ACCELERATING ERASURE CODE WITH MULTI-PROCESSING

LOH HONG KHAI

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons.) Electronic & Communications Engineering

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

May 2016

DECLARATION

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

Signature	:	
Name	:	
ID No.	:	
Date	:	

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"ACCELERATING ERASURE CODE WITH MULTI-PROCESSING"** was prepared by **LOH HONG KHAI** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons.) Electronic & Communications Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature	:	
Supervisor	:	
Date	:	

Specially dedicated to

my beloved grandfather, grandmother, mother and father

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Dr. Lai An-Chow and co-supervisor Dr. Chong Zan-Kai for their invaluable advice, guidance and their enormous patience throughout the development of the research. Brainstorming sessions with Mr. Chong Sin Ran, the Master's research student also proved to be great for coming up with ideas.

In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me encouragement. Without them, this would be a joyless project. My parents have been accommodating and helpful in making sure I obtain all the resources that I need. Special thanks to my good friend Fredderich Yii Ying Sen who has since the inception of this project been amazingly encouraging. He was generous enough to allow me to code and run computations on his machine so that measurements could be taken.

ACCELERATING ERASURE CODE WITH MULTI-PROCESSING

ABSTRACT

Erasure code has long been used for binary erasure channels, in which a message with k original symbols is transformed into larger n encoded symbols where a subset of n can be decoded to retrieve the original message. Fountain code is a rateless erasure code in that n can be potentially infinite and the receiver can decode the message with 99.99% success probability given k’ symbols where k’ is only slightly larger than k. Fountain codes such as LT code and Raptor code are very efficient for long messages with linear time encoding and decoding through pre-coding and backward substitution method, or simply by using Gaussian Elimination. However, network traffic is mostly short messages, hence fountain codes like Stepping Random (SR) and systematic Random (SYSR) codes are proposed at the expense of higher computational complexity. The method of decoding involves Gaussian Elimination (GE) in Galois field of 2 elements GF(2). For typical GE and its variations, the decoding complexity is $O(k^3)$. Unique to GE of GF(2), the elimination method involves only component-wise XOR operation. With this in mind, this report explores the use of multi-processing to parallelize the elimination method and the pivot search. This includes the comparison of GE in multi-core central processing units (CPU) and general purpose graphic processing units (GPU). Ideally for multi-processors, the speed up can be N times where N is the number of concurrent threads. This report proposes & analyses the use of multi-core CPU and GPU, both available off-the-shelf, in accelerating the GE of GF(2) for the decoding use of SYSR & SR codes. The result shows that speedup on the GPU increases with the increase in the number of symbols in the message. The speedup can even go up to more than 20 times using the range of symbol length and number of symbols utilized in this project. The bottleneck of the realistic implementation on GPU was found out to be memory access, which should be targeted for future works.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

GE	Gaussian Elimination
GF(2)	Galois field of 2 elements
GF(256)	Galois field of 256 elements
RSA	Rivest-Shamir-Adleman cryptosystem
ALU	Arithmetic Logical Unit
GFLOP	Giga floating point operations

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 Erasure Codes & Fountain Codes

The most common networking protocol, and the network communication of the majority of internet, is the Transport Control Protocol (TCP). TCP treats data as packets that are in sequence, using retransmission to guarantee the packets are received in order from the receiver's viewpoint (Mitzenmacher, 2004). However, the acknowledgement and retransmission mechanism of TCP consumes substantial bandwidth from the network and may not be suitable for some applications.

To improve on the transmission efficiency, erasure code is introduced into TCP. Erasure code is a forward error correction that is capable of recovering *k* data packets from *n* data packets transmitted, where *n* is larger than *k*. We have:

$$n = (1 + \mathcal{E})k \tag{1.1}$$

n = encoded message length

k = original message length

ε = overhead/redundancy

k/n = code rate

Given *k* linearly independent packets received, the receiver can then decode *k* input packets, regardless of which packets received. To overcome the limitation of finite coded symbols, an alternative paradigm known as digital fountain is introduced. A digital fountain has properties similar to a fountain of water: when you fill your cup from the fountain, you do not care what drops of water fall in, but only want that your cup fills enough to quench your thirst (Mitzenmacher, 2004).

Luby introduced the first fountain code that is practical known as Luby Transform (LT) code and was afterwards improved by Shokrollahi as Raptor code. Despite the breakthrough in LT code and Raptor code, they are only efficient for long-length messages (Chong et al., 2013). Chong et al. proposed Stepping-Random (SR) and systematic Random (SYSR) codes to accommodate smaller, shorter length packets for the internet. This comes at a price of computation, where the decoding of such codes is O(k^3) using Gaussian Elimination. Since the implementation of the SR and SYSR codes is in binary field, this report therefore examines the effect of using multi-processing in accelerating the decoding process.

1.2 Gaussian Elimination in GF(2)

Named after Carl Friedrich Gauss, Gaussian elimination is the one of the most common methods used to solve system of linear equations. It transforms a matrix into a row (column) echelon form or into its reduced row (column) echelon form; in the later case, the algorithm is known as Gauss-Jordan elimination (Morancho, 2015). Linear equations appear in many cases of various research and application fields, notably in cryptanalysis such as the RSA encryption schemes. Therefore, many literatures have proposed solutions related to it. However, many of the proposed algorithms involving GE based algorithms such as LU decomposition and Cholesky decomposition target the generic, real number field. For SYSR and SR code where we look for a targeted approach to solving GE in GF(2), to the best of my knowledge there is no implementation of GE in GF(2) on off-the-shelf products such as CPU or GPGPU (General Purpose GPU) in the literature. Hardware-specific solutions like the parallel hardware architecture by Bogdanov et al. require higher human resource and financial cost.

SR and SYSR codes are both rateless erasure code for the use of internet networking. Hence, the flow of data comes from the internet, then to the processing unit(s). The streaming nature of the data transfer is a characteristic that has to be taken into account. Further work can be done through streaming the data into the CPU/GPU at the background during the decoding process. This in essence hides the latency of data transfer. The concept of data block transfer which coalesce the required data together for higher transfer bandwidth is also considered to attain higher throughout in general.

The binary nature of the received data allows us to consider only the GE acceleration in GF(2). Firstly, to accelerate the GE algorithm through multi-processing, the algorithm has to be broken down to separate parts that are parallelizable and those that are not. The portion that can be run simultaneously should then be distributed to different processing units, either in a multi-core CPU or a GPU. In the case of GPU, the data has to be copied from the main memory to the GPU's memory before a kernel can be initiated.

1.3 Parallelization

The approach that is looked into is one of parallel computing. Parallel computing arises from the exponential growth of digital devices in this century. Without a doubt, the explosion of the amount of digital data in current age has changed the demographics of computer users. The limited frequency of scaling for single core is an obstacle. By increasing core count, performance can be greatly improved.

Traditionally, computing has been written for serial computation. This means that a problem is broken into a sequence of discrete instruction, where each is executed right after another. Only one instruction may be executed at any single time. In a simple sense, parallel computing uses multiple compute resources to solve a problem which can be broken into discrete parts. These parts are then solved concurrently, with the result of needing an overall coordination mechanism. The growth of parallel computing in current years is encouraging. An example of classification for parallel computer is shown below:

SISD Single Instruction stream Single Data Stream	SIMD Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	MIMD Multiple Instruction stream Multiple Data stream

Table 1.1: Flynn’s taxonomy

However, one challenge that has to be considered is Amdahl’s Law. Amdahl’s Law states that the potential program speedup is limited by the fraction of code (P) that can be parallelized:

$$speedup = \frac{1}{1-P} \tag{1.2}$$

If none of the code can be parallelized, there will not be any speedup even with parallel computing. If all the code is parallelized – an ideal case – then the speedup is infinite in theory. The relationship can then be modelled as below when the number of processing units (N) is added in:

$$speedup = \frac{1}{\frac{P}{N}+S} \tag{1.3}$$

S is the serial fraction and P is the parallel fraction. This sets a limit into the power of parallelism, which we have to also take into consideration. One of the goals of this project is to make sure the parallelizable part is maximized, with the serial overhead kept to a minimum.

1.4 Aims and Objectives

The aim of this project is to come up with an accelerated method of solving system of linear equations in the Galois field of 2 elements, through multi-processing units i.e.

GPU and multi-cores. Since GF(2) entails component-wise XOR, the algorithm has to be customized for the use of SYSR code in internet networking context. In interest of this objective, the tasks are set out as below:

- To analyse the overhead and the bottleneck of the implementation of GE in GF(2)
- To propose a new algorithm suited for multi-processing
- To implement such algorithm in GPU using NVIDIA CUDA C
- To evaluate the performance of the proposed solution in different architectures and compare it with traditional CPU implementation

CHAPTER 2

LITERATURE REVIEW

2.1 Fountain Codes

Fountain codes are rateless in the sense that the number of encoded packets that can be generated from the source message is potentially limitless; and the number of encoded packets generated can be determined on the fly. Fountain codes are universal because they are simultaneously near-optimal for every erasure channel. Regardless of the statistics of the erasure events on the channel, we can send as many encoded packets as are needed in order for the decoder to recover the source data. The source data can be decoded from any set of *K'* encoded packets, for *K'* slightly larger than *K*. Fountain codes can also have fantastically small encoding and decoding complexities (MacKay, 2005). LT code, developed by Luby (2002) is the first practical implementation and Raptor code by Shokrollahi (2006) is the improved version.

To facilitate understanding, the implementation of LT code is studied. Encoding is straightforward and easy: the encoded packets are generated first by choosing a random number of packets, and then adding them together with XOR operation. The number of input packets used to encode depends on the degree distribution like the Solitun distribution. This distribution is also at heart of the code since it will determine whether decoding is successful or not.

Figure 2.1 illustrates the decoding process. The decoding process initiates with all symbols being uncovered. Then, all degree one decoding symbols get released to cover their unique neighbours. The set of covered message symbols that have not been processed is known as the ripple. The process then randomly selects a set to process and removes that encoding symbol as neighbours from all encoding symbols. Any symbols with degree one are released and its neighbours covered. The process continues until all packets are decoded (Luby, 2012). An improvement of the decoding process is proposed by (Lu et al., 2009) called full rank decoding to prevent the LT code from terminating early due to lack of symbols in the ripple.

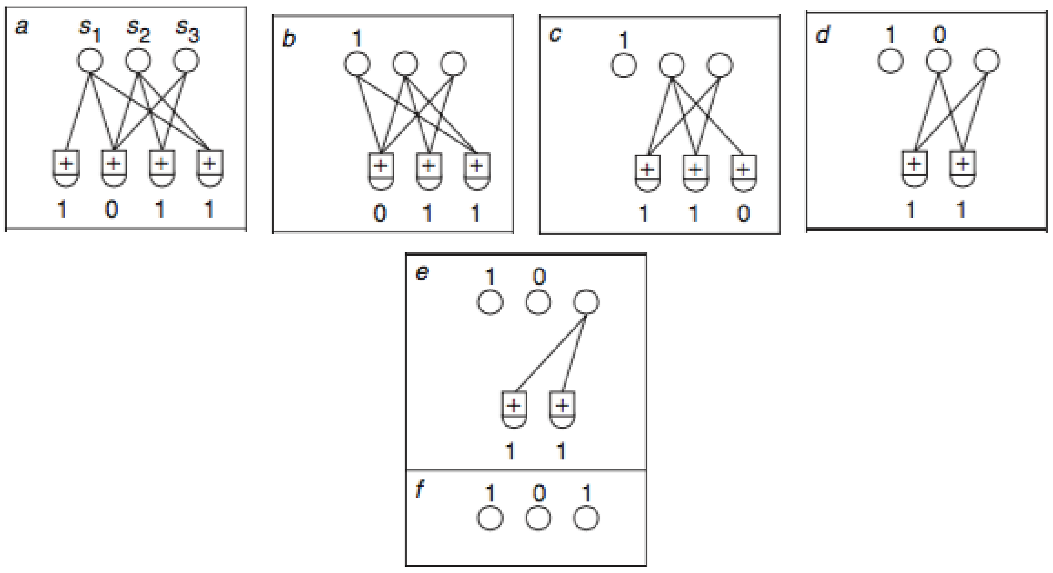

Figure 2.1: Example decoding for a fountain code with K = 3 source bits and N = 4 encoded bits (MacKay, 2005)

However, LT and Raptor codes require large input packet length to deliver near-optimal transmission rate. For k > 100,000, Raptor code has a decoding inefficiency of $\varepsilon \approx 0.03$ (Chong et al., 2013). Hence, Chong et al. (2015) considers shorter packet length in expense of higher computational complexity using Gaussian Elimination in decoding in the proposal of SYSR code. Just like typical systematic rateless erasure code, a message of *k* symbols will form part of the encoded symbols (termed as Part I coded symbols) and the rest of coded symbols from *k+1th* onwards (termed as Part II coded symbols) are generated by adding the k message symbols randomly, i.e. multiplying the random matrix with the message (Chong et al., 2015).

The Part I of the SYSR code is just the original message. It can be denoted as a matrix by having *k* symbols with *l* bits each, i.e. $\mathbf{M}^{k \times l}$. The Part I generator matrix is just an identity matrix. For Part II, each coded row matrix $\mathbf{M}^{1 \times l}$ is independently generated by multiplying a random row matrix $\mathbf{G}^{1 \times k}$ with $\mathbf{M}^{k \times l}$, i.e.:

$$\mathbf{X}^{1 \times l} = \mathbf{G}^{1 \times k} \times \mathbf{M}^{k \times l} = \begin{bmatrix} g_{0,0} \cdots g_{0,k-1} \end{bmatrix} \times \begin{bmatrix} m_{0,0} & \cdots & m_{0,l-1} \\ \vdots & \ddots & \vdots \\ m_{k-1,0} & \cdots & m_{k-1,l-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} \cdots g_{0,k-1} \end{bmatrix}$$

(2.1)

The receiver is assumed to receive both the coded symbol $\mathbf{X}^{1 \times l}$ and the corresponding generator matrix $\mathbf{G}^{1 \times k}$ at the same time as they are embedded in the same packet during the transmission. Subsequently, the receiver reconstructs the original message with Gaussian elimination when it has full rank generator matrix (Chong et al., 2015). To illustrate the decoding, the coded symbol matrices $\mathbf{X}^{k+10 \times l}$ is then augmented with the generator matrix $\mathbf{G}^{(k+10) \times l}$ such as below:

$$\left[G^{(k+10)\times k} | X^{(k+10)\times l}\right] = \left[\begin{array}{ccc|ccc} \tilde{g}_{0,0} & \cdots & \tilde{g}_{0,k-1} & \tilde{x}_{0,0} & \cdots & \tilde{x}_{0,l-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \tilde{g}_{\alpha-1,0} & \cdots & \tilde{g}_{\alpha-1,k-1} & \tilde{x}_{\alpha-1,0} & \cdots & \tilde{x}_{\alpha-1,l-1} \\ \hline g_{\alpha,0} & \cdots & g_{\alpha,k-1} & x_{\alpha,0} & \cdots & g_{\alpha,l-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ g_{k-9,0} & \cdots & g_{k-1,k-1} & x_{k-9,0} & \cdots & g_{k-1,l-1} \end{array}\right] \quad (2.2)$$

g_{ij} denotes the *i*th row *j*th entry in the generator matrix, x_{ij} denotes the *i*th row *j*th entry in the coded symbol matrix. The tilde denotes Part I row matrices. To reduce the matrix down to an upper triangle matrix, The following steps are taken (Chong et al., 2015) :

Step 1. Identify a Part I generator row matrix. Then, denote the position of the selected Part I generator as row *i* and its non-zero entry at column *j*.

Step 2. Add (XOR) row *i* of both generator and coded symbols row matrices to those Part II row matrices, for which their *j* entries are non-zero.

Step 3. Remove row *i* and column *j* from the augmented matrix.

Step 4. Repeat from Step 1 until all the Part I row matrices are removed.

If Part I is received intact in whole, then no decoding is needed. However, it has to wait for $k + 10$ Part II symbols to arrive in the incident that Part I is not received fully intact. Since the decoding complexity using GE is $O(k^3)$ when Part I is not fully received, an accelerated version of GE through multi-processing is desired so that the decoding can be done quickly.

The following example taken from (Chong et al., 2015) assumes a message of $k = 4$ and a symbol size of $l = 2$. The receiver has 2 Part I coded symbols and 12 Part II coded symbols. The augmented received matrix is then:

$$[G^{14\times 3}|X^{14\times 2}] = \left[\begin{array}{cccc|cc} 1 & 0 & 0 & 0 & \tilde{x}_{0,0} & \tilde{x}_{0,1} \\ 0 & 1 & 0 & 0 & \tilde{x}_{1,0} & \tilde{x}_{1,1} \\ \hline 1 & 0 & g_{2,2} & g_{2,3} & x_{2,0} & x_{2,1} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & g_{13,2} & g_{13,3} & x_{13,0} & x_{13,1} \end{array}\right] \tag{2.3}$$

In Step 1, the first row of Part I symbol is selected and then the first non-zero entry is searched, i.e. $i = 0$ and $j = 0$. Then in Step 2, this row is added into the rows in Part II where column j is also non-zero. Then, row i and column j are removed in Step 3, resulting to:

$$[G^{13\times 3}|X^{13\times 2}] = \left[\begin{array}{ccc|cc} 1 & 0 & 0 & \tilde{x}_{1,0} & \tilde{x}_{1,1} \\ \hline 0 & g_{2,2} & g_{2,3} & x_{2,0}\oplus\tilde{x}_{0,0} & x_{2,1}\oplus\tilde{x}_{1,0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & g_{13,2} & g_{13,3} & x_{13,0} & x_{13,1} \end{array}\right] \tag{2.4}$$

Notice that there is still one row left in Part I of the matrix. Step 1 and 2 are repeated until no rows are left, giving us:

$$[G^{12\times 2}|X^{12\times 2}] = \left[\begin{array}{cc|cc} \hline g_{2,2} & g_{2,3} & x_{2,0}\oplus\tilde{x}_{0,0} & x_{2,1}\oplus\tilde{x}_{1,0} \\ \vdots & \vdots & \vdots & \vdots \\ g_{13,2} & g_{13,3} & x_{13,0}\oplus\tilde{x}_{1,0} & x_{13,1}\oplus\tilde{x}_{1,1} \end{array}\right] \tag{2.5}$$

When Part I is fully eliminated, Gaussian Elimination is used to continue with the augmented matrix to decode it.

2.2 Gaussian Elimination

2.2.1 Gaussian Elimination on GPU

A general GE produces an upper triangle matrix, in which back-substitution is required to solve for x. If Gaussian-Jordan elimination is done instead, the resulting solution is trivial because the matrix A will be an identity matrix. Gaussian-Jordan Elimination method, or just commonly referred as Gaussian Elimination, is very stable and widely used. It involves pivoting and then reducing the matrix to row echelon form. This comes at an expense of a cubic operation instead of a quadratic operation of back-substitution. Given the widespread use of GE, the improvement of the algorithm and the acceleration are desired.

An early paper by Saad in 1986 describes the minimum communication time for GE on multiprocessor is $O(N^2)$, regardless of number of processors. It was tested on ring and bus type architectures (Saad, 1986). However, no proper test results were given since this was a theoretical paper that was not implemented on actual multiprocessors available nowadays. The paper by Galoppo et al. in 2005 presented a novel GPU-based algorithm for solving dense linear systems. The spatial coherence between elementary row operations and fast parallel data transfer to move data on GPU are exploited (Galoppo et al., 2005). This shows great potential for using GPU and any other multi-processing units in implementing GE. Another paper by Buluc et al. also provided proof that running 4 separate GE-based algorithms can be faster on GPU, specifically in this case a NVIDIA 8800 GPU was compared to an Opteron. The temporal locality was exploited instead in this approach. While the previous paper avoided matrix multiplication, Buluc et al. managed to use alternate schedule of path computations us to cast almost all operations into matrix-matrix multiplications on a semiring. Since matrix-matrix multiplication is highly optimized and has a high ratio of computation to communication, the implementation does not suffer from the premature saturation of bandwidth resources as iterative algorithms do (Buluç et al., 2008). Similar work for solving partial differential equations using Finite Element

Method was also done in (Leow et al., 2011) which showed that the performance to execute forward elimination can be up to 185x faster when using GPU instead of a quad-core i5 Intel CPU. On decoding of Raptor GF(2) and GF(256), the serial and parallel implementation were explored for practical real-time requirement in multimedia broadcast/multicast service (Linjia Hu, 2013). However, this is on Raptor codes which have different matrix characteristics and sparsity.

The literature has given enough confidence that GPU can greatly enhance the GE method. However, none of them addressed the unique GF(2) properties when using GE on GF(2). An implementation of GE on GF(2) on GPU and similar multi-processing architecture have not been compared and contrast. Since GF(2) is a limited field and operations involve only XOR and AND operations, it is worthy to explore the effectiveness and performance enhancement using GPU. A specific algorithm to accelerate GE in GF(2) contributes to the decoding speed of aforementioned SYSR and SR codes.

2.2.2 Galois field in two elements

GF(2) is the Galois field with two elements, the smallest finite field. The two elements in it are called 0 and 1 as they are the additive and multiplicative identities respectively. The field's addition operation corresponds to the logical XOR operation, while the field's multiplication operation corresponds to the logical AND operation. Because of the unique properties of GF(2), it is somewhat simpler yet important in the implementation of modern computers with binary code as data representation.

Several literatures have involved dedicated hardware or customized, specific hardware implementation for the improvement of GE over GF(2). A method has been developed that needs no extra storage to store the history of the elimination with a program to work with up to 4096×4096 matrices has been developed for the ICL-DAP (Parkinson & Wunderlich, 1984). In 1991 by the work of of Koc and Arachchige, an algorithm was proposed employing the binary search technique. The algorithm takes $2m^2 + mlog_2n -2m$ bit operation to triangularize an $n \times m$ matrix on a bit array with n

processors and $m+2+log_2n$ bits of vertical memory per processor (Koç & Arachchige, 1991). The implementation however was on the Geometric Arithmetic Parallel Processor (GAPP), a mesh-connected array of single-bit SIMD processing elements. Both the parallel hardware in Parkinson & Wunderlich as well as the GAPP are not commonly used nowadays, and they are dedicated hardware developed in the previous decades for calculation, hence rendering them obsolete.

Bogdanov et al. presented a parallel hardware architecture for fast GE over GF(2) using an FGPA implementation. A hardware-optimized algorithm for matrix-by-matrix multiplication over GF(2) which runs in linear time and quadratic space was proposed. The algorithm incorporates shifts to increase the speed of calculation. A hardware implementation for LSEs up to a dimension of 50 × 50 has been realized using VHDL. The design can be implemented for LSEs up to dimension of 1000 × 1000 yielding conventional chip sizes , and larger matrix has to be broken down by Strassen's algorithms (Bogdanov et al., 2006). However, the use of FPGA and the dimension limit of 1000 × 1000 makes it less attractive since it will be costlier and restrictive. The FGPA nature also means that the algorithm cannot be easily updated over-the-air, unlike software updates.

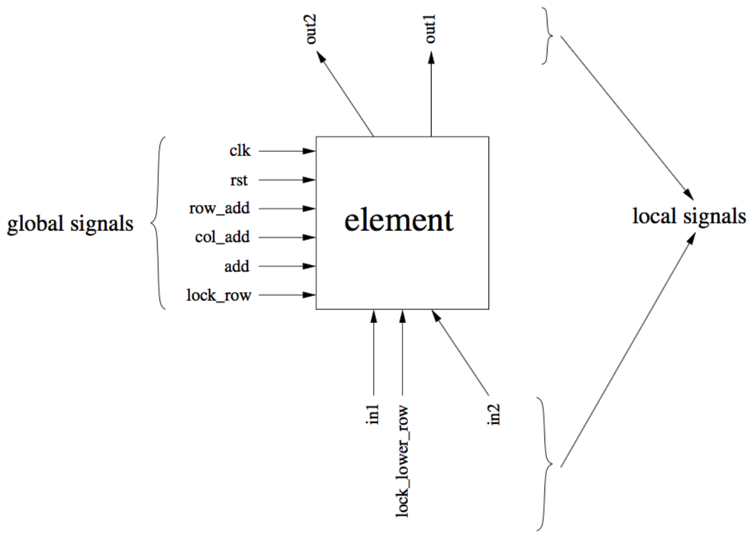

Figure 2.2: I/O Signal of basic matrix element (Bogdanov et al., 2006)

The most recent approach was the vector implementation of GE over GF(2) in 2015. By exploiting the vector implementation of modern processors. Performance results show that the larger the vector length, the smaller the execution time. For the same vector length, implementations specialized for this case study outperform generic implementations by between 3.7X and 3.9X (Morancho, 2015). This approach does not include the use of GPU to increase the acceleration, so there is still room for improvement in accelerating GE over GF(2), in which will be discussed in this report.

As a summary, literature available shows that many attempts at solving systems of linear equations are done with infinite field of real numbers, including those using GPU. Literature also shows proposal of using dedicated hardware for the Gaussian Elimination over GF(2). However to the best of my understanding, there is no dedicated solution for parallelizing GE over GF(2) in the context of GPU and comparing the performance with that of multi-cores. Such a solution will be useful in accelerating the erasure codes i.e. SR and SYSR codes.

CHAPTER 3

METHODOLOGY

3.1 Profiling of Gaussian Elimination

3.1.1 Galois field in two elements

Galois field of two elements is the simplest finite field. In it, addition and multiplication is defined to be:

$0 + 0 = 0$	$0 + 1 = 1$	$1 + 0 = 1$	$1 + 1 = 0$
$0 \times 0 = 0$	$0 \times 1 = 0$	$1 \times 0 = 0$	$1 \times 1 = 1$

Table 3.1: Addition and Multiplication of GF(2)

With 0 and 1 viewed as bits, it is easy to see that the addition in GF(2) is exclusive OR, and the multiplication is AND. Matrices formed by elements in GF(2) are called binary matrices or bit matrices. The advantage of utilizing a finite field is that there is no round off errors when using computers, like those of infinite fields.

3.1.2 Amdahl's Law and Limitations

As introduced before, Amdahl's Law plays a large role in determining the potential speedup that is expected from a parallel implementation. The formula, for the sake of discussion, is given again as:

$$speedup = \frac{1}{\frac{P}{N}+S} \tag{3.1}$$

Speedup is tied to the size of the fraction by both the parallel (denoted as P) and serial part (denoted as S) of a particular algorithm. Theoretically, the number of parallel processing units (denoted as N) can tend towards infinity to better understand the limitations of the algorithm given infinite resources. P and S are both less than 1 as they are a fraction of the code.

The denominator of the speedup equation has S plus P divided by N. S is a summation of the strictly serial part of the algorithm, the synchronization cost required, the switching involved and other overhead costs caused by a certain parallel implementation. For this discussion, since the speedup of a theoretical unlimited number of parallel processing unit is assumed, the overhead costs are all lumped together in S for sake of simplicity. S by itself contains a fix amount of serial work that is inherent in the algorithm. Addition to that, as N increases, there is a chance that the synchronization and overhead communication costs will increase proportionally. This is the first limit to the speedup.

Subsequently, the next term of the denominator, the P/N term is examined. To have a small denominator so that a large speedup is obtained, the ideal case is to have P to tend towards 1 and hence S will tend towards 0. However, in realistic cases, S is substantial and P can only be minimized to a certain extend. It is also impossible that unlimited processing power, the N term, can be obtained. It has a physical limit depending on the machine or GPU that is used.

To better understand the fraction of P and S, a distillation of all the elements in the algorithm can be done. The different elements in the parallel fraction and serial fraction of the algorithm can reveal which part of it is the culprit for the slowdown. It can be computation, memory access, synchronization or many other elements. This way, the bottleneck can also be targeted for improvement.

3.1.3 Implementation

The transformations applied by both Gaussian and Gauss-Jordan eliminations are three elementary row operations:

i) swapping rows
ii) multiplying all the row elements by a nonzero scalar and
iii) adding to a row the scalar multiple of another row

By understanding the steps of Gaussian Elimination, the algorithm can then be applied to the specific GF(2). That way, the bottlenecks and dependencies can be found.

For Gaussian Elimination, the three elementary row operations are dependent on each other. The second step has to wait for the completion of the first step and the third step can only start when the second step is finished. Essentially, the pivot element has to be found and swapped before forward elimination and backward substitution can be done. Before moving to the next column, all the previous steps have to be completed. Another inherent issue is that communication and syncing are required for the algorithm to run in parallel. The pivot row has to be communicated so that all row elimination (forward elimination and backward substitution) can be done in parallel.

The method of parallelizing the algorithm is proposed as follows:

i) The search for pivot can be done by splitting the matrix and dividing them to different cores, or only done by 1 core if the matrix is dense enough
ii) The pivot row is distributed to all spawned parallel operations
iii) The matrix is divided and split to each operation so that row elimination can be done concurrently
iv) All individual operations have to be sync and wait for the total completion before moving on to the next column

Taking that into consideration, the parallel algorithm in CPU and GPU can be implemented to compare their performance, with the system as given in Section 3.2. Profiling of the algorithm will be done to check the complexity and time taken to complete the whole GE. We proceed by:

1. Implementing GF(2)-specific GE in single-threaded environment using C, with a size of under 32 × 32 bits
2. Improving the algorithm to fit the largest vector, which is unsigned long long.
3. For matrix larger than 64 × 64 bits, we need to split them up to multiple vector containers.
4. For sizes that do not fit exactly the vector containers, we need to pad them up with 0 or 1.

Once the algorithm works in a CPU environment, it is then ported to a GPU environment. The code is run in GPU under CUDA first with a single thread, then extended to multiple, concurrent threads. In a GPU, the following points have to be considered:

1. The data is first copied from the CPU (host) to the GPU (device).
2. The CPU then initiates a kernel that runs multiple concurrencies. Each of them work on a separate set of data.
3. Synchronization is needed in between the code. However, communication is expensive and should be kept minimum.
4. When the process is done, the control is returned to the CPU.

3.2 System Specifications

The project is run on a system with the following specification:

CPU: Intel Core i5 4690 3.50GHz

GPU: NVIDIA GeForce GTX 960

Memory: 8GB 1600MHz DDR3

Intel Core i5 is a quad-core CPU targeted for the consumers, mostly for the gaming market and normal graphical usage. Implementation of GE over GF(2) on multi-cores will be implemented on this same system. This is done using the C language, one of the most popular programming language. For the GPU implementation, CUDA platform is selected.

3.3 GPU Architecture

With the growth of computing and the use of better and higher definition graphics, the market demand for better image processing has allowed the growth of Graphic Processing Units (GPU). Coupled together by the increasing difficulty of extracting instruction level parallelism (ILP) from instruction stream, the insatiable demand from the market for real-time, 3D graphics has turned the GPU into a highly parallel, multithreaded, multi-core processor with tremendous computational horsepower and very high memory bandwidth suitable for high performance computing. Now GPGPU (General Purpose GPU) serves as both a programmable graphics processor and a scalable parallel computing platform. Most systems are heterogeneous, combining a GPU with a CPU. Performance wise, the floating point operations per second by the GPU greatly exceeds that of CPU, as illustrated in Figure 3.1.

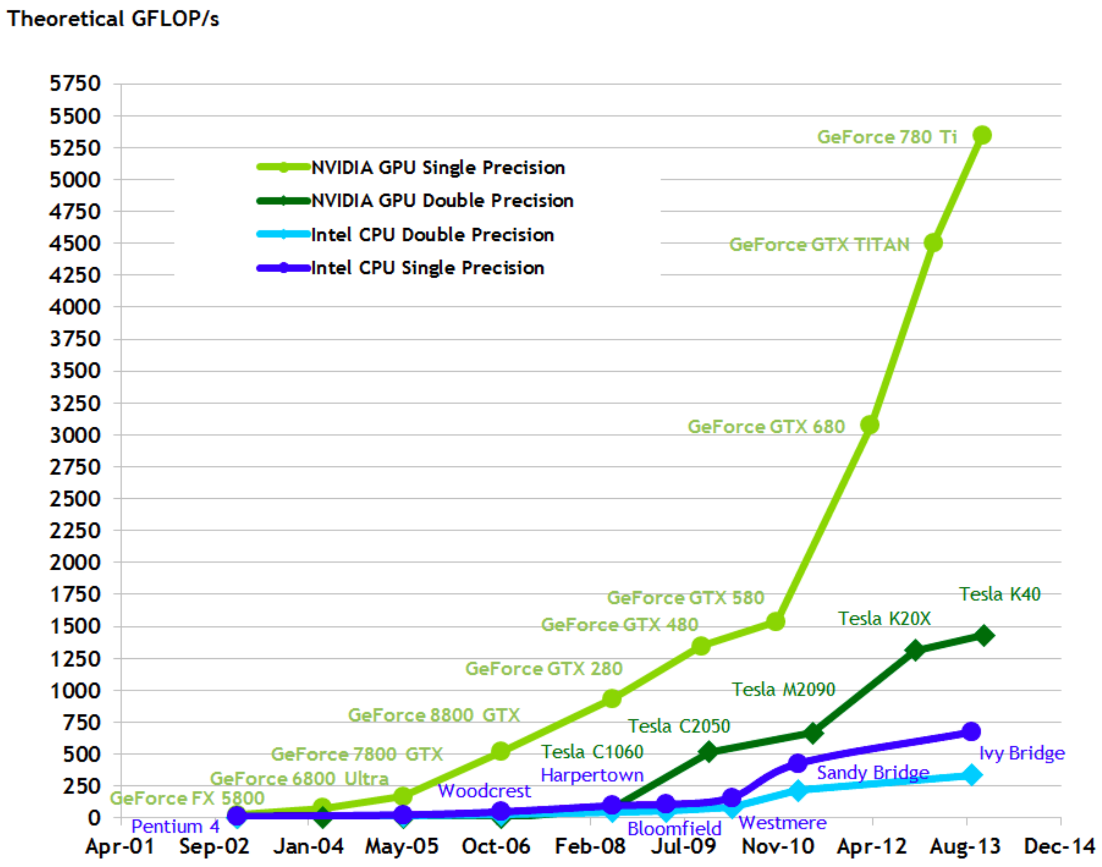

Figure 3.1: Floating-Point Operations per Second for the CPU and GPU

(NVIDIA Corporation, 2015)

The reason behind the disparity between the floating-point capability of GPU and CPU is because GPU is designed for graphic rendering, a process that is highly computing intensive and requires parallel computing. A lot more data is devoted to data processing than flow control. The difference between the 2 structures is shown in Figure 3.2.

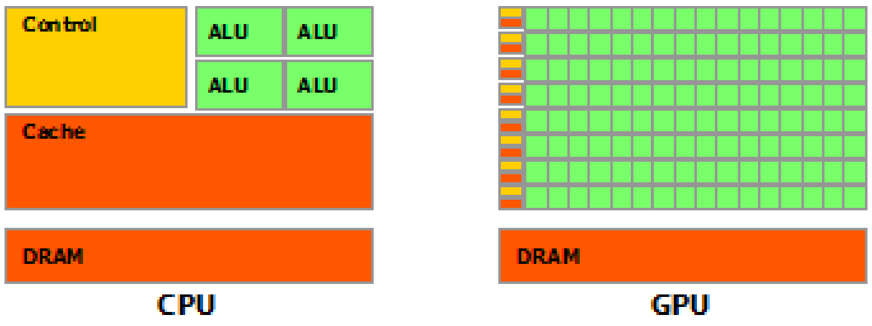

Figure 3.2: The GPU Devotes More Transistors to Data Processing

The GPU architecture consists of two main components: global memory and streaming multiprocessors (SM). The global memory acts like a random access memory (RAM) in a CPU server. This portion of the GPU is accessible by both the GPU and CPU, so that data can be transferred between CPU and GPU. The SMs perform the actual computation. Just like a small processor, an SM has its own control units, registers, execution pipelines and caches.

GPU is well-suited to tackle problems that involve data-parallel computations. This means that a program that is executed on many data elements – often with high arithmetic intensity – can utilize the ability of GPU. A program with high ratio of computation over memory access means there is a lower requirement for flow control, and hence memory latency can often be hidden behind with calculations.

3.3.1 NVIDIA CUDA Platform

In November 2006, NVIDIA introduced CUDA®, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU (NVIDIA Corporation, 2015). CUDA comes in an environment where programmers can code in one of the most common language, the C language. It also allows multiple different higher level programming language and interface like FORTRAN and Python.

With the advent of multi-core CPUs and GPUs, it is obvious that our current computing systems are mostly parallel systems. The challenge onwards lies in developing application software and algorithms that are parallel in nature, meaning that they can transparently scale their parallelism to leverage the increasing numbers of processing cores. Using a familiar language like C, CUDA can attract more programmers to tackle this problem and develop more application software that exploits the GPU's power.

CUDA at its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions (NVIDIA Corporation, 2015). These abstractions allow more fine-grained parallelism both in data and thread, through the use of higher level, coarse-grained data and task parallelism. A programmer is guided to divide the task into coarse sub-problems, which are then more finely defined and divided. This kind of decomposition allows thread to work together, meaning they can scale and work regardless of the number of memory partitions or multiprocessors. CUDA will work for different GPUs having different capacity and power.

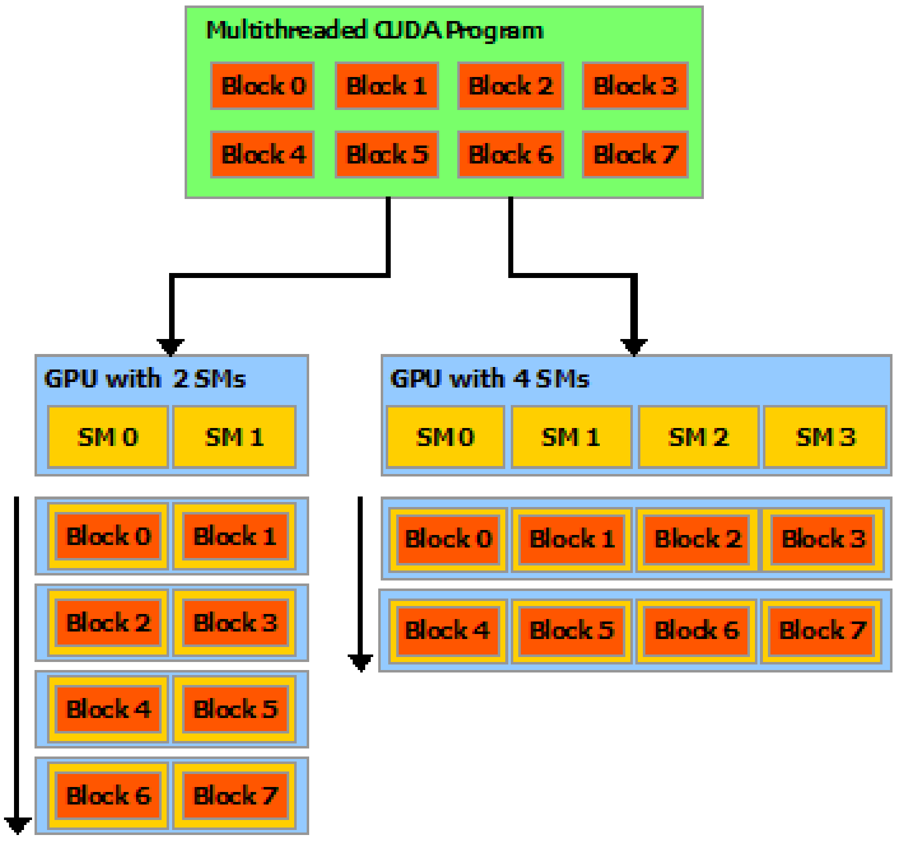

Figure 3.3: Automatic Scalability (NVIDIA Corporation, 2015)

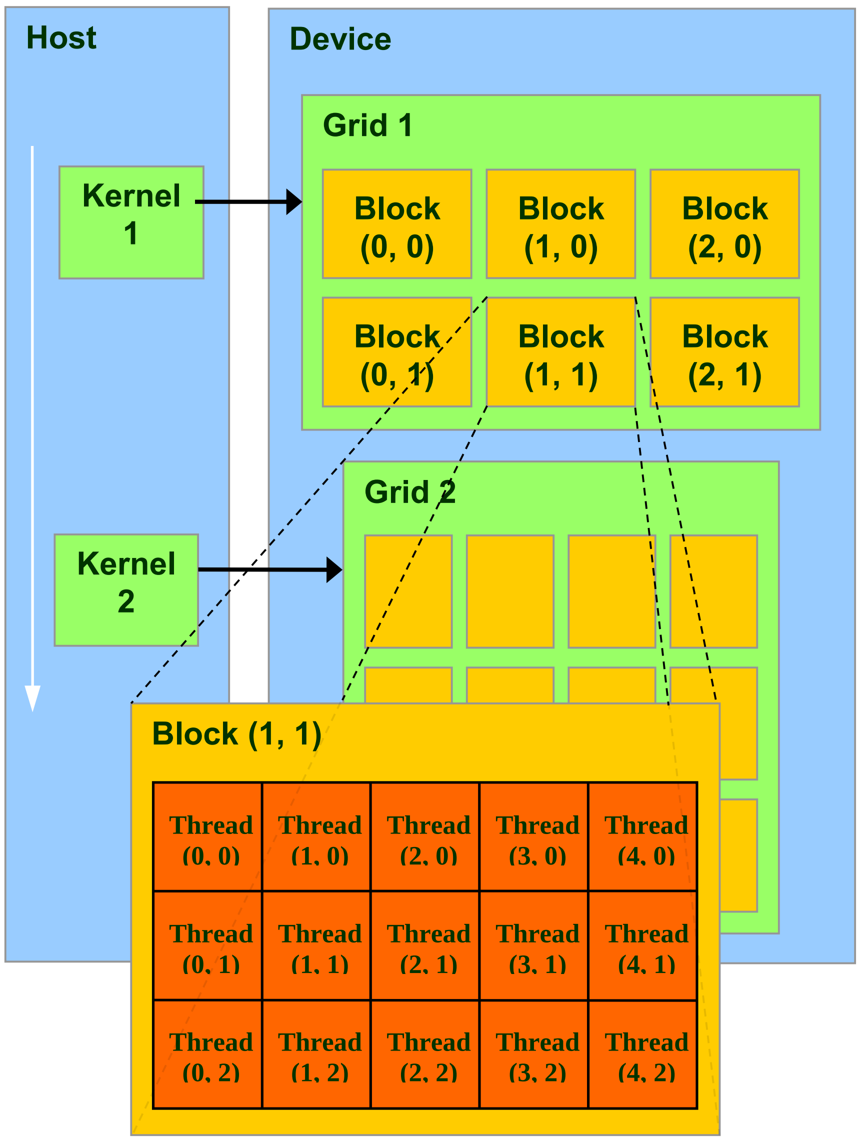

Figure 3.4: Hierarchy and division of process

As shown in Figure 3.4, the hierarchy and division of process can be done to a low-level. First, the host launches kernels that are run by a grid of blocks. These blocks can have threads in them processing on the data. The difference between the blocks and the threads is that parallel threads have mechanisms to communicate and synchronize. It should be noted that the kernel launches by the host is just like the normal program: it runs serially which then spurs the grids in GPU.

What programmer expresses in CUDA are computation partitioning, data partitioning, data management and orchestration, as well as concurrency management. Computation partitioning deals with the problem of where does the computation occur.

Declaration of functions using __host__ __global__ and __device__ clarifies the computation location. Mapping of thread programs to device are done with <<gs, bs>>(<args>) where gs is number of blocks started, bs is number of threads started and <args> is the argument passed. Data portioning deals with the location of the data as well as the scope and access method of the data. Declarations include __shared__ and __constant__. Data management and orchestration is responsible of copying to and from the host and device. This is done mostly through the cudaMemcpy() function. Lastly, __syncthreads() is used for concurrency management, making sure that concurrent processes are managed well and no race conditions take place. Equipped with the understanding of GPU architecture and CUDA abstraction, programming can be made better and the algorithm to run the parallel programming on GPU can be optimized.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Profiling of Realistic Bottlenecks for Implementation on GPU

4.1.1 Understanding Gaussian Elimination's steps

According to (Linjia Hu, 2013), the most time consuming parts of the Gaussian Elimination are the backward substitution and forward elimination steps. This is especially evident given that the encoding method utilized in this project is a random code encoding matrix with a density of 0.5. This effectively means that the other two steps required to complete a cycle of Gaussian Elimination – pivot search and row swapping – are generally done rather quickly since the pivot can be found quickly.

Fortunately, both forward elimination and backward substitution are processes that are amenable to parallel processing. There is potential for speedup in this particular section given that once the pivot search is done and swapping completed, every row of the matrix has to be checked and XOR-ed if the condition is met. Since this process is independent for each row, there is big room for acceleration, and it is then posited that the effect will be apparent when the size of the matrix is large. The other reason is the availability of many vacant processing units or cores in the GPU. Since the process is independent, the CUDA drivers are free to allocate the best resources and optimize the queueing and data processing.

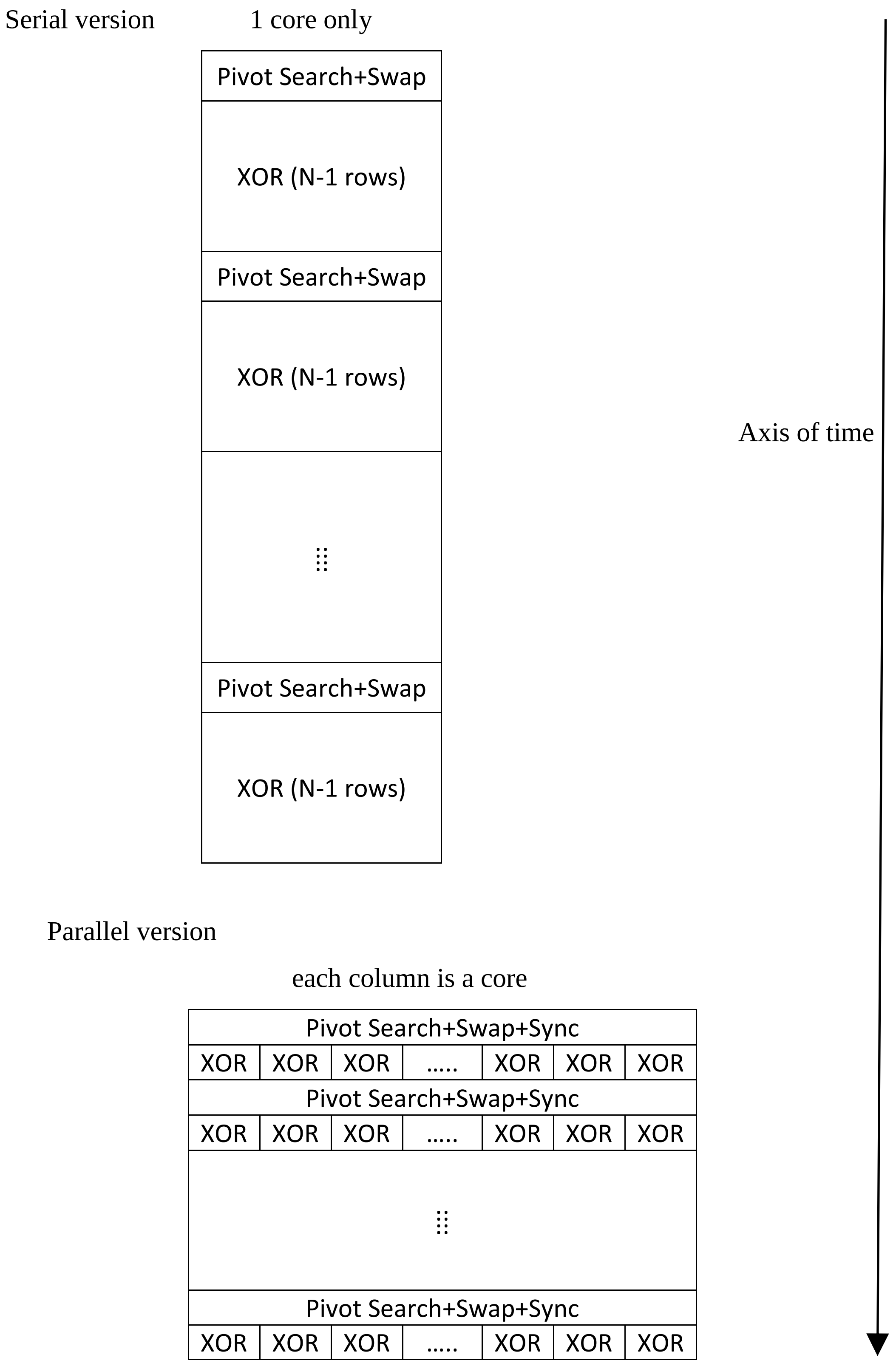

Figure 4.1: Graphical Comparison of Serial & Parallel Implementation

Gaussian Elimination partitions the workload to three distinct steps, mainly the pivot search, swap and then XOR. The parallelizable part of the algorithm lies mainly in the XOR operations.

The workload can be shrunk when the XOR operations are done quickly and in parallel by the parallel processing units, in this case the cores of the GPU. The serial implementation has the XOR operation taking up a longer time, from Figure 4 it is shown that the XOR operations in serial implementation is longer vertically taking more time (not drawn to scale). When done in parallel, the XOR operations divides to many cores that can run quickly and in shorter time frame. The aim is to have the XOR operations reduced in time substantially. However, the parallel implementations include a synchronization cost, which will be discussed later in the results to be minimal, hence not affecting the performance too much.

4.1.2 Proposed Algorithm

The proposed algorithm is as given and implemented.

Require: Generator matrix G and encoded matrix X, augmented G|X. G & X each has *n*+10 rows and m columns, so the augmented matrix has *n*+10 rows with *2m* columns.

```
1:  procedure GEonGPU(G|X)
2:      (Host)Import matrix G|X from text file, parse into vectorised form of unsigned
3:              long arrays;
4:      (Host→Device)Copy the whole G|X matrix from host to device;
5:      for $i \leftarrow 1, (n-1)$ do
6:          (Kernel 1)   $j \leftarrow i$;
7:                       Search for pivot row on the $i$-th column from row $j$ to row $(n+9)$.
8:                       Then, replace pivrow[] with the whole row of the pivot found.
9:                       Replace $j$-th row with $i$-th row;
10:         (Kernel 2)   for all rows except row $i$:
11:                          if $i$-th column is 1, then
12:                              XOR pivrow[] with that row;
```

```
13:                 end for
14:         (Kernel 3)   Replace i-th row with pivrow[];
15:     end for
16:     (Device→Host) Copy the matrix X to host. Export to text file as hexadecimal
17:                   characters.
18: end procedure
```

During the XOR operation, each row of the matrix is handled by each single thread. Each thread will traverse through the whole row to execute the XOR operation. Therefore, when more than 1024 rows are in a matrix, there must be more than 1 block that is initiated. This is because the limit of the number of threads that can be launched per block is 1024 for NVIDIA CUDA C. However, if the row number is smaller than 1024 and can be processed by only 1 block, then the configuration of block and thread counts can be more flexible.

4.1.3 Elements of CUDA C Implementation

Global memory coalescing is a method by NVIDIA CUDA C to allow better use of the memory bandwidth. The device coalesces global memory loads and stores into as little transactions as possible thereby minimizing the DRAM bandwidth. When a warp of 32 threads access data that is within locality, the access is simply coalesced together as 1 request. As the number of threads are increased and the data is arranged in a fashion that is contiguous when accessed, we can compare and measure the effect of global memory coalescing.

Shared memory is confined to a single block, residing in a multiprocessor. The limit of the shared memory is very small relative to the global memory, and depends on the device's limit. Shared memory bank conflict arises when multiple addresses of a memory request maps to a common memory bank. This will make the memory request to the shared memory block to be serialized, decreasing the bandwidth to a factor equal to the number of separate memory requests. However, since this algorithm cannot utilize shared memory, this is not of concern.

Synchronization includes block level synchronization using __syncthreads() and also kernel launches since they are serialized in a stream. Sync in a block happens when every warps and threads have to reach the __syncthread() function before the whole code can proceed. This might delay what could have been an acceleration if they are not dependent, however we have no way to make sure data independence. Therefore, synchronization is inevitable. The kernel launch time and __syncthreads() are the overheads that we can measure when the elements are distilled down.

Branch divergence happens when treads inside a warp branches to different execution paths. Depending on how many branches are created from the divergence of a warp, the performance loss can be significant. One way to mitigate this is to have the branches not do anything when they diverge. Indeed, as the CUDA C implementation has divergence when some threads require XOR operation on its respective row, while other threads do not. However when there is no XOR operation required, there is no further operation down the line. This is a better solution to reduce performance loss.

4.1.4 Filtering out the Bulk of Operation

The understanding of Gaussian Elimination proves to be the first step into implementing an effective algorithm onto the NVIDIA GPU for acceleration. Given that the previous deductions are mostly theoretical and do not take into account the realistic issues of global memory coalescing, shared memory bank conflicts, overhead of synchronization, branch divergence and so on. To better drill down on the substantial part that is affecting the performance of the decoding using GPU, the implementation has to be timed and each component stripped out so that the elements can be quantified.

Any implementation of a parallel processing algorithm will consist of overhead costs. Amdahl's Law, as mentioned before, succinctly puts a formula that can be used to investigate the cost and return of parallelization. Only when the return given by utilizing many processes are greater than the overhead and synchronization, it is a worthy venture. A general process is assumed when implementing the parallel data processing in NVIDIA CUDA, and it is given as follows:

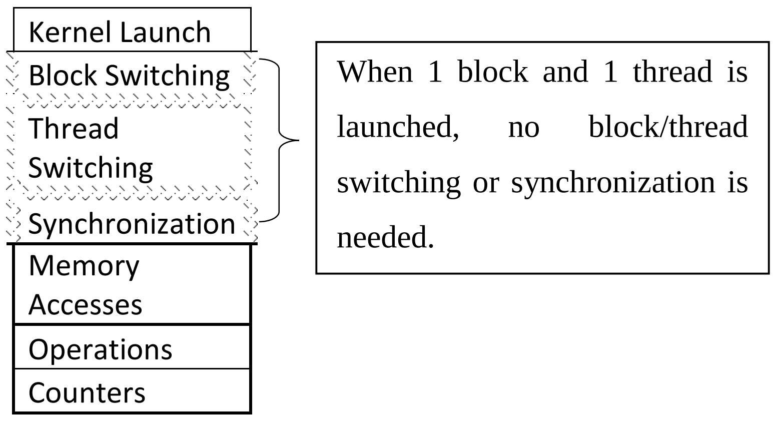
Kernel Launch
Block Switching
Thread Switching
Synchronization
Memory Accesses
Operations
Counters
When 1 block and 1 thread is launched, no block/thread switching or synchronization is needed.

Figure 4.2: Elements of a GPU Kernel

When the whole process is broken down into elements, the algorithm can be tweaked so as to boil down the cost of each element. The question now is, which element takes more time and which part should we focus on improving? To be able to filter out the block switching, thread switching and synchronization part out of the calculation, 1 single thread from 1 single block is used to run the whole algorithm. Essentially, there is no parallelization except that there are costs for kernel launches since this is done in GPU. Running a single thread to process the whole matrix has proven to be time-consuming. It took almost half an hour to completely decode the original message, as shown in table below.

To further boil down on the elements' time costs, the memory access element is removed from the algorithm. This is done by doing the same amount of operations but only using the registers which are very near to the processing ALU, which means that the speed is very close to having to use no memory access at all in comparison. Understandably, the result of decoding is not correct since the required data is in global memory. However, this method is useful to remove memory access in the equation, but keeping all the algorithms and operations still the same.

Furthermore, the use of counters to keep track of the algorithm is also omitted. This way, the barebones are left and the time that is used to execute each element can be clearly separated from each other.

To solve a GF(2) matrix with size of (N+10) × 2N , where N = 4096:

Elements	Time taken (ms)	Ratio
Kernel Launch + Memory Accesses + Operation+ Counters	1705214.0	100%
Kernel Launch + Operations + Counters	295240.0	17.31%
Kernel Launch + Operations	217626.0	12.76%
From above, we can calculate:		
Memory accesses	~1409974.0	82.69%
Counters		4.51%

Table 4.1: Breakdown of elements for 1 threads

The table above shows that memory access takes up the bulk of the operation time, using around 83% of the total time. A few reasons can be given for this time-consuming memory access. Firstly, since there is only 1 thread executing it, only the memory cache and registers for 1 core, at the maximum, can be utilized. The core has a lot more capacity that is wasted, and could have contributed in reducing memory access counts.

Secondly, and most crucial, is that the way the GPU is designed is not good for only single thread execution. The architecture of GPU is such that it combines or coalesce neighbouring data into one single memory access, reducing the total number of memory access required if done individually. Global memory loads and stores are usually done with warps. Warps in this case, are a bundle of 32 threads – though some concern has to be taken since older GPUs has semi-warps of 16 threads. If these threads are accessing adjacent data, the GPU can immediately coalesce 32 of them into just 1 single read or write. This can improve the utilization of the DRAM bandwidth. Instead of doing that, by using 1 sole thread to run we are essentially throwing away the 31 other adjacent data points when we request for data, and then subsequently request again. Only 4 bytes out of the 128 bytes are utilized.

In short, the total number of warp memory requests are the same as the total number of XOR operations done if we are only utilizing 1 single thread. The aim now is to reduce the number of warp memory requests so that it is very much smaller than the total number of operations done. Bear in mind GPU has the advantage of many ALUs that can operate on data concurrently, and that is the leverage beneficial to the project.

The next step forward is to increase the number of threads. This has two obvious advantage:

1. The memory accesses of a warp can be fully utilized instead of being discarded
2. Memory that are contiguous are loaded and processed in parallel fashion

By increasing the thread counts, the synchronization and thread switching costs are examined. Even though performance using 1 thread shows the result being memory access as the major bottleneck, the thread count should be increased to see if this issue is mitigated when more threads are initiated.

With the same data size, but launching with 256 threads:

Elements	Time taken (ms)	Ratio
Kernel Launch + Thread Switching + Synchronization + Memory Accesses + Operation + Counters	18602.7	100%
Kernel Launch + Thread Switching + Synchronization + Operation + Counters	2067.0	11.11%
Kernel Launch + Thread Switching + Operation + Counters	2056.8	11.06%
From above, we can calculate:		
Memory accesses	~16535.7	88.88%
Synchronization	~10.2	0.05%

Table 4.2: Breakdown of elements for 256 threads

When more than 1 thread is used, a block level synchronization for all threads in the same block is used. The nature of Gaussian Elimination requires that all backward substitutions are done before the next pivot search can be started. Since only a block level synchronization is done, we use the __syncthreads() function available from the CUDA C library.

The results from the 256 thread launch shows that the overall speed has increased significantly. Where as 1 single thread needs nearly half an hour to decode, 256 threads take less than 20 seconds to decode the same size! That is a speedup of 91 times. Memory accesses took around 89% of the total processing time. This is an increase from the 83% previously used by 1 thread.

The reason why memory access has taken up more percentage when there is more threads launched is because as the thread count increases, there is more capacity per core to execute the XOR operation. The workload of XOR operation is then distributed to many other processing units. This means that averagely, each processing unit has a smaller amount of work to do. When the work load is significantly smaller, the ratio of execution over the total execution time drops. Memory accesses on the other hand, can only be coalesced to 32 threads in 1 warp, regardless of the distribution of the workload as long as it is more than 32 threads launched.

The next thing to notice is also the use of __syncthreads() for synchronization purposes contributed only less than 0.1% of the total time. It is a very lightweight and good synchronization function, although it can only be done in the block level, not in the grid level in CUDA terms. Memory access pertains to be the bottleneck that is hindering further performance enhancement.

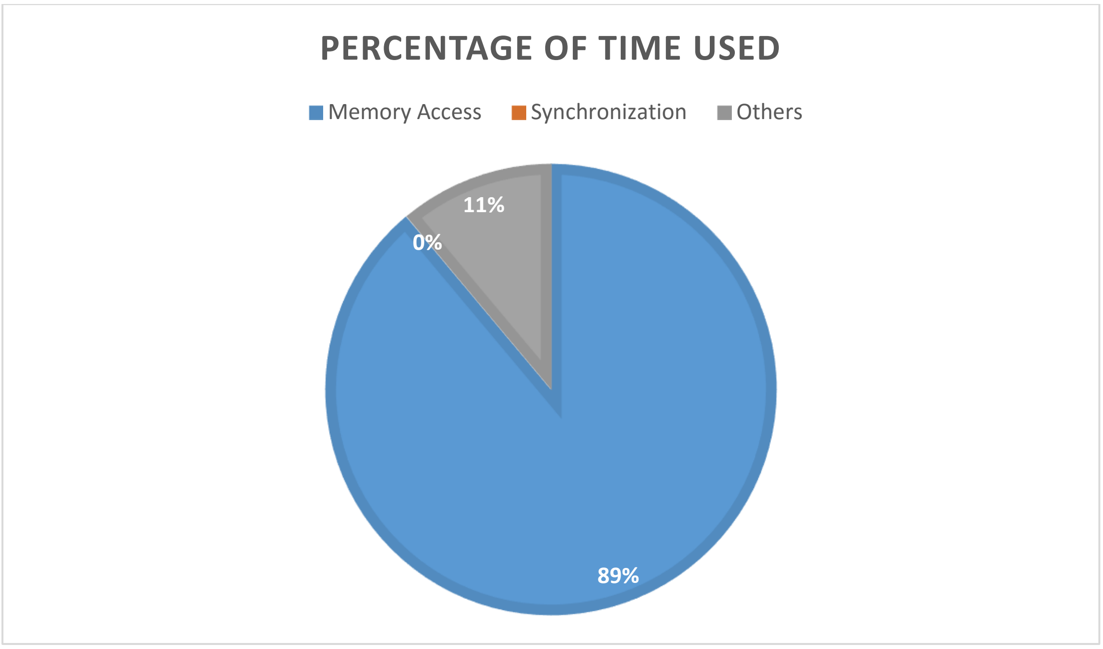

Figure 4.3: Bulk of Operation in GPU Implementation

To see how the thread counts and block counts affect the speed of decoding, various thread counts and block counts are tested to see which of them yields the best performance.

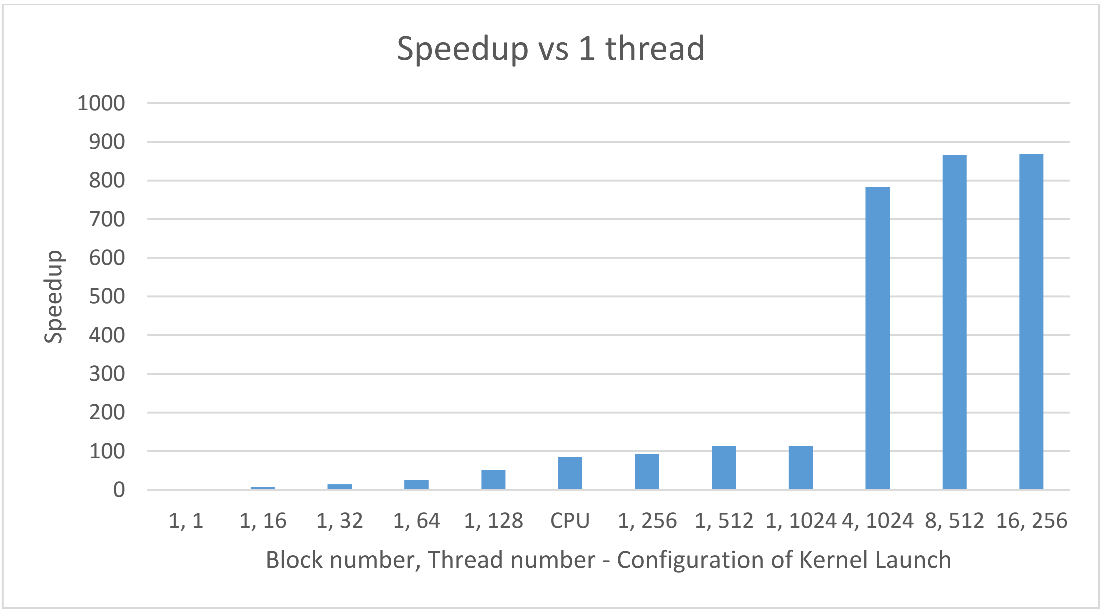

Figure 4.4: Speedup vs 1 thread

Block and thread number configuration are factors affecting the performance of the decoding. To better understand how different configurations work, a comparison is made for all the configurations. The speedup approaches plateau when it hits 256 threads, and increases only nominally when the thread count is increased to 512 and 1024 threads. This is processed with 1 block – which basically is only 1 Streaming Multiprocessor used out of the 8 available. When we increase the total thread count by launching more blocks, the speedup goes through a significant growth as shown. This tallies with our expectation since the operations are distributed out when more threads in general are used, but is limited by the physical multiprocessor and core counts. It should be noted that threads and blocks are simply abstract concept for the ease of programming.

With the observations, it seems that with 128 multiprocessors, they can handle 2 datasets before plateauing in performance, giving rise to 256 threads as a good optimum count for thread counts. Threads are executed by cores in a multiprocessor. Block counts on the other hand corresponds better to the number of streaming multiprocessors. In simple terms, each block is mapped to a single multiprocessor. Since we have 8 SMs in the GPU, and having 8 blocks launched is the optimum block number for the results, increasing the block number per SM does not greatly affect the results after that. This shows that a multiprocessor saturates when 256 threads are executed and the extra ones are queued, while the total multiprocessor saturates at 8 blocks, as that is the total number of physical multiprocessor available in GTX 960.

In conclusion, as the total number of threads increases, the speedup increases, given that the block and thread count configuration is optimized and it is limited by the physical count of multiprocessors and cores available to the GPU. Memory access takes up the bulk of the processing time especially when the execution work load is distributed to many ALUs.

4.2 Performance

4.2.1 Message Generation and Encoding

Since the message generation and encoding are not under the acceleration scope of this project, they are done using MATLAB for ease of implementation and assurance of correctness. MATLAB has the ability to generate Galois field of 2 elements and performs operations on them, so a simple code is written to generate a random message of desired length with desired number of symbols, which are subsequently encoded with a corresponding matrix of generated elements. It should be noted also that the number of encoded symbols has 10 extra compared to the number of symbols of the original message due to Kolchin's Theorem to make sure we have a probability of complete decoding (PCD) of 99.999% (Chong et al., 2013). To export out the message matrix, the encoded message matrix and also the generator matrix, the binary matrix is converted into hexadecimal representation in text and exported as a text file. The generator matrix is augmented with the encoded symbols matrix. Each symbol takes up a whole row of characters, while the symbols are separated by a new line.

The encoded data can have different combinations of symbol length and number of symbols. The length of the symbols is varied from 1024 bits to 16384 bits. The number of symbols tested is also varied from 32 to 16384 rows.

4.2.2 Decoding Performance

The total time required for the processing is given by the tables below.

Rows/Columns	1024	2048	4096	8192	16384
	Time required in ms (CPU)				
32	0.05	0.15	0.25	0.34	0.94
64	0.2	0.43	0.72	1.44	3.18
128	0.84	1.53	3.89	5.63	13.66
256	3.74	6.13	11.35	31.52	88.57
512	16.67	27.07	73.97	157.12	323.37
1024	86.15	252.24	432.65	815.2	1796.5
2048	1056.8	1454.93	2492.24	4081.6	12464.5
4096	11304.2	14250.2	18709.67	36885.9	68325.7
8192	122325.2	132005.6	160336	222951.2	337316.2
16384	1125262	1168553	1337740		

Table 4.3: Time required for CPU

Rows/Columns	1024	2048	4096	8192	16384
	Time required in ms (GPU)				
32	2.49	3.96	7.12	13.5	25.87
64	4.79	8	14.17	26.49	51.34
128	9.91	16.08	28.49	53.22	100.15
256	21.27	33.52	58.32	99.98	184.15
512	48.9	73.7	110.33	198.06	455.75
1024	109.95	153.77	242.47	512.2	967
2048	310.28	430.75	716.04	1167.7	2066.3
4096	1277.4	1498.2	1968.11	2911.8	4792.3
8192	5258	5864.8	6942.8	9587.2	13843.6
16384	40380.5	42688.5	49149.5		

Table 4.4: Time required for GPU

When both of them are compared for performance, we can obtain the speedup as follows:

Speedup					
Rows/Columns	1024	2048	4096	8192	16384
32	0.020	0.038	0.035	0.025	0.036
64	0.042	0.054	0.051	0.054	0.062
128	0.085	0.095	0.137	0.106	0.136
256	0.176	0.183	0.195	0.315	0.481
512	0.341	0.367	0.670	0.793	0.710
1024	0.784	1.640	1.784	1.592	1.858
2048	3.406	3.378	3.481	3.495	6.032
4096	8.849	9.512	9.506	12.668	14.257
8192	23.265	22.508	23.094	23.255	24.366
16384	27.866	27.374	27.218		

Table 4.5: Speedup vs CPU

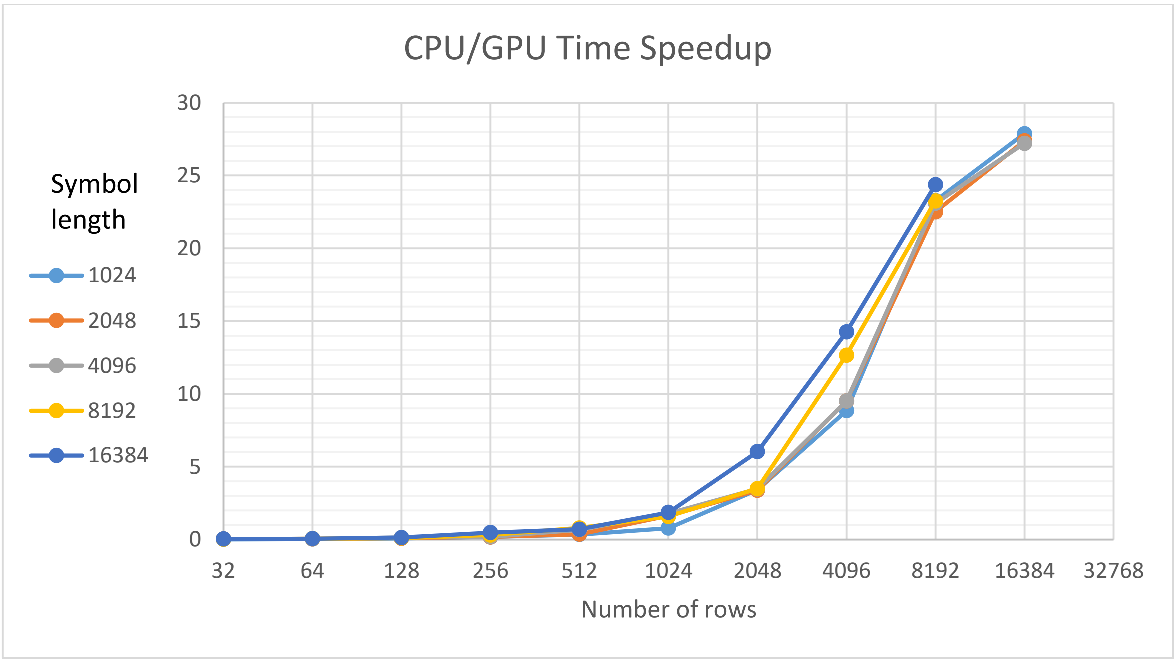

Figure 4.5: Speedup vs 1 GPU for different symbol length and number of symbols

The speedups highlighted in red in Table 4.5 are those where CPU performance is better than GPU performance using the proposed algorithm. For number of symbols 512 or less, CPU performs better in all column size that we tested. For 1024 rows with 2048 columns and above size, we can see that GPU algorithm outperforms CPU. As the size of encoded message gets bigger, the speedup of GPU is more significant. This tallies with previous hypothesis which is GPU is suited for large dataset processing that reduces the relative cost of overhead and synchronization by GPU kernel launches.

Another interesting observation is that as the size of the column gets bigger when the number of row is kept constant, the increase is speedup is not as steep as when the number of rows is increased. For example, when the number of rows is 4096, the speedup when the column size is increased from 1024 to 16384 ranges from 8 to 14. However, when the number of rows is doubled to 8192, the speedup starts from 23 folds. To put it simply, a twice increase in size in column has around 7% increase in speedup, but a twice increase in size in row has 162% increase in speedup. In networking terms, the number of symbols instead of symbol length has a greater impact on the speed of decoding.

This can be attributed to the fact that the matrix is already vectorised. Each symbol is vectorised into type of unsigned long, so that memory access is fully utilized. Every 32 bits of data – corresponding to 32 columns – will only add 1 extra vector of unsigned long. Yet each row starts with a new vector array. Hence, it is logical that adding extra symbols contributes to the longer processing time more significant than adding symbol length to it.

CHAPTER 5

CONCLUSION AND RECOMMENDATIONS

5.1 Recommendations and Future Works

It should be noted that the matrix utilized in this project is moderately dense and randomly generated. The performance is measured under this premise. Future work can include testing the algorithm with very sparse matrix or very dense matrix. This can yield different results because of the steps required to obtain a pivot, and the total number of operations needed when different densities of matrix is used can be very different.

Another future work that can be done is utilizing the nature of interaction between host and device to come up with an algorithm that allows the host and device to both compute at the same time. Usually termed 'on-the-fly', this kind of algorithm divides the workload to both CPU and GPU. One of such ideas is to utilize the CPU to search for pivot since it is a serial operation, and then let the GPU handle the XOR operation for massive parallel, independent operation.

NVIDIA CUDA C has a lot to offer, and different approaches can result in varying speedups. A new algorithm that allows each thread to access only a single vector can also be promising, but the overhead and synchronization costs should also be included. This is a direction worth exploring.

Finally, by tackling the memory access problem, future work can better improve memory performances and reduce the decoding time needed. This is a substantial bottleneck that should be researched and solved to accelerate Gaussian Elimination on GPU.

5.2 Conclusions

The bottleneck of the algorithm is shown to be the memory access which takes up the bulk of processing time. An algorithm suited for GPU implementation of Gaussian Elimination for Galois field of 2 elements is proposed and implemented using NVIDIA CUDA C. The performance has been shown to outperform normal CPU implementation when the number of symbols is 1024 and above. When the size of encoded message is bigger, the speedup gets bigger, more so when the number of symbols is increased. The block and thread number configurations are also affecting the performance and depends on the hardware for optimum acceleration.

REFERENCES

Mitzenmacher, M., 2004. Digital fountains: a survey and look forward. In *Information Theory Workshop*. San Antonio, 2004. IEEE.

Chong, Z.-K. et al., 2013. Stepping-Random Code: A Rateless Erasure Code for Short-Length Messages. *IEICE Transactions on Communications,* 96(7), pp.1764-71.

Morancho, E., 2015. A Vector Implementation of Gaussian Elimination over GF(2): Exploring the Design-Space of Strassen's Algorithm as a Case Study. In *2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Turku, 2015. IEEE.

MacKay, D.J.C., 2005. Fountain codes. *IEE Proceedings-Communications,* 152(6), pp.1062 - 1068.

Luby, M., 2012. LT Codes. In *Proceedings of The 43rd Annual IEEE Symposium on Foundations of Computer Science.*, 2012. IEEE.

Lu, F., Foh, C.H., Cai, J. & Chia, L.-T., 2009. LT codes decoding: Design and analysis. In *IEEE International Symposium on Information Theory ISIT 2009*. Seoul, 2009. IEEE.

Chong, Z.-K. et al., 2015. Systematic rateless erasure code for short messages transmission. *Computers & Electrical Engineering*, 45, pp.55-67.

Saad, Y., 1986. Communication complexity of the Gaussian elimination algorithm on multiprocessors. *Linear Algebra and its Applications*, 77, pp.315-40.

Galoppo, N., Govindaraju, N.K., Henson, M. & Manocha, D., 2005. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware. In *Proceedings of the ACM/IEEE SC 2005 Conference.*, 2005. IEEE.

Buluç, A., Gilbert, J.R. & Budak, C., 2008. *Gaussian Elimination Based Algorithms on the GPU*. Under review for the Special Issue of Parallel computing on parallel matrix algorithms and applications.

Leow, Y.K., Akoglu, A., Guven, I. & Madenci, E., 2011. High Performance Linear Equation Solver Using NVIDIA GPUs. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. San Diego, 2011. IEEE.

Koç, Ç.K. & Arachchige, S.N., 1991. A fast algorithm for gaussian elimination over GF(2) and its implementation on the GAPP. *Journal of Parallel and Distributed Computing*, 13(1), pp.118-22.

Parkinson, D. & Wunderlich, M., 1984. A compact algorithm for Gaussian elimination over GF(2) implemented on highly parallel computers. *Parallel Computing*, I(1), pp.65-73.

Bogdanov, A. et al., 2006. A Parallel Hardware Architecture for fast Gaussian Elimination over GF(2). In *FCCM '06 Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, 2006. IEEE.

Morancho, E., 2015. A Vector Implementation of Gaussian Elimination over GF(2): Exploring the Design-Space of Strassen's Algorithm as a Case Study. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Turku, 2015. IEEE.

NVIDIA Corporation, 2015. *CUDA C Programming Guide*. [Online] Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide [Accessed August 2015].

Linjia Hu, S.N.T.M., 2013. Forward Error Correction with Raptor GF(2) and GF(256) Codes on GPU. *IEEE Transactions on Consumer Electronics*, 59(1), pp.273-80.

APPENDICES

APPENDIX A: Computer Programme Listing

MATLAB

```
row = 4096;
col = 4096;
m = 1;
extra_row = 10;

for fileid = 1:100
   mes_mat = randi((0:(2^m)-1), row ,col);
   gen_mat = randi((0:(2^m)-1), row+extra_row,row);
   mes_matgf2 = gf(mes_mat,m);
   gen_matgf2 = gf(gen_mat,m);
   encoded_matgf2 = gen_matgf2*mes_matgf2;
   encoded_matgf2_double = double(encoded_matgf2.x);
   gen_matgf2_double = double(gen_matgf2.x);
   mes_matgf2_double = double(mes_matgf2.x);

   file1name = sprintf('%d_%d_20.txt', row, fileid);
   fileID = fopen(file1name,'w');
   for a = 1:size(encoded_matgf2_double,1)
      enc_hex_str = binaryVectorToHex(encoded_matgf2_double(a,:));
      gen_hex_str = binaryVectorToHex(gen_matgf2_double(a,:));
      fprintf(fileID,gen_hex_str);
      fprintf(fileID,enc_hex_str);
      fprintf(fileID,'\n');
   end
   fclose(fileID);

   file2name = sprintf('%d_%dans_20.txt', row, fileid);
   fileID = fopen(file2name,'w');
   for a = 1:size(mes_matgf2_double,1)
      mes_hex_str = binaryVectorToHex(mes_matgf2_double(a,:));
      fprintf(fileID,mes_hex_str);
      fprintf(fileID,'\n');
   end
   fclose(fileID);
end
```

NVIDIA CUDA C – GPU

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <algorithm>

// shared memory in GPU buffer
__device__ __constant__ int rownum_d;
__device__ __constant__ int arr_count_d;
__device__ __constant__ int num_bits_in_type_d;

// GPU kernel declaration
__global__ void XORKernel(unsigned long *dev_arr,unsigned long *dev_pivrow,
                          const int curcol, const int curarr)
{
    unsigned long checkbit = 1;
    int i =  blockDim.x*blockIdx.x + threadIdx.x;
    if (i<rownum_d) {

        if (dev_arr[curarr*rownum_d+i] & (checkbit<<(curcol%num_bits_in_type_d))) {
            for (int k = 0; k<arr_count_d; k++) {
                dev_arr[k*rownum_d+i] = dev_arr[k*rownum_d+i]^dev_pivrow[k];
            }
        }
    }
}

__global__ void pivotSearchKernel(unsigned long *dev_arr, unsigned long *dev_pivrow,
                                  const int curcol,const int curarr,const int currow)
{
    if (0 == threadIdx.x) {
        unsigned long checkbit = 1;
        for (int i = currow; i < rownum_d; i++) {
            if (dev_arr[curarr*rownum_d+i] & (checkbit<<(curcol%num_bits_in_type_d)))
{
```

```
                for (int k = 0; k<arr_count_d; k++) {
                    dev_pivrow[k]= dev_arr[k*rownum_d+i];
                    dev_arr[k*rownum_d+i] = dev_arr[k*rownum_d+currow];
                    //dev_arr[k*rownum_d+currow] = dev_pivrow[k];
                }
                break;
            }
        }
    }
}

__global__ void pivotReplaceKernel(unsigned long *dev_arr, unsigned long *dev_pivrow,
                                   const int currow)
{
    if (0 == threadIdx.x) {
        for (int k = 0; k<arr_count_d; k++) {
            dev_arr[k*rownum_d+currow] = dev_pivrow[k];
        }
    }
}

// CPU function declarations
char** importToStr(const char* filename, const int rownum, const int symbol_bitsize,
                   const int bits_per_char_rep, const int symbol_count)
{
    /* IMPORT DATA */
    char* line;
    int tempsize = (symbol_bitsize*2/bits_per_char_rep)+2; // 2 extra bytes for \n\0
    line = (char*)malloc(tempsize*sizeof(char));    //1024bytes maximum per line imported from file
    char **inputdata;
    FILE* fr;           // pointer to file

    // Memory allocation for importing data
    // allocate for symbol_count+10=rownum.
    // allocate symbol_count*(2)/(4 bits per character): assumes symbol length =< bits/symbol.
    // +1 extra for EOS character.
    // --- *2 because random gen matrix concatenated with msg matrix
    inputdata = (char**)malloc((rownum) * sizeof(char*));
    for (int i = 0; i < (rownum); i++)
        inputdata[i] = (char*)malloc( ((symbol_bitsize*2/bits_per_char_rep)+1) * sizeof(char) );

    // Open file
    fr = fopen (filename, "rt");  // open the file for reading
    // "rt" means open the file for reading text
    if (fr == NULL)
    {
        printf("Error opening file!\n");
        exit(1);
    }
    else
        printf("File %s opened!\n", filename);

    // Import each line into array of strings, inputdata
    int import_row = 0;
    while(fgets(line, tempsize, fr) != NULL)    // max 1024 characters per line to be read per fetch;
    {
        strncpy(inputdata[import_row], line, symbol_bitsize*2/bits_per_char_rep);
        inputdata[import_row][symbol_bitsize*2/bits_per_char_rep] = '\0';  // last character
        import_row++;
    }
    fclose(fr);  /* close the file prior to exiting the routine */

    return inputdata;
}

unsigned long* parseToUL(char** inputdata, const int rownum, const int arr_count, const int hexchar_required_in_type,
                         const int hexcharnum)
{
    unsigned long* host_arr;
    host_arr = new unsigned long [(rownum*arr_count)];
```

```

        //
        /* Splitting parts of the strings into arrays of unsigned long */
        //
        char* tempstring;
        tempstring = (char*) malloc((hexchar_required_in_type+1)*sizeof(char));
        for (int i = 0, post = 0; i<arr_count; i++, post = post+hexchar_required_in_type) {
            for (int j = 0; j<rownum; j++) {
                strncpy(tempstring, inputdata[j]+post, hexchar_required_in_type);
                tempstring[hexchar_required_in_type] = '\0';
                host_arr[i*rownum+j] = strtoul(tempstring, NULL, 16);
                if (i==arr_count-1) {                       // if last array, left shift appropriately
                    host_arr[i*rownum+j]= host_arr[i*rownum+j]<<((hexchar_required_in_type-(hexcharnum%hexchar_required_in_type))*4);
                }
            }
        }
        return host_arr;
    }

    char** ULtoStr (const unsigned long* host_arr, const int rownum,  const int arr_count,
                    const int hexcharnum, const int hexchar_required_in_type)
    {
        /* Convert back to hexa-represented strings */
        char **outputdata;

        outputdata =(char**) malloc(rownum * sizeof(char*));
        for (int i = 0; i < rownum; i++)
            outputdata[i] =(char*) malloc((hexcharnum+1) * sizeof(char));
        for (int i = 0; i < rownum; i++) {
            for (int j = 0, pos = 0; j < arr_count; j++){
                char* temp = (char*)malloc(hexchar_required_in_type+1);
                char str[15], z = '0', p ='%';
                sprintf(str, "%c%c%dlx\n",p, z, hexchar_required_in_type);
                sprintf(temp, str, host_arr[j*rownum+i]);
                strncpy(outputdata[i]+pos, temp,hexchar_required_in_type);
                pos = pos+hexchar_required_in_type;
            }
            outputdata[i][hexcharnum] = '\0';
        }

        return outputdata;
    }

    void exportTxt (char* fileout, char** outputdata, const int symbol_count, const int hexcharnum)
    {
        FILE* fr;           // pointer to file
        fr = fopen(fileout, "w");
        if (fr == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
        }
        else
            printf("File %s created!\n", fileout);

        for (int i = 0; i<symbol_count; i++) {
            fprintf(fr, "%s\n", outputdata[i]+(hexcharnum/2));
        }
        fclose(fr);
    }

    void freeAllmem(char** inputdata, char** outputdata, unsigned long* host_arr, const int rownum)
    {
        for (int i = 0; i < (rownum); i++){
            free(inputdata[i]);
        }
        free(inputdata);

        for (int i = 0; i < rownum; i++){
            free(outputdata[i]);
        }
        free(outputdata);

        delete [] host_arr;
    }

    // main routine
    int main()
    {
        // Device-side pointers
        unsigned long *dev_arr;
        unsigned long *dev_pivrow;

        /* Declaration of variables */
        const int num_bits_in_type = (sizeof(unsigned long)*8);
        const int symbol_count = 1024;    // how many symbols in matrix
        const int symbol_bitsize = symbol_count; // Assuming square msg matrix
        const int bits_per_char_rep = 4;    // // assumes GF2 matrix is represented by hexa characters
        const int hexchar_required_in_type = num_bits_in_type/4;  // A hex-char represents 4 bits
        const int rownum = symbol_count+10; // 10 extra rows(symbols) needed for 99.99% PCD.
        cudaError_t cudaGEStatus;


        /* ITERATE over all the files */
        for (int iter = 1; iter < 100; iter++)
        {
            // Pass file name
            char* filename = (char*)malloc(20*sizeof(char));     //  20 means the filename can be at most 10 characters.
            sprintf(filename, "%d_%d.txt", symbol_count,iter);     // import file name format
            char **inputdata;
            inputdata = importToStr(filename, rownum,symbol_bitsize, bits_per_char_rep, symbol_count);

            int hexcharnum = 0; // Count the number of hex-characters in the 1st string, assuming each str same size
            while (inputdata[0][hexcharnum]!='\0') {
                hexcharnum++;
            }
            const int colnum = (hexcharnum)*bits_per_char_rep;
            const int last_arr_hexchar = hexcharnum%hexchar_required_in_type;
            const int arr_count = (hexcharnum/hexchar_required_in_type)+(last_arr_hexchar?1:0); // the number of arrays needed

            unsigned long* host_arr;
            host_arr = parseToUL(inputdata, rownum, arr_count, hexchar_required_in_type, hexcharnum);

            // Choose which GPU to run on, change this on a multi-GPU system.
            cudaGEStatus = cudaSetDevice(0);
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaSetDevice failed!  Do you have a CUDA-capable GPU installed?");
                goto Error;
            }

            // Allocate GPU buffers.
            cudaGEStatus = cudaMemcpyToSymbol(rownum_d,&rownum,sizeof(int));
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaMemcpyToSymbol failed!");
                goto Error;
            }

            cudaGEStatus = cudaMemcpyToSymbol(arr_count_d,&arr_count,sizeof(int));
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaMemcpyToSymbol failed!");
                goto Error;
            }

            cudaGEStatus = cudaMemcpyToSymbol(num_bits_in_type_d,&num_bits_in_type,sizeof(int));
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaMemcpyToSymbol failed!");
                goto Error;
            }

            cudaGEStatus = cudaMalloc((void**)&dev_arr, arr_count* rownum * sizeof(unsigned long));
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaMalloc failed!");
                goto Error;
            }

            cudaGEStatus = cudaMalloc((void**)&dev_pivrow, arr_count* sizeof(unsigned long));
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaMalloc failed!");
                goto Error;
            }

            cudaGEStatus = cudaMemcpy(dev_arr, host_arr, rownum * arr_count * sizeof(unsigned long), cudaMemcpyHostToDevice);
            if (cudaGEStatus != cudaSuccess) {
                fprintf(stderr, "cudaMemcpy failed!");
                goto Error;
            }

            // Loop through the data, launching XOR kernels for every row.
            int num_of_block= 1, num_of_threads = std::min(1024,rownum);
            if (rownum>num_of_threads)
            {
                num_of_block += (rownum/num_of_threads);
            }
            for (int currow = 0, curarr = 0, curcol = (arr_count*num_bits_in_type)-1; currow < symbol_count; currow++, curcol--) {
                pivotSearchKernel<<<1, 1>>>(dev_arr, dev_pivrow, curcol, curarr, currow);
```

```
                    //cudaGEStatus = cudaDeviceSynchronize();
                    XORKernel<<< num_of_block , num_of_threads >>>(dev_arr, dev_pivrow, curcol, curarr);
                    //cudaGEStatus = cudaDeviceSynchronize();
                    if (!(curcol%num_bits_in_type)){
                        curarr++;
                    }
                    pivotReplaceKernel<<<1,1>>>(dev_arr, dev_pivrow, currow);
                    //cudaGEStatus = cudaDeviceSynchronize();
                }


                // Check for any errors launching the kernels
                cudaGEStatus = cudaGetLastError();
                if (cudaGEStatus != cudaSuccess) {
                    fprintf(stderr, "addKernel launch failed: %s\n", cudaGetErrorString(cudaGEStatus));
                    goto Error;
                }

                // cudaDeviceSynchronize waits for the kernel to finish, and returns
                // any errors encountered during the launch.
                cudaGEStatus = cudaDeviceSynchronize();
                if (cudaGEStatus != cudaSuccess) {
                    fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaGEStatus);
                    goto Error;
                }

                // Copy output vector from GPU buffer to host memory.
                cudaGEStatus = cudaMemcpy(host_arr, dev_arr, rownum * arr_count* sizeof(unsigned long), cudaMemcpyDeviceToHost);
                if (cudaGEStatus != cudaSuccess) {
                    fprintf(stderr, "cudaMemcpy2 failed!");
                    goto Error;
                }

                /* Convert back to hexa-represented strings */
                char **outputdata;
                outputdata = ULtoStr(host_arr, rownum, arr_count, hexcharnum, hexchar_required_in_type);

                /* EXPORT to file */
                char* fileout= (char*) malloc(20*sizeof(char));
                sprintf(fileout, "%d_%dout.txt", symbol_count,iter);
                exportTxt (fileout,outputdata, symbol_count, hexcharnum);
        Error:
                cudaFree(dev_arr);
                cudaFree(dev_pivrow);
                freeAllmem(inputdata, outputdata, host_arr, rownum);
            }
            return 0;
        }
```