A Parallelized Implementation of Raptorq Using Nvidia Cuda

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Citable link</td>
<td><a href="https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364551">https://nrs.harvard.edu/URN-3:HUL.INSTREPOS:37364551</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at <a href="http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA">http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA</a></td>
</tr>
</tbody>
</table>
A PARALLELIZED IMPLEMENTATION OF RAPTORQ USING NVIDIA CUDA

De Zhi Huang

A Thesis in the Field of Information Technology for the Degree of Master of Liberal Arts in Extension Studies

Harvard University
March 2018
Abstract

The recent increase of users of cloud computing and internet technologies is causing the demand for data to explode. Data loss during transmission often leads delay or corruption of data packets. As example, Cisco predicts that consumer video traffic will dominate other type of traffics by 2019, taking 80% of the global market. Video streaming can be vastly affected by as little as 5% of loss causing glitches, frame drop, tearing of video. A back channel is commonly necessary to relay the loss information back to sender and loss data will be retransmitted. Since the sender has to keep track of the status of each its client, this is considered an expensive option. With the recent advancement in computer hardware, Forward Error Correction (FEC) algorithms have become a viable and economical option for protecting data against loss without using a back channel.

IETF RFC 6330 RaptorQ Forward Error Correction algorithm has gained much interest in research and practice in recent years. Its linear runtime and high data recovery probability are making it an appealing solution. This RFC uses a patented technique called Inactivation Decoding (ID method) which is a technique that combines belief-propagation and Gaussian Elimination to attain linear runtime for both its encoder and decoder. Despite its benefits, ID method is not suitable for throughput-oriented architecture hardware like the general processing Graphic Processing Unit (GPU). This project comes up with a highly parallelized implementation of RaptorQ encoder and decoder for GPU; and compares its performance against an open source version. Although at the proposed version did not outperform the ID method, this is mainly due to the limitation of my current hardware but is still sufficient to protect data in real-time. The proposed method could best the CPU implementation on newer GPU.
Acknowledgement

I would like to express my gratitude to my work mentor, Brian Sperlongano for recruiting and mentoring me on this project. Without his funding, this project would not have been possible. Next, I would like express my deepest appreciation to my thesis advisor, Jose Luis Ramirez Herran and Dr. Jeff Parker, ALM-IT Director who guided me on every step on to the completion of this thesis.
# Table of Contents

Chapter 1 - Introduction  
1.1 Problem  
1.2 Outline of Thesis  

Chapter 2 - Prior Works  

Chapter 3 - Overview of FEC  

Chapter 4 - Introducing the RaptorQ Codes  

Chapter 5 - Proposed GPU Design and Implementation  
5.1 Encoder Design  
5.2 Decoder Design  
5.3 Data Interleaver and Deinterleaver  
5.4 Hardware Dependency  
5.5 Summary  

Chapter 6 - Test setup and Evaluation  

Chapter 7 - Results  

Chapter 8 - Conclusion  

Chapter 9 - Glossary  

Chapter 10 - References
List of Figures

Figure 1 Simple Communication Model 2

Figure 2 Illustration of packet loss (source - Qualcomm RaptorQ
https://www.youtube.com/watch?v=xuAZVemBeOw) 2

Figure 3 Picture with no data loss (Right) and Picture with data loss (Left) - (source: Glitch Transition Effect https://www.youtube.com/watch?v=sZBj0eXj6hY) 3

Figure 4 Example of block FEC algorithm 7

Figure 5 The three stages of Raptor Code encoding 8

Figure 6 Graph and matrix representation from RaptorQ Technical Overview Page 4 10

Figure 7 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322 10

Figure 8 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322 11

Figure 9 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322 11

Figure 10 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322 12

Figure 11 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322 12

Figure 12 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322 13
Figure 13 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322

Figure 14 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322

Figure 15 ID example from: A. Shokrollahi and Luby “Raptor Codes” Foundations and Trends in Communications and Information Theory Vol. 6 Nos. 3 -4 (2009) 213 - 322

Figure 16 - Intermediate Symbols generation. Intermediate symbols can be resolved either via ID method or simply multiply the inverse of A by SS. L is equal to S + H + k'. P is the length of the source symbols and the intermediate symbols. P normally is the data length

Figure 17 Property of LT codes

Figure 18 - Tuple Generator or Luby Transform Code Generator

Figure 19 - An example of a sender using RaptorQ to protect the data from a binary erasure channel with 20% erasure. The green bars represent the source symbols and the red bars represent the repair symbols

Figure 20 - Comparison between the sender and the receiver matrix A. On the right, it is sender’s constraint matrix A. On the left, it is receiver’s reconstructed constraint matrix A. The receiver lost a single symbol. The red bar represents the patched repair symbol

Figure 21 - Illustration of naive matrix multiplication A by B on GPU. C block is the solution. Source:
http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda#TOC-The-matrixMul-Problem

Figure 22 - Illustration of tiling matrix multiplication of A by B. All the matrices are partitioned into submatrices. Source:
http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda#TOC-The-matrixMul-Problem
Figure 23 - Visualization of Sherman Morrison formula per a particular update. The red bar is a dense vector computed based on the update to matrix A. The runtime is quadratic because the inverse A matrix only multiplying by a row vector.

Figure 24 - Visualization of Sherman Morrison Formula for a particular element update. SM is the simplified matrix form of content inside of the parentheses of (3). i is the row index and j is the column index. m is the row index corresponded to where the update

Figure 25 - Illustration of data interleaving and deinterleaving. The above is a block of source code. Each row represents a symbol.

Figure 26 - Proposed Encoder Architecture Diagram

Figure 27 - Proposed Decoder Architecture Diagram

Figure 28 - Overview of the role of Packet Dropper

Figure 29 - Full Demo

Figure 30 - BEC Flowchart

Figure 31 - RaptorQ Encoder Flowchart

Figure 32 - RaptorQ Decoder Flowchart

Figure 33 - OpenRQ vs gRRQ Encoder Runtime

Figure 34 - OpenRQ and gRRQ Decoder Runtime

Figure 35 - Screenshot of Real-Time Video Streaming Demo
Chapter 1 - Introduction

RaptorQ Forward Error Correction (FEC) code is the most advanced version of Raptor Code that offer linear time encoding and decoding and great flexibility in error control. A new GPU-based encoder and decoder are proposed in this project.

1.1 Problem

Data losses are undesirable and inevitable effect found in information transmission. For binary files, a few missing packets can render the entire file useless. 5% to 10% loss to video stream can greatly affect user experience (Mansfield & Antonakos 2010). Today’s internet, data are transmitted in the form of a datagram packets and route through a number of medium/channels such as routers, switches, fiber cables etc... These data packets are vulnerable to network congestion, bad weather, channel fading, buffer overflow, and random noises. As a result, packets get corrupted and dropped during transmission causing the loss of information. Consider the simple model of communication in Figure 1. The sender possesses some data that need to be sent to the receiver. Sender pushes data into this channel and receiver collects the data from other end. The transfer medium is abstracted with a simple channel block. To simplify the scope of analysis, the medium (routers, fiber cables, satellite, Ethernet cables, Wi-Fi) are collectively referred to as communication channels. In practice, data passing through this block is subjected to loss or data erasure. This channel block is commonly referred to as unreliable channel or binary erasure channel (BEC). As illustrated in Figure 2, the effect of packets get dropped is causing video distortions. These distortions can vastly affect video quality. For video stream, loss and corruption result in “glitches” or tearing to the picture and audio as shown in Figure 3.
Figure 1 Simple Communication Model

Figure 2 Illustration of packet loss (source: Qualcomm RaptorQ https://www.youtube.com/watch?v=xuAZVemBeOw)
The loss of data packet is nondeterministic and is affected by network status, equipment, weather, time and varies from client to client. Clients living in remote location may experience a higher loss rate. To recover from such a loss, a back-channel is commonly employed to ensure reliable data transfer (e.g. the Transmission Control Protocol). A back channel is used by the client to relay their loss information back to the sender. So the sender can retransmit the missing packet(s). However, this is considered an expensive option because it requires the sender to allocate a small amount of resources for each of its client. In addition, this back channel is also subjected to loss and exhibited different level of erasure. For real-time broadcasting, back channels introduce additional round trip latency that could affect user experience. Hence, as the advancement and the decline of cost of computational hardware, using FEC to protect data against data loss has become a more economically option. Despite adding computational
overhead to both sender and receiver(s), FEC eliminates the need to keep a back channel. A FEC Sender could encode its data and add redundancy. A FEC receiver would reconstruct the data upon the packets that it receives. FEC algorithms can be found in many one-to-many communication (broadcast/multicast) or one-to-one (unicast) communication applications. Historically, FEC is not cheap computation wise. Although most computer nowadays have multiple cores but sacrificing one of the core to continuous perform error correction is a little expensive.

1.2 Outline of Thesis

This thesis is structured as follows. In Chapter 2, a brief mention of previous works attempt to address this problem is mentioned in Chapter 1. Then, it will follow by an overview of FEC in Chapter 3 and introduction to RaptorQ in Chapter 4. This should give the reader a general sense of FEC. In Chapter 5, this thesis will dive into the proposed design of the encoder, decoder, data interleaver, hardware dependency and a summary comparing of the current implementation versus the proposed implementation. Chapter 6 outlines the design of the experiment and the real-time video streaming demo. Chapter 7 will follow up with the results; and chapter 8 will conclude the all findings.
Chapter 2 - Prior Works

The first GPU implementation published earlier in 2013 attempted to find the inverse of constraint matrix of RaptorQ (Hu, Nooshabadi and Mladenov 2013). Their work shows promising results on GPU. They summarized that the ID method cannot satisfy the real-time requirement for the speed of the HSDPA (High Speed Downlink Packet Access) protocol when the block size is too large. In Lu et al introduced the use of Sherman Morrison decoder to find the update inverse of constraint matrix. Their work concludes their decoder can reduce computational complexity by as much as 6.5% of that required by the direct matrix inverse on CPU (Hu, Nooshabadi and Mladenov 2013). This project shall deliver a pure GPU based RaptorQ encoder and decoder.

Although Qualcomm discontinued their business and support with their RaptorQ API, an open source alternative called “OpenRQ” is available. It is a Java implementation of RFC 6330 (Lopes and Neves 2014). It will be used as a baseline RFC 6330 to compare between the GPU implementation.
Chapter 3 - Overview of FEC

The fundamental concept of FEC is to encode the data and add redundancy. FEC can be broken down into two classes - block codes and convolutional codes. Their difference is the encoder. Block codes would partition data into block(s) and process each block separately. It encodes a resultant code word by combining any of its code word in a block. On the other hand, Convolutional Codes would treat the entire data as a single dimension array and use a sliding window function to scan through this array to create resultant code word. This thesis will only focus block codes because it is more suitable for GPUs.

As illustrated in Figure 4, a block code FEC algorithm takes the binary source data and divides it into \( k \) symbols (This paper uses packets and symbols, interchangeably). These symbols are passed into an encoder to convert into \( n \) symbols where \( n \) is greater than \( k \). While transmitting packets over the unreliable channel, a portion of the symbols get thrown out. The receiver receives \( p \) symbols, where \( p \) must be at least equal or greater than \( k \) before it can decode the message. This project will refer the redundant data added during encoding stage as “repair symbols”. The received symbols \( p \) are passed to decoder to reconstruct the original \( k \) symbols of source data. Any additional symbols received more than \( k \) is considered overhead.
Despite FEC benefit, the encoding and decoding adds additional computations to both the sender and receiver. Such computational cost is a deciding factor for FEC algorithm. In 1963, Low Density Parity Check (LDPC) code was first introduced but it was then ignored because LDPC was deemed impractical to implement onto hardware at the time (Gallager 1963). Many years later, the work of Gallager gave birth to the creation of irregular LDPC code called Fountain Code/Luby Transform (LT) Codes (MacKay 2004). LT codes use a fixed-degree distribution pseudorandom number generator. This allows the encoder to generate a large amount of encoding symbols with a spread data redundancy. (Hyytia, Tirronen & Virtamo 2006). However, the encoding and decoding complexity of LT codes are still high. This motivates the invention of Raptor Codes. Raptor Code was introduced (Shokrollahi 2006). The basic idea of Raptor Code is to “precode” the source symbols into intermediate symbols, $I_S$. Then, LT code is applied to $I_S$ to generate encoding symbols. It can be broken down into 3 stages as depicted in Figure 5.

The Precoding stage is where LDPC is applied. Symbols with redundant information are generated. Next, the LT stage uses a pseudorandom number generator to pick which of the symbols to use for the encoding stage. The generator uses two separate distributions for deciding the number of symbols to use and which of the symbols to pick. The number of picked symbols is commonly referred to as “degree”. The higher
the degree, the more redundancy is created thru XOR other symbols. This lead to more encoding symbols and computations needed to decode original source symbols. Source symbols have degree of 1. Raptor Code uses Robust Soliton Distribution generator to ensure optimal degree per encoding symbol and the uniform distribution generator to decide which symbols to choose from a block (Shokrollahi & Luby 2007). Finally, the candidate symbols are combined via an exclusive-or operation to create an encoding symbol. Raptor Code was quickly adopted by Internet Engineering Task Force as IETF RFC 5053 and by the 3rd Generation Partnership Project for mobile broadcast and multicast. RaptorQ or IETF RFC 6330 was introduced in 2011 and posted as the successor of Raptor Codes.

Figure 5 The three stages of Raptor Code encoding
Chapter 4 - Introducing the RaptorQ Codes

RaptorQ advertises linear time encoding and decoding, it consists of steps that requires non-trivial computation. Per encoding or decoding, an intermediate step to solve $\text{IS}$ (a matrix that has the same dimension as the source symbols) are required. According to RFC 6330 section 5.3.3.4.2 Example Method for Calculation of $\text{IS}$, can be derived by the following identity below (1).

$$A \ast \text{IS} = \text{SS or IS} = A^{-1} \ast \text{SS} \quad (1)$$

$A$ is the square constraint matrix and $\text{SS}$ is the block of source symbols. As illustrated in Figure 6, $A$ is an L by L square matrix. This matrix begins with S rows of Low Density Parity Check (LDPC, a sparse matrix) and H rows of High Density Parity Check (HDPC, a dense matrix). As defined in RFC 6330 section 5.3.3.3, they are considered part of the precoding relationship with the $\text{SS}$. The remaining $k'$ rows are Luby Transform (LT) codes (another sparse matrix). The $\text{SS}$ matrix has L row but P column. P is the length of the packet excluding the header information. Each transaction (encode/decode) can process only $k'$ number of packets or $k'$ times P bytes. The LT codes in $A$ and the source symbols have a one-to-one relationship. This will be clarified later in the section. Instead of using (1), the current RFC 6330 uses a patented method called Inactivation Decoding (ID) Gaussian Elimination (Mladenov 2011). The ID method can simply be viewed as a technique for solving a system of linear equations. Consider the following example from Professor Amin Shokrollahi below. A graph and its associated system of equations in matrix form illustrating the relationship between constraint matrix A (the sparse matrix on the left, the intermediate symbols (all the x) and the source symbols (the entire y).
To solve for all the $x$ or the intermediate symbols, first identify the row with the degree of one (circle of 1). The entire column of that row is colored in pink indicating a particular $x$ has been solved.
Then, another row with the degree of one is selected (circle of 2).

During the next step, the remaining unselected rows have degree greater than 1. This is where inactivation part kicks in. A column is chosen to be inactive. The last column is chosen and is highlighted in blue (circle of $i_j$).
The third row has a degree of one. Hence, it is selected next.

Again, none of the remaining rows has a degree of one. Another column is chosen to be inactive. Column 4 is chosen and highlighted in blue ((circle of \(i_2\)).
Sixth row is picked as the next row with degree of one.

At last, the fourth row is picked. Note - It is completely arbitrary to pick the forth row. It can be second row or the fifth row.
Re-order the constraint matrix according to the circle number. Note: the same reordering is applied to the intermediate and the source symbols.
Row elimination is applied up to the inactive columns. The elimination steps on source symbols, $y$ to obtain $\hat{y}$ as illustrated below:

Normally, Gaussian elimination is applied to the lower right submatrix above. However, from inspection, all the intermediate symbols can be obtained as follows:

\[
\begin{align*}
x_1 &= \hat{y}_4 + x_4 \\
x_2 &= \hat{y}_6 + x_4 + x_7 \\
x_3 &= \hat{y}_7 \\
x_4 &= \hat{y}_2 + \hat{y}_5 \\
x_5 &= \hat{y}_2 + \hat{y}_5 \\
x_6 &= \hat{y}_1
\end{align*}
\]

Qualcomm advertises the total running time of RaptorQ using ID method for resolving intermediate symbols is $O(k')$, where $k'$ is the number source symbols.
Figure 16 - Intermediate Symbols generation. Intermediate symbols can be resolved either via ID method or simply multiply the inverse of A by SS. L is equal to S + H + k’. P is the length of the source symbols and the intermediate symbols. P normally is the data le

Once obtained the IS, encoding or decoding can be commenced. LT codes and IS have a very interesting property as illustrated in Figure 16. The product of LT code and IS will yield a new class of symbol called the encoding symbol. Since RaptorQ is systematic, its encoding symbols also consist of its source symbols. The non-source encoding symbols are called the Repair Symbols (RS). These RS contain redundant information of the source symbols.
The nature of the LT code is controlled by tuple generator or LT code generator defined by RFC 6330 section 5.3.5.4. As depicted in Figure 18, it takes two positive integers, \( k' \) and Internal Symbol Identifier, \( ISI \). For \( ISI \) greater than or equal to \( k' \), the generated LT codes yield \( RS \) when multiplying by \( IS \). While for those \( ISI \) less than \( k' \), the generated LT code yields \( SS \) when multiplying by \( IS \). The LT codes and the \( RS \) also have a one-to-one relationship like \( SS \). During encoding, the \( k' \) rows of \( A \) consist of generated LT codes less than \( k' \). Once resolved \( IS \), the sender estimates the channel loss and determine the number of \( RS \) to generate. A general formula to determine the number of \( RS \) to generate or \( S_{\text{repair}} \) can be expressed as (2):

\[
(2)
\]
\[ S_{\text{repair}} = \frac{k' + S_{\text{overhead}}}{(1 - P_{\text{erasure}})} - k' \] (2)

\( S_{\text{overhead}} \) is the number of overhead symbols and \( P_{\text{erasure}} \) is the estimated probability of channel erasure.

Consider a simple example in Figure 19. A sender estimates a channel with 20% erasure. In other words, this BEC will randomly drop 2 packets for every 10 packets that it sees. On the right hand side, receiver 1 received all the source symbols. It does not need to go through the decoding process. Meanwhile, receiver 2 was less fortunate because its 2nd and 5th source symbols were dropped by the BEC. But it was able to patch the two missing symbols with 2 repair symbols.

Figure 19 - An example of a sender using RaptorQ to protect the data from a binary erasure channel with 20% erasure. The green bars represent the source symbols and the red bars represent the repair symbols

To recover the two source symbols, receiver 2 resolves \( IS \) via using the ID method. Receiver 2’s constraint matrix \( A \) will have to update its LT codes like the receiver in Figure 20 before solving for \( IS \).

It has to update original LT codes will be replaced by the two LT codes corresponded to the \( RS ISI \). Then, ID method is applied to derive \( IS \). Once obtained \( IS \), receiver 2 may generate LT codes with \( ISI \) matching the two patched symbols and have them multiply by \( IS \) to recover the original source symbols. In some
cases, the patching \( RS \) might not contain the necessary information to re-derive \( IS \) on the receiver’s end. This problem arises when trying to invert \( A \) with the \( RS \) respective LT codes. If “extra” \( RS \) are available, they may be used in place. These extra \( RS \) that a receiver receives is known as the “overhead”. RaptorQ guarantees that with 2 overhead symbols, it can decode the original data with 99.9999% probability.

Figure 20 - Comparison between the sender and the receiver matrix \( A \). On the right, it is sender’s constraint matrix \( A \). On the left, it is receiver’s reconstructed constraint matrix \( A \). The receiver lost a single symbol. The red bar represents the patched repair symbol.
Chapter 5 - Proposed GPU Design and Implementation

The aim of this project is to produce a RaptorQ design with higher data encoding and decoding throughput. This project chooses general purpose NVIDIA GPU because its massive parallel architecture and commercially available in most laptops and desktops. This project uses both C++ and NVIDIA CUDA C. It will be delivered as a dynamic link library (.DLL) in Windows and shared library (.SO) in Linux.

\( \mathbf{IS} \) can be resolved with either ID method or via dense multiplication. ID method is not suitable for GPU because it is an iterative method consists of Gaussian Elimination and Belief Propagation methods (Shokrollahi & Luby 2007). The alternative method is to use the identity from (1) but the inverse of \( \mathbf{A} \) and the multiplication to the \( \mathbf{SS} \) incur \( O(k'^3) \) runtime. This runtime is based on the assumption that the platform is a CPU. For GPUs, the matrix multiplication runtime is \( O(k') + \) overhead where overhead is the time it requires to copy to and from the GPU global memory. Unlike RFC 6330 ID method, the proposed encoder and decoder uses a different architecture. Consider the design below.

5.1 Encoder Design

One question arises is the inverse of \( \mathbf{A} \) will still add a cubic runtime to the overall runtime. Interestingly, the encoder is not necessary need to solve for inverse of \( \mathbf{A} \) at every transaction. \( \mathbf{A} \) is remaining constant. Since only \( \mathbf{SS} \) would be subject to change at every transaction, \( \mathbf{A} \) inverse can be solved once and retain in the memory. Then, the \( \mathbf{IS} \) for encoder is simply just multiplying \( \mathbf{A}^{-1} \) to \( \mathbf{SS} \).
There are several techniques for GPU multiplication; and they are - naive, tiling, coalescing, avoid memory bank conflicts, computation optimization, loop unrolling, and pre-fetch (Ye 2015). Consider the Figure 21 below.

![Figure 21 - Illustration of naive matrix multiplication A by B on GPU. C block is the solution. source: http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda#TOC-The-matrixMul-Problem](http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda#TOC-The-matrixMul-Problem)

Each of the elements in C is assigned to a GPU thread. Each thread copies a row of A and a column of B, then multiply the respective elements and sum up the products. Hence, the theoretical runtime is $O(k')$. Instead of using naive implementation, this project chooses to start with tiling because it is the foundation to the other advanced techniques. Tiling implementation has shown to be more efficient than the naive method and it makes use of the NVIDIA shared memory, which is the second fastest cache memory on the GPU to avoid unnecessary global memory copying. Note: $A^T$ and $S$ resides in GPU global memory.

Consider Figure 22 below.
Matrix A, B and C are all partitioned into submatrices. Differ from the naive method; the GPU thread in submatrix C has access to every element in submatrix A and submatrix B. This reduces the number of times to access the global memory because each thread will copy its assigned elements from A and B then share with others. If time permits, more advanced optimization to matrix multiplication will be included.

The other part of encoding is the generation of RS. Figure 5 has shown that a RS can be obtained by multiplying LT codes vector with ISI equal to or greater than $k'$ by IS. LT code generated by the Tuple Generation in Figure 6 is a sparse vector. It consists of elements with either 1s or 0s. When a large number of RS is needed, this is equivalent to multiplying a sparse matrix by a dense matrix (IS).

Applying the previous tiling matrix multiplication would waste GPU memory and incur unnecessary memory copy to the shared memory cache. Hence, this project shall represent the LT codes in Compressed Sparse Row (CSR) format. Bell et al in his studies shows the GPU sparse matrix multiplication kernels outperform quad-core Intel Xeon system by almost ten times. This project shall implement their CSR sparse matrix multiplication kernel (Bell & Garland 2008). If time permits, other sparse matrix multiplication kernels will be implemented and evaluated.
5.2 Decoder Design

Unlike encoder and without the ID method, decoder will require finding the inverse of $A$ whenever it encounters a missing source symbol to its block. The matrix $A$ of the receiver with patched $RS$ would be having a different LT codes at the patched row as illustrated in Figure 9. The receiver reconstructed constraint matrix, $A'$ is not so much different from $A$. Solving the inverse of matrix would add additional $O(k^3)$ runtime to the overall runtime.

An alternative method is to use Sherman Morrison formula (3). It is a numerical technique allowing one to find the update of inverse $A$ where $u$ and $v$ are perturbation vectors. This technique has $O(k^{2})$ runtime per update as illustrated in Figure 23.

\[ A_{update}^{-1} = (A + uv^T)^{-1} = A^{-1} (I - \frac{uv^TA^{-1}u}{1 + v^TA^{-1}v}) \]  

(3)

The denominator of (3) can be used to detect if the patched $RS$ contains the required information to derive $IS$. If the denominator is 0, then the receiver may swap out the patched $RS$ and avoid performing the expensive multiplication of $A'$ operation.

Figure 23 - Visualization of Sherman Morrison formula per a particular update. The red bar is a dense vector computed based on the update to matrix $A$. The runtime is quadratic because the inverse $A$ matrix only multiplying by a row vector.
On GPU, the update multiplication step can be achieved in constant time. Consider Figure 13.

![Figure 24 - Visualization of Sherman Morrison Formula for a particular element update. SM is the simplified matrix form of content inside of the parentheses of (3). i is the row index and j is the column index. m is the row index corresponded to where the update](image)

Each GPU thread at their index (i, j) can simultaneously compute its solution. This kernel is called Sherman Morrison kernel. After resolved the update of inverse $A$, tiling multiplication is apply to re-derive $IS$. Missing source symbol(s) can be recovered by multiplying the LT codes generated for the corresponded $ISI(s)$.

### 5.3 Data Interleaver and Deinterleaver

Interleaving is a technique commonly employed on top of FEC algorithm to help protecting data against burst loss by reshuffling the data as depicted in Figure 15. This allows better distribution of error to the data. This technique will increase the chance of recovering source symbols. When burst error occurs, a consecutive number of packets get dropped. Without interleaving, an entire consecutive of information will be missing. The repair symbols may not be able to cover those losses. Hence, the decoder will fail to decode for $IS$. On the other hand, with interleaving, only a fragment will be missing of the overall data will be missing. The likelihood of decoding failure is greater reduced. For pixelated images or video with a few missing pixels can hardly affect user’s experience.
Currently, OpenRQ does not support this feature but it is believed that Qualcomm’s version support interleaving. Data interleaving is applied prior to the encoding stage and data deinterleaving is applied post the decoding stage. If time permits, a constant runtime interleaving and deinterleaving kernels will be added to the implementation.

5.4 Hardware Dependency

This project heavily exploits the features of NVIDIA GPU such as Streams (since CUDA 5), unified memory (introduced in CUDA 6 and updated in CUDA 8) and texture memory. CUDA Streams are basically the equivalent of threads in CPU. NVIDIA introduces asynchronous Streams to CUDA 7. This new framework enables better distribution of tasks and dynamic parallelization. This project uses NVIDIA Tesla K40. The unified memory is a feature that allowing programmer to create a single copy of memory that is being shared by both CPU and GPU. The transfer of memory is being handled implicitly by the API itself. Finally, this project uses a separate class of memory on GPU called “texture memory”. It is a read-only memory and commonly used for store texture for gaming 3D models. According to RFC 6330, it recommends to use finite field log and exponential tables to compute the multiplication and division. Instead this project implements a texture memory based lookup multiplication table. Studies shows texture memory has better performance than GPU global memory for read-only data (Iandola, Sheffield, Anderson, Phothilimthana and Keutzer 2013).
5.5 Summary

To summarize, the proposed encoder has the following step to resolve for intermediate symbols.

1. Copy the source data/symbols into pre-allocated GPU memory buffer
2. Perform interleaving (invoke Interleaving Kernel optional)
3. Multiply the source data by the pre-computed $A^t$ matrix to solve for $IS$ (invoke Tiling Matrix Multiplication Kernel)
4. Multiply LT codes who $ISI$ greater than or equal to $k'$ to obtain $RS$ matrix (invoke Sparse Matrix Multiplication Kernel)

The proposed decoder will resolve intermediate symbols according to the following:

1. Copy the received source data/symbols into pre-allocated GPU memory buffer
2. Sequentially resolve for the updated version of $A^t$ (invoke Sherman Morrison Kernel) and store loss information
3. Multiply the source data by the updated $A^t$ matrix (invoke Tiling Matrix Multiplication Kernel)
4. Recover missing source symbol(s) (invoke Sparse Matrix Multiplication Kernel)
5. Perform de-interleaving (invoke Interleaving Kernel optional)
Of course, solving for \( IS \) is merely an intermediate step of the encoder and decoder to generate either encoding symbols or re-generate source symbols, respectively. The encoder \( IS \) would be multiplied by another sparse matrix to generate a \( RS \) matrix. The decoder intermediate symbols would be multiplied by a sparse matrix whose dimension corresponds to the number of loss symbol to recover the information. De-interleaving shall be run after the recovery of loss symbol(s). A table summarizing the runtimes between the RFC 6330 and the proposed design is shown below. At first glance, the proposed decoder may be solver than the ID method but for longer symbol length, it may favor the GPU decoder.
Chapter 6 - Test setup and Evaluation

Finally, to test and to evaluate the RFC 6330 and the proposed design, a tunable packet dropper program will emulate a BEC. The test setup comprises of 3 separate processes. The encoder process take in any binary object and encode it into RaptorQ encoding UDP packets. These packets will be multicast to a socket. Meanwhile, the packet dropper subscribes to that multicast socket and buffer up the encoding symbol packets. Once the dropper has received sufficient encoding symbol packets, it will randomly remove a number of packets and forward the rest to the decoder as illustrated in Figure 28. The decoder will attempt to recover the original data message. Delay/latency and throughput will be measured and used to evaluate the performance.

![Figure 28 - Overview of the role of Packet Dropper](image.png)

Tests are written in unit test case. A separate project called gRRQunitTest (for C++) and another project called OpenRQTest (for Java). Please see appendix section for unit test code used for comparing performance.

Real –time Video Streaming Demo
In addition, a demonstration application is created for this thesis to illustrate its capability in performing real-time error correcting. This demo basically uses FFmpeg to stream a video via UDP multicast. Two channels are created – one with the proposed design FEC protection and another without. This will allow easier compare and contrast of video quality. The full demo schematic is illustrated in Figure 29.

A video clip will be streamed to 225.0.0.2:1234. Video at this address has no packet loss. Next, two instances of simulated BEC are created. One of the BEC will subscribe directly to 225.0.0.2:1234. BEC’s job is to consume UDP packets, randomly remove the packets and send out the remaining to 225.0.0.4:11. The design of the BEC is illustrated in Figure 30. At this stage, the client that plays the video from this address will experience glitches. But this channel will act as a control for our demo.

Meanwhile on the other channel, a RaptorQ encoder (flowchart in Figure 31) will consume packets from 225.0.0.2:1234, convert the consumed packets into encoded packets and multicast it back out to 225.0.0.3.
Now the other BEC is generated, consumes the packets, remove some of the packets and send out the remaining to 225.0.0.3:6. In this stage, the packets are still encoded packet. Hence, a RaptorQ decoder (flowchart in Figure 32) is created consumes the packets from BEC, decode message and send the original data to 225.0.0.4:10. Client that subscribes to this address shall experience little to no video glitches.
Figure 31 - RaptorQ Encoder Flowchart
Figure 32 - RaptorQ Decoder Flowchart
Chapter 7 - Results

A test project is created for both OpenRQ and this project (gRRQ). Each test is expected to have 50% channel erasure. Block size range from 10 to 800 is used. The runtime has exceeded 1 second for block size of 800. Hence, this project stops evaluation at this block size. When the block size is less than or equal to 200, the runtime begins to deviate. gRRQ has comparable performance to OpenRQ.

Figure 33 - OpenRQ vs gRRQ Encoder Runtime
According to NVIDIA Visual Profiler, the dense matrix multiplication is the most time consuming process. It accounts for 80% of the runtime. However, with block size of 100, the proposed design is capable of performing real-time video forward error correction. As illustrated in Figure 35, with 50% channel erasure, the control video stream can barely view the video (bottom right). Meanwhile, the protected stream (top right) quality is same as the source (center left) except there is about 1 ~ 2 seconds delay. This is mainly due to the need for emulating channel loss between the encoder and decoder. In actual deployment, the delay should be negligible.
Figure 35 - Screenshot of Real-Time Video Streaming Demo
Chapter 8 - Conclusion

RaptorQ is a block forward error correction algorithm that is commonly used for protecting data loss or corruption. Some notable application includes living high definition video streaming, massive online multiplayer video game, GPS, file storage and etc… from data loss during transmission. Although the proposed method did not outperform the DI method, it provides an alternative implementation of RFC 6330. Error correction is known to be computation expensive. The proposed implementation offer a new method that offload computation to highly parallelized computation platform such as the GPU. The main drawback of this method is the dense matrix multiplication which could take up 80% or above of the computation time. When block size is less than or equal to 200, the proposed encoder and decoder are comparable to OpenRQ. As one can see from the real-time video streaming demo, even with block size of 100, it is still capable of protecting data in real-time. The proposed design offloads most of the computation to GPU. This implementation increases utilization of external hardware and reserve CPU resource for other tasks. This GPU used for this project is NVIDIA Tesla k40c with a clock rate of 875 MHz. The time this thesis is written, many new generations of GPUs have release; and the clock rate has near tripled. Hence, the proposed method could potentially best the DI method in newer GPU with high clock rate.
Chapter 9 - Glossary

- BEC - Binary Erasure Channel
- IETF - Internet Engineering Task Force
- RFC - Request for Comments
- GPU - Graphics Processing Unit
- CUDA - Compute Unified Device Architecture
- GE - Gaussian Elimination
- FEC - Forward Error Correction
- IS - Intermediate Symbols
- ES - Encoding Symbols
- LDPC - Low-Density Parity Check
- HDPC - High-Density Parity Check
- ID - Inactivation Decoding
- TCP - Transmission Control Protocol
- UDP - User Datagram Protocol
- AL - Application Layer
- IP - Internet Protocol
- HSDPA - High Speed Downlink Packet Access
- DVB-H - Digital Video Broadcasting - Handheld
Chapter 10 - References


Mackay, D. (2004). Fountain codes. IEE Seminar on Sparse-Graph Codes (Turbo Codes, Low Density Parity-Check Codes and Fountain Codes). doi:10.1049/ic:20040504

Patent US7956772 - Methods and apparatus employing FEC codes with permanent inactivation of

47
symbols for encoding and decoding processes. (n.d.). Retrieved May 02, 2017, from
https://www.google.com/patents/US7956772


   Euro Med Telco Conference (EMTC). doi:10.1109/emtc.2014.6996641

Chapter 11 - Appendix