GE is systematic, it imposed an entry time complexity of $O(k^3)$, Without a computer, it will take more than 10 pages of A4 papers that are full of equations to solve a $k = 100$ linear systems; for REC case, although GE is able to be used for decoding in most of the known REC, the decoding time complexity of REC is still $O(k^3)$. For instance, the operation count for GE will be demonstrated next to determine the time complexity.

3.2.3 Operational count for Gaussian Elimination

For a set of n packets matrix in Random code, the total operational count of pivoting search is:

$$\text{Total pivot search in GE} = \sum_{\text{pivot}=1}^{\text{k}} \tau_{\text{pivot}}$$

The process of pivoting search is just as simple as iterate through the main diagonal of the G in the received packets $(G|X)$.

$$\text{Total row swap in GE} = \sum_{\text{pivot}=1}^{\text{k}} \sum_{\text{swap}=1}^{\text{k+l}} \tau_{\text{swap}}$$

After the pivoting is searched, it will be swapped to the pivoting row. This process will basically consist $k + l$ swapping operations in each pivot iteration.

$$\text{Total XOR operation in GE} = \sum_{\text{pivot}=1}^{\text{k}} \sum_{\text{total row}=1}^{\text{n}-1} \sum_{\text{XOR}=1}^{\text{k+l}} \tau_{\text{XOR}}$$

Then the XOR operation will be performed on all the n rows that have a '1' in the pivoting column except the pivoting row. This required $n - 1$ operation counts that consist of $k + l$ XOR operations each because each packet will require $k + l$ XOR operations.

Total operation count in GE

$$= \sum_{\text{pivot}=1}^{k} \tau_{\text{pivot}} + \sum_{\text{pivot}=1}^{k} \sum_{\text{swap}=1}^{k+l} \tau_{\text{swap}}$$

$$+ \sum_{\text{pivot}=1}^{k} \sum_{\text{total row}=1}^{n-1} \sum_{\text{XOR}=1}^{k+l} \tau_{\text{XOR}}$$

$$= \sum_{\text{pivot}=1}^{k} \left(\tau_{\text{pivot}} + \sum_{\text{swap}=1}^{k+l} \tau_{\text{swap}} + \sum_{\text{total row}=1}^{n-1} \sum_{\text{XOR}=1}^{k+l} \tau_{\text{XOR}} \right)$$

$$= k\left(\tau_{\text{pivot}} + (k+l)(\tau_{\text{swap}}) + (n-1)(k+l)(\tau_{\text{XOR}})\right)$$

$$\approx k(n)(k+l)\tau_{\text{XOR}} \qquad (3.1)$$

Hence for the whole Gaussian elimination operational count, it can be seen that the XOR operations parts are actually the dominating part, where the profiling in GE shows that 99% of the operation is performed in the XOR operations, while the other 1% is from the swap and pivot search, hence the swap and pivot operation is generally negligible.

The table and graph below show the effects of the entries message size k imposed by GE. The GE is constructed based on the typical REC decoder shown in the study of RaptorQ (Hu, et al., 2012), and further modifications with our best effort from several papers that emphasize on GE optimisation are done in this research.

At 4GHz workstation XEON E3 computer configuration:

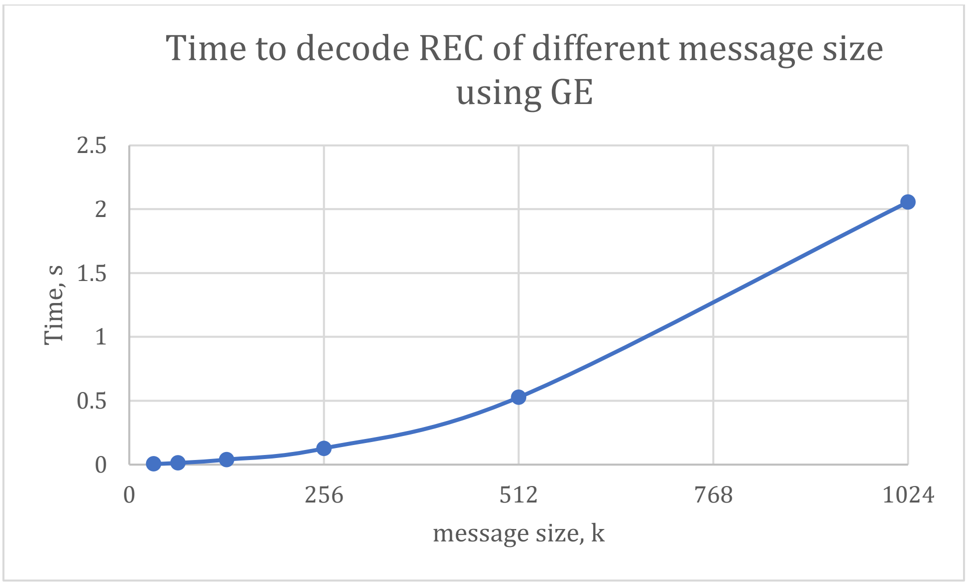


Figure 3.1: Effects of message size k towards the decoding speed.

In Figure 3.1, it can be seen that even at a 4GHz workstation XEON E3 computer, the time consumed in decoding (Gaussian Elimination) the larger k messages increase exponentially.

3.3 Redimensioning

When we study into the decoding complexity of GE, which is $O(k^3)$ (Bioglio, et al., 2009), the large data size could yield a very large complexity. The decoding time of a very large file can be made unreasonable long. Even with the aids of GPU parallel processing, the decoding time will still increase exponentially as the file size gets bigger due to the hardware limitation.

By addressing such issue, we propose a technique called *redimensioning* which is applicable to almost all the REC; it is a process of partitioning one big file into γ numbers of fixed-size subsets and decode each subset in a less complex manner. In this case, the speed of GE decoding process can be enhanced significantly.

3.3.1 Encoding of Redimensioning

The basic procedure for redimensioning starts with:

1.) Before the encoding, the stream of a message will be first divided into γ numbers of equal size subsets.

Example 3.3.1a:

$$k = 4, \qquad l = 5, \qquad \gamma = 2$$

$$M = 01000100111000100100 \rightarrow \overbrace{\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}}^{l}\Big\}k \; {}_{k\times l}$$

If $\gamma = 2$, message, M will be partition into 2 equal size subsets (M^1 & M^2) with the dimension of $\frac{k}{\gamma} \times l$ as shown:

$$k = 4, \qquad l = 5, \qquad \gamma = 2$$

$$\begin{array}{c} M^1_{\frac{k}{\gamma}\times l} \\ \hline M^2_{\frac{k}{\gamma}\times l} \end{array} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{array}{c} 2\times 5 \\ \\ 2\times 5 \end{array}$$

2.) The γ subsets of the M will be encoded separately with random generated G to form respective encoded symbols.

$$\boldsymbol{G^1, G^2 .. G^\gamma \ is\ the\ subset\ of\ G\ after\ partition}$$

$$\boldsymbol{M^1, M^2 .. M^\gamma \ is\ the\ subset\ of\ M\ after\ partition}$$

$$\boldsymbol{X^1, X^2 .. X^\gamma \ is\ the\ subset\ of\ X\ after\ partition}$$

Example 3.3.1b:

$$\boldsymbol{G^1}_{\boldsymbol{n}\times\frac{k}{\gamma}} \times \boldsymbol{M^1_{\frac{k}{\gamma}\times l}} = \boldsymbol{X}^1_{\boldsymbol{n}\times l}$$

$$\boldsymbol{G^2}_{\boldsymbol{n}\times\frac{k}{\gamma}} \times \boldsymbol{M^2_{\frac{k}{\gamma}\times l}} = \boldsymbol{X}^2_{\boldsymbol{n}\times l}$$

In packet form ($\boldsymbol{G|X}$):

$$\left(\boldsymbol{G^1}_{\boldsymbol{n}\times\frac{k}{\gamma}}\middle|\boldsymbol{X^1_{n\times l}}\right) \ \boldsymbol{and} \ \left(\boldsymbol{G^2}_{\boldsymbol{n}\times\frac{k}{\gamma}}\middle|\boldsymbol{X^2_{n\times l}}\right)$$

3.3.2 Decoding of Redimensioning

After receiving sufficient packets to decode all γ redimensioned packets ($G^1|X^1, G^2|X^2 \dots G^\gamma|X^\gamma$) back into original messages ($M^1, M^2 \dots M^\gamma$), new message will have a new size k' which is equivalent to $\frac{k}{\gamma}$, and also new n' for each redimensioned matrix that requires $k' + \varepsilon$ for a 99.99% PCD, the total overhead for redimensioned matrix will become:

$$M^1 \to n' = k' + \varepsilon$$

$$M^2 \to n' = k' + \varepsilon$$

$$\vdots$$

$$M^\gamma \to n' = k' + \varepsilon$$

Example 3.3.2a:

$$k' = \frac{k}{\gamma}, \qquad n' = k' + 10,$$

Received packets:

$$\left(\boldsymbol{G}^{1}_{n'\times k'} \middle| \boldsymbol{X}^{1}_{n'\times l}\right) \ \boldsymbol{and} \ \left(\boldsymbol{G}^{2}_{n'\times k'} \middle| \boldsymbol{X}^{2}_{n'\times l}\right)$$

$$\downarrow$$

$$\left(\boldsymbol{G}^{1}_{n'\times k'} \middle| \boldsymbol{X}^{1}_{n'\times l}\right) \xrightarrow[\boldsymbol{Gaussian\ elimination}]{} \left(\begin{array}{c|c} \boldsymbol{I}_{k'\times k'} & \boldsymbol{M}^{1}_{k'\times l} \\ \boldsymbol{Z}_{10\times k'} & \boldsymbol{Z}_{10\times l} \end{array}\right)$$

$$\left(\boldsymbol{G}^{2}_{n'\times k'} \middle| \boldsymbol{X}^{2}_{n'\times l}\right) \xrightarrow[\boldsymbol{Gaussian\ elimination}]{} \left(\begin{array}{c|c} \boldsymbol{I}_{k'\times k'} & \boldsymbol{M}^{2}_{k'\times l} \\ \boldsymbol{Z}_{10\times k'} & \boldsymbol{Z}_{10\times l} \end{array}\right)$$

$$\begin{matrix} \boldsymbol{M}^{1}_{k'\times l} \\ \boldsymbol{M}^{2}_{k'\times l} \end{matrix} \to \boldsymbol{M}_{k\times l}$$

I = identity matrix

Z = zero matrix

As calculated earlier in section 3.2, the operational count of decoding using GE without redimension is:

$$\text{GE operational count} = k(n)(k+l)\tau_{\text{XOR}}$$

For the operational count of GE after redimension:

$$\text{M}^1 \text{ GE operational count} = k'(n')(k'+l)\tau_{\text{XOR}}$$

$$\text{M}^2 \text{ GE operational count} = k'(n')(k'+l)\tau_{\text{XOR}}$$

$$\vdots$$

$$\text{M}^\gamma \text{ GE operational count} = k'(n')(k'+l)\tau_{\text{XOR}}$$

$$\text{Total redimension operation count} = M^1 + M^2 + \cdots M^\gamma$$

$$= \gamma(k'(n')(k'+l)\tau_{\text{XOR}}) \qquad (3.2)$$

To compare the speed up of redimensioning towards the normal GE decoding:

$$\text{Speedup}_{\text{redimension}} = \frac{\text{GE}}{\text{Redimension GE}} = \frac{k(n)(k+l)\tau_{\text{XOR}}}{\gamma(k'(n')(k'+l)\tau_{\text{XOR}})}$$

Since $l \gg k$ and $l \gg k'$:

$$k + l \approx l, \qquad k' + l \approx l$$

$$\frac{n}{n'} = \frac{k+10}{\frac{k}{\gamma}+10} \approx \frac{k}{\frac{k}{\gamma}} \approx \gamma$$

$$\text{Speedup}_{\text{redimension}} = \frac{k\gamma(l)\tau_{\text{XOR}}}{\gamma\frac{k}{\gamma}(l)\tau_{\text{XOR}}} \approx \gamma \qquad (3.3)$$

From eq. 3.3, it can be seen that the redimension technique can speed up the decoding process by a factor of γ, and it is proven experimentally by using CPU in the table below.

Table 3.1: Time for GE with Redimension decoding of Random Code in different $\boldsymbol{k}$ at $\boldsymbol{l = 8192\ bits}$, and $\boldsymbol{\gamma = 2}$

k	Time to decode REC using GE	Time to decode REC using GE with redimensioning, $\gamma = 2$	Speedup $\frac{\text{GE}}{\text{GE redimensioned}}$
32	0.004992	0.003828	1.3040
64	0.013208	0.009801	1.347
128	0.039508	0.0264	1.4965
256	0.126859	0.08012	1.5833
512	0.526941	0.2731	1.9295
1024	2.056845	1.0610	1.9386

.

From Table 3.1, at $k = 32$ and $\gamma = 2$, the speedup is different from the eq 3.3 Where it is supposed to reach a speedup of 2. This is because based on the assumption that made in the formation of eq.3.3, the k is large. Where 10 becomes negligible when k is large, at small k the 10 had a certain weightage that contributes to a slower decoding time, and it is proven when $k \geq 512$, the speed up converges back to γ.

With this technique, the large file ($k > 512$) with high time complexity is no longer a large issue for decoding, and the only things to focus is to enhance the performance for decoding a smaller file of size $k \leq 512$.

3.4 First Degree Parallel Processing

As mentioned, GE would impose an entry time complexity of $O(k^3)$ during the REC decoding; this means that the process of GE computation will eventually go slower as the k (message size) increases. Although the improvements over CPU implementations have previously been achieved for GE in terms of raw speed (using faster computer) and the redimensioning technique (mathematical technique), however the utilization of the underlying available computational resources was still low, for instance parallel processing that can be done in almost all modern computers that contain the GPU.

In this section, the propose method to solve the complexity issue by using the state of art of GPU will be discussed.

3.4.1 Implementation of GPU Unit for Gaussian Elimination

Gaussian elimination (GE) decoding algorithms have triple nested loops computationally which led to an entry complexity of $O(k^3)$. Such complexity has led researchers to approach them in a parallel processing manner.

3.4.1.1 Parallelising Code in CUDA

CUDA is a parallelisation platform developed by NVIDIA. It is designed in such a way that applications of parallel programming can be executed on both CPU and GPU. For parallelisation, the CPU which is the host, will initialize the data to be transferred and executed in the GPU device and the GPU will allocate the pre-defined threads in the <<<blocks, threads>>> bracket to invoke the GPU kernel.

After GPU completes its calculation in parallel by using the allocated threads, the program in device is now considered completed, and the output data will be copied back to the host before the device can release the storage space in GPU and get ready for the next task. The process is shown in the following example:

Example 3.4.1.1.a:

```
1. //CPU CODE
2. main(){                                  //Host function
3.       int x[i];                          //Declare Variable
4.       for (int i=0;i<3;i++){             //3 for Loops {0,1,2}
5.             x[i]=i+i;
6.       }
7.
8.       for (int i=0;i<3;i++){             //print the output
9.             cout<<x[i]<<endl;            //end line after printing
                                            one value
10.      }
11. }
```

Output:

```
0
2
4
```

From the CPU code as shown in example 3.4.1.1.a, the additional operation (line 5) in the algorithm is independent of each other, even though the loops of these addition operations are not arranged in sequence, the output will be the same, and this is how parallel processing come into play, because rather than computing these additions sequentially, multiple of them can be concurrently executed and still yield the same outputs.

Parallelised addition operation in GPU:

```
//GPU CODE
__global__ CallGPUkernel(int *x){                //GPU kernel
     i=threadid.x;                                //identify what
                                                   thread to use
     If(i<3){
           x[i]=i+i;                              //simple addition
     }
}

main(){                                    //Host function
     int x[i];                             //Declare Host variable
     int*dev_x;                            //Declare Device variable

     cudaMemcpy(dev_x,x,3*sizeof(int),cudaMemcpyHostToDevice);
      //memory copy from host to device to perform parallelisation
      //(device variable, host variable, size of variable, memory
        from host to device)

     CallGPUkernel<<<1,3>>>(dev_x);
      //invoke GPU kernel with <<< blocks amount, threads amount>>>
     (input variable)

     cudaMemcpy(x,dev_x,3*sizeof(int),cudaMemcpyDeviceToHost);
      //memory copy from device to host to perform
     //(host variable, device variable, size of variable, memory
        from device to host)

     cudaFree (dev_x);                     //free the device memory


     for(int i=0;i<3;i++){                 //print output
           cout << x[i]<<endl;
     }
}

```

Output:

```
0
2
4
```

3.4.2 Ideal vs. Practical Parallelisation

The idea of parallelisation is to simultaneously perform multiple independent operations at once, hence reducing the overall operational time. In an ideal case, parallelisation equation will be represented below:

$$T_{\text{parallel}} = \left\lceil \frac{y \times \tau_{\text{operation}}}{\text{threads allocated}} \right\rceil \tag{3.4}$$

$T_{\text{parallel}} =$ Time taken to complete all operations after parallelisation
$y =$ Amount of independent operation
$\tau_{\text{operation}} =$ Time taken to complete one operation
threads allocated $=$ Threads used in parallelisation

However, for a parallelisation to perform practically, parallel overheads such as physical limitation, data reuse, memory dependency, kernel call delay, threads allocation, etc., should be considered in a GPU device. In this thesis, they are all categorized under parallel overheads $p_{\text{operation}}$.Hence practically, the parallelisation equation is represented as below:

$$T_{\text{parallel}} = \left\lceil \frac{y \times \tau_{\text{operation}}}{\text{threads allocated}} \right\rceil + p_{\text{operation}} \tag{3.5}$$

$T_{\text{parallel}} =$ Time taken to complete all operations after parallelisation
$y =$ Amount of independent operation
$\tau_{\text{operation}} =$ Time taken to complete one operation
threads allocated $=$ Threads used in parallelisation
$p_{\text{operation}} =$ parallel overhead

3.4.3 Parallelisation of Gaussian Elimination

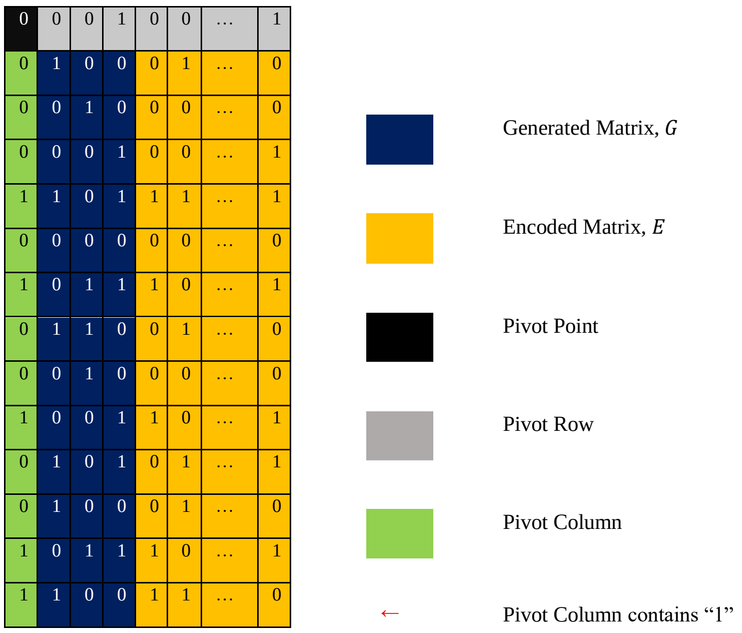
The parallelization of GE algorithms is a challenging process. The number of threads and block allocation must be decided carefully. Attention should be paid to the synchronization part to get accurate results, because a slight desynchronize value might end up ruining the results.

To start the parallel processing in $\mathrm{GE}_{\mathrm{parallel}}$, the steps that used is the same as the steps for $\mathrm{GE}_{\mathrm{serial}}$ in Chapter 3.2 and most importantly it yields the same result like $\mathrm{GE}_{\mathrm{serial}}$ does:

1. Searching for pivot;
2. Swap pivoting rows to the right position;
3. XOR row operation.

3.4.3.1 Pivoting Search

The pivoting search step is represented in the outer most nested loop of the code in appendix. During this step, the parallelisation is not required because it is as simple as searching the right condition (if pivoting row and column not "1") to progress to step 2- Row swap.



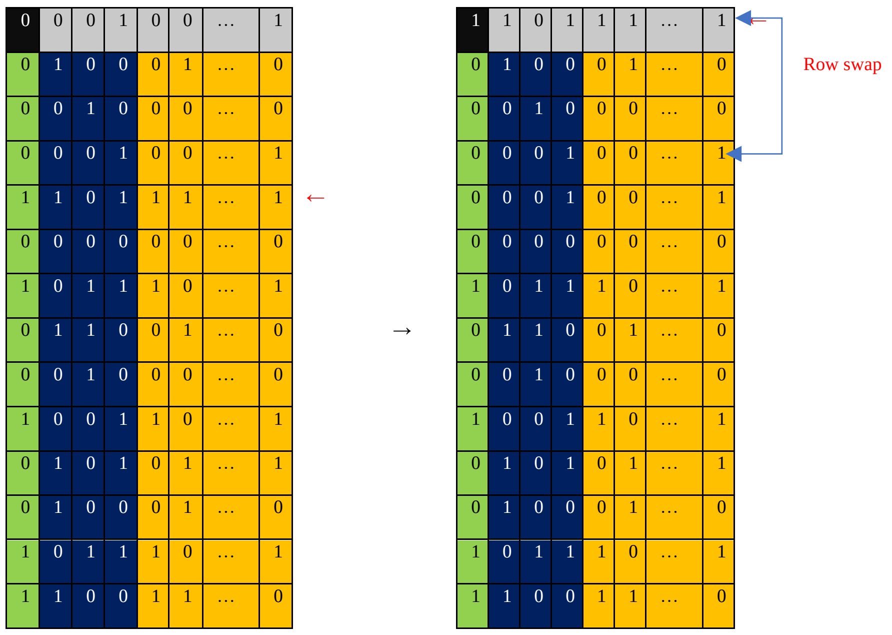
0	0	0	1	0	0	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
1	1	0	1	1	1	…	1
0	0	0	0	0	0	…	0
1	0	1	1	1	0	…	1
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
1	0	0	1	1	0	…	1
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
1	0	1	1	1	0	…	1
1	1	0	0	1	1	…	0

Figure 3.2: Pivoting Search

3.4.3.2 Row Swap

If the pivoting search in the first step meets the condition to proceed with row swapping, which is represented in the second nested loop, the row swapping will be executed.

Step 1: Pivoting Search Step 2: Row Swap



Step 1: Pivoting Search

0	0	0	1	0	0	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
1	1	0	1	1	1	…	1
0	0	0	0	0	0	…	0
1	0	1	1	1	0	…	1
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
1	0	0	1	1	0	…	1
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
1	0	1	1	1	0	…	1
1	1	0	0	1	1	…	0

Step 2: Row Swap

1	1	0	1	1	1	…	1
0	1	0	0	0	1	…	0
0	0	1	0	0	0	…	0
0	0	0	1	0	0	…	1
0	0	0	1	0	0	…	1
0	0	0	0	0	0	…	0
1	0	1	1	1	0	…	1
0	1	1	0	0	1	…	0
0	0	1	0	0	0	…	0
1	0	0	1	1	0	…	1
0	1	0	1	0	1	…	1
0	1	0	0	0	1	…	0
1	0	1	1	1	0	…	1
1	1	0	0	1	1	…	0

Figure 3.3: Row Swap

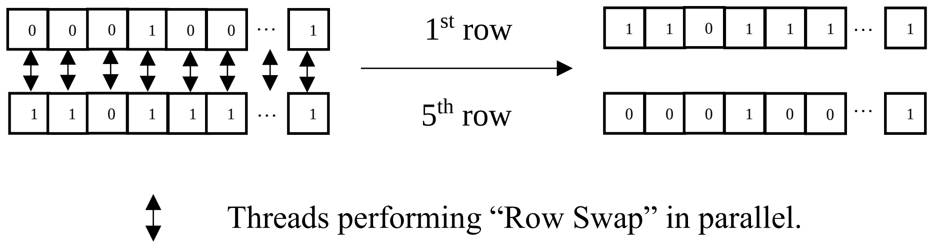


Figure 3.4: Threads allocation for Swap Row

For example, in Figure 3.3, when the row swapping process is executed, two rows will generally switch their positions with each other. Another example shown in Figure 3.4 is that, in a CPU case, the row swapping process will undergo 6 iterations to swap the corresponding elements in all the 6 columns. For parallelisation, the 6 iterations can be reduced to one iteration in terms of processing time by performing parallelisation of 6 XOR operation columns with 6 GPU's threads simultaneously as shown in Figure 3.4.

3.4.3.3 XOR Operation

After the row swap step is done, here comes the step 3- XOR operation where the others row that have a “1” in pivoting column will perform XOR operation with the pivoting row as shown in Figure 3.5.

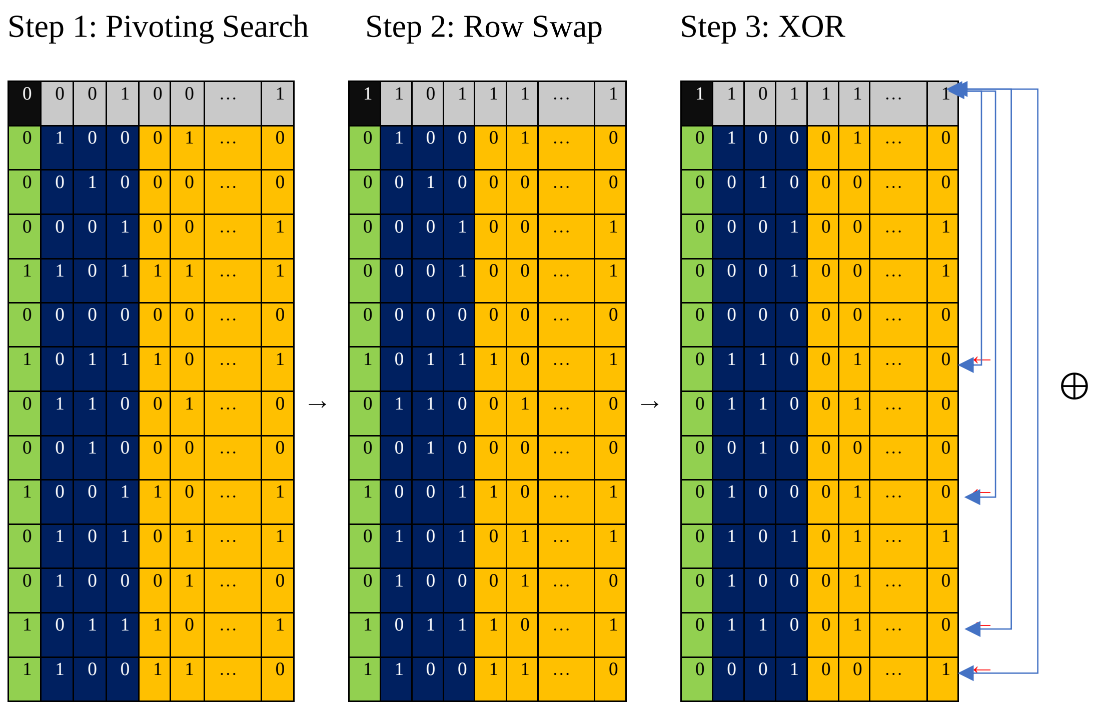


Figure 3.5: XOR operations

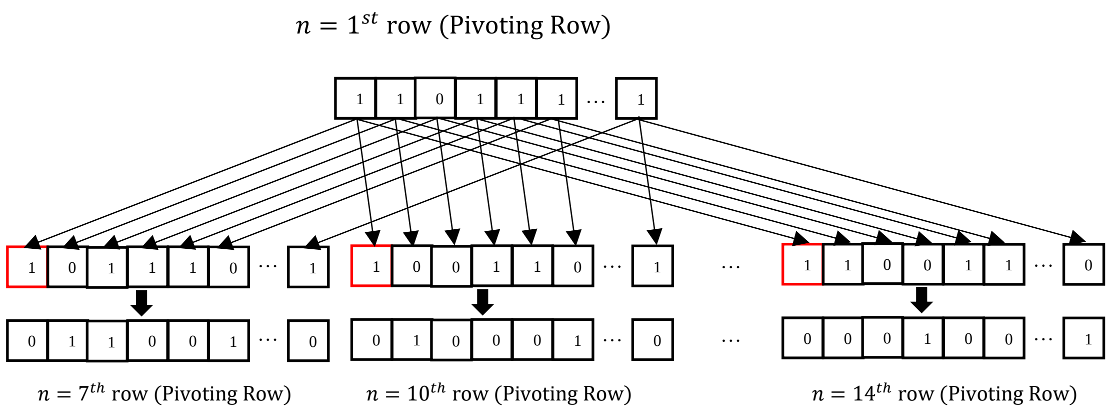

$n = 1^{st}$ row (Pivoting Row)
$n = 7^{th}$ row (Pivoting Row)
$n = 10^{th}$ row (Pivoting Row)
$n = 14^{th}$ row (Pivoting Row)

Figure 3.6: Threads allocation for XOR operation

For example, in Figure 3.6, the XOR operation is performed on the row that consists of '1' (in red square) in their respective pivoting column, in this case, other than pivoting row, the pivoting column that contains "1" in other rows will be basically eliminated. In a GE_{serial} case, the XOR operation will iterate from

2nd row, 1st column $=$ 2nd row, 1st column $\oplus$ pivoting row, 1st column to 10th row, 6th column $=$ 10th row, 6th column $\oplus$ pivoting row, 6th column, which is a total of 10×6 steps XOR operations to iterate through.

While for $\text{GE}_{\text{parallel}}$, the 10×6 steps of XOR operation will be allocated with 10×6 threads each for parallel processing, which means that the 60 steps of XOR operation in serial CPU operation can be reduced to complete in one step ideally.

3.4.4 Construction of $\text{GE}_{\text{parallel}}$ in CUDA

Gaussian Elimination in Parallel

```
threadX = threaded.X × threadX workload                        //mapping Threads X
threadY = threaded.Y × threadY workload                        //mapping Threads Y

//GPU kernel
define Swap_Row (G|X[n][k+l], pivot)                           //Swap row kernel
    for row = pivot to n-1 do                                  //check all n rows
        if (G|X[pivot][pivot]≠1&& G|X[row][pivot]==1) then     //if pivot point is zero
                                                                while another row had
                                                                pivot column of 1

            for offset=0 to threadY workload do                //mapping Threads Y
                col=threadY + offset                           //allocating workload
                Swap G|X[row][col] to G|X[pivot][col]          //Swap between rows
            End for
        End if
        BREAK                                                  //Break the whole swap
                                                               operation when one row
                                                               swap is done
    End for

define Pivot_check (G|X[n][k+l], check[n], pivot)              //Check pivot in GPU
    for offset=0 to thread X workload do                       //one thread will handle
                                                               how many workload/rows
        row= threadX+offset
        if (row < n) then                                      //thread allocate cannot
                                                               more than n
            check[row]=G|X[row][pivot]                         //allocate pivot point
                                                               value into the check
                                                               array
        End if
    End for

define XOR (G|X[n][k+l], check[n], pivot)                      //the XOR kernel in GPU
    for offset=0 to thread X workload do                       //one thread will handle
                                                               how many workload/rows
```

```
27.
28.          row= threadX+offset
29.          if(G|X[row][pivot]==1 && row ≠ pivot) then
30.              for offsetY=0 to thread Y workload do
31.                  col=threadY + offsetY                   //one thread will handle
                                                             how many workload/columns
32.
33.                  G|X[row][col]= G|X[row][col] ⊕ G|X[pivot][col]//XOR between rows
34.              End for
35.          End if
36.      End for
37.
38.  Host to Device memory copy
39.
40.  For pivot=0 to k-1 do                                   //iterates through all k
                                                             pivot point
41.      Swap_Row <<<blocks, threads >>> (G|X[n][k+l], pivot)           //Swap Row
42.      Pivot_check <<<blocks, threads >>> (G|X[n][k+l], check[n], pivot)
43.      XOR <<<blocks, threads >>> (G|X[n][k+l], check[n], pivot)      //XOR
44.  End for
45.
46.  Host to Device memory copy
```

In GPU, threads are the key element for parallelisation, hence they will be identified into threadX and threadY for handling the row and column of the matrix G|X.

All the GPU call function kernel will be constructed following the 3 steps in GE, so when the G|X matrix is received with the dimension of $n \times (k + l)$, the code can fully decode G|X back into the original messages.

First will be the pivoting search, since this step is just iterating through the GE process, it will be handled by the for-loop in line 40.

Next, the swap row GPU kernel is constructed in line 4-14, and called in line 41, the kernel is called to execute the row swapping step with $G|X$ as the input, and piv represents the pivoting point location and $<<< threads, blocks >>>$ brackets show the threads and block amount that are needed to be allocated for the kernel.

After this, pivoting value will be recorded in a new set of the array for checking purpose as shown in line 17-23, and call this a new set of array - $check[n]$ where n represents the size of this array.

Then come to the last step, which is the XOR operation step, such GPU code is constructed in line 25-36, where the input data from $G|X$ will be used to perform XOR operations in between rows.

From the pseudocode, there are offsets in line 9 and line 18 etc. In general, such offsets are used to map the GPU threads into the right positions to handle a certain amount of threads.

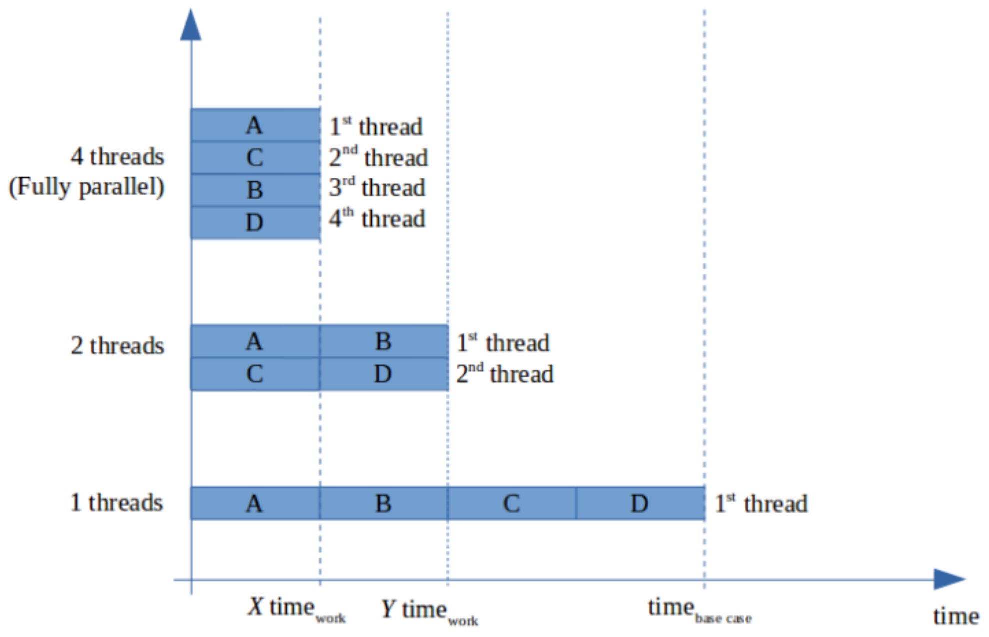


Figure 3.7: Offset of threads distribution.

If there are 4 workloads that is represented in A, B, C and D as shown in Figure 3.7, full parallelisation will be allocating maximum 4 threads by distributing one thread into every workload.

However, to study the effects of threads towards the speedup of such workload, 1 thread will be allocated to handle all A, B, C and D workload and this will be called a "base case", and the time to execute all the workload by using one thread will be called $\text{time}_{\text{base case}}$. Other than this, when there are 2 threads allocated to the 4 workloads, these threads will be mapped accordingly to handle the distributed workload, e.g. first thread will handle workload A and B while the second thread will handle C and D, and the time to execute all the workload by threads mapping will be $\text{time}_{\text{work}}$.

$$\text{Speed Up} = \frac{\text{time}_{\text{base}}}{\text{time}_{\text{work}}} \qquad (3.6)$$

According to the pseudo-code, the operational count of $\text{GE}_{\text{parallel}}$ will be formulated later and apply on the $\text{GE}_{\text{serial}}$ operational count in eq 3.1, and further elaborate it on the $\text{GE}_{\text{parallel}}$:

$$k\big((n)(k+l)\tau_{\text{XOR}}\big) \xrightarrow{parallelisation} k\left(\left\lceil\frac{(n)(k+l)}{\text{threads allocated}}\right\rceil \tau_{\text{XOR}} + p_{GE}\right) \quad (3.7)$$

$k =$ loops in line 40

$n =$ loops that goes through all the rows

$k + l =$ loops that goes through all the columns

threads allocated $=$ threads for parallelisation $(1,2,3 \ldots (n) \times (k+l))$

$\tau_{XOR} =$ operation constant in line 33

p_{GE} = overal parallel overhead

Assuming that maximum threads, $k(k+l)$, will be allocated:

$$\mathrm{GE}_{\text{operational count}} = k(1\tau_{\mathrm{XOR}} + p_{GE}) \tag{3.8}$$

Parameters added in this $\mathrm{GE}_{\text{parallel}}$ operational count is the "threads allocated" and the "p_{GE}"; but, a thread is not capable of allocating more than its workload ($(n) \times (k+l)$ in this case). Other than that p_{GE} is a general idea of the parallel overheads in every parallelisation, for example, in ideal case, N amount of workload executed in parallel should yield ideal N times faster than N amount of workload executed in series, however this parallel overheads will practically inhibit the performance of such parallelisation, and yield probably only 0.5N speedup, such parallelisation overhead will be visualised in the Result chapter where the performance of GE will be demonstrated to show how parallel overhead will affect its performance.

3.5 Gaussian Elimination with Matrix Multiplication (GEMM)

In this thesis, the new decoding algorithm that propose is the Gaussian Elimination with Matrix Multiplication (GEMM). As we had explained previously, GE is an iterative algorithm different from matrix multiplication (MM), a widely-recognized candidate for a fast GPU implementation, which does not suffer from the premature saturation of GPU bandwidth resources as GE does. Hence, the idea of GEMM advocates the combination of GE and MM to perform the REC decoding.

$$\textbf{Received packets} - \; G_{n\times k}|X_{n\times l}$$

$$G_{n\times k}|\begin{matrix} I_{k\times k} \\ Z_{10\times k}\end{matrix} \xrightarrow{\text{Gaussan Elimination}} \begin{matrix} I_{k\times k} \\ Z_{10\times k}\end{matrix}|\begin{matrix} G^{-1}_{k\times k} \\ Z_{10\times k}\end{matrix} \tag{3.9}$$

$$G^{-1}_{k\times k} \times X_{k\times l} = M_{k\times l} \tag{3.10}$$

When the packets are received in the form of $G|X$, the basic idea of GEMM is to first inverse the $G \rightarrow G^{-1}$ by using GE, and the G^{-1} will be used to matrix multiply with X to retrieve the original message M.

3.5.1 Inversion, INV

Since the Gaussian Elimination can be used for inversion ($G|I \rightarrow I|G^{-1}$) as well as the direct decoding previously ($G|X \rightarrow I|M$), the inversion using Gaussian elimination will be called INV while the direct decoding in Section 3.2 using Gaussian Elimination remains as GE.

During the decoding of the received packets in the form of $G|X$ using GEMM, INV is the first phase of GEMM which is the eq 3.8 where:

$$G_{n\times k}|\begin{matrix} I_{k\times k} \\ Z_{10\times k}\end{matrix} \xrightarrow{\text{Gaussan Elimination}} \begin{matrix} I_{k\times k} \\ Z_{10\times k}\end{matrix}|\begin{matrix} G^{-1}_{k\times k} \\ Z_{10\times k}\end{matrix}$$

G will be used to perform INV, It will be augmented with a set of value that is made up of $G_{n\times k}|\begin{matrix} I_{k\times k} \\ Z_{10\times k}\end{matrix}$, however, due to the non-square matrix formed from the packets, the I needs to be dynamically allocated to the correct position to align with the "useful" information, (useful information will slowly shift to first k rows. The idea is quite similar to GE algorithm with certain additional steps as shown below:

1.) G will be first augmented with a set of zero matrix, Z as shown in example 3.5.1.a.

Example 3.5.1.a:

$k = 3, l = 8192, n = k + 10$

$$G_{3\times 13}\,|X_{13\times l} = \left(\begin{array}{ccc|c} 0 & 0 & 1 & x_1 \\ 0 & 1 & 0 & x_2 \\ 0 & 1 & 1 & x_3 \\ 1 & 1 & 0 & x_4 \\ & \vdots & & \vdots \\ 1 & 1 & 1 & x_{13} \end{array}\right)$$

$$G_{3\times 13} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \\ & \vdots & \\ 1 & 1 & 1 \end{pmatrix} \quad X_{13\times l} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_{13} \end{pmatrix} \quad G_{3\times 13}\,|Z_{3\times 13} = \left(\begin{array}{ccc|ccc} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ & \vdots & & & \vdots & \\ 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

2.) After this, $G|Z$ will be inversed by using Gaussian Elimination that starts from first step GE- pivoting search; the first row that contains ‘1’ (red 1) in pivoting column in example 4.2.1.b.

Example 3.5.1.b:

Pivoting (row, column) = (1, 1)

$$G_{3\times 13}\,|Z_{3\times 13} = \left(\begin{array}{ccc|ccc} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ \color{red}{1} & 1 & 0 & 0 & 0 & 0 \\ & \vdots & & & \vdots & \\ 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

3.) Then, the first row of G that contains '1' in pivoting column will be swapped to the pivoting row. (row in red will swap with row in blue in example 4.2.1.c) in this case the "useful" information will be swapped to the correct position including the X.

Example 3.5.1.c:

Pivoting (row, column) = (1, 1)

$$G_{3\times 13}\,|Z_{3\times 13} = \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ & \vdots & & & \vdots & \\ 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right) \qquad X_{13\times l} = \begin{pmatrix} x_4 \\ x_2 \\ x_3 \\ x_1 \\ \vdots \\ x_{13} \end{pmatrix}$$

4.) Now, we can confirm that the first row will contain useful information, then in the pivoting row (red in example 3.5.1.c), a dynamically allocate Identity matrix, I will be added to align with the pivoting row of $G|Z$ to form $G_{n\times k}|\begin{smallmatrix} I_{k\times k} \\ Z_{10\times k} \end{smallmatrix}$.

Example 3.5.1.d:

Pivoting (row, column) = (1, 1)

$$G_{3\times 13}\,|Z_{3\times 13} = \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ & \vdots & & & \vdots & \\ 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right) \qquad I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$G_{13\times3}\left|\begin{matrix}I_{1\times3}\\ Z_{12\times3}\end{matrix}\right. = \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 1 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ & \vdots & & & \vdots & \\ 1 & 1 & 1 & 0 & 0 & 0 \end{array}\right)$$

5.) Then come to the Step 3 XOR operation part, the pivoting row(red) will be XOR-ed with all other rows that contain '1' in their respective pivoting columns.

Example 3.5.1.e:

Pivoting (row, column) = (1, 1)

$$G_{13\times3}\left|\begin{matrix}I_{1\times3}\\ Z_{12\times3}\end{matrix}\right. = \left(\begin{array}{ccc|ccc} 1 & 1 & 0 & 1 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 1 & 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0\\ & \vdots & & & \vdots & \\ 0 & 0 & 1 & 1 & 0 & 0 \end{array}\right)$$

6.) Repeats step 1 to 5 by moving to next pivoting point in ascending order. At the end, it will result in inversing the $G \rightarrow G^{-1}$(in purple) .

Example 3.5.1.f:

$$G_{k\times n}\,|Z_{k\times n} \rightarrow G_{n\times k}\left|\begin{matrix}I_{k\times k}\\ Z_{10\times k}\end{matrix}\right. \xrightarrow{\text{Gaussan Elimination}} \begin{matrix}I_{k\times k}\\ Z_{10\times k}\end{matrix}\left|\begin{matrix}G^{-1}_{k\times k}\\ Z_{10\times k}\end{matrix}\right.$$

Useful information

$$\begin{matrix}I_{3\times3}\\ Z_{10\times3}\end{matrix}\left|\begin{matrix}G^{-1}_{3\times3}\\ Z_{10\times3}\end{matrix}\right. = \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 1 & 0\\ 0 & 1 & 0 & 0 & 1 & 0\\ 0 & 0 & 1 & 0 & 1 & 1\\ \hline 0 & 0 & 0 & 0 & 0 & 0\\ & \vdots & & & \vdots & \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \qquad X_{13\times l} = \begin{pmatrix} x_4\\ x_2\\ x_3\\ x_1\\ \vdots\\ x_{13} \end{pmatrix}$$

In the end of this phase, G^{-1} will be found and encoded symbol X will be in the correct position aligned with the useful information (in first k rows). In this case the others non-useful information will not be used in the next phase-MM, where they can be eliminated.

3.5.1.1 Constructing INV in CUDA

INV in Parallel

```
$threadX = threaded.X \times threadX\ workload$          //mapping Threads X
$threadY = threaded.Y \times threadY\ workload$          //mapping Threads Y

//GPU kernel
define Swap_Row (G[n][k], X[n][l], $G_i$[n][k], pivot)       //Swap row kernel
    for row = pivot to n-1 do                               //check all n rows
        if (G[pivot][pivot]≠1&& G[row][pivot]==1) then
       //if pivot point is zero while another row had pivot column of 1

            for offset=0 to threadY workload do             //mapping Threads Y
                col=threadY + offset                        //allocating workload
                If col<k then
                    Swap G [row][col] to G [pivot][col]     //Swap between rows
                    Swap $G_i$[row][col] to $G_i$ [pivot][col]
                 End if

                 If col<l then
                    Swap X [row][col] to X [pivot][col]     //Swap between rows

                 End if

             End for
         End if
         BREAK                                              //Break the whole swap
                                                            operation when one row
                                                            swap is done
      End for

 define Pivot_check (G [n][k], $G_i$[n][k], check[n], pivot)  //Check pivot
     If threadX==0 then
         $G_i$[pivot][pivot]= $G_i$[pivot][pivot] $\oplus$1      //inverse the $G_i$ pivoting point

     for offset=0 to thread X workload do                   //one thread will handle
                                                             how many workload/rows
         row= threadX+offset
         if (row < n) then
             check[row]=G[row][pivot]
          End if
      End for
```

```
36.
37.  define XOR (G[n][k], Gi[n][k], check[n], pivot)
38.      for offset=0 to thread X workload do                  //one thread will handle
                                                                 how many workload/rows
39.
40.          row= threadX+offset
41.          If (G [row][pivot]==1 && row ≠ pivot) then
42.              for offsetY=0 to thread Y workload do
43.                  col=threadY + offsetY                      //one thread will handle
                                                               how many workload/columns
44.                  G [row][col]= G [row][col] ⊕ G [pivot][col]  //XOR between rows
45.                  Gi [row][col]= Gi [row][col] ⊕ Gi [pivot][col]//XOR between rows
46.              End for
47.          End if
48.      End for
49.
50.  Host to Device memory copy
51.
52.  For pivot=0 to k-1 do                        //iterates through all k pivot point
53.      Swap_Row <<<blocks, threads >>> (G[n][k], Gi[n][k], X[n][l], pivot)
54.      Pivot_check <<<blocks, threads >>> (G [n][k], Gi[n][k], check[n], pivot)
55.      XOR <<<blocks, threads >>> (G [n][k], Gi[n][k], check[n], pivot)
56.  End for
57.  MM(Gi[n][k], X[n][l], M[n][l])               //Matrix multiplication to retrieve
                                                  message
58.
59.  Host to Device memory copy
60.
```

During The first stage of INV, the coding process is 90% the same with $\mathrm{GE}_{\mathrm{parallel}}$. The different thing in INV and $\mathrm{GE}_{\mathrm{parallel}}$ is, the input data $G|X$ in GE is replaced with G & G_i in INV as shown in line 12, where G_i is originally a zero matrix that will be transformed into the inverse of G after INV. Other than this, minor modifications are done to ensure the useful information is swapped to the right position. Such modification is done on line 12 and 17 to eliminate the unwanted information after INV, so that workload in MM later, can be reduced.

Furthermore, the placing of rows of identity matrix in $\mathrm{G_i}$ is done as in line 28. In this case, rows of Identity matrix will be allocated to each row in each iteration to the useful information position. After this, the G_i which is the inverse of G will be used to perform the MM in the later section. As for such pseudocode, it can be formulated that the operational count of INV can be obtained as shown:

Since INV approximately the same with $\mathrm{GE}_{\mathrm{parallel}}$, equation of $\mathrm{GE}_{\mathrm{parallel}}$ is applicable to INV with modification of $l \rightarrow k$:

$$\mathrm{INV}_{\mathrm{operational\ count}} = k\left(\left\lceil\frac{(n)(k+k)}{\text{threads allocated}}\right\rceil \tau_{\mathrm{XOR}} + p_{INV}\right) \tag{3.11}$$

p_{INV} = for visual purpose, parallel overhead in MM is identified as p_{INV}.

3.5.2 Matrix Multiplication, MM

After all the non-useful information are eliminated, G^{-1} is extracted to perform matrix multiplication with the encoded symbols X. By extracting the value from example 3.5.2.g:

Example 3.5.2.g:

$$G_{3\times3}^{-1} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \quad X_{13\times l} = \begin{pmatrix} x_4 \\ x_2 \\ x_3 \end{pmatrix}$$

$$G_{k\times k}^{-1} \times X_{k\times l} = M_{k\times l}$$

$$M_{k\times l} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} x_4 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_4 \oplus x_2 \\ x_2 \\ x_2 \oplus x_3 \end{pmatrix}$$

The product of $G_{k\times k}^{-1} \times X_{k\times l}$ will become the decoded value.

3.5.2.1 Parallelisation of MM

To parallelise Matrix multiplication is a simple and efficient process, as mentioned in (Lee, et al., 2015), matrix multiplication is a good suite algorithm that fits naturally in GPU parallelisation.

Example 3.5.2.1.a:

$$\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} x_4 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_4 \oplus x_2 \\ x_2 \\ x_2 \oplus x_3 \end{pmatrix}$$

To parallelise the MM above. The MM process will be separated into a series version for better visualisation.

In series (typical CPU implementation) $x_4 \oplus x_2$ will be done first, and then continue with x_2 and lastly $x_2 \oplus x_3$, and there will be these 3 steps for this case. One thing we know here is that all the steps here are independent of each other, which means that all of them can be simultaneously performed together unlike the 3 steps in GE, where third step must wait for the second step to complete. MM will just have to parallel process the 3 MM steps by allocating them with sufficient threads:

Table 3.2: Threads allocation for parallelisation in MM

Threads	Operation	Answer
➔ 1	$1x_4 \oplus 1x_2 \oplus 0x_3$	$x_4 \oplus x_2$
➔ 2	$0x_4 \oplus 1x_2 \oplus 0x_3$	x_2
➔ 3	$0x_4 \oplus 1x_2 \oplus 1x_3$	$x_2 \oplus x_3$

In **Table 3.2**, since 3 threads are allocated to the three operations to perform parallelisation, the original time taken to complete the 3 MM operation will be reduced to one MM operation time if the parallelisation performs ideally.

3.5.2.2 Constructing MM in CUDA

Matrix Multiplication in Parallel

```
1. threadX = threaded.X × threadX workload                //mapping Threads X
2. threadY = threaded.Y × threadY workload                //mapping Threads Y
3.
4. Define MM ($G_i$[n][k], X[k][l], M[k][l])                  //matrix multiplication
                                                           kernel
5.     for offset=0 to thread X workload do
6.         for offset=0 to thread Y workload do
7.             row=threadX+offset                         //one thread will handle
                                                          certain workload/rows
8.             col=thread+offset                          //one thread will handle
                                                          certain workload/columns
9.             if(row<k && col<l) then
10.                for z=0 to k-1 do
11.                    $M[row][col] = X[row][col] \oplus (G_i[row][z] * X[z][col])$  //$G_i \times X = M$
12.                End for
13.            End if
14.        End for
15.    End for
16.
```

After the INV produce the output G_i, it will be brought over to matrix multiply with the X. The output of this will be the decoded message.

As for the operational count for MM according to the pseudocode:

$$\mathrm{MM}_{\text{operational count}} = k(l \times k) \xrightarrow{parallelisation} \left\lceil \frac{l \times k}{\text{threads allocated}} \right\rceil k\tau_{\mathrm{MM}} + p_{MM}$$

$l \times k = \text{line } 5 - 6 \text{ loops}$

$k\tau_{\mathrm{MM}} = \text{line 11 matrix multiplication core operations constant}$, that consist of k numbers of loops.

$$p_{MM} = \text{Parallel overhead in MM is identified as } p_{MM}$$

Hence in general, GEMM:

$$\mathrm{GEMM}_{\text{operational count}} = \mathrm{INV}_{\text{operational count}} + \mathrm{MM}_{operational\ count}$$

$$\mathrm{GEMM}_{\text{operational count}} = \left(k\left(\left\lceil\frac{(n)(k+k)}{\text{threads allocated}}\right\rceil \tau_{\mathrm{XOR}} + p_{INV}\right)\right) + \left(\left\lceil\frac{l\times k}{\text{threads allocated}}\right\rceil k\tau_{\mathrm{MM}} + p_{MM}\right) \quad (3.12)$$

Assuming that if all of them are fully parallelised, where threads allocated is at maximum, there are $(n)(k+k)$ and $l \times k$ numbers of threads respectively for INV and MM.

$$\mathrm{GEMM}_{\text{operational count}} = \left(k(\tau_{\mathrm{XOR}} + p_{INV})\right) + (k\tau_{\mathrm{MM}} + p_{MM}) \quad (3.13)$$

To compare the speed between GEMM and GE, their operational count will be used as the reference:

- $\mathrm{GE}_{\text{operational count}} = k(1\tau_{\mathrm{XOR}} + p_{GE})$
- $\mathrm{GEMM}_{\text{operational count}} = \left(k(\tau_{\mathrm{XOR}} + p_{INV})\right) + (k\tau_{\mathrm{MM}} + p_{MM})$

The parallel overhead increases proportionally to the operating matrix size:
Size of GE is approximately same with MM, where $(n) \times (k+l) \approx k \times l$ assuming l is very large ($l \gg k\ in\ REC\ case$), while INV has the smallest parallel overhead for having smallest operating matrix size of $(n) \times (k+k)$.
filling approximation here will be:

$$p_{MM} \approx p_{GE} \approx \frac{(n)\times(k+l)}{(n)\times(k+k)} p_{INV} \approx \frac{l}{k} p_{INV}$$

Hence:

$$\mathrm{GE}_{\text{operational count}} = k\left(\tau_{\mathrm{XOR}} + \frac{l}{k} p_{\mathrm{INV}}\right)$$

$$\text{GEMM}_{\text{operational count}} = \left(k(\tau_{\text{XOR}} + p_{\text{INV}})\right) + \left(k\tau_{\text{MM}} + \frac{l}{k} p_{\text{INV}}\right)$$

If $l \gg k$:

$$\text{GE}_{\text{operational count}} = \tau_{\text{XOR}} k + l p_{\text{INV}} \approx l p_{\text{INV}} \qquad (3.14)$$

$$\text{GEMM}_{\text{operational count}} = (\tau_{\text{XOR}} + \tau_{\text{MM}})k + \frac{k^2+l}{k} p_{\text{INV}} \approx \frac{l}{k} \text{p}_{\text{INV}} \qquad (3.15)$$

From these equations, it can deduce that if $l >> k$ GEMM will perform faster than GE, which suit the properties of REC where l is usually $100 - 1000$ times bigger than k. Hence in theory, it is proven that GEMM can perform faster decoding speed than $\text{GE}_{\text{parallel}}$.

3.6 Second Degree Parallelisation

Previously, the decoding algorithms were designed based on one assumption where the device only decodes one file at a time; a file that is $k < 512$ in message size. But thousands of packets from distinct sources could be received in bulk, there is an opportunity where it can further exploit the power of parallel GPUs and this will be studied in the following section.

3.6.1 Bulk Decoding

In a high-speed networking link, thousands of packets are being transferred at once. In this case, the traditional way of CPU handling the received data often suffers from congestion issue, resulting in poor performance because the CPU can only handle the received data streams one by one in series.

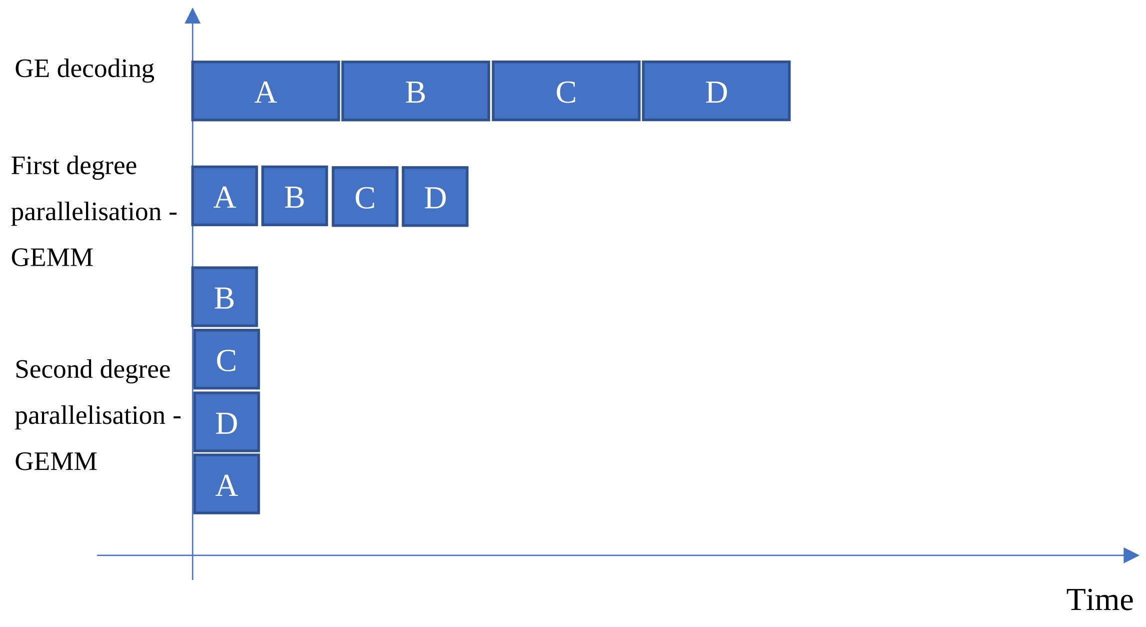

GE decoding
A
B
C
D
First degree parallelisation - GEMM
A
B
C
D
Second degree parallelisation - GEMM
B
C
D
A
Time

Figure 3.8: Timeline of GPU decoding vs the first and second-degree parallelisation.

Figure 3.8 show a new bulk decoding method by handling all the distinct files (A, B, C, D) in parallel. For example, upon receiving a stream of packets that consist of distinct files- A, B, C and D, we propose a second degree of parallelisation on top of GEMM explained previously. Instead of decoding the files using GEMM in ascending order from A to B to C then to D like the traditional decoding method, the receiver will perform GEMM decoding of all the 4 files all at once in parallel. Below are the steps to perform the bulk decoding:

1.) The bulky amount of received packets will be categorized into distinct files and an example is illustrated below:

Example 3.6.1.a:

4 files $(\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D})$, each file contains: $k = 2, l = 1, n = 12$

$$A = \begin{matrix} \overbrace{\begin{matrix}0 & 1\\ 1 & 1\\ & \vdots\\ 1 & 0\end{matrix}}^{k} & \overbrace{\begin{matrix}1\\ 0\\ \\ 1\end{matrix}}^{l}\end{matrix} \quad B = \begin{matrix}1 & 1 & 1\\ 1 & 0 & 0\\ & \vdots & \\ 1 & 1 & 1\end{matrix} \quad C = \begin{matrix}0 & 0 & 0\\ 1 & 0 & 0\\ & \vdots & \\ 1 & 1 & 1\end{matrix} \quad D = \left.\begin{matrix}1 & 1 & 1\\ 1 & 0 & 0\\ & \vdots & \\ 1 & 1 & 1\end{matrix}\right\} n$$

2.) The rearranged files will be augmented as shown in example 4.3.1.b and will be bulk decoded later.

Example 3.6.1.b:

$$\begin{matrix}A\\ B\\ C\\ D\end{matrix} = \left(\begin{array}{c|c} G_A & X_A\\ G_B & X_B\\ G_C & X_C\\ G_D & X_D\end{array}\right) = \left(\begin{array}{cc|c} 0 & 1 & 1\\ 1 & 1 & 0\\ & \vdots & \\ 1 & 0 & 1\\ \hline 1 & 1 & 1\\ 1 & 0 & 0\\ & \vdots & \\ 1 & 1 & 1\\ \hline 0 & 0 & 0\\ 1 & 0 & 0\\ & \vdots & \\ 1 & 1 & 1\\ \hline 1 & 1 & 1\\ 1 & 0 & 0\\ & \vdots & \\ 1 & 1 & 1\end{array}\right)$$

3.) All the G in the distinct file will be inversed into G^{-1} all at once. Three major steps of INV which had already demonstrated in the previous chapter will be applied in all the distinct file in parallel.

Example 3.6.1.c:

$$G|I \xrightarrow[Inverse]{} I|G^{-1}$$

$$\left(\begin{array}{c|c} G_A & I_A \\ G_B & I_B \\ G_C & I_C \\ G_D & I_D \end{array}\right) \xrightarrow[Inverse]{} \left(\begin{array}{c|c} I_A & {G^{-1}}_A \\ I_B & {G^{-1}}_B \\ I_C & {G^{-1}}_C \\ I_D & {G^{-1}}_D \end{array}\right)$$

All the G^{-1} will be used to matrix multiply with their respective X in parallel for the bulk decoding and below is the example.

Example 3.6.1.d:

$$4 \text{ files (A, B, C, D), each file contains: } k = 2, l = 1, n = 12$$

$$G^{-1} \times X = M$$

$$\begin{array}{c} {G^{-1}}_A \\ {G^{-1}}_B \\ {G^{-1}}_C \\ {G^{-1}}_D \end{array} \times \begin{array}{c} X_A \\ X_B \\ X_C \\ X_D \end{array} = \begin{array}{c} M_A \\ M_B \\ M_C \\ M_D \end{array}$$

$$\begin{array}{cc} G_{1A}^{-1} & G_{3A}^{-1} \\ G_{2A}^{-1} & G_{4A}^{-1} \\ G_{1B}^{-1} & G_{3B}^{-1} \\ G_{2B}^{-1} & G_{4B}^{-1} \\ G_{1C}^{-1} & G_{3C}^{-1} \\ G_{2C}^{-1} & G_{4C}^{-1} \\ G_{1D}^{-1} & G_{3D}^{-1} \\ G_{2D}^{-1} & G_{4D}^{-1} \end{array} \times \begin{array}{c} X_{1A} \\ X_{2A} \\ X_{1B} \\ X_{2B} \\ X_{1C} \\ X_{2C} \\ X_{1D} \\ X_{2D} \end{array} = \begin{array}{c} G_{1A}^{-1}.X_{1A} \oplus G_{3A}^{-1}.X_{2A} \\ G_{2A}^{-1}.X_{1A} \oplus G_{4A}^{-1}.X_{2A} \\ G_{1B}^{-1}.X_{1B} \oplus G_{3B}^{-1}.X_{2B} \\ G_{2B}^{-1}.X_{1B} \oplus G_{4B}^{-1}.X_{2B} \\ G_{1C}^{-1}.X_{1C} \oplus G_{3C}^{-1}.X_{2C} \\ G_{2C}^{-1}.X_{1C} \oplus G_{4C}^{-1}.X_{2C} \\ G_{1D}^{-1}.X_{1D} \oplus G_{3D}^{-1}.X_{2D} \\ G_{2D}^{-1}.X_{1D} \oplus G_{4D}^{-1}.X_{2D} \end{array} = \begin{array}{c} M_A \\ M_B \\ M_C \\ M_D \end{array}$$

And the assignment of workload to the threads is shown in Table 3.3 by properly mapping the threads into different MM operations for bulk decoding to be performed in parallel.

Table 3.3: Threads allocation for parallelisation in MM

Threads	Operation
➔ 1	$G_{1A}^{-1}.X_{1A} \oplus G_{3A}^{-1}.X_{2A}$
➔ 2	$G_{2A}^{-1}.X_{1A} \oplus G_{4A}^{-1}.X_{2A}$
➔ 3	$G_{1B}^{-1}.X_{1B} \oplus G_{3B}^{-1}.X_{2B}$
➔ 4	$G_{2B}^{-1}.X_{1B} \oplus G_{4B}^{-1}.X_{2B}$
➔ 5	$G_{1C}^{-1}.X_{1C} \oplus G_{3C}^{-1}.X_{2C}$
➔ 6	$G_{2C}^{-1}.X_{1C} \oplus G_{4C}^{-1}.X_{2C}$
➔ 7	$G_{1D}^{-1}.X_{1D} \oplus G_{3D}^{-1}.X_{2D}$
➔ 8	$G_{2D}^{-1}.X_{1D} \oplus G_{4D}^{-1}.X_{2D}$

CHAPTER 4

RESULT AND DISCUSSION

4.1 Experiment Platforms

Both the proposed GEMM and GE are implemented on the GPU using CUDA and these two algorithms are used to compare with the base case (an optimised and vectorised GE according to one that is used from the paper (Hu, et al., 2012). All our experimental CUDA code and serial C++ code are available in the appendix.

The serial and parallel version of GE implementation used the code given in the paper of Raptor GF(2). Such GE has been previously employed in paper (Hu, et al., 2012) to prove the applicability of GPU on decoding and found to be more effective than CPU decoding. However, as stated previously, there will be parallel overhead in GPU that prevents the GE parallelisation from getting linear speedup. Hence in this chapter, the algorithms will be tested from different aspects to compare their decoding speeds.

For our experimental platforms, the GPU code ran on a Nvidia Quadro K620 with CUDA 8.0 and a workstation XEON E3 CPU. The Nvidia Quadro K620 has 2GB GDDRAM3, a core clock of 1000MHz, and CPU-to-GPU bandwidth of 29 GB/s. It consists of 3 SMs, each containing 128 cores, making up a total of 384 cores. For comparison, the base case is the serial execution of the decoding code on XEON E3 4 GHz with 16 GB RAM.

4.2 Overhead test in GE and GEMM

As a solution to solve the decoding speed issue, GEMM, should perform faster than GE without increasing the overheads. Also, data received that is decodable using GE will be decodable using GEMM. Those packets that are non-decodable in GE will not be decodable using GEMM.

It had been mentioned that the Random code requires $n = k + 10$ amount of received packets for 99.99% of PCD using GE (Chong, et al., 2015), hence in GEMM, it is also required to have $n = k + 10$ amount of received packets to achieve 99.99% PCD. In this experiment, 10000 samples of distinct packets are used for decoding by GE and GEMM where:

$$n = k + \varepsilon, k = 32, l = 8192, \varepsilon \leq 10$$

Table 4.1: Overall PCD of GEMM and GE

Number of packets received, n	PCD of GE	PCD of GEMM
k	26.76%	26.76%
$k + 1$	57.78%	57.78%
$k + 2$	79.21%	79.21%
$k + 3$	89.02%	89.02%
$k + 4$	94.88%	94.88%
$k + 5$	96.71%	96.71%
$k + 6$	98.45%	98.45%
$k + 7$	99.22%	99.22%
$k + 8$	99.45%	99.45%
$k + 9$	99.81%	99.81%
$k + 10$	99.99%	99.99%

It is shown in **Table 4.1** that the PCD of GE and GEMM are the same, will not even the slightest change in PCD found in the comparison between the GE and GEMM. Furthermore, at $k + 10$ case, when both GE and GEMM are decoding the same 10000 packets, both algorithms are unable to decode sample no. 4831, which also means that overheads of GE and GEMM are the same.

4.3 First Degree Parallelisation - $\mathbf{GE_{parallel}}$ V.S. GEMM

To compare the parallel performance of GE and GEMM, the pseudocode used is available in the paper (Hu, et al., 2012) as the base case for GE. In addition, the same experimental model will be reworked and made comparison between:

- $\mathbf{GE_{serial}}$ using XEON E3 Workstation CPU (base case)
- $\mathbf{GE}_{\text{parallel}}$ using Quadro K620 GPU
- $\mathbf{GEMM}$ using Quadro K620 GPU

4.3.1 k Variation

In this section, the effects of complexity of GE will be shown, especially when the file size k gets bigger. In this case, the speedup is calculated based on:

$$\text{Speedup of } \mathrm{GE_{paralllel}} = \frac{\mathrm{GE_{serial}}}{\mathrm{GE_{parallel}}} \qquad (4.1)$$

$$\text{Speedup of GEMM} = \frac{\mathrm{GE_{serial}}}{\mathrm{GEMM}} \qquad (4.2)$$

Figure 4.1 demonstrates the time complexity (graph of $O(k^3)$ with respect to the size k for the base case GE:

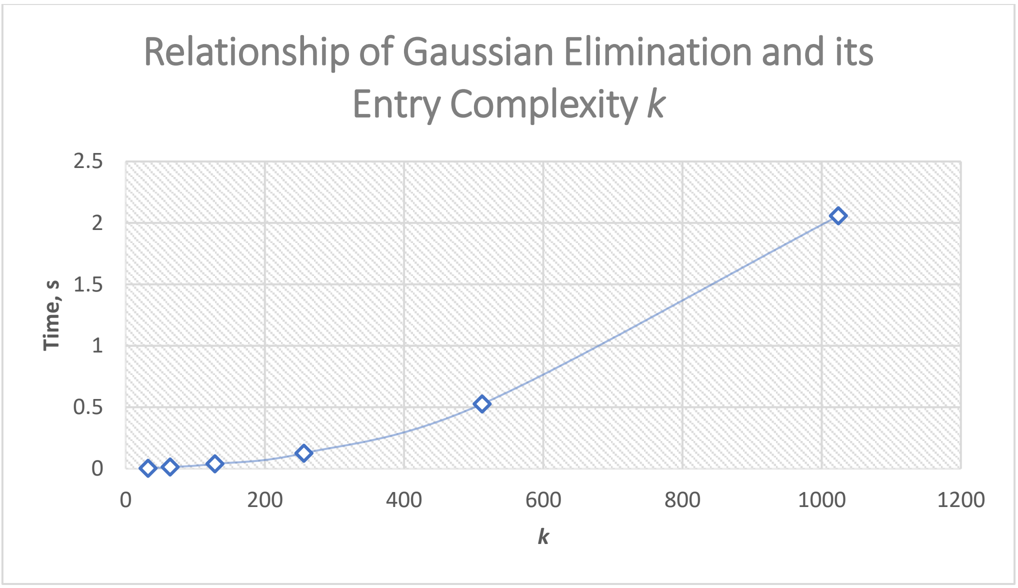


Figure 4.1: Complexity of GE in Time used to solve GE vs. k.

$$32 < k \leq 1024, \qquad l = 8192, \qquad threads\ allocated = 100000$$

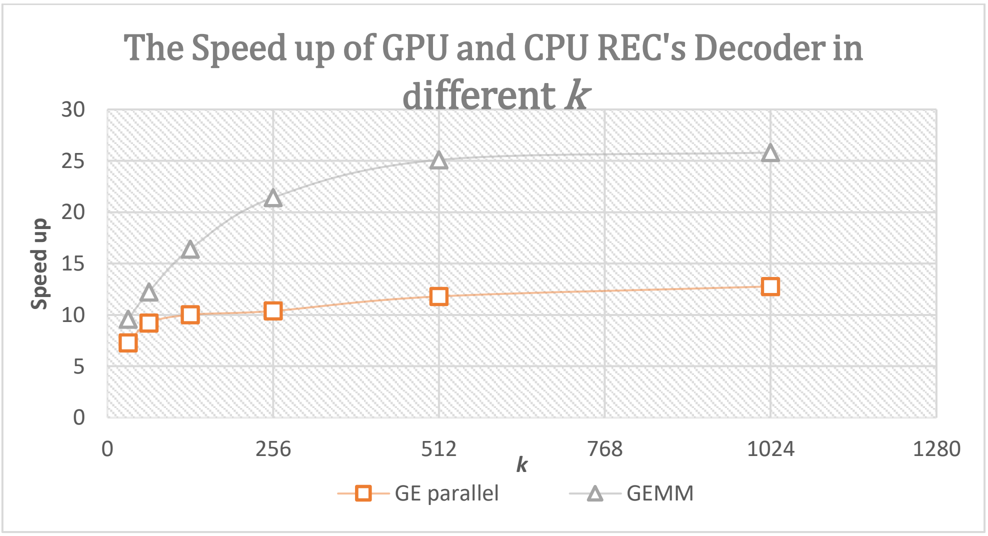


Figure 4.2: Speedup of $\text{GE}_{\text{parallel}}$ and GEMM after parallelisation with $\text{GE}_{\text{serial}}$ as the base case.

From the graph in Figure 4.2, $\mathrm{GE}_{\mathrm{parallel}}$ outperforms $\mathrm{GE}_{\mathrm{serial}}$ in terms of decoding time, whereby the speed up of $\mathrm{GE}_{\mathrm{parallel}}$ is significantly higher when k increases. Furthermore, the algorithm GEMM that this research emphasized have a significantly better speedup compared to $\mathrm{GE}_{\mathrm{parallel}}$, which makes GEMM the fastest and most efficient parallelised REC decoder.

The parallelised algorithm GEMM and $\mathrm{GE}_{\mathrm{parallel}}$ will eventually reach a speedup of 26 and 13 respectively. From this experiment, the GPU is better in handling larger size workload as the speed up can reach higher as k grows bigger.

4.3.2 l Variation

In chapter 2.6, the l is considered a very important element that affects the decoding speed of GE, in this section variations of l will be tested:

- $\mathbf{GE}_{\mathbf{serial}}$ using XEON E3 Workstation CPU
- $\mathbf{GE}_{\mathrm{parallel}}$ using Quadro K620 GPU
- **GEMM** using Quadro K620 GPU

The result in Figure 4.4 shows that GEMM can perform better parallelisation as l increase. As l is getting larger than k the speedup of GEMM is better than GE; this proves that the equation forms in section 3.5 is practically true.

$$k = 256, \qquad l = \{256, 8192, 73728\}, \qquad threads\ allocated = 100000$$

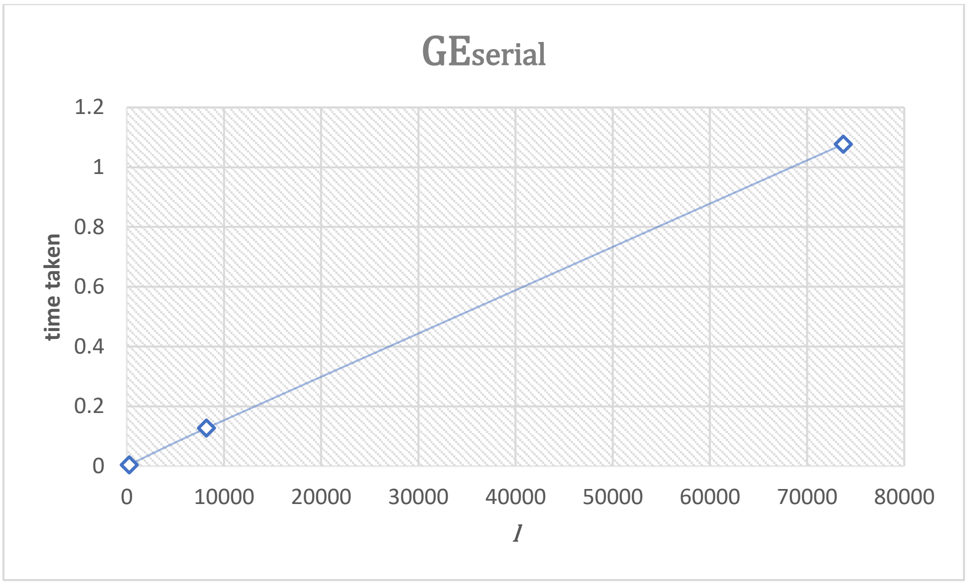


Figure 4.3: Complexity of GE in Time used to solve GE vs. the variation in l.

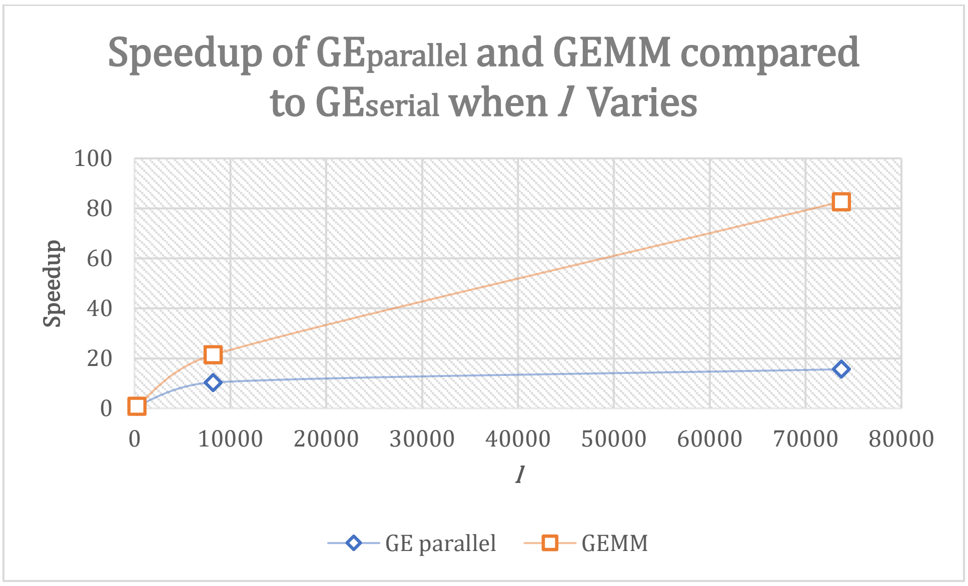


Figure 4.4: Speedup of $\text{GE}_{\text{parallel}}$ and GEMM after parallelisation with $\text{GE}_{\text{serial}}$ as the base case in the variation of l.

4.3.3 Threads allocation Variation

From the previous result (k and l variation), the GPU can decode better than CPU, where $\mathrm{GE}_{\mathrm{parallel}}$ and GEMM outperform the $\mathrm{GE}_{\mathrm{serial}}$ experimentally.

In this section variations of threads allocation will be tested in:

- **$\mathrm{GE}_{\mathrm{parallel}}$** uses Quadro K620 GPU (base case in this section: implemented using one thread one block for the execution, at $k = 256, l = 8192$ and the time to execute the base case here is 1.472s. Other than that, the packets received n=k+10, which is according to the random code standard.)
- **GEMM** uses Quadro K620 GPU

$$k = 256, \qquad l = 8192, \quad 0 < threads\ allocated < 100000, \qquad n = k + 10$$

$$\text{Speedup of } \mathrm{GE}_{\mathrm{parallel}} = \frac{\mathrm{GE}_{\mathrm{parallel}}(\text{in 1 thread})}{\mathrm{GE}_{\mathrm{parallel}}\ (\text{multiple threads})} \tag{4.3}$$

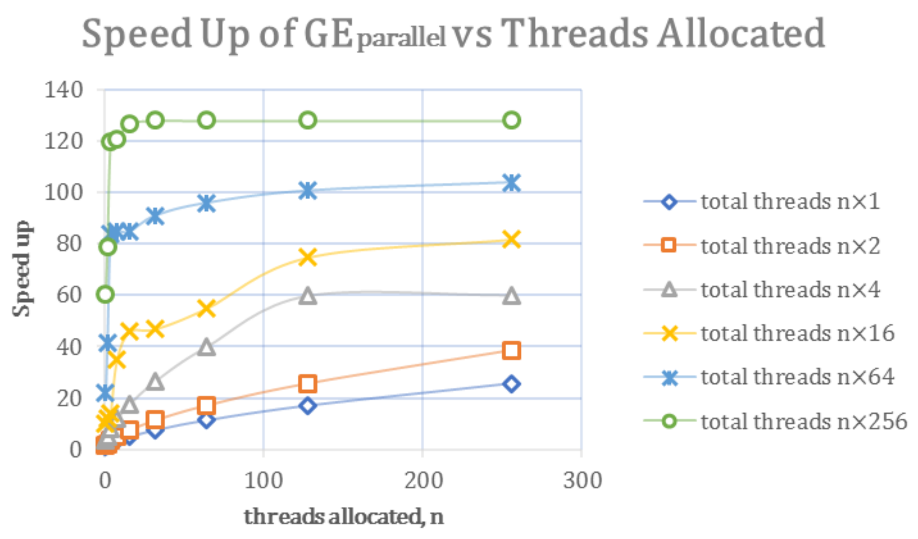


Figure 4.5 Sensitivity of threads allocated toward the speedup of $\mathrm{GE}_{\mathrm{parallel}}$.

Figure 4.5 shows that the speed up of $\mathrm{GE}_{\mathrm{parallel}}$ using multiple threads as compared to $\mathrm{GE}_{\mathrm{parallel}}$ using one thread (base case) reaches a saturation speedup of 129 times when 8×256 threads are allocated for $\mathrm{GE}_{\mathrm{parallel}}$.

As discussed earlier in chapter 4, the GEMM is separated into 2 parts:

1. Inverse (INV) - A variant of GE that is used for inversion of matrix in Chapter 3.5.1.
2. Matrix Multiplication (MM) - Refer to Chapter 3.5.2.

In this section, GEMM (INV+MM) will be analysed on how it can perform faster than GE. First, we will look into the sensitivity of threads allocation towards the speedup of INV:

$$\text{Speedup of } \mathrm{INV}_{\mathrm{parallel}} = \frac{\mathrm{GE}_{\mathrm{parallel}}(\text{in 1 thread})}{\mathrm{INV}_{\mathrm{parallel}}\ (\text{multiple threads})} \qquad (4.4)$$

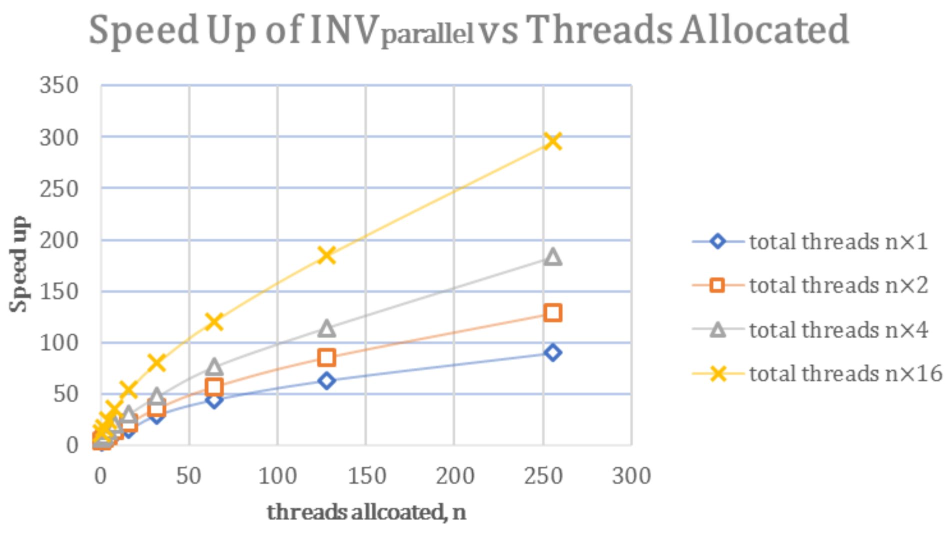


Figure 4.6: Sensitivity of threads allocated toward the speedup of $\mathrm{INV}_{\mathrm{parallel}}$.

Figure 4.6 demonstrates the speed up of INV when different number of threads are allocated for parallelisation. When it is compared to the speed of $\mathrm{GE}_{\mathrm{parallel}}$, which is the base case where $\mathrm{GE}_{\mathrm{parallel}}$ uses only one thread, INV can reach a speedup of approximately 300 times at the maximum performance of the GPU, where 256×16 threads are allocated for $\mathrm{INV}_{\mathrm{parallel}}$.

$$\text{Speedup of } \mathrm{MM}_{\mathrm{parallel}} = \frac{\mathrm{GE}_{\mathrm{parallel}}(\text{in 1 thread})}{\mathrm{MM}_{\mathrm{parallel}}\ (\text{multiple threads})} \tag{4.5}$$

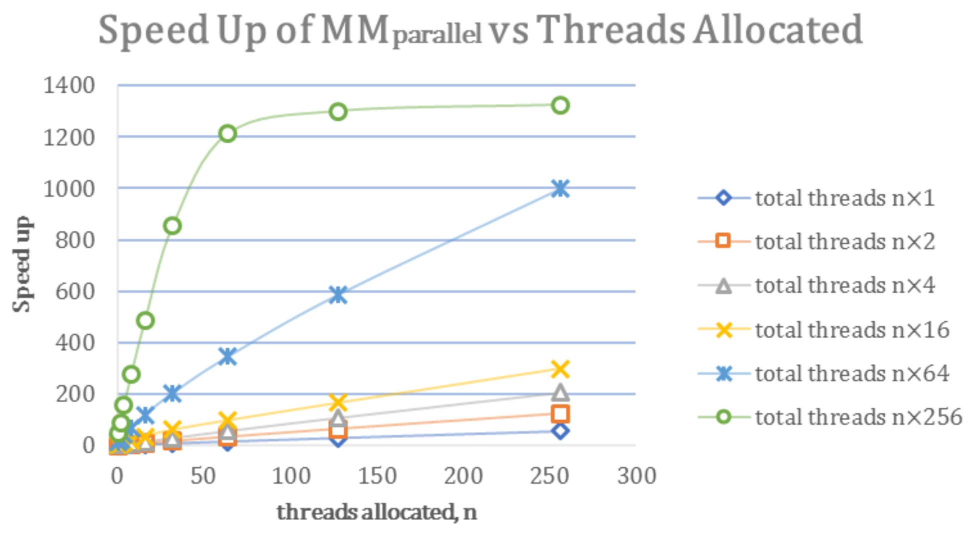


Figure 4.7: Sensitivity of threads allocated toward the speedup of $\mathrm{MM}_{\mathrm{parallel}}$.

Figure 4.7 shows that the performance sensitivity of allocating more threads for MM as compared to the $\mathrm{GE}_{\mathrm{parallel}}$. The saturation point of MM speedup can be seen when 64×256 threads are allocated for parallelisation; the speedup of MM as compared to the $\mathrm{GE}_{\mathrm{parallel}}$ can reach a factor of 1300 times faster.

However, for all the previous results, we realise that the speedup achieved in these experiments can be further improved; for instance there are only 10 times speedup instead of 100 times speedup for 100 threads used in parallelisation, e.g. parallelisation of GE from Figure 4.7 can only reach the speed up of 130x by providing it with maximum GPU resources (few thousands of threads allocated), whereas Figure 4.6 demonstrated the maximum INV speed up at 300x. Furthermore, the speedup of MM in Figure 5.7 is able to reach the speedup of 1300x as compared to the $\mathrm{GE}_{\mathrm{parallel}}$.

Such inefficiency in speedup proves the existence of parallel overheads, p which is discussed earlier in Chapter 3.

4.4 Analysis and Discussion

As mentioned in Chapter 3, the parallel overheads is the reason that concludes the major cause of the parallelisation inefficiency (CUDA, 2015) in GE or GEMM, it includes:

- Matrix size
- Algorithm nature

When it comes to parallelisation, size and the algorithm nature matters the most. The first issue to tackle is the input matrix size for GE decoding.

From Figure 4.5 and Figure 4.7, the saturation curves indicate the hardware limitation of GPU has been reached; there will be no further speedup even though more threads are added to the execution of the algorithm. Hence the bigger the workload size the easier the limitation is hit, where the speed up of parallel processing will saturate eventually.

Other than GPU's hardware limitation, the threads allocation delay will also be affected by the input matrix size. In GPU's architecture, threads are required to be scheduled before parallelisation start and this will be the threads allocation stage; for example:

Example 4.4.a:

If there are A, B and C workloads to be executed, CPU will serially execute from A to C one by one.

CPU execution:

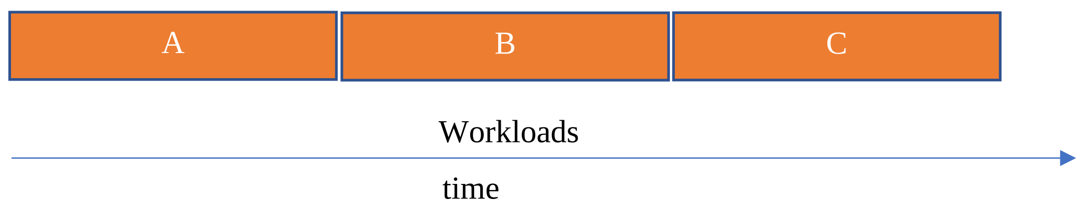


Figure 4.8: The CPU execution timeline

In GPU workload A, B and C will be parallel process and execute simultaneously.

GPU execution (ideal):

GPU execution(practical):

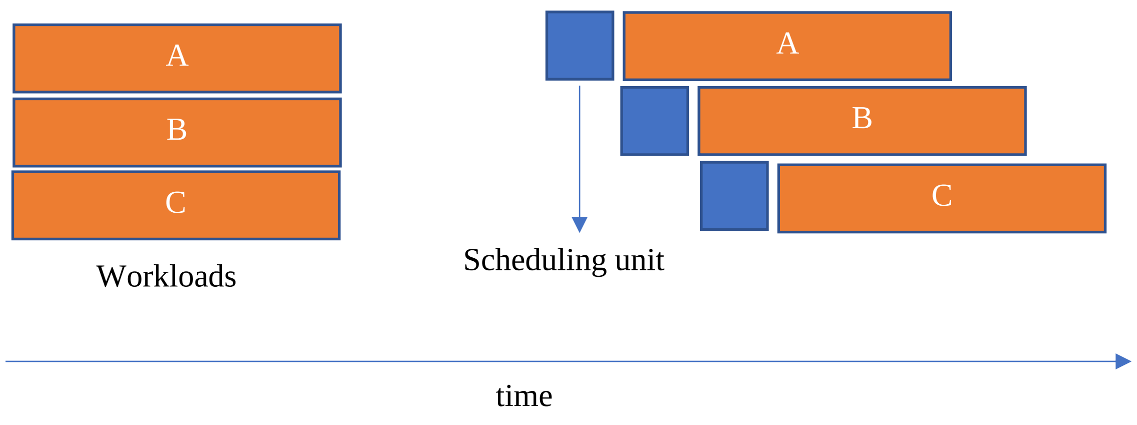


Figure 4.9: The GPU execution timeline

Ideally in Figure 4.9, GPU should execute all the workload at once. However, in practice, threads need to be scheduled and distributed before performing their specific tasks (in this case tasks represented by A, B and C will all work on same operation), e.g., a thread is needed to be tagged along with a thread identity {0, 1, 2, 3….} and allocated to the right position before parallelisation. Once the first threads are scheduled (scheduled duration are represented by blue box in example 4.4.a), the workload will be executed immediately as shown example 4.4.a. Using the Nvidia profiler of CUDA, the details of parallel process of GE and GEMM timeline can be seen clearly.

CUDA NVIDIA profiler result for GE_{parallel}:

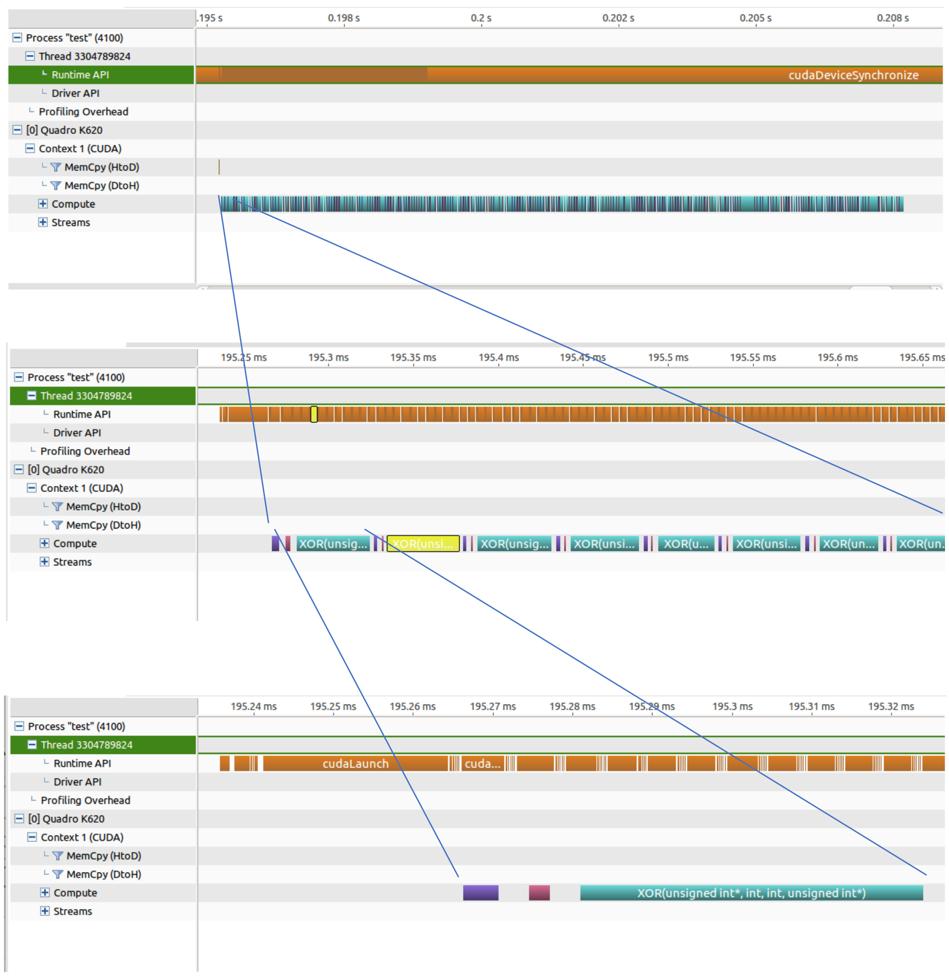


Figure 4.10: Structure of GE_{parallel} in Nvidia Profiler

The brown row in Figure 4.10 indicates the flow of Runtime API, columns of purple represents the swap row operations duration while pink column represents the pivoting check duration and finally the blue column indicates the XOR operations duration, which means all the scheduling flow and threads allocation, as well as call kernel duration mentioned earlier, are visualised.

From the result of profiling of the GE, XOR operation (blue columns) are the dominating one that occupied more than 80% even after parallelisation. Furthermore, the call kernel time occupied approximately 10% of the execution duration. In other words, to optimise a parallelised algorithm, one of the key element is to reduce the amount of calling kernel. However, the iterative nature in GE requires the execution to keep invoking the kernels at least k amount of times to complete the decoding and such accumulated delay of calling kernel will drastically reduce the speed of executing GE in parallel.

GEMM:

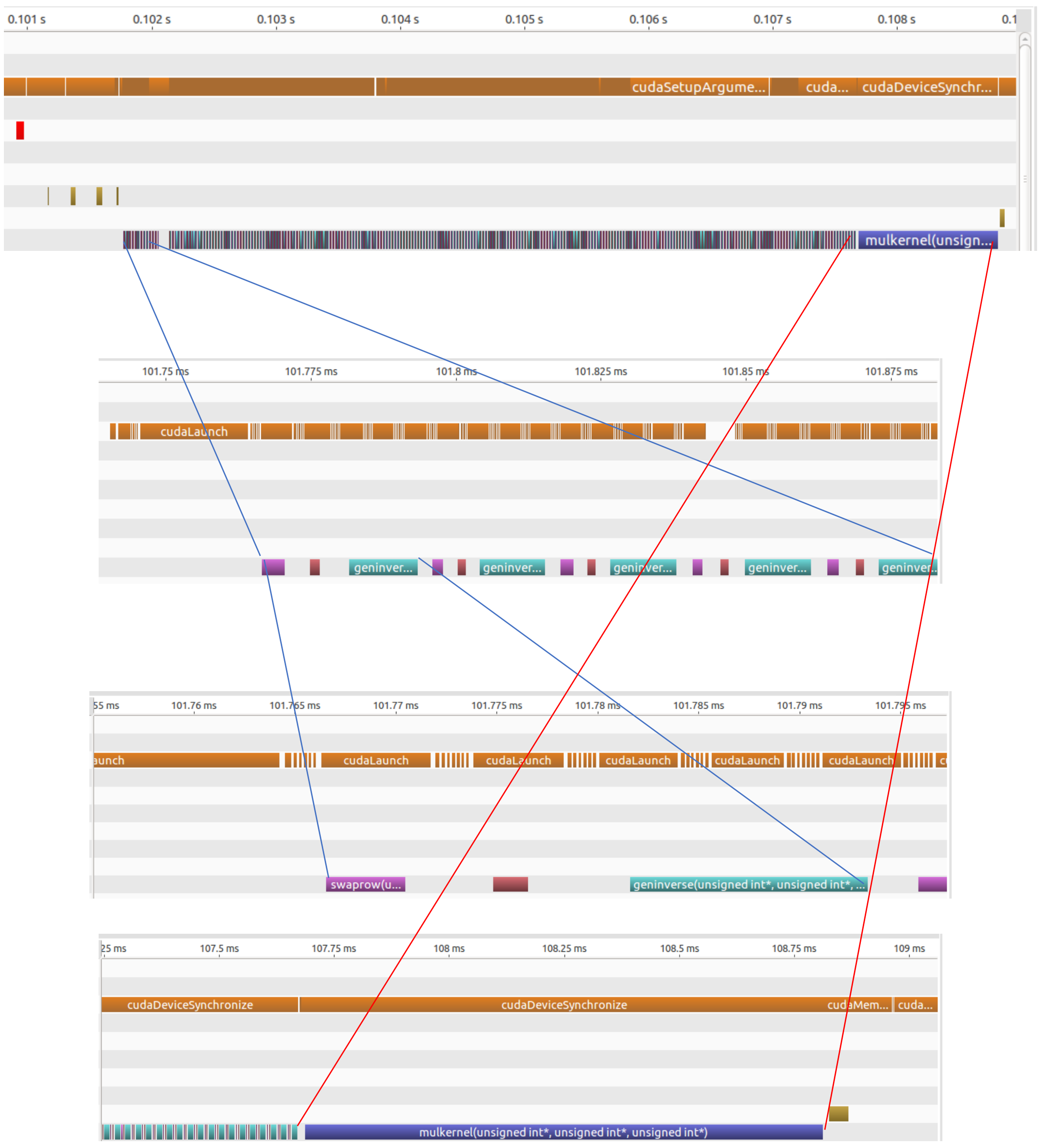


Figure 4.11: Structure of GEMM in Nvidia Profiler

GEMM consist of INV and MM and Figure 4.11 shows that the calling kernel time of the MM algorithm is much less as compared to $GE_{parallel}$.

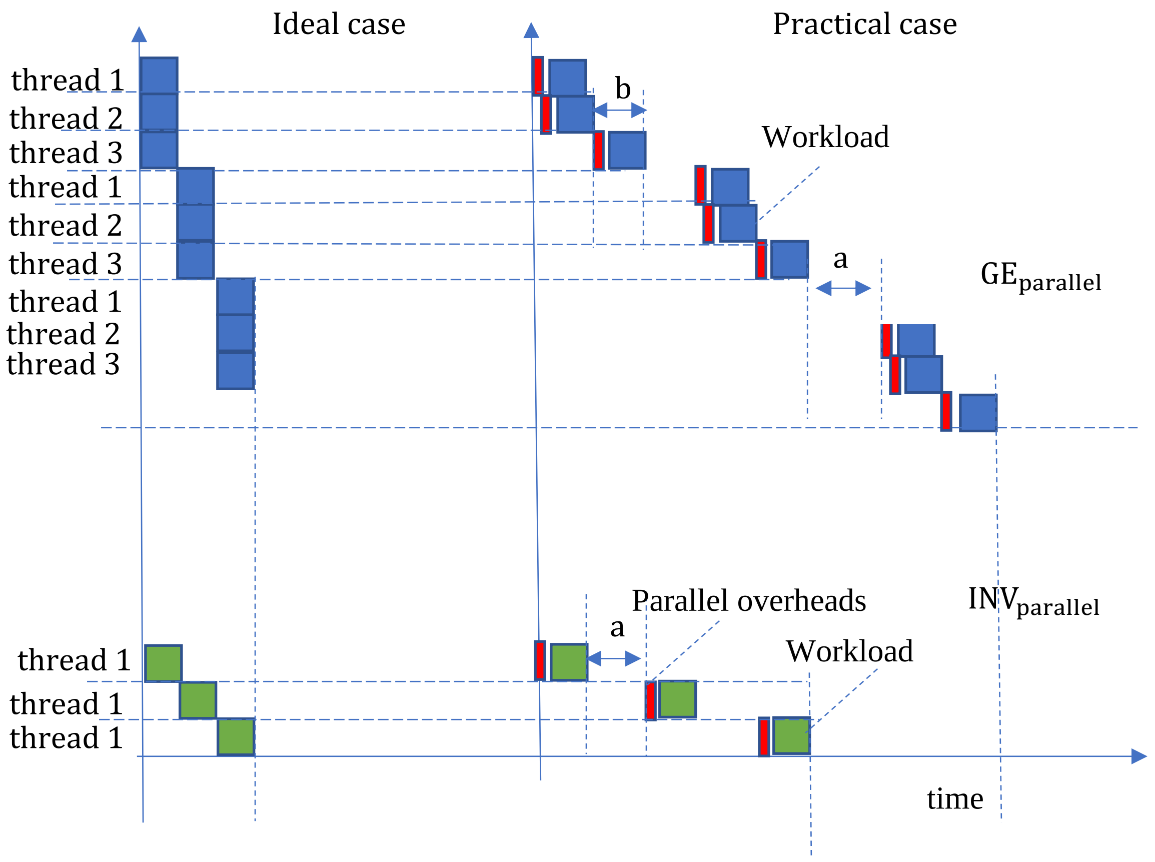


Figure 4.12: Breakdown of the profiled INV$_{\text{parallel}}$ & GE$_{\text{parallel}}$ in summary.

Figure 4.12 shows the summary of the profiling result of GE$_{\text{parallel}}$ and INV$_{\text{parallel}}$, when they are compared under ideal parallel condition where there are no parallel overheads such as the threads allocation delay, calling kernel delay, etc. They yield the same execution speed, but practically GE$_{\text{parallel}}$ that had a bigger matrix size of $n \times (k + l)$, will be executed slower than INV$_{\text{parallel}}$ that had a smaller matrix size of $n \times (k + k)$, and the block in red colour indicates the initialisation of GPU before parallelisation start. In this thesis, this thing will be considered under the coverage of parallel overheads.

Such an idea can be easily visualised. Larger matrix size has a larger workload to deal with and that means more memory access time, more loops and more threads are needed to be allocated to perform parallelisation. However, due to the limited resources in GPU, larger matrix will hit its performing saturation point earlier as compared to a smaller matrix as shown in “b” of Figure 4.12, while “a” indicates the calling kernel time. Since both $GE_{parallel}$ and $INV_{parallel}$ suffer from the same calling kernel time and also same number of iterations, as explained in Chapter 3.2, the main reason why $GE_{parallel}$ is slower than $INV_{parallel}$ is due to the limited resources in GPU in handling a larger matrix.

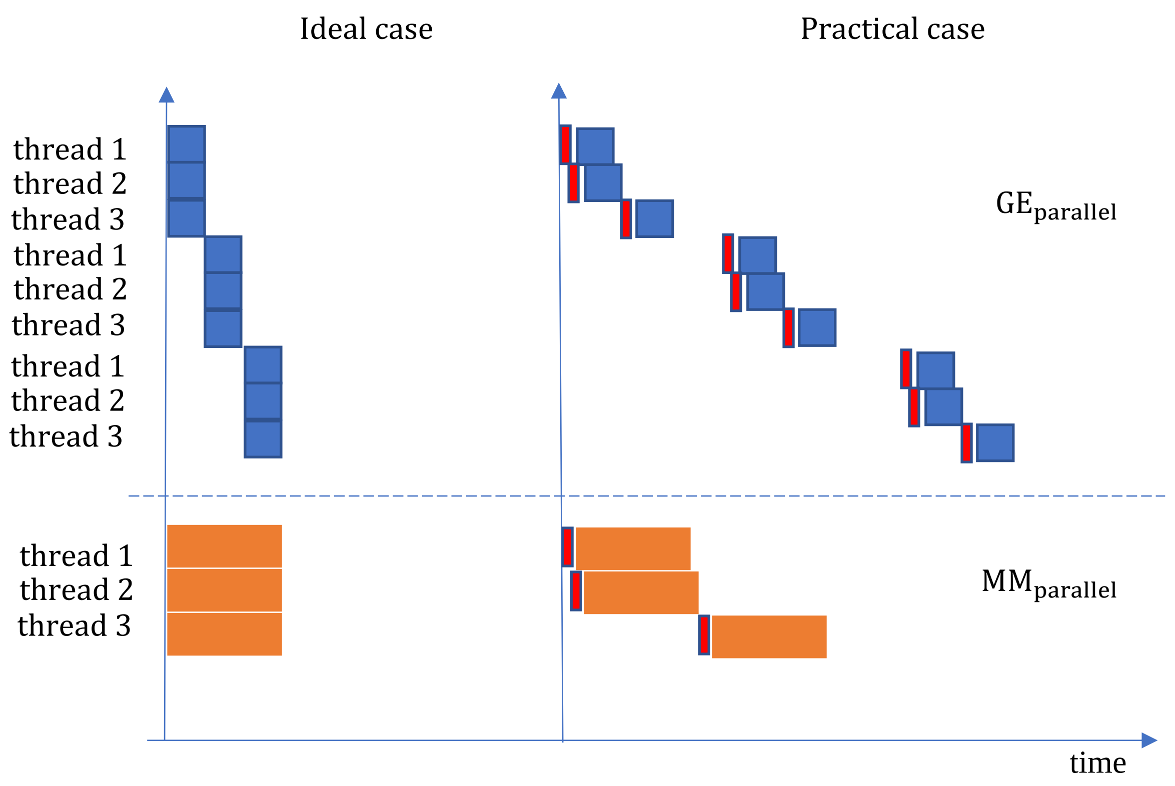


Figure 4.13: Breakdown of the profiled $MM_{parallel}$ & $GE_{parallel}$ in summary.

Figure 4.13 is the comparison of $GE_{parallel}$ and $MM_{parallel}$ in Figure 4.11, it can be seen that the structure of executing MM and GE is different; $GE_{parallel}$ has 3 iterations with each iteration having 3 tasks to be done in parallel using 3 threads, while $MM_{parallel}$ has only one kernel that call 3 tasks to be done in parallel using 3 threads, i.e., each thread will handle relatively more workload as compared to the threads in $GE_{parallel}$.

Ideally $\mathrm{GE}_{\mathrm{parallel}}$ and $\mathrm{MM}_{\mathrm{parallel}}$should perform at the same execution speed, but in practice, when parallel overheads are considered, $\mathrm{GE}_{\mathrm{parallel}}$ that appears to be the iterative algorithm in nature will perform parallelisation slower than $\mathrm{MM}_{\mathrm{parallel}}$. According to the Figure 4.13, MM only suffers from insufficient GPU resource, while GE suffers from insufficient GPU resources and the calling kernel time that is accumulated throughout the iterations.

4.4.1 GEMM vs $\mathrm{GE}_{\mathrm{parallel}}$

In terms of size, INV is working on the smaller matrix size with a size of $k \times 2k$ as MM works on $k \times l$ matrix size and GE works on $k \times (k + l)$ matrix size. It shows that the parallel overhead also increases with the matrix size where $\mathrm{GE}_{\mathrm{parallel}} > \mathrm{MM} > \mathrm{INV}$, and it is proven in the experiment that the workload size can inhibit the performance of GPU speedup. e.g. GE and INV are executed using the same method with different matrix size and slight modification. Figure 4.5 shows that the larger size GE hits the speedup limitation faster at 130x the speed of its base case. While INV hit the speedup limitation at 300x the speed of its base case in Figure 4.6.

In terms of algorithm nature, INV and GE are both iterative structures, making the parallelisation less efficient for the accumulated delay shown in Figure 4.12; whereas MM in the profiler of Figure 4.13 shows a parallelisation friendly structure that can contribute to a higher parallel efficiency, e.g., speedup of MM in Figure 4.7

can reach 1300x as compared to INV and GE, as shown in Figure 4.6 and Figure 4.5, which only reach a maximum speedup of 300 and 127, respectively.

4.5 Second Degree Parallelisation - Single File Parallelisation vs Bulk Decoding Parallelisation

IoT is the trend of the future network and majority of the uplink traffic are short messages if the experiment deploys the GEMM in the context of IoT. We are assuming a server to process thousands of short messages that are encoded with REC in real life applications, since the IoT server in nature are getting feedbacks from thousands of internet devices.

Beside accelerate GE with GPU in the previous cases, the throughput will be further accelerated by having multiple decoding session running concurrently on different short messages. Such multitude of acceleration further leverages the speed up to the factor of 60× at the end of this experiment.

4.5.1 Experimental Result

Two set of machines are tested in the experiment:

- **Bulk decoding $\mathbf{GE}_{\text{serial}}$** using XEON E3 CPU
- **Bulk decoding $\mathbf{GE}_{\text{parallel}}$** using Quadro K620 GPU
- **Bulk decoding GEMM** using Quadro K620 GPU

Generally, the bulk decoding of α = 1024 unique files in the experiment with each message size $k = 32, l = 8192$ bits. Theoretically, the REC (Random code in this study) will encode the message into a potentially infinite number of packets in the form of G|X. The decoder will initiate the decoding algorithm, i.e., GEMM once k + 10 = 42 packets are received for complete decoding. While for bulk decoding GEMM, when there is α number of files to be decoded, $\alpha \times (k + 10)$ relevant packets are required to be received to initiate the bulk decoding of GEMM, as to facilitate the comparison, the speedup is measured using:

$$\text{Speedup of bulk decoding} = \frac{\alpha \text{ unique message base case decoding time}}{\alpha \text{ unique messages decoding times}} \quad (4.6)$$

The base case here is referring to the duration of CPU $\text{GE}_{\text{serial}}$ that performs execution of $k = 32$ and $l = 8192\ bits$ file, which is 0.0049s when decoding $\alpha = 1$ files.

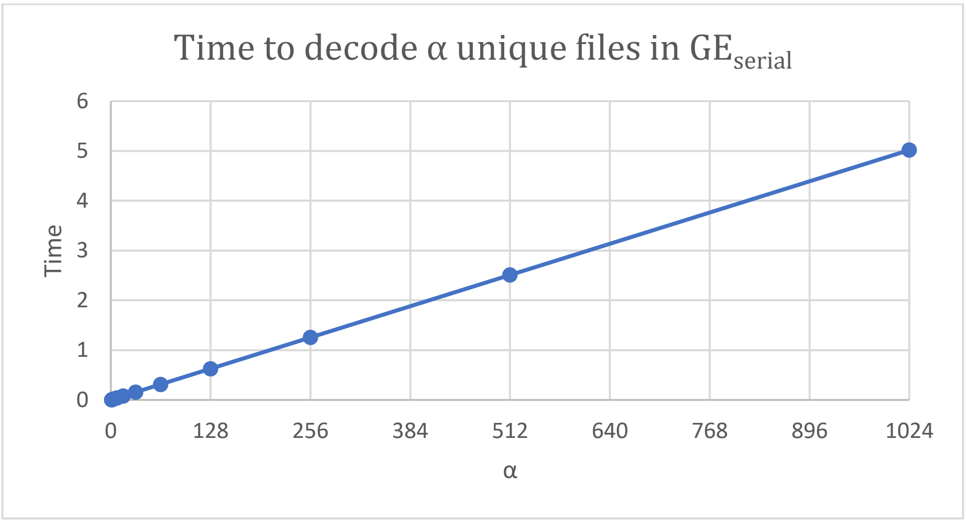


Figure 4.14: Time used to decode α files using $\text{GE}_{\text{serial}}$.

Experiment parameters:

$$k = 32, \quad l = 8192, \quad \mathbf{1} < \boldsymbol{\alpha} < \mathbf{1024}, \quad threads\ allocated = 1{,}000{,}000$$

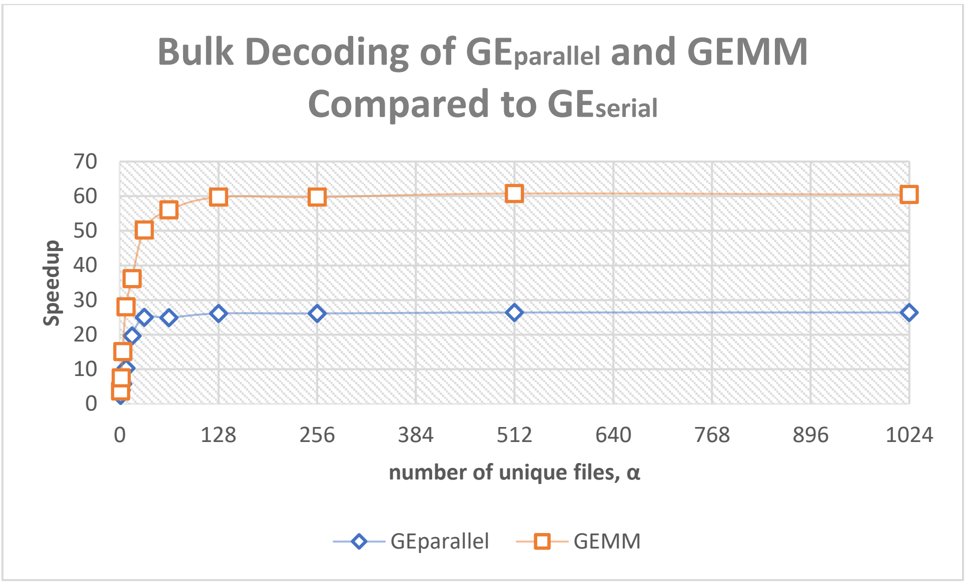


Figure 4.15: Speedup of Bulk decoding $GE_{parallel}$ and $GEMM$ as compared to GE_{serial}.

In the configuration, the large value of allocated threads is not the actual value of threads that will be used in parallelisation, e.g., if there are 2 tasks for parallelisation, only maximum 2 threads will be used and the excessive threads will be eliminated by the CUDA system automatically.

Figure 4.13 shows the speedup of GEMM on bulk decoding over various number of files, α in parallel on different CPUs and GPUs. The best GE computational times on Intel Xeon E3 for each value of α are taken as the base cases. The results show that when the volume of unique messages is small ($\alpha < 32$), both GE and GEMM experience limited speedup. However, GEMM manifests the benefits of higher parallelism degree reaches the speedup of a factor 60 at $\alpha = 1024$.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

While the trend of the future network and majority of the networking traffic are short messages, REC is employed to improve reliable transmission and the decoding algorithm of REC – GE, plays an important role in ensuring the high probability of complete decoding. Hence, the component that contributes to a high computational time of GE and accelerates the algorithm using the state of art GPU will be analysed. However, due to the interactive properties in GE that scale inefficiently in GPU that performs 13 times faster than the $\mathrm{GE}_{\mathrm{serial}}$, GEMM is proposed to further minimise the computational time with the integration of parallel matrix multiplication. As a result, GEMM is capable of performing at approximately 26 times faster than $\mathrm{GE}_{\mathrm{serial}}$.

As to further accelerate the decoding speed of GE and GEMM using GPU, multiple files will be executed in bulk by using GE and GEMM decoding method concurrently to solve thousands of messages and able to achieve the speedup in the factor of 28 and 60 for speedup of bulk decoding $\mathrm{GE}_{\mathrm{parallel}}$ & GEMM respectively compared to bulk decoding of $\mathrm{GE}_{\mathrm{serial}}$.

Table 5.1: Overall speedup performance of GEMM and $\mathrm{GE}_{\mathrm{parallel}}$

Speedup=$\frac{\mathrm{GE}_{\mathrm{serial}}}{\text{Tested algorithm}}$ / Tested algorithm	first degree parallelisation speedup	second degree parallelisation speedup (bulk decoding)
$\mathrm{GE}_{\mathrm{serial}}$	1	1
$\mathrm{GE}_{\mathrm{parallel}}$	13	28
GEMM	26	60

Finally, the results have achieved all the objectives. First, GE as our main study object is proven to be the most efficient linear code solver in terms of PCD. We conclude this from several papers such as (Bioglio, et al., 2009), (Yeqing, et al., 2013) etc., where these papers implements the hybrid of GE and another method to improve the PCD in solving the linear code. Secondly, we are able to propose a new parallel decoding algorithm- GEMM, by making use of the state-of-art GPU. Furthermore, the algorithm undergoes test experimentally to figure out its capability toward the current standard protocols as well as potential future implementations. In the end of the experiment, GEMM is proven scalable towards the given resources such that, GEMM performance can be further improved. When more GPU resources are provided, more information can be decoded in parallel at once.

5.1 Future Work

In this section, the future works to further enhance on the proposed algorithm will be briefly discussed.

5.1.1 Hardware Variety

Previously, GEMM is proven to be a faster decoding algorithm for REC in parallelisation. Due to resources problem, we are only able to test it on GPU K620 Quadro GPU (low end GPU) and a CPU workstation XEON E3 (High end CPU), hence in future we will suggest to have a range of GPU and CPU to be tested on.

5.1.2 Machine Learning

Machine learning is the computational task that processes a bunch of data to discover a pattern that can be used to predict or categorise the new incoming data.

Lately, machine learning appears to be the attractive topic to improve the Rateless erasure code in terms of overheads and decoding speed, i.e., it can be used to find the perfect generated matrix G with the least overheads required for linear independence. Other than that, decoding speed can be improved by creating its own algorithm by studying the millions of patterns of decoding.

5.1.3 Protocol Design

In order for the Rateless Erasure Code to function appropriately (as in to utilise the network bandwidth), typical data transmission protocols such as TCP will be needed to be modified for fully utilisation of the characteristic of REC, e.g., the protocols design of REC should focus on minimising or completely abolish the needs of acknowledgment as such acknowledgment mechanism is the main reason for inefficient transmission in TCP, especially when packet loss occurs, the throughput will be reduced exponentially. Much simulation work needed to be done to find its efficiency in terms of transmission error percentage and bandwidth utilisation. We will leave all the detailed study in future works.

References

Alqahtani, A. H., Sulyman, A. I. & Alsanie, A., 2016. *Loss-tolerant large-scale MU-MIMO system with rateless space time block code.* s.l., s.n., pp. 342-347.

Anghel, B., Vasile, B. & Aurel, V., 2011. *Study of Decoding Complexity for Decoding Rateless Erasure Code.* s.l., ACTA TECHNICA NAPOCENSIS: Electronics and Telecommunications.

Anghel, B., Vasile, B. & Mihai P, S., 2011. *Performance evaluation of rateless erasure correcting codes for content distribution.* Iasi, Romania, Roedunet International Conference (RoEduNet).

Anghel, B., Zsolt, A. P. & Zsuzsanna, I. K., 2010. *FECTCP for High Packet Error.* Bucharest, Romania, IEEE.

Assefa, T. D., Kralevska, K. & Jiang, Y., 2016. *Performance analysis of LTE networks with random linear network coding.* s.l., s.n., pp. 601-606.

Bioglio, V., Grangetto, M., Gaeta, R. & Sereno, M., 2009. On the fly gaussian elimination for LT codes. *IEEE Communications Letters,* December, 13(12), pp. 953-955.

Brownlee, N. & Claffy, K. C., 2002. Understanding Internet traffic streams: dragonflies and tortoises. *IEEE Communications Magazine,* Oct, 40(10), pp. 110-117.

Chen, S., Zhang, Z., Zhang, L. & Yao, C., 2013. *Belief propagation with gradual edge removal for Raptor codes over AWGN channel.* s.l., s.n., pp. 380-385.

Cheong, S. T. & Fan, P., 2016. *Novel degree function over finite field for LT codes.* s.l., s.n., pp. 1-5.

Chong, S., Lai, A. C. & Chong, Z. K., 2016. *Improve the decoding process of rateless erasure code and network coding with graphics processing unit in IoT.* s.l., s.n., pp. 436-439.

Chong, Z. K. et al., 2015. Improving the probability of complete decoding of random code by trading-off computational complexity. *IET Communications,* 9(18), pp. 2281-2286.

Chong, Z. K. et al., 2016. *Improving Reliable Transmission Throughput with Systematic Random Code.* s.l., s.n., pp. 539-542.

CUDA, N., 2015. *Issue Efficiency,* http://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm: s.n.

CUDA, N., 2017. *CUDA Toolkit Documentation,* http://docs.nvidia.com/cuda: s.n.

Guo, Y. L., Pan, Y. & Cai, L., 2016. *OPNET-based analysis of MTU impact on application performance.* s.l., s.n., pp. 1-5.

Hagedorn, A., Starobinski, D. & Trachtenberg, A., 2008. *Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes.* s.l., s.n., pp. 457-466.

Hu, L., Nooshabadi, S. & Mladenov, T., 2012. *Implementation and evaluation of Raptor code on GPU.* s.l., s.n., pp. 1-6.

Hu, L., Nooshabadi, S. & Mladenov, T., 2013. Forward error correction with Raptor GF(2) and GF(256) codes on GPU. *IEEE Transactions on Consumer Electronics,* February, 59(1), pp. 273-280.

Jamshid, A., Siavash, F. & Konstantinos, N., 2011. *Raptor codes in wireless body area networks.* Toronto, ON, Canada, Personal Indoor and Mobile Radio Communications (PIMRC).

Julia, M. & Thinn, T., 2011. *Management of Data Replication For PC Cluster Based Cloud Storage System.* s.l., International Journal on Cloud Computing: Services and Architecture(IJCCSA).

Kevin, W., 2015. *Erasure Coding in Distributed Storage Systems.* Zurich, Department of Informatics, University of Zurich.

Kim, B. & Lee, J., 2004. A simple model for TCP loss recovery performance over wireless networks. *Journal of Communications and Networks,* Sept, 6(3), pp. 235-244.

Lee, S. Y., Arunkumar, A. & Wu, C. J., 2015. *CAWA: Coordinated warp scheduling and Cache Prioritization for critical warp acceleration of GPGPU workloads.* s.l., s.n., pp. 515-527.

Lidl, R. & Niederreiter, H., 1997. *Finite Fields.* 2 ed. s.l.:Cambridge University Press.

Li, H. et al., 2014. *Work in progress: A new algorithm to improve the decoding success probability of Raptor code.* s.l., s.n., pp. 271-274.

Luby, M., 2002. *LT codes.* s.l., s.n., pp. 271-280.

Lu, W., Lin, X., Lin, J. & Niu, K., 2012. *A novel construction method of fountain codes.* Chengdu, China, Communication Technology (ICCT), IEEE 14th International Conference.

M, A. S. & S, L. K., 2006. *Systems and processes for decoding a chain reaction.* United States of America, Patent No. US6856263B2.

Mathis, M., Jeffrey, S., Jamshid, M. & O.Teunis, 1997. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *SIGCOMM Comput. Commun. Rev.,* #jul#, 27(3), pp. 67-82.

Mladenov, T., Nooshabadi, S. & Kim, K., 2012. Efficient GF(256) raptor code decoding for multimedia broadcast/multicast services and consumer terminals. *IEEE Transactions on Consumer Electronics,* May, 58(2), pp. 356-363.

Molnar, S., Moczar, Z. & Sonkoly, B., 2014. *How to transfer flows efficiently via the Internet?.* s.l., s.n., pp. 462-466.

Ren, Z., Wang, Z. & Guo, Q., 2014. *Rateless codes based file delivery protocols in deep space communications.* s.l., s.n., pp. 1-6.

Rossi, M. et al., 2010. SYNAPSE++: Code Dissemination in Wireless Sensor Networks Using Fountain Codes. *IEEE Transactions on Mobile Computing,* Dec, 9(12), pp. 1749-1765.

Salyers, D. C., Striegel, A. D. & Poellabauer, C., 2008. *Wireless reliability: Rethinking 802.11 packet loss.* s.l., s.n., pp. 1-4.

Shaneel, N. & Paula-Rayond, L., 2013. *Network Performance Evaluation of Jumbo Frames.* s.l., 6th International Conference on Emerging Trends in Engineering and Technology.

Shokrollahi, A., 2006. Raptor codes. *IEEE Transactions on Information Theory,* June, 52(6), pp. 2551-2567.

Shokrollahi, A. & Luby, M., 2011. Raptor Codes. *Foundations and Trends in Communications and Information Theory,* 6(3–4), pp. 213-322.

Yeqing, W. et al., 2013. *A Fast Raptor Codes Decoding Strategy for Real-Time.* s.l., Canadian Center of Science and Education.

Yuan, X., Sun, R. & Ping, L., 2010. Simple capacity-achieving ensembles of rateless erasure-correcting codes. *IEEE Transactions on Communications,* January, 58(1), pp. 110-117.

Zhu, H., Zhang, G. & Li, G., 2008. *A novel degree distribution algorithm of LT codes.* s.l., s.n., pp. 221-224.

List of Publication

Ran-Chong, S., Lai, A. C. & Chong, Z. K., 2016. *Improve the decoding process of rateless erasure code and network coding with graphics processing unit in IoT.* s.l., s.n., pp. 436-439.

Chong, Z.K., Hiroyuku, O., Bryan, N., Goi, B.M., Ewe, H.T., & Ran-Chong, 2016. *Improving Reliable Transmission Throughput with Systematic Random Code*. IEEE Local Computer Networks (LCN), 2016 IEEE 41st Conference, pp. 539-542.