Lossless Data Compression by Representing Data as a Solution to the Diophantine Equations

by

Karandeep Singh Grewal

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Javier Gonzalez Sanchez, Chair
Ajay Bansal
Michael Findler

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

There has been a substantial development in the field of data transmission in the last two decades. One does not have to wait much for a high-definition video to load on the systems anymore. Data compression is one of the most important technologies that helped achieve this seamless data transmission experience. It helps to store or send more data using less memory or network resources. However, it appears that there is a limit on the amount of compression that can be achieved with the existing lossless data compression techniques because they rely on the frequency of characters or set of characters in the data. The thesis proposes a lossless data compression technique in which the data is compressed by representing it as a set of parameters that can reproduce the original data without any loss when given to the corresponding mathematical equation. The mathematical equation used in the thesis is the sum of the first N terms in a geometric series. Various changes are made to this mathematical equation so that any given data can be compressed and decompressed. According to the proposed technique, the whole data is taken as a single decimal number and replaced with one of the terms of the used equation. All the other terms of the equation are computed and stored as a compressed file. The performance of the developed technique is evaluated in terms of compression ratio, compression time and decompression time. The evaluation metrics are then compared with the other existing techniques of the same domain.

DEDICATION

To my mother and father for standing by me during the good and bad times.

To my family and friends for recognizing the flair inside me and pushing me forward.

ACKNOWLEDGMENTS

I want to express my deepest gratitude to the committee chair, Dr. Javier Gonzalez Sanchez, for all the guidance and knowledge he shared with me during my thesis research. There were many highs and lows during the research work. Still, he always motivated me to give my best and helped me believe that it is never too late to find a new direction and set another goal.

Secondly, I am thankful to Dr. Michael Findler and Dr. Ajay Bansal for reviewing my work and sharing their suggestions about the different things that helped improve the research's quality and scope. I would like to appreciate how Dr. Findler offers just the right amount of guidance to the students that initiate the interest in a topic and automatically push the student to self-study the topic in detail. I will also like to admire how Dr. Bansal offers a unique perspective to any topic he teaches. It allows the student to think out of the box and discover new ways of learning things.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Data Compression 1

1.2 Importance of Data Compression 2

1.3 Existing Solutions 3

1.3.1 Shannon-Fano Algorithm 3

1.3.2 Huffman Coding 6

1.3.3 Lempel-Ziv Algorithm - LZ77 10

1.4 Challenges 10

1.5 Proposed Solution 11

1.6 Evaluation Plan 13

1.7 Document Structure 14

2 BACKGROUND 15

2.1 Diophantine Equation 15

2.2 Geometric Series 16

2.3 Repdigits & Repunits 17

2.4 Brazilian Numbers 17

2.5 Parallel Programming 18

2.5.1 Rise of GPU Programming 19

2.5.2 Task Based Parallelism 20

2.5.3 Data-Based Parallelism 21

2.5.4 How GPU increases performance? 22

2.5.5 Programming Model - CUDA 22
2.5.6 Restrictions in CUDA Programming 26
2.6 Big Integer Calculations using GMP 26
3 APPROACH - PROPOSED SOLUTION 28
3.1 Non-Diophantine Approach 28
3.2 Diophantine Approach 30
3.2.1 Repunit Approach 30
3.2.2 Repdigit Approach 31
3.2.3 Neighbors Check 32
3.2.4 Padding Approach 33
3.3 Optimizing the Repunit Approach 35
3.4 Sliding Window Approach 35
4 IMPLEMENTATION 38
4.1 Wrapper over the GMP Library 38
4.2 Non-Diophantine Approach 38
4.3 Repunit Approach 42
4.3.1 CPU Implementation of the test 42
4.3.2 GPU Implementation 46
4.4 Repdigit Approach 49
4.5 Neighbors Approach 49
4.6 Padding Approach 51
4.7 Reducing Memory Complexity by variable reuse 52
4.8 Best Case Data Generation 54
4.9 Structure of the program 56

4.10 Encoding the variables in the compressed file 57
5 RESULTS AND EVALUATION 60
5.1 Results 60
5.1.1 Repunit Approach 60
5.1.2 Repdigit Approach 63
5.1.3 Improvements in memory requirements with variable reuse 64
5.1.4 Comparing neighbors and padding approach 65
5.2 Evaluation 65
5.2.1 Test 1 - Repunit Approach Only 66
5.2.2 Test 2 - Repdigit Approach 67
5.3 Comparison with existing algorithms 67
5.4 Conclusion and Future Work 69
5.4.1 Limitations and Potential Applications 70
5.4.2 Calculations directly on Binary or Hexadecimal Numbers 70
5.4.3 Dividing the data into smaller divisions 71
5.4.4 Effect of value of N on run time 71
5.4.5 Implementing the algorithm on GPU 72
5.4.6 Trying different factorization techniques 72
5.4.7 Primality Check to avoid unnecessary factorization 73
REFERENCES 75

LIST OF TABLES

Table | Page

1.1 Memory Requirements and Transmission Time of a Video File - With and Without Compression 3
1.2 Table of Characters and their Frequencies 5
1.3 Sorted Table of Characters and their Frequencies 5
1.4 Assigning the Digits in Each Iteration - Shannon Fano Algorithm 5
1.5 Final Encoding Table - Shannon Fano Algorithm 6
1.6 Table of Characters and their Frequencies after First Iteration for Step 4 - Huffman Coding 8
1.7 Table of Characters and their Frequencies after Iteration Two for Step 4 in the Huffman Coding 8
1.8 Final Encoding Tree - Huffman Coding 9
1.9 Comparing Various Compression Techniques for a File containing value of π upto One Billion Digits 12
4.1 Illustrating the two methods of work distribution 46
4.2 Order of Parameters in the Compressed File for Different Approaches 59
4.3 Binary Representation for the Used Encoding 59
5.1 Specifications of System used for both CPU and GPU Tests 61
5.2 Test Results from CPU Implementation for Repunit Approach 62
5.3 Test Results from GPU Implementation 63
5.4 Memory Requirements of Older Version of the program 64
5.5 Evaluation Metrics for Test 1 67
5.6 Evaluation Metrics for Test 2A 68
5.7 Evaluation Metrics for Test 2B 68
5.8 Comparison of developed approach with the existing techniques 69

LIST OF FIGURES

Figure Page

1.1 Huffman Tree after First Iteration of Step 4 - Huffman Coding 7

1.2 Huffman Tree after Iteration Two for Step 4 in the Huffman Coding 8

1.3 Final Tree - Huffman Coding 9

2.1 Task Based Parallelism (Applying Three Effects to Every Frame of a Video) 21

2.2 Data Based Parallelism (Applying Three Effects to Every Frame of a Video) 23

2.3 Increase in Performance of CPU & GPU from the Year 2006 to 2012 24

2.4 Thread and Memory Hierarchy on a CUDA Device 25

3.1 Illustration of the Window Approach 36

4.1 Wrapper Code over the GMP variables to avoid Unwanted Memory Leaks 39

4.2 Checking if a Given Number is a Repunit in Non-Diophantine Approach. 40

4.3 Changing the Values of `base` and `n` in Non Diophantine Approach 41

4.4 Checking if a Given Number is Repunit in Diophantine Approach 43

4.5 Distributing the numbers in different threads as continuous ranges 44

4.6 Generating the numbers to be tested as Discontinuous Distribution 45

4.7 Kernel Code for GPU 47

4.8 Launching the GPU Kernel 48

4.9 Checking if a Given Number is a Repdigit in Diophantine Approach 50

4.10 Code Snippet for Neighbors Approach.................................. 51

4.11 Code Snippet for Padding Approach 53

4.12 Code for Generating the Best Case Data for Repunit 55

4.13 Pipeline for the Process of Compression and Decompression 57

4.14 Pseudocode for the Merged Approach 58

5.1 Image used for Test 1 66

5.2 Image used for Test 2A (left) and 2B (right)................................ 68

CHAPTER 1

INTRODUCTION

We have seen a massive increase in the amount of digital data in the last two decades. According to an IDC Report [14], the global digital data surpassed the zettabyte barrier in 2010. One zettabyte equals 10^{12} Gigabytes. Another IDC report [26] states that the total digital data was approximately 45 zettabytes in 2019 and is expected to grow to 175 zettabytes in 2025. This means the total digital data will grow by a factor of 175 in a period of just 15 years. Several attempts have been made to avoid this problem of rapidly increasing data. One way to deal with this problem is to extract all the important knowledge from the present digital data and discard the data that is not required. However, the knowledge extraction techniques are in a very early stage, and better techniques are yet to come.

Therefore, discarding the extra data may not be an ideal option for now and needs to be stored. Several attempts have been made to store the data in a smarter and more compact manner. This is commonly known as data compression. It reduces the file size so that more data can be stored in the same memory space. Data compression also helps in transmitting more amount of data using the same network bandwidth.

1.1 Data Compression

Data compression consists of two processes: Compression and Decompression. Compression refers to the process that takes the input data X and converts it into another representation X_C that requires fewer bits. A bit is the smallest unit of memory for computers and can have a value of 0 or 1. Decompression refers to the process of generating the representation Y from X_C. This regenerated representation Y can or cannot be equal to X, depending on the used data compression technique. If Y is not

equal to X ($X \neq Y$), there is some loss of data during the compression process.

Data compression in which there is some loss of data is called Lossy Data Compression. Y is not equal to X in lossy compression, but it is identical to X up to some level of detail. The most common areas of use for lossy compression are image, audio, and video files because some level of data loss is acceptable in these files, and both the compressed and original representation are equally meaningful for the human senses. The compression in which there is no loss of data is called Lossless Data Compression. Generally, text files are compressed using lossless techniques because any change in these files can make them completely useless for the end users.

1.2 Importance of Data Compression

The main aim of a data compression technique is to reduce the size of a file and generate a compressed file in an acceptable amount of time. The same applies with the decompression phase that the technique should be able to reproduce the decompressed file in an acceptable amount of time. This reduction of file size is most beneficial for data storage and transmission purposes. Data storage means storing the data in a storage device like hard drives or pen drive. Data Transmission refers to transmitting data from one location to another on the same or another computer system. Data compression helps in reducing the costs of storage and transmission. Table 1.1 shows what a two-minute high-definition original video means in terms of memory requirements and transmission time. The video file is compressed using H.264 and H.265, two of the most widely recognized video compression standards. The file compressed using these compression standards (Table 1.1) is approximately 5 to 7 times smaller than the original file.

Frame Rate	30 frames per second
Video Length	2 minutes
Video Resolution	1920 x 1080
Color Depth	16 bit
Memory - Without Compression	13.9 Gigabytes
Memory Requirements - Compressed using H.264	27.3 Megabytes
Memory Requirements - Compressed using H.264	18.7 Megabytes
Transmission Speed	10 Megabits per second
Transmission Time - Without Compression	3.16 Hours
Transmission Time - H.264	20.82 Seconds
Transmission Time - H.265	14.26 Seconds

Table 1.1: Memory Requirements and Transmission Time of a Video File - With and Without Compression

1.3 Existing Solutions

The history of data compression dates back to the work of Shannon, Fano, and Huffman work in the late 1940s to early 1950s [19]. Numerous data compression techniques have been found since then, and each of them has succeeded in showing great performance in data compression. This thesis mainly deals with the lossless data compression, and the major existing techniques used for lossless data compression are discussed in this section.

1.3.1 Shannon-Fano Algorithm

ASCII stands for American Standard Code for Information and is used to represent textual data in computers. ASCII Table uses 8 bits for one character and contains 256

(2^8) characters. Example: According to ASCII, the letter *a* is represented as 97 in decimal number system or 01100001 in binary number system. Shannon Fano algorithm is one of the earliest methods that define an encoding table that decreases the average number of bits required to store a character in the textual data. The algorithm is as follows [20]:

1. Create a table of all the characters and their frequencies in the data. See table 1.2.
2. Sort the table in the order of decreasing frequency. See table 1.3.
3. Divide the table into two divisions such that frequencies in the upper division are almost equal to the sum of frequencies in the lower division .
4. Assign the digit 0 to the upper division, and the digit 1 to the lower division.
5. Repeat steps 3 and 4 recursively, to the upper part and lower parts until each division has only character. See table 1.4.
6. Use the digits assigned to each division in all the iterations to build the final encoding table. See table 1.5

According to table 1.5, E can now be represented as '00'. In the original form (ASCII), E requires 8 bits. However, Shannon-Fano algorithm decides to give E a shorter code because the frequency of E is higher than the other characters. This process of giving short-length codes to more frequent characters results in data compression. Also, the encoding table is created based on the characters present in the data. Example: If only two type of character (say A and B) are present in the data, then each of the characters can be represented by just one bit each for each character.

Character	Total Frequencies
A	15
B	10
C	9
D	7
E	21
F	10

Table 1.2: Table of Characters and their Frequencies

Characters	Total Frequencies
E	21
A	15
F	10
B	10
C	9
D	7

Table 1.3: Sorted Table of Characters and their Frequencies

Characters	Total Frequencies	1st Iteration	2nd Iteration	3rd Iteration
E	21	0	0	
A	15	0	1	
F	10	1	0	0
B	10	1	0	1
C	9	1	1	0
D	7	1	1	1

Table 1.4: Assigning the Digits in Each Iteration - Shannon Fano Algorithm

Memory Requirements with Shannon Fano Algorithm

$$\begin{aligned}\text{ASCII Table (Without Compression)} &= \text{Total Characters} \times \text{Number of bits per character}\\ &= (15+10+9+7+21+10)\times 8\\ &= 72\times 8\\ &= 576 \text{ bits}\end{aligned}$$

Character	Frequency	New Code	Code Length
E	21	00	2
A	15	01	2
F	10	100	3
B	10	101	3
C	9	110	3
D	7	111	3

Table 1.5: Final Encoding Table - Shannon Fano Algorithm

$$\begin{aligned}
\text{Shannon Fano Encoding Table} &= \sum(\text{Frequency} \times \text{Code Length}) \\
&= ((21 \times 2) + (15 \times 2) + (10 \times 3) + (10 \times 3) + (9 \times 3) + (7 \times 3)) \\
&= 42 + 30 + 30 + 30 + 27 + 21 \\
&= 180 \text{ bits}
\end{aligned}$$

1.3.2 Huffman Coding

Huffman Coding [16] is an optimum method developed by David A. Huffman, for minimizing the number of bits required to store a set of characters in text data. This technique was developed in the class assignment of the first ever class in the field of information theory taught by Robert Fano. That is why it is very closely related to the Shannon-Fano Algorithm. Huffman Coding also aims to create a new code table that gives short-length codes to more frequent characters, but the process is different from Shannon-Fano Algorithm. The Huffman Coding algorithm is as follows:

1. Create a table of all the characters in the data and their frequencies 1.2.

2. Sort the table in the order of decreasing frequencies 1.3 .

3. Create a leaf node for each character in the table from step 2. Use both the character and its frequency as the label (<CHARACTER, FREQUENCY>) for the leaf node. These leaf nodes are joined to create the final huffman tree in the later steps.

4. Find the two nodes with the least frequency (lowest nodes in the table) and join them together into a parent node. The label of the parent node is the combination of labels of the child nodes (<CHARACTER1 + CHARACTER2, FREQUENCY1 + FREQUENCY2>). The two nodes with the least frequencies are removed from the table, and the new parent node is inserted. The table is again sorted in the order of decreasing frequencies after the new parent node is added .

5. Repeat step 4 until all the characters are joined together, and it forms a tree.

6. Label all the left edges of the tree as 0, and the right edges as 1. See figure 1.3.

7. Build the final encoding table using the digits assigned in step 6. See table 1.8..

The same character and frequency table from Shannon-Fano Algorithm's illustration is used to explain Huffman Coding. See Table 1.3.

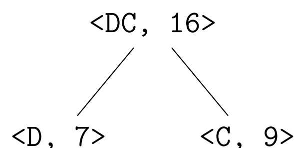

Figure 1.1: Huffman Tree after First Iteration of Step 4 - Huffman Coding

D and C are the nodes with the least frequency in table 1.3. They are removed from the sorted table and combined into a single node as <DC, 16>. See table 1.6 and figure 1.1.

Character	Total Frequencies
E	21
DC	16
A	15
F	10
B	10

Table 1.6: Table of Characters and their Frequencies after First Iteration for Step 4 - Huffman Coding

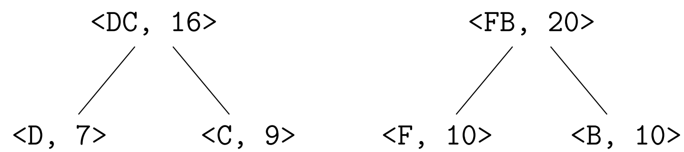

Figure 1.2: Huffman Tree after Iteration Two for Step 4 in the Huffman Coding

Character	Total Frequencies
E	21
FB	20
DC	16
A	15

Table 1.7: Table of Characters and their Frequencies after Iteration Two for Step 4 in the Huffman Coding

Step 4 is repeated until all the nodes are joined into a single tree. All the left edges are labeled as 0, and the right edges are labeled as 1. These labels are used to find the code for each character. The final tree and encoding table are given in table 1.8 and figure 1.3.

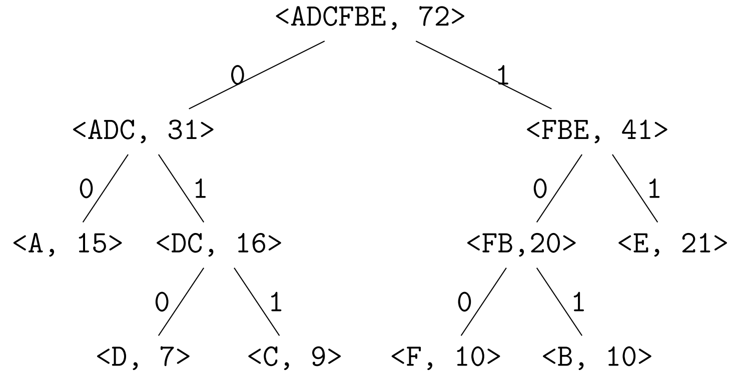

Figure 1.3: Final Tree - Huffman Coding

Character	Huffman Code
A	00
D	010
C	011
F	100
B	101
E	11

Table 1.8: Final Encoding Tree - Huffman Coding

Huffman Coding differs from the Shannon Fano approach in the order in which both of them assign codes to the nodes of the tree. Shannon Fano works in a top-down approach whereas Huffman Coding starts from the leaves of the tree and work in bottom-up manner [20]. Both Huffman Coding and Shannon Fano Algorithm requires the encoding table generated in the compression process to decompress the data.

1.3.3 Lempel-Ziv Algorithm - LZ77

LZ77 is another technique for lossless data compression and was created by Abraham Lempel and Jacob Ziv. It aims to find the repeating set of characters in a text. If there is a set of characters S and it repeats for n number of times in the text, LZ77 stores the first occurrence of S and use a pointer (pointing to the first occurrence) at other places of occurrence. The data compression in LZ77 lies in the fact that memory required to store the pointer is less than the data the pointer is pointing to. [29] [38]

LZ77 was the first algorithm to introduce the concept of 'sliding window' [25]. The sliding window is the area in which LZ77 looks for the repetition at any given time .Example: There is a textual data like "...Walter is going to Walt Disney...". Here, the word 'Walt' is repeated two times. According to the LZ77 Algorithm, the first occurrence is stored as it is, and the second occurrence is stored as a pointer <Jump Back Distance, Length>. Jump back distance is the number of characters between first occurrence and N^{th} occurrence of the repeating word ('Walt'). <Length> corresponds to the number of characters in the repeated word. According to this algorithm, this text data will be stored as "...Walter is going to <19,4> Disney". The actual representation of the pointer is not exactly <19,4> but an optimized way of representing the same thing. In LZ77, any repetition of characters is avoided, decreasing the memory required to store the same data. LZ77 does not requires a separate encoding table like Huffman Coding or Shannon Fano Algorithm because the pointers are sufficient to regenerate the original data.

1.4 Challenges

The compression techniques explained in the section 1.3 act as a foundation for most of the other existing techniques. Most of the new compression techniques are optimized

and context-specific implementations of these explained techniques. They all differ in the way how they generate an encoding table and replace the characters or set of characters in the text data using the generated encoding table. They differ in the way they look for repetitions in the data. However, there is one similarity in all existing solutions that they rely on the repeating parts of the data. This puts an limit on the amount of compression that can be attained using these techniques. Table 1.9 shows the memory requirements of a file [1] containing the value of pi (π) upto one billion digits when compressed using various state-of-the-art data compression algorithms. It shows how even the state-of-the-art techniques can achieve a compression ratio in the range of 1.59 to 2.35 (37.2%-57.55%). The calculation of compression ratio is discussed in the section 1.6. Example: Even if there is some imaginary modification of Huffman Coding that can store any character inside this file (containing π upto one billion digits) using 1 bit, the compressed file size will be around 125 megabytes, and the compression rate will be 7.632 (86.9%). Another way of storing this same file could be storing a program that generates the value of π upto N digits. This program to generate pi's value can range from a file size of 153 KB [15] to 29.2 MB multi-threaded pi-program by Alexander J. Yee [37]. In this approach with pi value generator program, only the value of N is required as an input to generate the file. This thesis proposes to use a general-purpose mathematical equation to represent any data using a set of parameters.

1.5 Proposed Solution

This thesis proposes to store data as the parameters of a mathematical equation such that original data can be reproduced when correct parameters are given to the equation. The proposed solution aims to compress the data beyond the maximum compression barrier. It also does not require an encoding table. It does not rely on the repetition of the data. It take the whole data as a single decimal number. Any data on the computer

Compression Technique	File Size	Compression Ratio
No Compression Technique	954 MB	1
Brotli [17]	405 MB	2.35
bzip2 [31]	411 MB	2.32
lzip [12]	416 MB	2.29
XZ [2]	421 MB	2.26
Zopfli - GZIP2 [11]	438 MB	2.17
Zopfli - DEFLATE [10]	438 MB	2.17
gzip [11]	447 MB	2.13
Zstandard [36]	467 MB	2.04
lzop [24]	567 MB	1.68
LZ4 [4]	599 MB	1.59

Table 1.9: Comparing Various Compression Techniques for a File containing value of π upto One Billion Digits

systems can be represented as a single decimal number. Therefore, the proposed technique is not meant for one particular file type.

Assumptions

Following assumptions are taken in account before developing the algorithm for the proposed compression technique:

- There exists a diophantine mathematical equation that can represent any number when correct parameters are provided. 'Sum of geometric series' is used as the potential mathematical equation. Several modifications are made to the equation in order to satisfy this assumption. The concept of diophantine equations is

discussed in the section 2.1.

- The memory required to store the correct parameters is less than the memory required to store the original data. This assumption is what enables the data compression.

1.6 Evaluation Plan

The proposed data compression technique is evaluated on the bases given below:

- Compression Ratio: is one of the major performance evaluation metrics used in data compression. It is the ratio of number of bits required to store original file to the number of bits required to store a compressed file. Compression ratio can also be interpreted as the reduction in the amount of memory required to store a file. [29]

$$\text{Compression Ratio} = \frac{\text{Uncompressed File Size}}{\text{Compressed File Size}}$$

$$\text{Compression Ratio (Percentage)} = 1 - \frac{\text{Compressed File Size}}{\text{Uncompressed File Size}} \times 100$$

If a compression technique compresses a file from 10 MB to 1 MB, the compression ratio is 10:1, and the compression ratio (percentage) is 90%.

- Compression Time: is the time required to generate the compressed file. It may vary due to multiple factors, including computer system specifications, algorithm design, size of the file being compressed, number of phases required to compress the file, and programming language used in the compression program.

- Decompression Time: is the time required to regenerate the original file from a compressed file. All the factors that affect the compression time also apply to the decompression time.

1.7 Document Structure

The remaining content of the thesis is divided as follows:

- Chapter 2 summarizes the background of the major concepts used in the thesis. All the existing literature about the used mathematical equations is discussed.
- Chapter 3 reveals the whole process of algorithm development for the proposed data compression technique. All the methods used to increase the scope and performance of the proposed technique are discussed in this chapter.
- Chapter 4 presents the implementation-specific details of the developed computer program.
- Chapter 5 evaluates and analyzes the performance of the developed program using the evaluation plan discussed in section 1.6. This chapter clearly outlines the results of the study and discusses the recommendations for future research.

CHAPTER 2

BACKGROUND

This chapter covers the existing work done in the area of the mathematical equations used for the data compression in this research. It also includes the key concepts and methods used for software development in the thesis. The chapter starts with introduction to concepts like geometric series, repdigits and Brazilian numbers. The proposed data compression technique assumes whole data to be a single decimal number and developing an algorithm with such a big number requires massive parallel processing and big integer calculations. The core concepts behind parallel processing and big integer calculations are explained in this chapter.

2.1 Diophantine Equation

Diophantine Equations are the equations for which integer solutions are the only solutions of interest. It means all the variables in the Diophantine equations should be integer values. Generally, the number of equations is less than the number of variable terms in the Diophantine equations [32]. One of the assumptions about the potential mathematical equations that can represent the data according the proposed method is that the equations should be Diophantine. This is because the whole data is assumed to be a single decimal number. Integer calculations are easier and simpler than floating point (decimal) number calculations. Also, the integer calculations tend to be the same on different computer systems platforms whereas the floating point number calculations tend to be different depending on the underlying algorithms and precision of the calculation.

2.2 Geometric Series

Geometric Series is one of the simplest infinite series, and each subsequent term in this series is obtained by multiplying the previous term with a constant term called ratio [35].

$$\text{General Form} = A, AB, AB^2, AB^3 \ldots AB^N \ldots \tag{2.1}$$

$$\text{Example} = 3, 3\times 2, 3\times 2^2, 3\times 2^3 \ldots \quad (\text{Let } A = 3, B = 2) \tag{2.2}$$

$$= 3, 6, 12, 24 \ldots \tag{2.3}$$

Equation 2.1 shows the general form for a geometric series with the first term being A and ratio B. It can be observed in the general form that the power of B in any N^{th} term of the series is $N-1$, leading to equation 2.4, which shows the formula for calculating the Nth term in a geometric series. Let S_N be the sum of first N terms in a geometric series.

$$T_N = AB^{N-1} \tag{2.4}$$

$$S_N = A + AB + AB^2 + AB^3 + \cdots + AB^{N-2} + AB^{N-1} \tag{2.5}$$

$$B \times S_N = AB + AB^2 + AB^3 + AB^4 + \cdots + AB^{N-1} + AB^N \tag{2.6}$$

Subtracting equation 2.5 from 2.6

$$(BS_N) - S_N = -A + (AB - AB) + (AB^2 - AB^2) + \cdots + (AB^{N-1} - AB^{N-1}) + AB^N \tag{2.7}$$

$$S_N(B-1) = -A + AB^N \tag{2.8}$$

$$S_N(B-1) = A(B^N - 1) \tag{2.9}$$

$$S_N = A \times \frac{B^N - 1}{B - 1} \tag{2.10}$$

Formula derivation for S_N is given in equations 2.5 to 2.10. Equation 2.10 gives the final formula for finding the sum of first N numbers in a geometric series with given first element A and ratio B.

2.3 Repdigits & Repunits

As discussed in [5], a repdigit or a monodigit is a number that consists of repetitions of the same digit in a given base B. Example: The decimal number 129 is repdigit because its representation in base 6 is 333 which consists of same digit 3.

$$129_{10} = 333_6 = 3 \times 6^0 + 3 \times 6^1 + 3 \times 6^2 = 3 \times (6^0 + 6^1 + 6^2) \quad (2.11)$$

Equation 2.11 shows how the number 129 is represented in base 6 and its mathematical meaning. It can be clearly compared with the equation 2.10. So, anytime the document mentions about repdigit approach means the equation 2.10 is being used.

If the repeating digit of the repdigit is 1, then it is called a repunit. Example: The decimal number 781 is a repunit because its representation in base 5 is 11111. Equation 2.12 shows the mathematical meaning of representing the number 781 in base 5 and can also be compared to the equation 2.10 but the value of A will always be equal to 1 as the repunits will always be in the form of 111.... Like repdigit approach, repunit approach refers to the use of equation 2.10 but with the value of first term A being equal to 1.

$$781_{10} = 111_5 = 1 \times 5^0 + 1 \times 5^1 + 1 \times 5^2 = 1 \times (5^0 + 5^1 + 5^2) \quad (2.12)$$

2.4 Brazilian Numbers

According to [30], any number $S > 0$ is Brazilian if there exists a base B such that $0 < B < S-1$ and the representation of the number S in base B is written with all equal digits. The condition for $B > 1$ because there is no number system using base 1. The condition

for $B < S-1$ because any number would be Brazilian in the base $S-1$ when written as $S = (11)_{S-1}$. Any given number can have more than one Brazilian representation (same repeating digit) in different bases [22].

According to [30], a given Brazilian number S in a given base b and repeating digit a can be represented as:

$$S = A(1 + B + B^2 + \cdots + B^{N-1}), \quad 1 \leq A < B < S-1 \tag{2.13}$$

$$399 = 3 \times (1 + 11^1 + 11^2) = (333)_{11} \tag{2.14}$$

The term N in equation 2.13 refers to the number of times digit A is repeated in the number when in base B. On the decomposition of S into the product of prime factors, all possible values of a $(A_1, A_2, A_3 \ldots)$ can be found. They are all divisors of S except itself.

If a given $S > 7$ is not Brazilian, S is a prime number or square of a prime number. So any even number greater than 7 is Brazilian [30]. This can also be written as:

$$\text{For all } S > 7, S = 2k = 2(k-1) + 2 = (22)_{k-1} \tag{2.15}$$

The brazilian number are like a variant of repdigits but with the condition that $B < S-1$ because every number is brazilian in the base $S-1$. This condition will also be taken into account whenever the repdigit approach is mentioned because the algorithm will always give the value of B equal to $S-1$ if this condition is not used. Without this condition, the assumption that requires the memory required to store the parameters to be less than the memory required to store the original data, will always fail.

2.5 Parallel Programming

One of the assumptions in the proposed technique is that there exists a mathematical equation that can represent any number when correct parameters are provided. The thesis starts with a basic version of the geometric series (repunit approach), as explained

in chapter 3. To check if the assumptions hold true for a given equation, a range of numbers are passed as one of the variables in the equation and is checked if the variables in the solution are integers or not. This allowed checking if the majority of the numbers can be produced/represented by the mathematical equation or not. Further modifications of the equation were made on the basis of this assumption check. The bigger the range of the numbers passed to the mathematical equation, the higher is the certainity that the assumption can be true for a given mathematical equation. Thus, it is a computationally heavy task when the range of numbers is high.

2.5.1 Rise of GPU Programming

CPU (Central Processing Unit) is the main circuit in a computer system and contains the main memory, control unit, and arithmetic-logic unit of the system. Main memory is a high-speed volatile memory on which all the running programs and data are stored. The control unit is responsible for passing the control of the processor from one operation to another. The arithmetic logic unit (ALU) is responsible for the arithmetic and logical operations of the system. The clock speed of a CPU is the number of operations that CPU does in one second. Clock speed of the CPU is what enables it to perform the tasks more quickly and was thought to be the only way to increase performance. Over the period from 1980s to 2010, the CPU's clock speeds have increased from 1MHz to more than 4GHz, which is nearly 4000 times faster. Still, the demand for performance in the market was not fulfilled by the systems with high clock speed. At this time, CPU manufacturers stopped trying to increase clock speed because of physical limits to transistor size, power, and heat restrictions, and they started looking for alternatives [34] [6]. In 2005, manufacturers' interest shifted from single-core CPUs to multi-core CPUs. This shift is sometimes referred to as the multi-core revolution . CPUs up to 80 cores (Ampere Altra) are available in today's

market. However, the multi-core systems also failed to keep up with the rapidly increasing demands of the market [13]. As manufacturers were looking for alternative ideas to improve the computer system's performance, they came across the concept of computing on a GPU Unit. GPUs themselves may not be very new, but the idea of general-purpose programming on GPU Units was new and gained popularity very quickly. GPUs were used in computer systems since the 1980s - 1990s to accelerate the 2D or 3D graphics rendering. OpenCL Library [33] was opened by Silicon Graphics in 1992 and is still a very popular library for GPU Programming. With this rise in GPU Programming for graphics, GPUs with programmable pipelines showed the potential to do something more than just standard graphics. Earlier, pixel color values were given to the GPU Unit for processing, but researchers thought to use this to process any type of data. [28]

2.5.2 Task Based Parallelism

A computer system is generally expected to perform multiple tasks at the same time. Example: An audio player is playing a song while the user is editing a document file in the text editor. In the given example, two different tasks are being performed on two different data files. So, each of the applications is run on a different core (in a multi-core processor) or a single core with time-shared threads. Each of the available resources (processors) is utilized by different tasks running on the system. This type of parallelism in which different tasks are performed on different processors is called Task Based Parallelism. In the context of achieving parallelism for data processing, a program can be written to pipe the data from one processor to another and perform different tasks on different processors. This is called pipeline processing. The processed data from one processor is input for another task of another processor. This level of parallelism is easier to implement and make changes based on the user's requirements. This can work

well until there are sufficiently powerful system resources. However, one issue that arises in this parallelism is that the overall data processing is as fast as the slowest task in the whole pipeline. Example: Three effects are applied to every frame of a video using task-based parallelism. Each effect is applied by a different component (processor), and the resultant frame is passed to the next component in the pipeline. Suppose one of the effects requires 10 seconds for each frame. It does not matter how fast the other two effects are applied, as long as that effect is taking 10 seconds per frame. In a task based parallel manner, even if most work is given to a set of two or more GPUs. As long as they do not account for the same amount of work, the performance will not be maximum. For a set of two GPUs performing the work in a ratio of 70 and 30, means the total runtime of the computation is decreased to the 70% of the total runtime. [9]

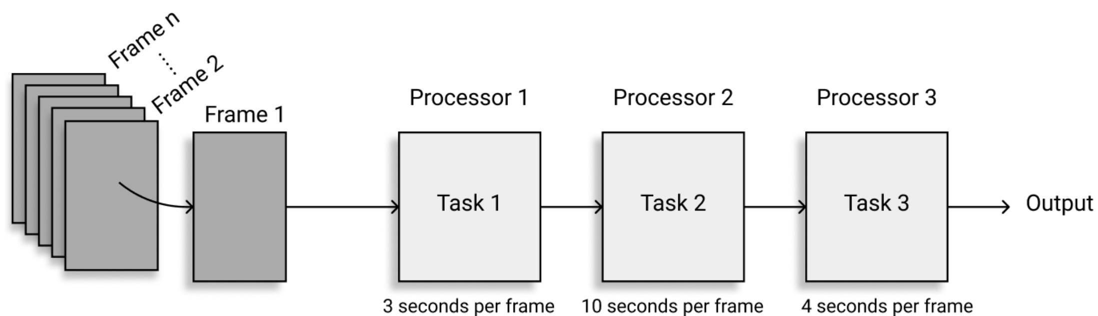

Figure 2.1: Task Based Parallelism (Applying Three Effects to Every Frame of a Video)

2.5.3 Data-Based Parallelism

Data-based parallelism focuses on achieving parallelism based on the data distribution instead of task distribution among different resources. It does allow to overcome the issue of having unequal load on different resources as discussed in task-based parallelism. Data-based approach aims to apply the same set of operations on different divisions of the same data but on different nodes of the parallel network. Example: Three effects are applied to

every frame of a video. Unlike the task-based approach, this approach distributes the video frames (divides data) to different CPU processors or GPU threads, and all the processing units apply three effects to the set of frames given to them. In this manner, each processing unit has an equal share of work, and thus data-based approach achieves more optimal possible parallelism than the task-based approach. After the whole process, subsets of processed data are collected and joined together into an expected final form. Data-based parallelism is the parallelism used on the GPU units to achieve parallelism.[9]

2.5.4 How GPU increases performance?

As mentioned in the previous section, GPUs were used to render graphics for computer systems. All the pixel values for a given screen size were given to the GPU Unit, and they all were processed in the same way. Thus, there was no need to have multiple control units. All the pixel values could be fed to a different ALU pipeline and same control unit could control all of them. So, GPU can be seen as a CPU but with more arithmetic units and lesser control units. GPUs like Nvidia Titan V can have up to 5120 cores which means 5120 tasks can be run simultaneously. This is how GPU can increase the level of parallel processing.

2.5.5 Programming Model - CUDA

CUDA stands for Compute Unified Device Architecture. It is a general-purpose parallel computing model for NVIDIA GPUs and is available in different programming languages and interfaces like C++, OpenACC, and FORTRAN. Just like C++ threads, CUDA has a declaration for a thread. CUDA kernels are the functions that execute on the GPU and process data as an array of threads in parallel. CUDA Programming Model assumes that CUDA threads run on a separate device (GPU) controlled by the

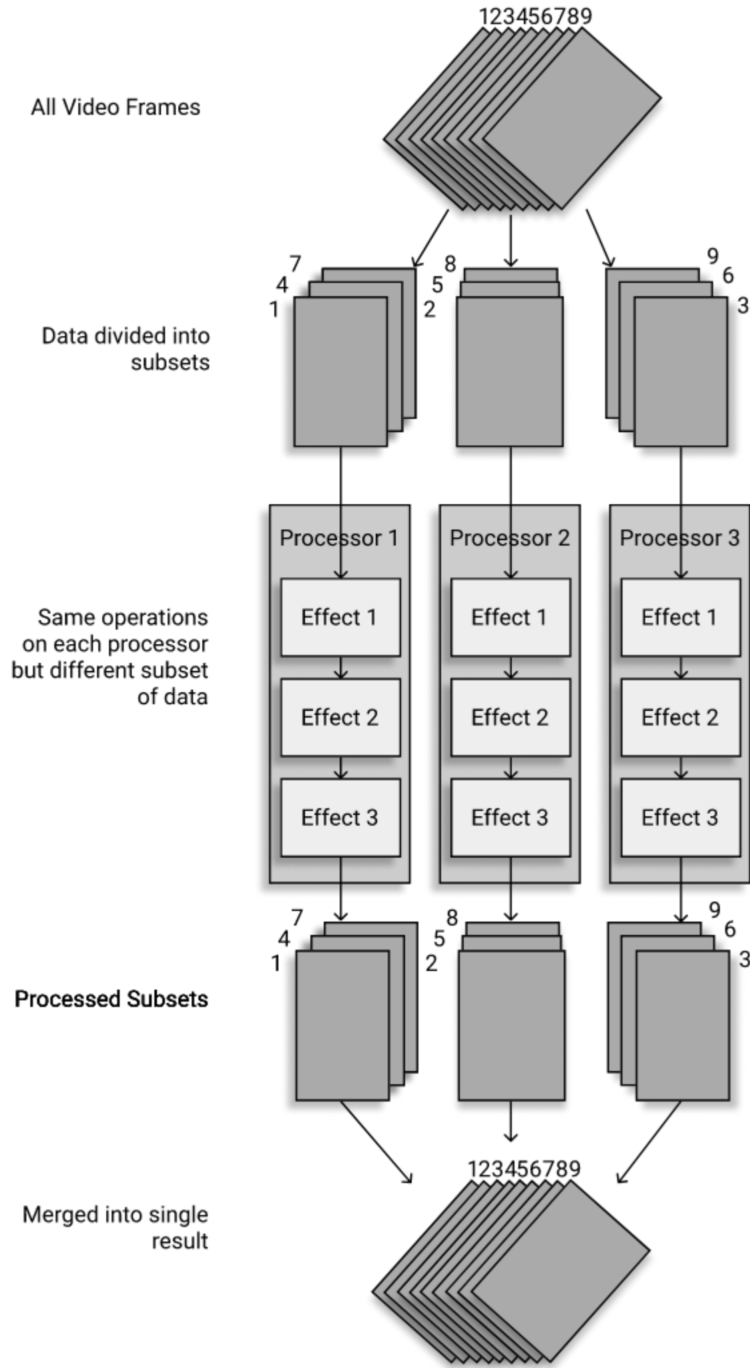

Figure 2.2: Data Based Parallelism (Applying Three Effects to Every Frame of a Video)

main host processor (CPU). It assumes that both devices (host and CUDA device) maintain their memories. All the memory allocations, deallocations, and data transfer

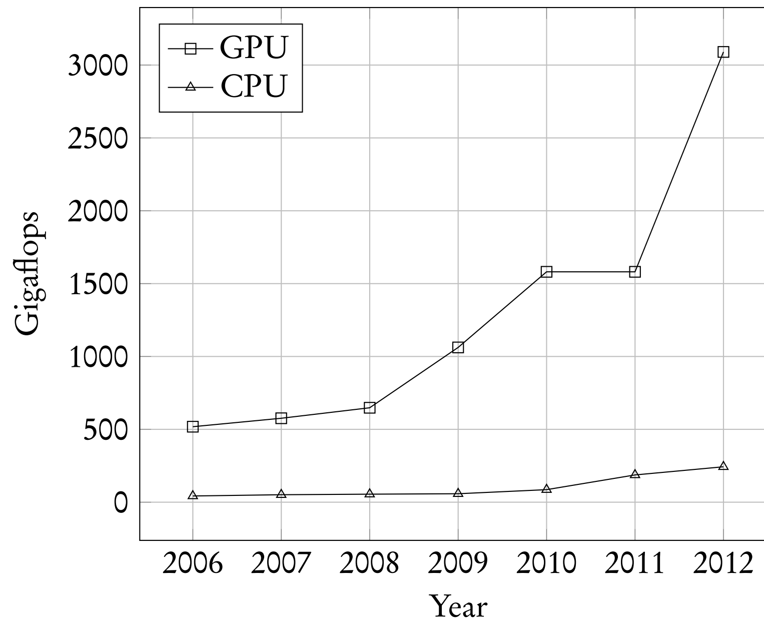

Figure 2.3: Increase in Performance of CPU & GPU from the Year 2006 to 2012

between the two devices are done using the CUDA Runtime Interface. There are three main levels of memory for a CUDA device - Thread, Block and Global. Local memory is only accessible by the thread. A group of threads is called a block, and block memory is shared between multiple threads that belong to that block. Global memory is the memory accessible by every thread on a CUDA device. There is one other type of read-only memory on a CUDA device - Constant Memory. Any thread on the CUDA device can access this memory and be used for data that does not change during the program execution. A group of thread blocks is known as a grid. On the programmer level, a thread block can be one-dimensional, two-dimensional, or three-dimensional, depending on the requirements. There is a limit on the number of threads that can belong to the same thread block. On current systems, this limit is up to 1024 threads per block. CUDA device creates, manages, schedules, and executes threads in a group of 32 parallel threads and is known as warps. All the threads in the same warp start together. The warp scheduler is responsible for the execution of threads in a warp. All the threads

on the same warp should have an identical execution path. Any difference between the execution path may disable all the threads that are not following the current execution path and resume when all the threads have the same state again. CUDA Programming Model also allows multiple CUDA devices, and the programmer has complete freedom to choose the device based on the requirements. If multiple CUDA devices are present, peer-to-peer memory access/copying between the two devices is possible by enabling the access permissions.[23]

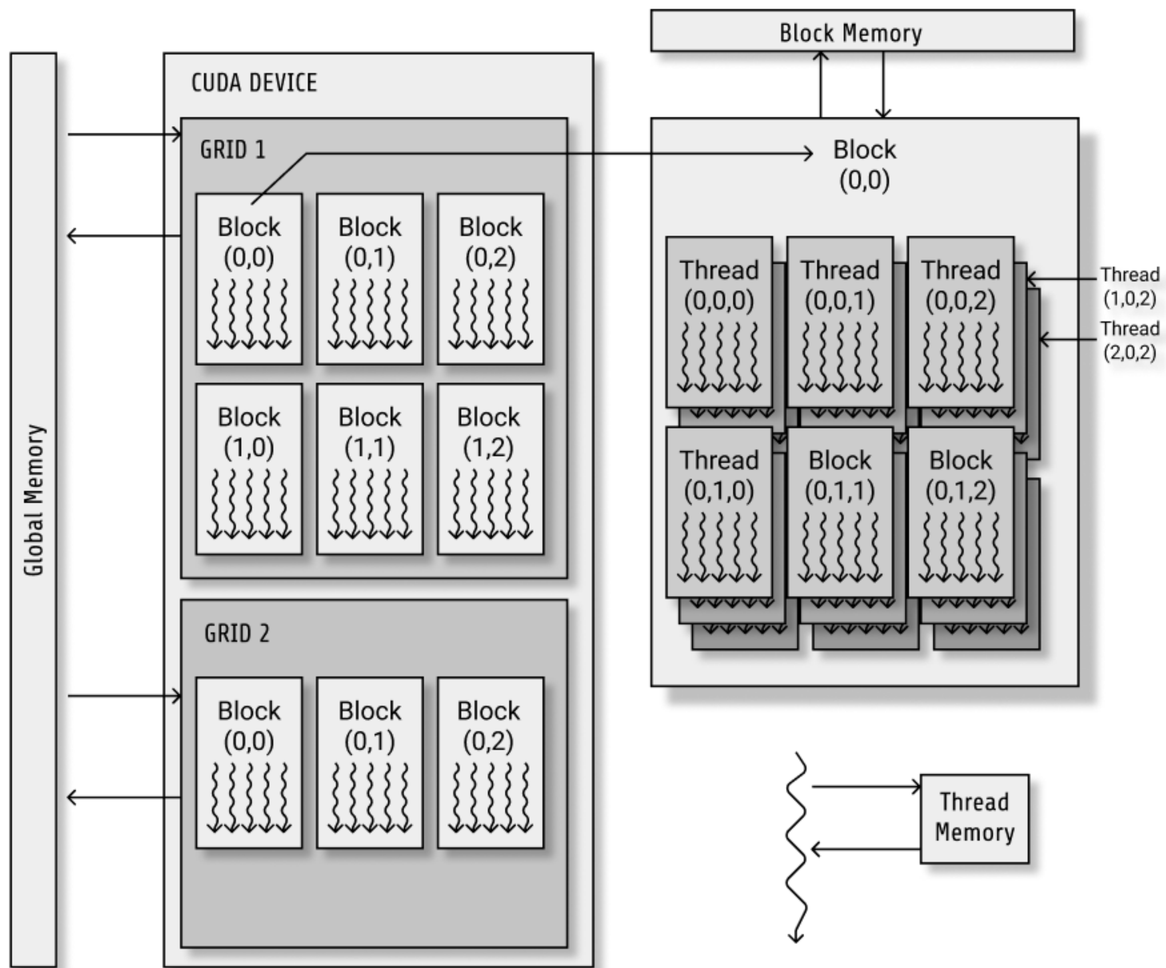

Figure 2.4: Thread and Memory Hierarchy on a CUDA Device

2.5.6 Restrictions in CUDA Programming

- There can be no direct memory access between the host, and CUDA device because CUDA assumes both have their own memory space, and only data transfer should allow the sharing of data between the two.

- Because all the threads should be running with minimal dependency on data from other threads or blocks, CUDA does not allow static variables. Using a static variable will turn the parallel execution into a dependent and serial execution.

- CUDA allows calling other kernel functions on/from the CUDA device, but it restricts recursive calls.

- CUDA does not allow a variable number of arguments for the functions that are run on the CUDA device.

- CUDA does not recommend accessing and writing to same memory address from different threads because the order of thread execution is not predictable and can give completely unpredictable results. It is always better to work on the data structure being used in the program or thread/block level variables to minimize the data sharing between multiple threads.

2.6 Big Integer Calculations using GMP

The thesis deals with whole data that will be compressed as a single number. Because the compression time and decompression time are critical, the programming is done in C++. The largest number possible in C++ is using '`unsigned long long`' and only goes up to 20 digits, but the practical size of data, if seen as a single decimal number, starts from thousands of digits. One option to deal with this issue was to consider the number as a string data type and write functions in C++ to perform the required

mathematical operations on the string data. Another option was to use any existing libraries that allow arbitrarily big integers. GNU Multiple Precision Arithmetic Library (GMP) is used for the arithmetic operations on arbitrarily large numbers (integers, floating point and rational numbers). According to [3], there is no practical limit to the precision except the constrains of available memory of the device on which it is running. The library was created in 1991 and now a part of GNU Free Software Project. The current build of this library is optimized only for CPU based computation and lack GPU support.

CHAPTER 3

APPROACH - PROPOSED SOLUTION

In the current computer system, there are different types of files based on the type of data they contain. On the system memory level, any file contains only binary values: 0 or 1. But on the user end level, they are interpreted as different file based on the interpretation of the binary data they contain. According to the proposed solution, the whole data in a file is taken as a single number and one of the terms in a mathematical equation is replaced by that single number. Values of all the other variables are computed and stored. These stored values are then used to regenerate the original data. The assumption behind this approach is that memory required to store the computed variables is lesser than storing the data. This results in a certain level of data compression depending on the mathematical equation and size of the data.

3.1 Non-Diophantine Approach

Section 2.1 explains how integer solutions are the prime interest in Diophantine equations. As the name indicates, any type of solutions (integer or decimal) are allowed in non-diophantine approach. The equation used in this approach is a simpler version of sum of geometric series (equation 2.10) in which the value of first term A is assumed to be one. The equation for this approach looks like equation 3.1.

$$S = \frac{B^N - 1}{B - 1} \tag{3.1}$$

According to the definition of a repunit, the representation of the number in some base B should be recurrence of the digit 1. Equation 3.1 can be seen as the formulation for a repunit where B is the base, S is repunit and N is the number of times the digit 1 is repeated. The value of B in case of repunits range from 2 to $S - 2$ and the value of N must be integer. In this approach, the value of B is allowed to be integer or decimal. A

brute force search is used in this approach and it requires to check all the possible values of B from 2 to $S-2$, and all the possible values of N. But this is not practical because of infinitely many decimal numbers between 2 and $S-2$. But the range of N can be calculated and corresponding values of B can be found with the help of that.

Range of N

For a given value of B and N, smallest value of number that can be formed is digit one followed by $(N-1)$ zeros. Example: If $B=3, N=4$, then $S(\text{Smallest}) = 1000_3 = 27_{10} = B^{N-1}$. Similarly the largest value of the number can be formed by having all digits equal to $B-1$. Example: If $B=3, N=4$, then S (Largest) $= 2222_3 = 80_{10} = B^N - 1 < B^{N-1}$
Therefore, the number S has N number of digits in base B if

$$B^{N-1} \leq S < B^N \tag{3.2}$$

$$N-1 \leq \log_B S < N \tag{3.3}$$

$$N = \lfloor \log_B S \rfloor + 1 \tag{3.4}$$

The value of B ranges between 2 and S-2. Therefore the value of N ranges from $\lfloor \log_2 S \rfloor + 1$ to $\lfloor \log_{S-2} S \rfloor + 1$.

Algorithm

1. Iterate through all the possible integer values of N from $(\lfloor \log_2 S \rfloor + 1)$ to $(\lfloor \log_{S-2} S \rfloor + 1)$.

2. For each value of N, find the value of $B < S-1$ that satisfies the equation 3.1. Store the value of N and B found in this step as one set.

3. After iterating through all the values of N and storing the sets of B and N, find the set that requires the least memory. Store it as the compressed file and discard the others.

The set of N and B stored in the last step of the algorithm is used to regenerate the original data in the decompression phase. The values of B and N are put into the equation 3.1 and original data comes out as the value of S in that equation. The problem with this approach is that the value of B is a decimal number and needs to be precise enough to generate the original data without any loss or change. Multiple tests were done to compress data using this approach but the amount of data compression is very low and thus focus of thesis was shifted to the Diophantine approach in search of better results. This technique is useful when some amount of loss in the data is acceptable. In such case, B being a decimal value can be changed. The least significant digits of fractional part of B can be discarded. Then, the data generated with changed B and N, is expected to be similar to the original data upto some level, depending on the number of discarded digits.

3.2 Diophantine Approach

All the approaches discussed in this section deal with the equations in which only integer solutions are permitted. The section starts with a basic repunit approach with only two variables that are computed to compress the data. The repunit approach is not able to compress any given number. So, various changes are made to that equation so that any data can be compressed using the equation.

3.2.1 Repunit Approach

The equation used in this approach is same as the one used for non-diophantine approach in section 2.1 but only integer solutions are accepted.

Algorithm

1. Iterate through all the possible integer values of N from $(\lfloor \log_2 S \rfloor + 1)$ to $(\lfloor \log_{S-2} S \rfloor + 1)$.

2. For each value of N, find the value of B that satisfies the equation 3.1.

3. If the value of B is an integer, the value is stored along the corresponding value of N as a pair. Otherwise it is discarded.

4. From all the stored pairs of B and N, the pair that requires least memory is stored and discard all the other pairs.

The pair saved in the last step of the algorithm is used to regenerate the original data as done in the non-diophantine approach. This approach turns out to be more efficient than Non-Diophantine approach for large numbers in terms of data compression but it is not able to compress every number. There are many cases in which algorithm is not able to find a single pair of N and B. A test is done to check the scope of this approach. The test shows how many numbers out of first 600 million numbers can be compressed using this technique. The results of the test are discussed in the section 5.1.1.

3.2.2 Repdigit Approach

In the previous approaches, the value of A in equation for sum of geometric series is taken as one. But now the value of A is not a constant anymore. Now, the number S is divided with each of its factors, and the approach from section 3.2.1 is used on the number S/A. The equation used in this approach can also be seen as the formulation for repdigit or brazilian numbers.

$$S = A \times \frac{B^N - 1}{B - 1} \tag{3.5}$$

Algorithm

1. Check if the number S can be compressed using the approach explained in the section 3.2.1. If yes, store the solution as $< A, N, B >$ with value of $A = 1$.
2. Find all the factors of the number S as A.
3. Divide the number S with one of its factors (A_x) and use the repunit approach from section 3.2.1 on the S/A_x. If the solution is found, get the values of N and B from the repunit approach, and store the solution as $< A_x, N, B >$. Repeat this step for all the values of A computed in step 2.

Finding the factors of the number S

Factor of a number S is a number that divides the number S with remainder equal to zero. The factor of a number S can not be 1 or the number S itself. There are couple of levels on which factors of the number can be found:

- All the numbers from 2 to $S - 1$ can be potential factors of the number.
- Only the prime numbers lesser than S can be potential prime factors of the number. In this approach, only prime factors are found.
- All the numbers, lesser than $\sqrt{S}$ can be potential factors of the number. In this situation, the number S/A_x (A_x is a factor less than $\sqrt{S}$) can also be counted as the factor of the number. This approach reduces the checks required to find all the factors greater than $\sqrt{S}$ but the number S/A_x needs to be computed.

3.2.3 Neighbors Check

The results of repunit and repdigit approach are discussed in the section 5.1.1 and 5.1.2. The repdigit approach proves to be better as it is able to compress more numbers than

the repunit approach but repdigit approach does not work for prime numbers and some non-prime numbers. To make it work for every number, a new variable D (difference) is introduced. Using this variable, a neighbor number is found that can be compressed using approach in section 3.2.2. Example: A number S can not be compressed using any of the previous approaches. In this case, the previous approaches are used on the number $S \pm D$). The equation for this approach with the new variable D looks like:

$$S = A \times \frac{B^N - 1}{B - 1} + D \tag{3.6}$$

Algorithm

1. Set initial value of D as zero.
2. Apply the approach from section 3.2.2 on $S - D$. Go to step 5 if the solution is found.
3. Apply the approach from section 3.2.2 on $S + D$ if the value of D is greater than zero. Go to step 5 if a solution is found.
4. Increase the value of D by one and go to step 2.
5. Store the value of D along the values of A, B, and N from the section 3.2.2.

3.2.4 Padding Approach

This approach is an improvement over the neighbor check approach discussed in section 3.2.3 in terms of data compression but with a performance trade-off. In the previous approach, difference between the original number and the compressed number is stored as D but in this approach, instead of adding/subtracting the D from the S, a given number of digits P are padded to the end of the original number S. For the storage purposes, only the number of digits padded, are required. Example: The number 1032 is

being compressed using this approach. Suppose the number of digits that will be padded to the original number to be 2. Therefore, this approach need to find a number between 103200 and 103299 that can be compressed using repdigit or repunit approach. The number 103200 can be compressed using the repdigit approach. To store the number 1032 in the compressed file, the values of A, B and N can be taken from the repdigit approach and the value of padded digits (2) will be stored along them. The equation of this approach can be written as:

$$S = \left(\frac{B^N - 1}{B - 1} \times 10^P\right) + X \tag{3.7}$$

where P is the number of padded digits and $0 \leq X < 10^P - 1$

Algorithm

1. Take the initial value of number of padded digits P to be zero.
2. Pad P zeros to the number S.
3. Use the repdigit or repunit approach from section 3.2.2 or section 3.2.1 to compress the padded number S. If a solution is found, store the solution along the current value of P.
4. If no solution is found and all the last P digits of the number S are not 9, increment the number S and go to step 3.
5. If no solution is found and all the last P digits of the number S are 9, remove last P digits of the number. Increment the value of P and go to step 2.

3.3 Optimizing the Repunit Approach

According to the definition of sum of first N terms in geometric progression,

$$S = 1 + B + B^2 + \cdots + B^{N-2} + B^{N-1} \tag{3.8}$$

$$S > B^{N-1} \tag{3.9}$$

$$\sqrt[N-1]{S} > B \tag{3.10}$$

Using Binomial Expansion,

$$(1+B)^{N-1} = 1 + \binom{N-1}{1}B + \binom{N-1}{2}B^2 + \cdots + \binom{N-1}{N-1}B^{N-1} \tag{3.11}$$

$$\tag{3.12}$$

The value of $\binom{N-1}{X}$ is always greater than 1 and equal to 1 for just two terms. The value of N will always be greater than 2. Therefore, the value of $\binom{N-1}{X}$ will be greater than 1 for at least one term. This leads to the following result:

$$(1+B)^{N-1} > 1 + B + B^2 + \cdots + B^{N-1} \tag{3.13}$$

$$(1+B)^{N-1} > S \tag{3.14}$$

$$(1+B) > \sqrt[N-1]{S} \tag{3.15}$$

From equation 3.10 & 3.15,

$$1 + B > \sqrt[N-1]{S} > B \tag{3.16}$$

Therefore, given the value of N, $\lfloor \sqrt[N-1]{S} \rfloor$ is the only candidate for value of B. The range of N is already known to be between $\lfloor \log_2 S \rfloor + 1$ and $\lfloor \log_{S-2} S \rfloor + 1$ (Section 3.1).

3.4 Sliding Window Approach

In the previous approaches, the whole data is considered to be a single number but now the whole data is scanned for numbers that can be directly compressed using the

repunit approach discussed in section 3.2.1. The motivation for the concept of sliding window is taken from the Lempel Ziv Algorithm. All the numbers in the data that can be compressed in that manner are replaced with the respective compressed representation.

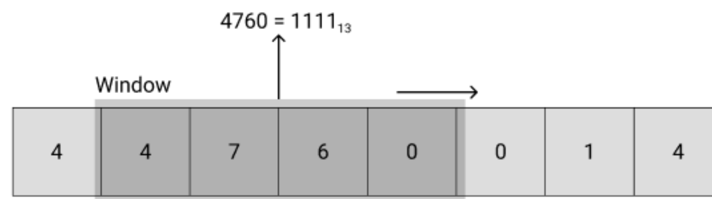

Figure 3.1: Illustration of the Window Approach

Algorithm

1. A window size is chosen that is close the size of the whole data. Example: If the whole data taken as a single number is X digits long, then the initial size of the window is taken to be close to X because bigger the size of window means potentially more compression.

2. Then the algorithm starts from first digit of the whole data and consider all the digits after it that comes in the range of that window. Example: If the size of window is 3, and the data is 12345, then starting from the first digit numbers in the range of window are 123.

3. The current number in the window is checked if it can be compressed using the approach from section 3.2.1.

4. If yes, the number is replaced with the respective representation ($< B, N >$).

5. Otherwise, the window is shifted forward by one digit (From digit x to digit x+1 in the right direction). Go to step 3, if the window has not reached the end of the data (window extends out of the data).

6. If the window size becomes small enough that the changing the representation results in increased file size, stop the algorithm. Otherwise, decrease the size of the window by one and go to step 2.

If the window is being moved across the data and the numbers inside the range of the window include the compressed representation of another number that was created in an earlier iteration, that position should be skipped and window should be moved to the digit to the position where the compressed representation ends. It means the algorithm should not try to compress the already compressed data. If the compressed data is chosen to be re-compressed, the decompression algorithm will also have to undergo multiple iterations over the compressed data in order to regenerate the original data because in each iteration of decompression, the data may include the partially decompressed representation. So the decompression algorithm will have to keep iterating over the file until no compressed representation are present in the data.

CHAPTER 4

IMPLEMENTATION

This chapter focuses on the implementation details of the various approaches discussed in Chapter 3. It discusses all the optimizations done to improve the time and space complexity of the developed program. At the end of the chapter, the overall structure of the developed program and the encoding used to store the variables in the compressed file is described.

4.1 Wrapper over the GMP Library

The functions/interface provided by the GMP library are complex and make the process of writing the code slow. Also, each GMP variable is stored in the heap memory and memory deallocation needs to be done manually by the programmer. To reduce the chances of memory leaks while making changes to the code and using more intuitive method names, a custom wrapper class is put over the functions required from the GMP class. Figure 4.1 contains the snippet of the code used to avoid unwanted memory leaks of the variables of GMP Library. This wrapper class helped in writing the code more quickly and enabled automatic memory deallocation. All the code snippets in this chapter where the type of used variables is `BigInt` or `BigFloat`, means this wrapper class is used.

4.2 Non-Diophantine Approach

This section discusses the implementation of the non-diophantine approach discussed in section 3.1 using GMP and the wrapper class discussed in the section 4.1. To implement this approach, a function called `check_repunit` is written assuming the value of A to be one. This function takes `base` and `n` as input and save the geometric sum

```
class BigFloat {
    public:
    mpf_t value;
    int precision = 200;
    BigFloat(){
        mpf_init2(value, precision);
    }
    ...
    ~BigFloat(){
        mpf_clear(value);
    }
}
```

Figure 4.1: Wrapper Code over the GMP variables to avoid Unwanted Memory Leaks

in a argument called `result`. This function is used to find the value of `S`, given the values of `base` and `n`. The code snippet for this function is given in the figure 4.2. Another function developed for this approach is the function `brute_force` that tries different values of `n` and `base` with the function `check_repunit` and check what values can result in getting the expected value of original data being compressed. In the `brute_force` function, an initial value is taken for `base` and `increment`. The value of `base` changes according to the value of `increment`. With the initial value of `base`, the sum of geometric series is calculated as `result` and compared to the original data. Then the value of `increment` is changed according to the results of the comparison. The corresponding change is made to the value of `base`. This process of changing the value of these two variables, computing the sum of geometric series, and then comparing the `result` with original data is done again and again until the value of the `result` is close

```
void check_repunit(BigFloat *result, BigFloat *base, long int n){
    BigFloat dividend, divisor;
    // dividend = b^n-1
    calculate_exponent(&dividend, base, n);
    dividend.subtract(1);
    // divisor = b-1
    divisor.copy_value(base);
    divisor.subtract(1);
    // result = (b^n-1)/(b-1)
    dividend.divide_by(&divisor);
    result->copy_value(&dividend);
}
```

Figure 4.2: Checking if a Given Number is a Repunit in Non-Diophantine Approach

to the value of original data. In this implementation of the method, a given range of the value for `n` is used and the `brute_force` function provides the value for `base` for each value of `n`. Any of the pairs for `base` and `n` generated by this approach can be used to regenerate the original data. The code snippet in the figure 4.3, the value of `n` ranges from 2 to 10. Therefore, eight solutions will be generated with this snippet. The range of n discussed in the section 3.1 can be used in the `brute_force` function. The `brute_force` function can be seen as a fine-tuner for the values of `base` and `n`. This property of fine-tuning can be used in lossy compression. In a case when certain level of data loss is acceptable, the algorithm can be stopped before achieving complete equality between the target data and the generated data.

```
void brute_force(BigFloat *target){
    BigFloat base, dividend, divisor, result, increment, difference;
    unsigned long int n;
    for (n=2;n<10;n++) {
        base.set("2");
        increment.set("100");
        check_repunit(&result, &base, n);
        while (true) {
            int comparison = target->compare(&result);
            while (comparison == 1) {
                check_repunit(&result, &base, n);
                base.add(&increment);
                comparison = target->compare(&result);
            }
            base.subtract(&increment);
            base.subtract(&increment);
            check_repunit(&result, &base, n);
            calculate_difference(&difference, target, &result);
            if (difference.compare(1) == -1)
            break;
            increment.divide_by(10);
        }
        result.do_ceil();
    }
}
```

Figure 4.3: Changing the Values of `base` and `n` in Non Diophantine Approach

4.3 Repunit Approach

In this approach, only integer solutions are accepted. The equation used in this approach was the sum of first N terms of a geometric series in which the value of A (first term) is taken as zero. The algorithm of this approach is discussed in the section 3.2.1. The implementation of this approach is discussed in this section. The optimization discussed in the section 3.3 is also used in the implementation. A function `check_repunit` was written for this approach and the code for that function without the use of GMP is given in the figure 4.4. The function `calculate_geometric_sum(R, B, N)` returns the sum of first N elements of the geometric series with ratio B and value of first element being 1. This approach fails to compress any given number because only integer solutions are accepted and given two integers N and B, not all numbers can be represented as S using the equation 3.1. A test is done to check how many numbers out of first 600,000,000 numbers can be compressed using this approach.

4.3.1 CPU Implementation of the test

In this implementation, all the numbers from 1 to 600,000,000 were divided into equal divisions and given to each thread. Each thread iterates through the range of numbers given to it and checks if the number is compressible using the repunit approach or not. The code to check if the number is compressible is given in the figure 4.4. The computer system on which the test was performed contains 8 cores and the test was performed on 6 cores for the sake of simplicity because it makes dividing the numbers into each core simpler. In this way, the core 1 gets all the numbers from 1 to 100,000,000 and the core 6 gets all the numbers from 500,000,001 to 6000,000,000. This distribution is named as Continuous Distribution.

This looks like an even distribution but the computation done in the compression

```
bool check_repunit(BigInt *target) {
    BigInt base, result;
    int n_min = 2, n_max;
    n_max = get_log(target, 2);
    for (int i = n_min; i <= n_max; i++) {
        calculate_root(&base, target, i);
        calculate_geometric_sum(&result, &base, i+1);
        if (result.compare(target) == 0) {
            return true; // Integer Solution Found
        }
    }
    return false;
}
```

Figure 4.4: Checking if a Given Number is Repunit in Diophantine Approach

algorithm highly depends on the number. It takes lesser time for small numbers and more time for bigger numbers. The problem with the distributing numbers to each core in this manner (continuous distribution) is that the core with last division has to deal with large numbers. To avoid this problem, a different type of distribution is used and is named as Discontinuous Distribution. Total numbers that will be handled by each core and the thread number is given to the thread as parameters. The thread then computes all the numbers that it has to test. According to this distribution, continuous ranges are not tested by different threads but all the numbers are divided across all the threads such each thread gets equal share of work load. The illustration of both the distributions is given in the table 4.1. The code snippet for the discontinuous distribution is given in the figure 4.6

```
void worker_thread(int start_number, int end_number){
    for(int number=start_number; number<=end_number; number++){
        // test code
    }
}
...
int TOTAL_CORES = 6;
int TOTAL_NUMBERS = 6000000;
const int NUMBERS_PER_CORE = TOTAL_NUMBERS / TOTAL_CORES;
int i = NUMBERS_PER_CORE;

for(int thread_num = 0; thread_num < TOTAL_CORES; thread_num++){
    std:thread thread(
    worker_thread, thread_num*NUMBERS_PER_CORE+1,
    thread_num*NUMBERS_PER_CORE+NUMBERS_PER_CORE);
}
```

Figure 4.5: Distributing the numbers in different threads as continuous ranges

In the discontinuous distribution, the size of the numbers that each thread gets is totally different. To implement this for a computer system, there are two ways in which it can be done. One way is to use create a list of all the numbers that each thread will get as an argument. Other way is to call each thread with a single input number and the host thread handles the distribution. Example: Threads 1 to 6 will be called for numbers 1 to 6. In the next iteration they will be called with numbers 7 to 12. Both of these techniques are inefficient because in the first technique, the size of the list will grow as the range of the numbers used for the test increase. The other technique is inefficient because creating

```
void worker_thread(int thread_number,
       int total_numbers_to_be_tested,
       int total_threads) {
   int number_to_be_tested = thread_number;
   for(int i = 0; i< total_numbers_to_be_tested; i++){
       number_to_be_tested += total_threads;
       ... // test code
   }}
   ...

   int TOTAL_CORES = 6;
   int TOTAL_NUMBERS = 6000000;
   const int NUMBERS_PER_CORE = TOTAL_NUMBERS / TOTAL_CORES;
   for(int i=0; i<TOTAL_CORES; i++){
       // create test threads with parameters as (i, NUMBERS_PER_CORE,
       // TOTAL_CORES)
   }
```

Figure 4.6: Generating the numbers to be tested as Discontinuous Distribution

a thread is a computationally heavy task and this technique requires to create the threads, same number of times as the range of numbers (600,000,000 threads will be created in this case). However, there is one way avoid these inefficiencies. It is to assign an index number to each thread and tell the thread how many numbers it needs to test. In this way, thread will use these variables to compute the numbers it needs to test. There is a slight tradeoff behind the computing the number but is better than other techniques in terms of overall time and memory performance.

Thread Number	Continuous Distribution	Discontinuous Distribution
1	1 to 100,000,000	1, 7, 13, 19...599,999,995
2	100,000,001 to 200,000,000	2, 8, 14, 20... 599,999,996
3	200,000,001 to 300,000,000	3, 9, 15, 21... 599,999,997
4	300,000,001 to 400,000,000	4, 10, 16, 22...599,999,998
5	400,000,001 to 500,000,000	5, 11, 17, 23...599,999,999
6	500,000,001 to 600,000,000	6, 12, 18, 24...600,000,000

Table 4.1: Illustrating the two methods of work distribution

4.3.2 GPU Implementation

Even with the multi-core implementation and both types of number distributions, the run time of the CPU implementation is very high. To decrease the run time of the test, GPU programming was used. The details of how the GPUs helps in improving the performance of a computation, are discussed in the section 2.5. The whole GPU implementation can be divided into two parts - Host code and GPU device code. The host code is the one that is written to transfer the data from the GPU device to the host CPU device and vice versa. The GPU device code consists of the computation that runs on the GPU device. The code snippets for host and device code are given in the figure 4.7 and 4.8 respectively.

In the GPU implementation, two different memory allocations are required for CPU and GPU. First, host (CPU) memory allocation is done as `h_C`. The size of the `h_C` is size of `int` multiplied by the count of numbers that will be used in the test (600,000,000). Same amount of memory allocation is done on the GPU device as `d_C`. Then the values for the number of threads per block and blocks per grid are assigned. The kernel (GPU Function) is then launched with the required arguments as shown in

```
__global__ void checkRepunit(int *C, int total_numbers)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if (id < total_numbers)
    {
        int base, result, n_max;
        n_max = log2f(id);
        for (float i = 2; i <= n_max; i++) {
            base = powf(id, 1 / i);
            result = (powf(base, (i + 1)) - 1) / (base - 1);
            if (id == result)
                C[id] = id;
        }
    }
}
```

Figure 4.7: Kernel Code for GPU

the figure 4.8. In the GPU function, the number to be tested is calculated using the block id, block dimensions and the thread id of the current thread. The number is checked using the algorithm from section 3.2.1 if it can be compressed using this approach or not. If a number S can be compressed, then the Sth element of the array `d_C` is set to 1. When all the computation on the GPU is complete, `d_C` is copied from GPU to CPU as `h_C`. This array `h_C` now contains all the information about the number that can be compressed using repunit approach. To check the count of total repunits in the tested range, program iterates through the `h_C`, and increment a counter variable whenever the value 1 is encountered in the array.

```
int total_numbers = 600000000;
size_t size = total_numbers * sizeof(int);


int *h_C = (int *)malloc(size);
if (h_C == NULL) exit(EXIT_FAILURE);
int *d_C = NULL;
err = cudaMalloc((void **)&d_C, size);
if (err != cudaSuccess) exit(EXIT_FAILURE);


int threadsPerBlock = 256;
int blocksPerGrid =
    (total_numbers + threadsPerBlock - 1) / threadsPerBlock;
checkRepunit<<<blocksPerGrid, threadsPerBlock>>>(d_C, total_numbers);
err = cudaGetLastError();
if (err != cudaSuccess) exit(EXIT_FAILURE);


err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
if (err != cudaSuccess) exit(EXIT_FAILURE);
int total = 0;
for (int i = 0; i < total_numbers; i++)
    if (h_C[i] != 0)
        total++; // COUNTING THE REPUNITS
// FREE GPU AND HOST MEMORY
```

Figure 4.8: Launching the GPU Kernel

In GPU Implementation, the work is equally divided in all the threads because the range of numbers that need to be tested is distributed by the host device but calculated by the GPU threads when during the function call. The variable `id` in the given code snippet corresponds to the number that is being checked for the data compression. Thus, GPU programming automatically enforces equal work distribution among all the tasks and enables massive parallel processing.

4.4 Repdigit Approach

This section discusses the implementation of the approach discussed in section 2.2. In this approach, the number along all the divisors of the number are checked if they can be compressed using the approach in section 3.2.1. In the first iteration, the original number S is checked (Factor, $A = 1$). In the later iterations, the number S is divided by one of its factors A, and the number S/A is checked for compression. As discussed there can be different ways to find the factors of the number - Prime factors, factors less than square root of the number, and all the possible factors. In the code snippet given in the figure 4.9, the factors are calculated as all the factors lesser than square root of the target number. In this implementation the `BigInt` version of the GMP wrapper used which corresponds to the arbitrarily large integer. The function `check_repdigit` in figure 4.9 finds all the factors of the target number and uses the function `check_repunit` from the figure 4.4.

4.5 Neighbors Approach

This section discusses the implementation of the approach discussed in section 3.2.3 in which all the numbers below and above the target number are checked for compression and the difference between the compressed number and stored number is stored to regenerate the original data later. Here a variable `counter` is maintained to

```
bool check_repdigit(BigInt *target) {
    BigInt factor, factor_max;
    factor.set("1"); // Start with factor = 1
    calculate_root(&factor_max, target, 2);
    while (factor.compare(&factor_max) != 1) {
        if(target->isDivisible(&factor)){
            BigInt b;
            b.copy_value(target);
            b.divide_by(&factor);
            if(check_repunit(&b))
                return true;
        }
        factor.add(1);
    }
    return false;
}
```

Figure 4.9: Checking if a Given Number is a Repdigit in Diophantine Approach

keep record of the difference between the original and compressed number. The code for this approach is given in the figure 4.10. All the neighbors below and above the target number are check with repunit or repdigit approach until a solution is found. The value of `counter` at the end of the `while` loop is a value stored along the other values (A, B, N) in the compressed file. The code given in figure 4.10 prints the value of counter to be zero if the solution is found without the use of this approach and non-zero when the solution is found for one of the neighbors.

```
void check_neighbours(BigInt *target){
    BigInt counter, negative_neighbour, positive_neighbour;
    negative_neighbour.copy_value(target);
    positive_neighbour.copy_value(target);
    counter.set("0");
    if(check_repunit(target))
        cout<<"COUNTER: 0";
    while (true){
        negative_neighbour.subtract(1);
        positive_neighbour.add(1);
        if (check_repunit(negative_neighbour))
        cout<<"Negative Counter";
        if (check_repunit(positive_neighbour))
        counter<<"Positive Counter";
        counter.add(1);
    }
    cout<<"COUNTER: "; counter.print();
```

Figure 4.10: Code Snippet for Neighbors Approach

4.6 Padding Approach

This section describes the implementation of the padding approach discussed in the section 3.2.4. The code for padding was written in the GMP way without the use of custom wrapper class and is given in the figure 4.11. The program starts by checking if the original number can be compressed using the `check_repdigit` function from the figure 4.9. Then it sets the value of total padded digits to be one. The digits are padded inside the `for` loop and the padded number is again check with repdigit approach. The

digits are padded and the value of total padded digits is increased until a repdigit is found. The function prints the value of total padded digits when the solution is found and returns back.

4.7 Reducing Memory Complexity by variable reuse

In the final program that includes combination all the discussed techniques, there are multiple loops and variable initialization is done in each of them. This means the variables of the GMP library are also initialized and cleared in each of the loops. According to the GMP documentation [3], the memory allocation of a variable is managed by the library and additional space is given to the variable when required. In most of the implementations of the thesis, `mpz_t` variables are used. According to the library documentation, the memory allocation to this variable is done in a way that the library never reduce the variable size. This helps in lesser reallocations. The documentation also recommends to avoid excessive initializing and clearing of the variables. In the implementations in which a wrapper is used over the library, `mpz_t` variables are cleared whenever they go out of scope. The wrapper helps to write the code easily but the program performance degrades due to excessive initializing and clearing. To avoid this problem, variable reuse is done in the final implementation. A minimum number of variables of `mpz_t` type are initialized in a global scope and they are used whenever a variable of that type is required. Valgrind [21] [7] is a multi-purpose memory analysis tool for Linux and is used to analyze the memory usage and allocations of the program and the results of this improvement are discussed in the section 5.1.3

```
void check_neighbours_with_padding(string target_str) {
   mpz_t target;
   mpz_init(target);
   const char *target_const_char = target_str.c_str();
   mpz_set_str(target, target_const_char, 10);
   if (check_repdigit(&target)) {
      cout << "PADDING : 0";
      return;
   }
   string padded_target_str;
   int total_padding = 1;
   while (true) {
      for (int i = pow(10, total_padding - 1);
            i <= pow(10, total_padding) - 1; ++i) {
         padded_target_str = target_str;
         padded_target_str.append(to_string(i));
         target_const_char = padded_target_str.c_str();
         mpz_set_str(target, target_const_char, 10);
         if (check_repdigit(&target)) {
            gmp_printf("PADDING: %d", total_padding);
            return;
         }
      }
      total_padding++;
   }
}
```

Figure 4.11: Code Snippet for Padding Approach

4.8 Best Case Data Generation

This section discusses a program written to generate the data that acts like a best case for this data compression algorithm in terms of compression ratio. Approximating the amount of compression that a file will undergo with the proposed techniques (without compressing the file) is a difficult and quite impossible task. To approximate the amount of compression, data needs to go under the same process that is used to compress the data. This best case data generation program takes small values for N and B, and generate an image which if compressed using the proposed technique will result in the same values. This data generation program puts the two values in the equation with two variables (equation 3.1) and generate the number S which is then converted into an image file. The code for best case data generation is given in the figure 4.12.

Algorithm

- Take small values of B and N.
- Put them in the equation 3.1 and get the value of S.
- Split the number S into multiple numbers such that they can be put as a list in which each value is less than 255.
- Take the square root of the length of the list. This gives the dimensions of the image that will be created. Other methods can also be used to produce image of expected aspect ratio.
- Iterate through the list and take pairs of three consecutive values, which becomes the color value (R, G, B) for each pixel of the image that will be generated.
- Encode the color values into an image file using a library like EasyBMP and save the image file.

```
BigInt base;
    long int n = 1020; // Setting value of N
    base.set("102030"); // Setting value of B
    BigInt result;
    calculate_geometric_sum(&result, &base, n);

    string str = result.toString();
    int total_values = str.length()/2;
    int total_colors = total_values/3;
    int image_length = sqrt(total_colors);
    int image_width = image_length;
    EasyBMP::RGBColor black(0, 0, 0);
    EasyBMP::Image image(image_width, image_length, "sample.bmp", black);

    int r, b, g, index;
    for (int i = 0; i < image_length; i++) {
        for (int j = 0; j < image_width; j++) {
            index = 6*(i*image_length+j);
            r = stoi(str.substr(index,2));
            g = stoi(str.substr(index+2,2));
            b = stoi(str.substr(index+4,2));
            image.SetPixel(i, j, EasyBMP::RGBColor(r, g, b));
        }
    }
    image.Write();
```

Figure 4.12: Code for Generating the Best Case Data for Repunit

4.9 Structure of the program

The developed program includes the components mentioned in the figure 4.13. The process starts from the input file and the file reader. File reader is the component that accepts a given file and convert the whole data into a decimal number. This decimal number is then passed to the compressor. The compressor is the main component which includes the main algorithms developed as the part of the thesis. Compressor operates on the input number to generate a set of parameters that will be stored in the compressed file. The compressor writes these parameter value to the compressed file according to the encoding guide discussed in the section 4.10 and store the generated parameter in a binary form. The compressed file can then be used to regenerate the original data. The decompressor component reads the compressed file and use the same encoding guide to convert the binary data inside the compressed file into decimal values. Using these values, a decimal number is generated by the decompressor which represents the original data. This decimal number is now passed to the file writer which creates the output file based on the file type of the input file. The current implementation of the file writer and reader is restricted to only text or image files. Also, the current implementation does not store the file type in the compressed file and this information is given manually to the file writer.

Merging all the approaches as one program

This section discusses how all the discussed approaches are merged and used to compress any given number. The first step of the compression algorithm is to check if the original number can be compressed using the repdigit. If the solution is not found, all the factors of the target number are found, and the repdigit approach is used. This combination of repunit and repdigit approach is expected to compress a majority of the number but may not always work when the target number is a prime number or not a repdigit composite

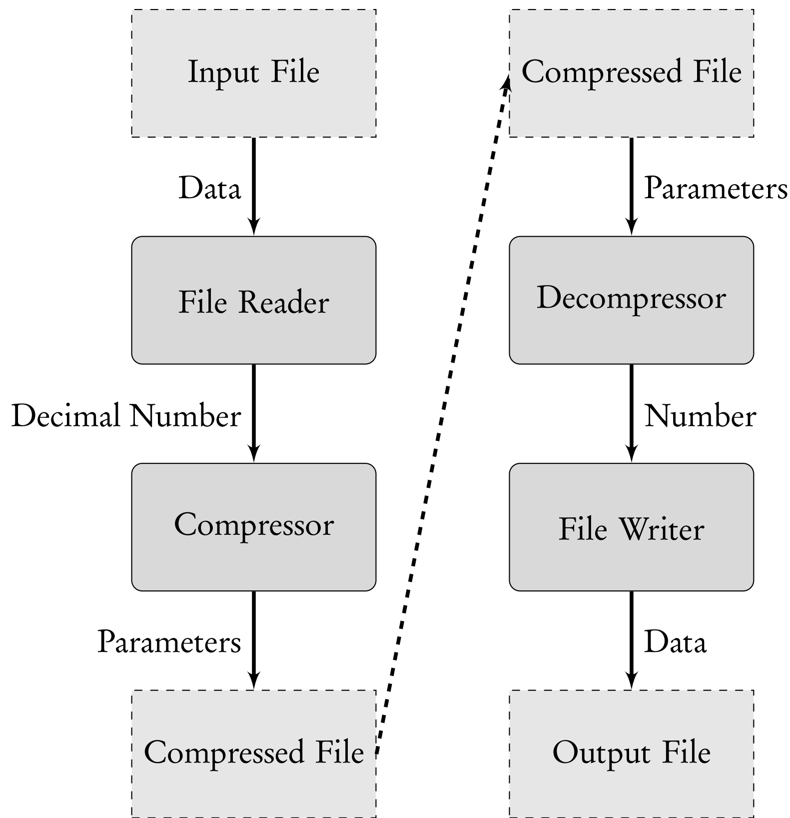

Figure 4.13: Pipeline for the Process of Compression and Decompression

number. In that case, the neighbor or the padding approach is used on the top of repdigit approach.

4.10 Encoding the variables in the compressed file

This section discusses the encoding guidelines used to store the parameters generated by the compressor component in a binary form. The parameters generated by the compressor are in decimal form. A separator character is required to separate the various parameters. A character is required to reflect the end of file. Therefore, the file will contain twelve characters - ten decimal numbers, one separator and one EOF(end of file) character. The nearest power of two that is larger than 12 is $2^4 = 16$. Therefore, any of these characters can be distinguished if each character is stored using 4 bits. Each digit of

```
read target_number
// Repunit
if(repunit(target_number))
   return b and n
else
   diff = 0
   while true
      // Repdigit
      for each factor of target_number
         new_target_number = target_number/factor
         // Repunit
         if(repunit(new_target_number))
            return factor, b, n, diff

      // Neighbors Approach for numbers below the target number
      // Can be replaced with Padding Approach
      target_number = target_number - 1
      diff = diff + 1
```

Figure 4.14: Pseudocode for the Merged Approach

the parameters is represented by the binary representation given in the table 4.3. The representation of the digits is done by simple conversion from base 10 to base 2. The order in which the parameters are stored also affect the final size of the compressed file. The parameters for B and N are always required whereas the other parameters may not always be used. Other parameters are - factor/repeated digit (repdigit approach), difference (neighbors approach) and number of padded digits (padding approach). The

repdigit approach is always used inside the neighbor or padding approach which means the `factor` parameter will always be there if any these approaches is used. Neighbors and Padding approach are two different approaches for almost the same thing. Only one of them is used at a time and there is no conflict in their parameter orders for them.

Repunit Approach	Base, N
Repdigit Approach	Base, N, Factor
Neighbors Approach	Base, N, Factor, Difference
Padding Approach	Base, N, Factor, Total Padded Digits

Table 4.2: Order of Parameters in the Compressed File for Different Approaches

Character	Binary Representation
Digits 0 to 9	0000, 0001, 0010, ..., 1001
Separator	1010
End of File	1011
Reserved for future use	1100, 1101, 1110, 1111

Table 4.3: Binary Representation for the Used Encoding

CHAPTER 5
RESULTS AND EVALUATION

This chapter starts with the results of various approaches discussed in Chapter 3 and how the various changes like variable reuse and padding approach help in improving the performance of the compression technique. It continues with the evaluation and comparison of the proposed techniques with other existing techniques. The chapter ends with the conclusion of the research and gives an idea of the future work that should be done to improve the performance and scope of the proposed technique.

5.1 Results

This section states the results of the Repunit approach and the CPU/GPU tests done to check the scope of the approach. It contains the results from the repdigit approach and quantitative idea about the improvements achieved by the variable reuse in the GMP code. It compares the neighbors and padding approach and mentions the pros and cons of each approach.

5.1.1 Repunit Approach

This section discusses the results from the Repunit approach discussed in section 4.3. After the implementation was done, different sets of numbers were given to the developed program but it was not able to compress majority of the numbers. To check how many numbers can be compressed using that implementation, a test was written. This test was written in a single threaded manner. The total time taken by the program when run for first 600,000,000 numbers was more than 35 minutes. A multi-threaded version was also written but still it could only increase the performance by around 6-7 times, depending on the number of cores available on the test system. As an improvement, a GPU-based

program was written to perform the same test. The specifications of the system used for the CPU and GPU based tests are given in the figure 5.1. The results of both CPU and GPU based tests are discussed next in this section.

Processor	Intel Xeon CPU E5-2623 v4
Clock Speed	2.60 GHz
Total Cores	8
RAM Memory	30.0 GB
Operating System	Windows Server 2018 Datacenter
GPU	NVIDIA Quadro M4000
GPU Memory	15.0 (Shared) + 8.0 (Dedicated) GB

Table 5.1: Specifications of System used for both CPU and GPU Tests

CPU Implementation

The CPU implementation performed well in terms of memory usage because not in terms of run time. This was one of the major motivations to opt for the GPU implementation. There were 6 threads in the CPU implementation and all of them were recording the results in a local variable. There were two kinds of number distributions discussed in section 4.3.1 - Continuous and Discontinuous. The discontinuous distribution was expected to perform better because all the threads were getting even amount of work load. But instead of the main thread telling each worker thread about the work they have to perform, the discontinuous distribution leaves this job to the worker threads. The worker threads are responsible to calculate the number they have to test. The calculation required to do this is very minimal but it still creates some additional computation as compared to the continuous distribution. This is a trade-off

for the discontinuous distribution. Both the distributions work equally when the data size is equal but the work distribution will be uneven among the different threads if the count of numbers to be tested is increased.

Total Numbers	1 to 600,000,000
Numbers that can be compressed	25,643
Run Time (Continuous Range Distribution, 6 cores)	5 Minutes 24 Seconds
Run Time (Discontinuous Uniform Distribution, 6 cores)	4 Minutes 50 Seconds
Run Time (Single Threaded)	$>$ 35 Minutes
Memory Usage	$<$ 1 Megabyte
CPU Usage	$\approx$ 80%

Table 5.2: Test Results from CPU Implementation for Repunit Approach

GPU Implementation

The GPU implementation shows a good amount of improvement in terms of the run time of the test. In the GPU algorithm, the result from each thread was stored in a different place in the array allocated in the GPU device. This was a trade-off for this implementation because the array on the GPU device led to high memory usage (3.8 GB). This high memory usage can be managed to some degree but will increase the run time.

The main result from both of these tests is that only 25,643 out of first 600,000,000 (0.000042%) numbers can be compressed using the Repunit approach. This makes this approach (in its current form) very impractical for the real-life data because the data is at most 9 digits. The data in real life has no bounds and the probability of a number being compressed by this approach is even lesser. This problem lead the research to make

Total Numbers	1 to 600,000,000
Numbers that can be compressed	25,643
Run Time (256 threads)	$\approx 6-8$ Seconds
Memory Usage	3.8 Gigabytes
CPU Usage	$\approx$ 13%

Table 5.3: Test Results from GPU Implementation

changes to this approach in order to compress more numbers, which eventually lead to the factorization method or the equation of sum of geometric series.

5.1.2 Repdigit Approach

This approach is a extension of the Repunit approach and introduced a new variable to the equation. This approach showed better results because if a number is not compressible by the Repunit approach, this approach is able to compress the number if the quotient found by dividing the target number by one of its factors, is compressible by the Repunit approach. If this approach is used, the newly introduced variable (factor by which the target number is divided) also needs to be stored as the part of the compressed file. This decreases the compression ratio of the algorithm but increases the solution space of the algorithm. The process of finding the factors for each number on a different GPU thread is not very efficient. So, GPU implementation was not created to test this approach. A CPU based test program was written to find the numbers that can be compressed using this approach. The test was not done on a large set of data as in the case of the Repunit approach. This test was done to find some kind of pattern between the numbers that can not be compressed using this approach.

The result from the test is that it cannot compress a prime number if the prime number

itself is not compressed using the Repunit approach. Also, there exists many composite numbers that cannot be compressed using this approach because dividing those number by any of the factors does not produce a number on which the Repunit approach can be used. These same results were also discussed by Bernard Schott in [30].

5.1.3 Improvements in memory requirements with variable reuse

Section 4.7 discusses how the variable reuse is used to decrease the memory requirements of the program. This section discusses the results of that implementation. Table 5.4 shows the memory usage and allocations of the program as given by Valgrind. There are three different sizes of numbers that are given to the program. In the older version of the program, there is an increase in memory allocations and usage when the input size increases but this does not happen in the new version. There can be a slight difference in the memory usage of the newer version because of the approaches involved and the size of the input number but the overall memory requirements are roughly the same.

Input Size	4 Digits	16 Digits	640 Digits
Allocations (Old)	119	3,024	211,860
Memory (Old)	84,940	153,809	29,996,619
Allocations (Improved)	16	18	17
Memory (Improved)	87,104	87,203	90,077

Table 5.4: Memory Requirements of Older Version of the program

5.1.4 Comparing neighbors and padding approach

Both of these approaches are able to compress any number that is given to them. Internally they both apply Repdigit approach to compress the number. Neighbors approach applies the repdigit approach to the neighboring numbers of the target number ($S \pm D$) as opposed to the padding approach which adds some digits to the end of the target number and applies the repdigit approach to that number. In the neighbors approach, the difference between the neighbor number and the target number is stored in the compressed file. This difference can range from 0 to quite a large number depending on the size of the number being compressed. In the padding approach, the number of digits that were added to the target number need to be stored in the compressed file. This value does not grow as fast as the difference value of the neighbors approach. It generally ranges between 2-6 and can be stored in as little as 3 bits of memory. But even if the difference value is say 100, will require 5 bits. This is how the padding approach is better than the neighbors approach in terms of compression ratio.

There is a tradeoff with the padding approach. Suppose the target number is N and the nearest number that can be compressed using the Repdigit approach is $N-1$, then the neighbors approach will find it in the first or second iteration. But if the padded approach is used, it may not be able to find it in several different iterations. Also, when digits are added to the end of the target number, the size of the target number increases. Thus the calculation is now done on a bigger number and increase the run time of the algorithm. This is how the run time of the padding approach can be more than neighbors approach.

5.2 Evaluation

This section discusses the evaluation of the developed algorithms in terms of compression ratio, compression time, and decompression time. Multiple tests are

performed to evaluate the developed algorithm. Bitmap images generated by the data generator discussed in the section 4.8 are used for the algorithm evaluation. Two levels of tests are performed. First case is when the data is a Repunit and the algorithm can directly use the Repunit approach. Second is when the data is a repdigit and the algorithm has to find all the factors of the data number and apply repdigit approach.

5.2.1 Test 1 - Repunit Approach Only

In the test 1, the algorithm is able to achieve very good compression because of test being the best case scenario. The algorithm only checks if the number is a Repunit and immediately gets the solution without the other approaches. In this case because the solution consists only two variables that need to be stored, the encoding of the solution is even more compact. The solution consists of two variables, one seperator character and one EOF (end of file) character.

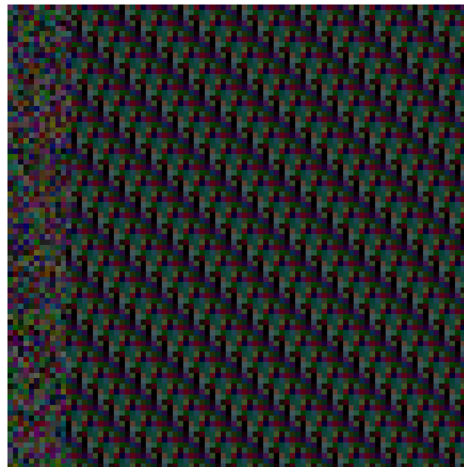

Figure 5.1: Image used for Test 1

File Size	22.5 KB
Compression Time	10.7 Seconds
Compressed File Size	48 Bits
Decompression Time	0.005 Seconds
Compression Ratio	30000:1 (99.996%)
Number of times Repunit check is done	1

Table 5.5: Evaluation Metrics for Test 1

5.2.2 Test 2 - Repdigit Approach

Following are the results of two tests done on two different images for which only the value of N is different. This test is done to demonstrate the impact of the value of N on the total run time when all the other factors are kept constant. The two tests are named as A and B. In the first test, data consists of a repdigit which means the solution will consist of three variables. The encoding of the solution in this case is done as three variables, two separator symbols, and one end of file symbol. In the test 2B, the value of N in test 2A is 1000 whereas it is 2000 in test 2B. The difference in size of the compressed files (80 Bits, 96 Bits) in both the tests is not very significant but the time taken by the test 2B is almost 7 times the time taken by test 2A. This shows how the value of N in the result hugely affects the computation time. The details of this observation are discussed in the section 5.4.4.

5.3 Comparison with existing algorithms

This section discusses the performance of the developed algorithm with the other existing techniques using the three image files that were compressed in the section 5.2.

The given table shows how the existing data compression techniques can compress the

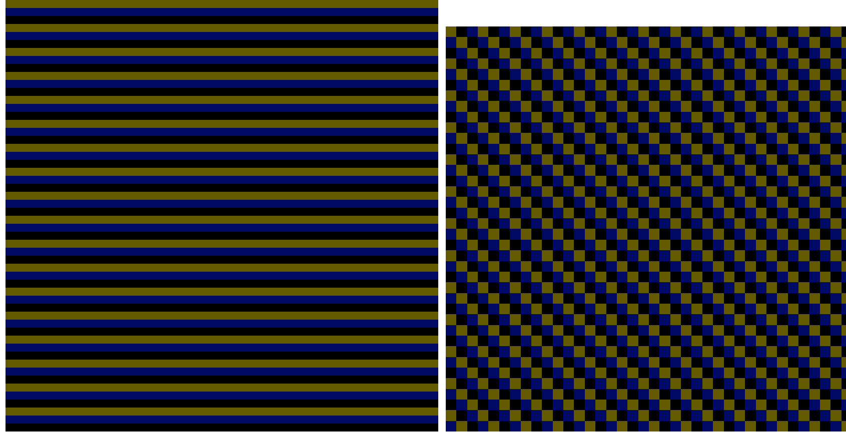

Figure 5.2: Image used for Test 2A (left) and 2B (right)

File Size	4.36 KB
Compression Time	1 Minute 7 Seconds
Compressed File Size	80 Bits (10 Bytes)
Decompression Time	0.002 Seconds
Compression Ratio	436:1 (99.77%)
Number of times Repunit check is done	1000

Table 5.6: Evaluation Metrics for Test 2A

File Size	8.7 KB
Compression Time	7 Minutes 38 Seconds
Compressed File Size	96 Bits (12 Bytes)
Decompression Time	0.009 Seconds
Compression Ratio	742.5 (99.865%)
Number of times Repunit check is done (N)	2000

Table 5.7: Evaluation Metrics for Test 2B

Algorithm	File - Test 1	File - Test 2A	File - Test 2B
Original File	22.5 KB	4.36 KB	8.7 KB
Brotli	2.84 KB	59 Bytes	62 Bytes
BZIP2	3.36 KB	99 Bytes	107 Bytes
LZIP	2.94 KB	103 Bytes	111 Bytes
XZ	2.96 KB	132 Bytes	144 Bytes
GZIP	3.16 KB	110 Bytes	165 Bytes
ZST	2.95 KB	77 Bytes	71 Bytes
LZOP	4.17 KB	183 Bytes	185 Bytes
LZ4	4.03 KB	121 Bytes	116 Bytes
Thesis Approach	6 Bytes	10 Bytes	12 Bytes

Table 5.8: Comparison of developed approach with the existing techniques

files upto some level but all of them are not able to go beyond a limit. Example: In case of file from test 1, all the existing techniques are able to compress the file from 22.5 KB to a range of 2 to 4 KB but the algorithm developed in the thesis go beyond that limit and compress it to 6 bytes. All the existing techniques try to compress the files on the basis of the repetition in the files but the thesis approach changes the complete representation of the data and repetition in the data using the concept of Repunit/Repdigit, and takes advantage of that repetition to compress the data beyond the current limits.

5.4 Conclusion and Future Work

This section discusses the conclusion of the whole study done in the context of compressing data as a solution to the Diophantine equations. Multiple tests were performed using the developed algorithm but were done on the file generated by the

program discussed in section 4.8. Compression Ratio and compression time were recorded for all of them, and various changes were made to improve them. There were significant improvements in space complexity but the time complexity of the algorithm is still high. Due to high run time, it was not possible to perform testing on big files.

5.4.1 Limitations and Potential Applications

The testing done in this thesis is limited to image files and the content of the files is either Repunit or Repdigit. The neighbors and padding approach was not involved for the testing as they significantly increase the computation time. One important conclusion is that the current implementation of the algorithm can have very high compression time for a given file. However, the decompression time is still low. Therefore, the potential area of application for the current implementation is compressing the data that is compressed once and distributed to various poor network areas. Example: A YouTube video is to be transmitted to multiple rural areas in South Africa. YouTube has all the powerful computer systems and can use this thesis' algorithm to compress the video and achieve great compression ratios. The compressed file can be transmitted over the network and can be decompressed on any average computer system in South Africa. There are several recommendations to improve the performance of the developed algorithm and are discussed in the following sections.

5.4.2 Calculations directly on Binary or Hexadecimal Numbers

In the current implementation, the whole data is converted from binary form into a decimal number and all the discussed approaches are used on that decimal number. However, this is not efficient because if the whole data is converted into one decimal number, is computationally heavy. Otherwise, if the binary data is broken into equal

chunks and then converted into decimal base, is not efficient for memory. The solution to this is to do the computation in the binary base. This will remove the need to convert into other bases. Another solution is to do the computations in hexadecimal base because 16 is a direct power of 2 ($16 = 2^4$) converting the binary data into hexadecimal base is more efficient.

5.4.3 Dividing the data into smaller divisions

In the current implementation, the whole data is taken a single decimal number but as an improvement, the data can be divided into multiple divisions and the compression can be done on these smaller chunks. In this way, the computation will be done on multiple smaller integers and can be done on multiple CPU cores or GPU devices. This will result in significantly less compression time but there will be a trade off in terms of compression ratio. So, a balance between the compression time and expected compression ratio is required to proceed with this approach.

5.4.4 Effect of value of N on run time

When the Repunit approach compresses a file, it takes it $\log_2 S$ (S is the target number) loops in the worst case scenario. In each loop, it calculates the B's and checks if the two values (B and N, where N is the loop count) can generate the original data. The current implementation of the algorithm starts from the value of N as 2 and goes up to $\log_2 S$. Suppose the value of N in a solution given by the Repunit approach is N_{Result}. Then if the value of N_{Result} is small, the algorithm did not check all the values of N but gets the result in first few iterations. In the worst case, where the value of $N_{\text{Result}} = \log_2 S$, the algorithm needs to go through all the values of N. From this observation, it is concluded that the way the algorithm iterates through the value of N massively affects the time taken

by the algorithm. If the algorithm iterates through all the values that are close to N, it will be able to find the result more quickly. As a future work, it is recommended to study how the value of N grows with the value of S using the Repunit approach and find a relation between the value of N_{Result} and S. This relation can be used to predict the range of numbers in which the value of N_{Result} is more likely to exist, the algorithm will take less time to find the result, leading to a less compression time.

5.4.5 Implementing the algorithm on GPU

Section 5.1.1 discusses the results of the Repunit check tests done on two different platforms - CPU and GPU. The GPU implementation showed how the algorithm performance can be accelerated by dividing the workload across hundreds of threads of the GPU device. The current implementation developed in the thesis uses GMP library for large integer calculations and is a CPU based library. As a future recommendation, the developed program can be migrated to a GPU version. The GPU will be used to perform large integer calculations. There exists a GPU based library called CAMPARY [18] which performs large integer calculations using NVIDIA CUDA. The repdigit approach requires finding the factors of a number and applies the Repunit check for each factor. For a large target numbers, there are a lot of factors and each Repunit check can be performed on a separate GPU thread. The padding approach can also be parallelized on the GPU platform when the number of padded digits are more than 2. In such cases, there are a lot of numbers that are further checked with the repdigit approach.

5.4.6 Trying different factorization techniques

The repdigit approach requires finding the factors of the number. There can be different types of factors as discussed in the section 4.4. As a future recommendation,

the repdigit approach should be performed for a significantly large set of numbers. The value of `factor` parameter in the solution of each number should be stored. There are multiple questions and corresponding results to check for the stored values and are given below. The time complexity of the algorithm will improve if the prime factors are proven to be the factors of interest. There are much more efficient factorization algorithms like Lehman's algorithm, MPQS, NFS and rho's algorithm [27] and should be incorporated in the future implementations of the repdigit approach.

- How many values are prime or non-prime?
  If the majority of values are prime, mean that only prime factors are the factors of interest. In that case, the factorization technique of the repdigit approach should be changed to algorithm that only computes the prime factors.

- How many values are less than the square root of the target number?
  If majority of the values are lesser than square root of the target number, only these factors should be computed.

5.4.7 Primality Check to avoid unnecessary factorization

Primality check of a number is the process of checking if a number is prime or not. This can be added to the repdigit approach to check if a number is prime or not before starting the factorization. The factorization won't be required in case of prime numbers. A test should be done between two implementation - With primality check and without primality check. The primality check is expected to avoid unnecessary factorization of the target number because the growth function of the prime number is strictly greater than the growth function of natural number [8].

These were the conclusions and recommendations that will drive the future work of the program. Other mathematical equations will be tested in both Diophantine and

Non-Diophantine manner for the possibility of data compression. All the current algorithms in the different components of the program will be replaced with more efficient and suitable algorithms. The results in the thesis offers promising results in the field and shows that the proposed technique holds the potential to be accepted for the real-life data storage and transmission purposes in the future.

[1] "Various mathematical constants to 1 billion digits", URL https://archive.org/details/Math_Constants (2016).

[2] "Xz utils", URL https://tukaani.org/xz/ (2020).

[3] "The gnu mp bignum library", URL https://gmplib.org/manual (2021).

[4] "Lz4 - extremely fast compression", URL https://lz4.github.io/lz4/ (2021).

[5] Beiler, A., *Recreations in the Theory of Numbers: The Queen of Mathematics Entertains*, Dover Recreational Math Series (Dover Publications, 1964).

[6] Bohr, M., "A 30 year retrospective on dennard's mosfet scaling paper", IEEE Solid-State Circuits Society Newsletter **12**, 1, 11–13 (2007).

[7] Bond, M., N. Nethercote, S. Kent, S. Guyer and K. McKinley, "Tracking bad apples: Reporting the origin of null and undefined value errors", vol. 42, pp. 405–422 (2007).

[8] Brilleslyper, M., N. Wakefield, A. Wallerstein and B. Warner, "Comparing the growth of the prime numbers to the natural numbers", Fibonacci Quarterly **54**, 65–71 (2016).

[9] Cook, S., *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012), 1st edn.

[10] Deutsch, P., "Deflate compressed data format specification version 1.3", URL https://www.ietf.org/rfc/rfc1951.txt (1996).

[11] Deutsch, P., "Gzip file format specification version 4.3", URL https://www.ietf.org/rfc/rfc1952.txt (1996).

[12] Diaz, A. D., "Lzip - lzma lossless data compressor", URL https://www.nongnu.org/lzip/ (2021).

[13] Esmaeilzadeh, H., E. Blem, R. St. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling", SIGARCH Comput. Archit. News **39**, 3, 365–376, URL https://doi.org/10.1145/2024723.2000108 (2011).

[14] Gantz, J. and D. Reinsel, "Extracting value from chaos", pp. 1–12 (2011).

[15] Gourdon, X., "Pifast : the fastest windows program to compute pi", URL http://numbers.computation.free.fr/Constants/PiProgram/pifast.html (2003).

[16] Huffman, D. A., "A method for the construction of minimum-redundancy codes", Proceedings of the IRE **40**, 9, 1098–1101 (1952).

[17] J. Alakuijala, Z. S., "Brotli compressed data format", URL https://tools.ietf.org/html/rfc7932 (2016).

[18] Joldes, M., J.-M. Muller, V. Popescu and W. Tucker, "Campary: Cuda multiple precision arithmetic library and applications", Mathematical Software – ICMS 2016 Lecture Notes in Computer Science p. 232–240 (2016).

[19] Lelewer, D. A. and D. S. Hirschberg, "Data compression", ACM Comput. Surv. **19**, 3, 261–296, URL https://doi.org/10.1145/45072.45074 (1987).

[20] Nelson, M. and J.-L. Gailly, *The Data Compression Book (2nd Ed.)* (MIS:Press, USA, 1995).

[21] Nethercote, N. and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation", in "Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation", PLDI '07, p. 89–100 (Association for Computing Machinery, New York, NY, USA, 2007), URL https://doi.org/10.1145/1250734.1250746.

[22] Noe, T. and T. O. F. Inc., "The on-line encyclopedia of integer sequences", URL https://oeis.org/A220136 (2012).

[23] NVIDIA, "Cuda c++ programming guide", (2021).

[24] Oberhumer, M. F., "oberhumer.com: Lzo real-time data compression library", URL https://www.oberhumer.com/opensource/lzo/ (2017).

[25] Rathore, Y., M. K. Ahirwar and R. Pandey, "A brief study of data compression algorithms", Networking and Communication Engineering **5**, 9 (2013).

[26] Reinsel, D., J. Gantz and J. Rydning, "The digitization of the world - from edge to core", (2020).

[27] Richard P., B., "Recent progress and prospects for integer factorisation algorithms", Tech. rep., GBR (2000).

[28] Sanders, J. and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming* (Pearson Education, 2010).

[29] Sayood, K., *Introduction to Data Compression, Fourth Edition* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012), 4th edn.

[30] Schott, B., "Les nombres brésiliens", http://dx.doi.org/10.1051/quadrature/2010005 (2010).

[31] Seward, J., "bzip2: Documentation", URL https://sourceware.org/bzip2/docs.html (2019).

[32] Smart, N. and J. Bruce, *The Algorithmic Resolution of Diophantine Equations: A Computational Cookbook*, London Mathematical Society Student Texts (Cambridge University Press, 1998).

[33] Stone, J. E., D. Gohara and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems", Computing in Science Engineering **12**, 3, 66–73 (2010).

[34] Sutter, H., “The free lunch is over a fundamental turn toward concurrency in software”, (2013).

[35] Wells, W., *Advanced Course in Algebra*, Wells’ Mathematical Series (D.C. Heath & Company, 1904).

[36] Y. Collet, E., M. Kucherawy, “Zstandard compression and the application/zstd media type”, URL https://tools.ietf.org/html/rfc8478 (2018).

[37] Yee, A. J., “Y-cruncher - a multi-threaded pi-program”, URL http://www.numberworld.org/y-cruncher/ (2021).

[38] Ziv, J. and A. Lempel, “A universal algorithm for sequential data compression”, IEEE Transactions on Information Theory **23**, 3, 337–343 (1977).