Department of Computer Science and Engineering - University of New South Wales

Compression of Text using Canonical Huffman Encoding

COMP4601 Project Report

Authors

Quynh Boi Phan (z5205690)

Khang Trinh (z5211203)

Jasmine Young (z5206275)

Date Submitted: 06 August 2021

Table of Contents

1 Introduction

The traditional *Huffman Encoding* (HE) algorithm is used for compression of data. By taking the frequency of the characters in a text file, a sorted binary tree is generated with the leaf nodes carrying the characters. The more frequent characters would be closer to the root. By traversing through the tree, a *Huffman Code* (HC) is created for each character that represents the path from the root. This code is then used to encode the characters in the text file, subsequently compressing the file.

For traditional HE, the decoding process required traversing a tree which is complex and utilises a lot of hardware components. To avoid this, we use *Canonical Huffman Encoding* (CHE). Although it takes the same input as the HE algorithm, the generated table of *Canonical Huffman Codes* (CHC) however allows simple encoding and decoding, subsequently reducing the memory utilisation and computational hardware.

2 Project Objectives

The objective of this project is to compress a text file using the CHE algorithm in hardware. Once completed, the CHE algorithm is accelerated to achieve better performance. To verify the correctness of the compression algorithm, the compressed file is also decompressed to compare the decompressed file with the original text file.

3 Investigation

The project comprises three different components: text file compression using CHE, decompression of compressed file, and acceleration of the CHE algorithm.

3.1 System Design

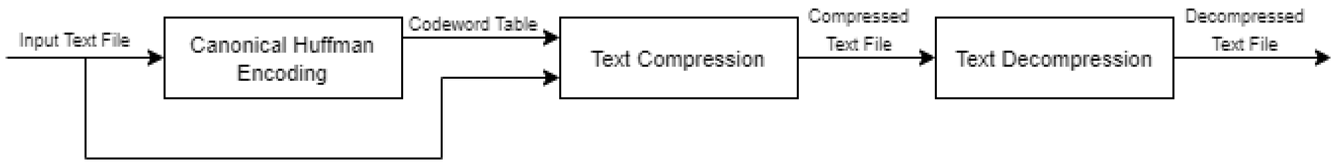

Figure 3.1: System design flowchart.

Figure 3.1 depicts the flow of the main system, in which the input text file must first be passed into the CHE algorithm to generate the CHC of each character. The characters of the input text file are then converted to its bit encoding from the CHC table. Every 8 bits is then converted into a char to create the compressed file. The correctness of the compressed file is then verified using the decompression algorithm, in which the content inside the decompressed text file should be the same as the input text file.

3.2 Compression of Text

The compression algorithm reduces the size of a text file by encoding characters into shorter bit lengths. In this project, the CHE algorithm (in *Figure 3.2*) generates the CHC for the ASCII characters, including the extended ASCII, to encode the characters in the text file.

3.2.1 Canonical Huffman Encoding Algorithm

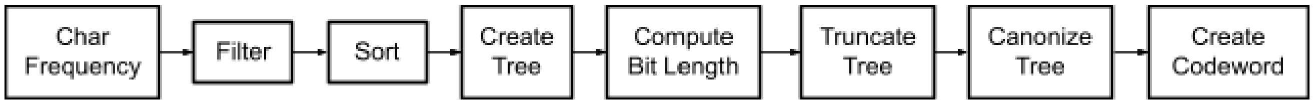

Figure 3.2: Canonical Huffman Encoding flow chart.

The CHE process begins with *filtering*, by removing unused characters from the frequency table and counts the number of non-zero frequency characters. It then *sorts* the input characters based on their frequency values, with the least frequent characters at the beginning, to *create a Huffman binary tree* based on the sorted list order. The tree generated would allow more frequent characters to have shorter node depths in the tree, resulting in shorter codeword lengths. From the generated tree, it *computes the bit length* to count the number of leaf nodes in each depth of the tree. Tree nodes with a depth longer than the maximum codeword length are then reorganized in the *truncate tree* process to avoid excessively long codewords. The tree is then *canonized* (in *Figure 3.3*) to determine the number of bits for each character to finally *create the codeword table* (in *Figure 3.4*) which contains the encoding of each character.

```
ap_uint<SYMBOL_BITS> length = TREE_DEPTH;
ap_uint<SYMBOL_BITS> count = 0;
// Iterate across characters from lowest frequency to highest
// Assign lowest frequency characters with largest bit length and vice versa
process_symbols:
    for(int k = 0; k < num_symbols; k++) {
        if (count == 0) {
            do { // find next non-zero bit length
                    length--;
                    count = codeword_length_histogram[length]; // n = # of characters with this bit length
            } while (count == 0);
        }
        symbol_bits[sorted[k].value] = length; // assigns bit length to characters
        count--; // keep assigning that bit length until we counted off n characters
    }
```

*Figure 3.3: The **canonize tree** module, returns a table of characters and its node depths.*

```
Codeword first_codeword[MAX_CODEWORD_LENGTH]; // MAX_CODEWORD_LENGTH = 27
first_codeword[0] = 0;
first_codewords: // Computes the initial codeword value for a symbol with bit length i
    for(int i = 1; i < MAX_CODEWORD_LENGTH; i++)
        first_codeword[i] = (first_codeword[i-1] + codeword_length_histogram[i-1]) << 1;
        Codeword c = first_codeword[i];
    }

assign_codewords: // modified the codewords base on the order of each character
    for(int i = 0; i < INPUT_SYMBOL_SIZE; ++i) {
        CodewordLength length = symbol_bits[i];
        make_codeword:
            if(length != 0) {
                Codeword c_out = first_codeword[length];
                // encoding: bits [32:5] for codeword, bits [4:0] for codeword length
                encoding[i] = (c_out << CODEWORD_LENGTH_BITS) + length; // len_bits = 5
                first_codeword[length]++; // codes with the same length differs by 1 bit
            } else encoding[i] = 0;
    }
```

*Figure 3.4: The **create codeword** module, generates a codeword table for each character.*

3.2.2 Compression Algorithm

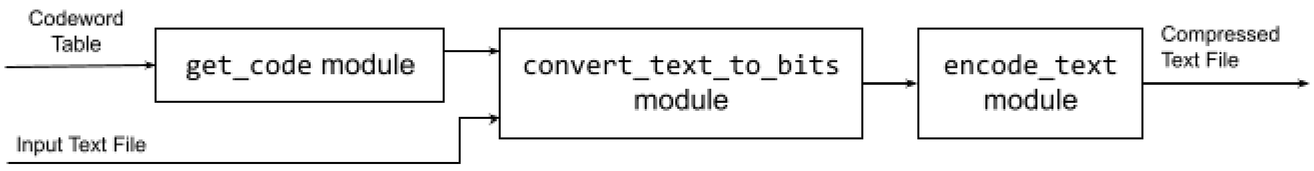

Figure 3.5: Compression algorithm flow chart.

The first step of the compression process is to convert the text file into a bitstream of CHC. The `convert_text_to_bits` module (*Figure 3.6*) replaces every character of the input text file by their generated CHC from the codeword table. It then constructs a bitstream based on each character's occurrence. Next, the `encode_text` module (*Figure 3.8*) converts every 8 bits of the bitstream to its equivalent standard unsigned ASCII char value and prints it into the output file. The compressor also stores the codeword table, and other trivial information that helps the decompressor decode the compressed file.

```c
int convert_text_to_bits(FILE *fi, FILE *ft, PackedCodewordAndLength *encoding,
                         int *extra) {
    unsigned int n, i, code, lastXbits, ex;
    int c, count = 0;
    while ((c = fgetc(fi)) != EOF) {
        n = (unsigned int) encoding[c]; // get codeword with padding leading bits
        code = get_code(n, &lastXbits); // extracted the codeword without leading bits
        print_bit_encoding(ft, code, lastXbits - 1); // print encoding into temp file
        count += lastXbits; // calculate the length of the total bit stream
    }
    ex = 0; // the bitstream length must divisible by 8 so every bit is turn into a char
    while (count % 8 != 0) {
        fprintf(ft, "0" ); // padding the bitstream so its length is divisible by 8
        count++;
        ex++;
    }
    *extra = ex;
    return count;
}
```

Figure 3.6: The ***`convert_text_to_bits`*** *module, converts the input text file into a binary bitstream by replacing each character in the text by their canonical codewords.*

```c
unsigned int get_code(unsigned int n, unsigned int *lastXbits) {
       *lastXbits = n & MASK; // n & 31 (0b11111) to get last 5 bits - the code length
       unsigned int code = (((1 << *lastXbits) - 1) & (n >> (5)));
       return code;
}
```

Figure 3.7: The ***`get_code`*** *module, gets the code word of a character from the codeword table, the module extracts the necessary bits and removes unused leading bits.*

```c
void encode_text(FILE *fo, FILE *ft) {
    int c, interval = 0;
    unsigned char writing = 0;
    while ((c = fgetc(ft)) != EOF){
        if (c == '1') {
            writing = writing << 1;
            writing += 1;
        } else if (c == '0') writing = writing << 1; // reading from the bit stream
        if(interval == 7) { // turn every 8 bits into a standard unsigned ASCII char
            fprintf(fo, "%c", writing);
            interval = 0;
            writing = 0;
        } else interval++;
    }
}
```

Figure 3.8: The ***encode_text*** *module, converts every 8 bits of the binary bitstream into a char, and prints it into the output file.*

3.3 Decompression of Text

The decompression algorithm reverts the compressed text file into its original form. The main purpose of this algorithm is to verify that the text file was successfully compressed without any error.

3.3.1 Decompression Algorithm

Upon opening the compressed file, every character and its encoding in the text file is read, including the number of bits to read and the number of characters in the compressed text. When completed the algorithm proceeds to reading the compressed text.

The compressed text is first translated into its bits form before translating back to the original text. To do this, the algorithm converts each character into its bit form and saves the bit form in a temporary file to save memory. This is useful if the bitstream is large.

```
    while((c = fgetc(fc)) != EOF) {          // While reading individual characters
        for(i = 7 ; i >= 0 ; i--) {          // from compressed file fc.
             bit =  c & (1 << (i));          // Get bit at i index of character.
             if(bit == 0) fprintf(ft, "0" ); // Save bit in a temporary file ft.
             else fprintf(ft, "1" );
         }
    }
```

Figure 3.9: Reading bits of every character in compressed file.

After saving the compressed text in the binary form, the bits are then read to retrieve the original text; decompressing the file. It does this by first adding one bit to a buffer string array, and then checking if the buffer temporarily matches with any of the characters' bit encodings. If there are multiple matches, the algorithm proceeds to add the next bit to the buffer and compare again. This subsequently reduces the number of matches until only one encoding matches with the buffer. With only one match found, the character is added into the decompressed file and the buffer is cleared to read the next set of bits.

```
bool is_match(char buffer[], char encoding[]) {
    // Encoded string would not match if the encoding is shorter than the buffer
    if (strlen(encoding) < strlen(buffer)) return false;
    for (int i = 0; i < strlen(buffer); i++) {
        if (buffer[i] != encoding[i]) return false; // The bits does not match
    }

    return true;
}
```

Figure 3.10: Checking if the buffer string temporarily matches with a character's bit encoding.

This process is repeated until all bits are read. The temporary file is then deleted upon completion.

3.4 Canonical Huffman Encoding Acceleration Design

As the CHE algorithm is implemented in hardware, this allows the opportunity for acceleration to reduce the compression time as it involves many operations. The CHE was optimised by creating a baseline solution on the `xc7z020clg484-1` device and determining which functions had the greatest latency. *Figure 3.11* shows that the sort function took the most time, so the majority of our focus went to analysing and optimising this function before attempting to optimise the other functions.

Instance	Performance					
	Max Latency (cycles)		Max Latency (absolute)		Max Interval Cycles	
	Baseline	Optimised	Baseline	Optimised	Baseline	Optimised
Sort	15258	4459	0.153ms	35.672us	15258	4459
Create_tree	1021	1276	10.210us	10.208us	1021	1276
Truncate_tree	1231	1018	12.310us	8.144us	1231	1018
Compute_bit_length	1082	796	10.820us	6.368us	1082	796
Create_codeword	822	270	8.220us	2.160us	822	270
Canonize_tree	2306	2306	23.060us	18.448us	2306	2306

Figure 3.11: Performance time of each function in the baseline and optimised solutions.

The optimisations performed include unrolling, task and cycle pipelining, array partitioning and rewriting some parts of the code. Loops that had arrays that were completely partitioned or an iteration latency of 1 were unrolled. This allowed some operations to be completed faster due to the increased memory accesses allowed. Other loops that did not have any inter dependencies were pipelined to increase concurrency, reduce the latency and keep the area cost smaller compared to unrolling. The directives applied are listed in *Figure 3.12*.

```
set_directive_loop_tripcount -min 254 -max 254 -avg 254 "compute_bit_length/traverse_tree"
set_directive_loop_tripcount -min 64 -max 64 -avg 64 "compute_bit_length/init_histogram"
set_directive_unroll "sort/init_histogram"
set_directive_pipeline "sort/compute_histogram"
set_directive_unroll "sort/find_digit_location"
set_directive_pipeline "sort/re_sort"
set_directive_pipeline "sort/copy_in_to_sorting"
set_directive_array_partition -type complete -dim 1 "sort" digit_histogram
set_directive_array_partition -type complete -dim 1 "sort" digit_location
set_directive_unroll "truncate_tree/copy_input"
set_directive_unroll "truncate_tree/copy_output"
set_directive_unroll "compute_bit_length/init_histogram"
set_directive_pipeline "compute_bit_length/traverse_tree"
set_directive_unroll "create_codeword/first_codewords"
set_directive_pipeline "create_codeword/assign_codewords"
set_directive_array_partition -type complete -dim 1 "create_codeword" first_codeword
set_directive_pipeline "filter/filter_label1"
set_directive_pipeline "huffman_encoding/copy_sorted"
set_directive_dataflow "huffman_encoding"
```

Figure 3.12: `directive.tcl` file of final solution.

3.5 Performance Results

3.5.1 Canonical Huffman Encoding Optimisation Results

Solution	Estimated Clock Period	Performance						Resource Utilisation		
		Latency (cycles)		Latency (absolute)		Interval Cycles		BRAM	FFs	LUTs
		Min	Max	Min	Max	Min	Max			
Baseline	8.333ns	4476	24546	44.760us	0.245ms	4476	24546	19	1011	3176
Optimised	7.000ns	2080	8838	16.640us	70.704us	911	4460	21	3420	8013

Figure 3.13: Synthesis result of baseline solution and optimised final solution.

From *Figure 3.13*, the latency has improved significantly with over 70% decrease in the maximum absolute latency. This increase in performance comes at the cost of a greater area. More than triple the number of

FFs were needed as some arrays were stored in registers and more pipeline registers were needed. Some pipelined iterations and unrolled loops could also not share the same operation hardware so more LUTs, FFs and BRAMs were used.

3.5.2 Compression using Canonical Huffman Encoding Performance Results

The performance of the compression algorithms and the compressed file size directly depends on the size of the input text file. From *Figure 3.14*, the total latency of the compression process increases at a linear rate, as shown by the orange curve. This trend still remains even after optimising the encoding process. The grey curve emphasises a very stable improvement of roughly 200 ms with the optimised design compared to the base design. This is regardless of the variations in the size of the input file. On the contrary, the output file size over input file size percentage results in the compressed file size decreasing as text size increases. The blue curve plateaued at around 10000 bytes and remains in the range of 55%-60%.

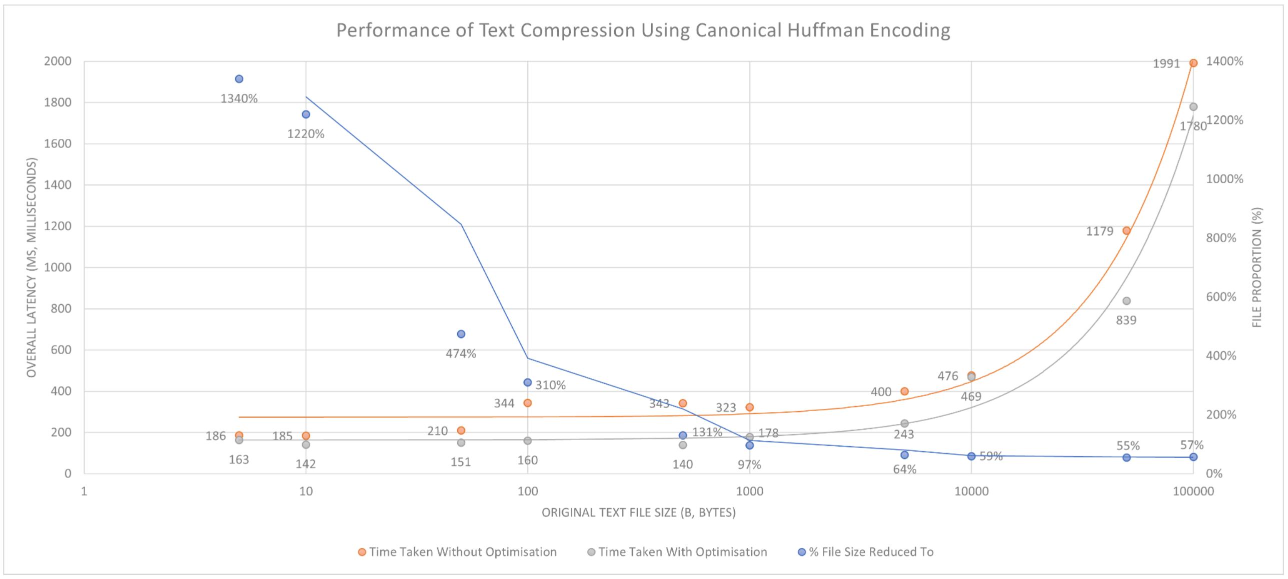

Figure 3.14: Performance result of compression algorithm.
NOTE: The x-axis (original text file size) is graphed as a logarithmic scale.

4 Discussion

4.1 Compression of Text Performance

Compressing large sized text files resulted in a significant decrease in the compressed file size as the number of bits needed for an ASCII character in the text is reduced based on the generated CHC.

A character requires 8 bits, however for example, if the CHC generated results in only 2 bits to represent the character, the character bits are joined together until it forms a new 8 bit character. Hence, a character in the compressed file may represent up to 4 characters of the original text. By continuously reducing the bits to encode the characters in the original file it will subsequently reduce the file size; compressing the file.

This is however not true when compressing smaller sized text files. As the character encodings are also stored with the encoded text itself, this would increase the file size as more space is needed to store the encoding as opposed to the space actually needed for the compressed text itself.

By increasing the file size, the performance time of the compression increases at a linear rate as more time is required to construct the frequency table, bitstream, and the output file. The frequency table is first constructed before passing as an argument in the CHE module. As it counts the frequency of each character inside the text file, increasing the file size would consequently increase the time to generate the table, therefore increasing the performance time. The increase in performance time is also true for the compression process itself, as it would also have to loop through *x* number of characters to encode after generating the CHC table.

The linear trend is also observed after optimising the CHE, however the latency of the process is reduced by 200ms. The linear trend is due to the unoptimised functions outside the CHE process, that is, the frequency table generation and the compression algorithm itself. These processes would still have the same runtime as the baseline solution, however they do not affect the CHC generation as they are outside the CHE algorithm. With only the CHE algorithm optimised, this allowed the consistent improvement in performance time we observed in the optimised solution over the base design.

4.2 Canonical Huffman Encoding Optimisation

4.2.1 Huffman Encoding

To optimise the top level function, the dataflow directive was used. This directive allowed the subfunctions to overlap which increased the concurrency and throughput and therefore decreased the latency. Although the function is only being executed once, if the compression was to be optimised later on, data could be streamed into the hardware.

The function also contained a for loop used to copy the sorted arrays and meet the dataflow requirements. This loop was pipelined and an initiation interval of 1 and an iteration latency of 2 were achieved. An initiation interval of 1 could be achieved because the memory holding the sorted array was dual-ported.

4.2.2 Filter

The filter function creates a new symbol array that does not include any symbols with a frequency of 0. It was optimised by pipelining the for loop. An initiation interval of 1 was achieved and the iteration latency was 5, reducing the total loop latency.

4.2.3 Sort

The sort function arranges the symbols based on their frequency in the input file. It is the most computationally intensive function in the CHE algorithm so multiple directives were needed to optimise the sort function and its subfunctions. The sort function involves many operations with arrays and some of them such as the `digit_histogram` and `digit_location` can be partitioned to allow for more concurrent accesses. With these arrays completely partitioned into registers, the `init_histogram` and `find_digit_location` loops could be unrolled. By unrolling these loops, the performance greatly improved as these loops only involved simple write and addition operations with partitioned arrays. Less resources were also used compared to pipelining these loops as less pipeline registers and loop bound logic were needed. The other loops in the sort function were pipelined as this allowed for more concurrent operations and better performance. An initiation interval of 1 was achieved for the `copy_in_to_sorting` loop and an initiation interval of 2 was achieved for the `re_sort` and `compute_histogram` loops due to the limited memory access on the input and output arrays.

Changing the radix value in the sort function was also explored. Decreasing the radix would decrease the number of items stored in the `digit_histogram` and `digit_location` arrays, the number of bits in `Digit` and the number of iterations of `find_digit_location` but it would also increase the iterations of the outer for loop, `radix_sort`. Although the area decreased with a smaller radix of 4, the number of `radix_sort` iterations increased, resulting in a worse performance. Increasing the radix also resulted in the synthesis not completing so a radix of 16 was kept.

4.2.4 Create Tree

The `create_tree` function creates the binary tree to represent the Huffman Codes. It could not be optimised because each iteration is dependent on the previous iteration as the tree is being built sequentially. These dependencies can be seen in the `in_count` and `tree_count` variables that store the number of inputs consumed and the number of intermediate nodes assigned a parent.

4.2.5 Compute Bit Length

The `compute_bit_length` function involves two loops, one to initialise the histogram array and another to traverse the tree. Loop unrolling was used for the `init_histogram` loop because it allowed two addresses from each array to be initialised at one time. Although array partitioning could further improve the performance, the Vivado HLS tool cannot optimise functions across their boundaries without a long synthesis time for code this large and one of the arrays is an output array. Partitioning one array would therefore not improve the performance as it is limited by the performance of writing to the output array.

To optimise the `traverse_tree` loop, it was pipelined as it allows for some of the operations to be completed in parallel. An initiation interval of 3 and an iteration latency of 5 were achieved. The initiation interval could not be further reduced because there is a memory access limit caused when `internal_length_histogram` is being read twice in the if/else conditions and also when its value is assigned to a `length_histogram` address.

The code was further optimised by rewriting the code to remove unnecessary variables and changing the nested if statements to an if/else if statement. This reduced the number of resources used as less registers and logic were needed.

4.2.6 Truncate Tree

The `truncate_tree` function has two for loops for copying the input and output arrays. This is used to pass the same data to the canonize tree and create codewords functions and meet the single consumer, single producer dataflow constraint. To optimise these loops, they were completely unrolled. Unrolling was used as it allows for two read/write operations to occur concurrently, reducing the overall loop latency.

The `truncate_tree` function also contains the main for loop, `move_nodes`, to reorder the tree if there are nodes at a length greater than the target length. This loop could not be optimised because each iteration is dependent on the previous iteration, as the reordering operations are done inplace.

4.2.7 Canonize Tree

The `canonize_tree` function has the next greatest latency after the sort function but it could not be optimised. The main `process_symbols` for loop has a do/while loop and an inter-loop dependency on the count variable that makes pipelining pointless. It also contains an initialising for loop that could be slightly optimised with unrolling. The amount of extra resources this would use makes it not worthwhile to do especially when there are many other optimisations that could be applied to the other functions.

4.2.8 Create Codeword

The `create_codeword` function is used to assign each symbol a codeword. It uses two loops, one for calculating the first codeword for each bit length and another to assign codewords. Since both loops involve reading and writing to the `first_codeword` array, array partitioning is used to help improve the performance when the loops are unrolled and pipelined. The `first_codewords` loop was unrolled because it allowed multiple adds and shifts to occur in one cycle and fill up the “dead time”. Pipelining would result in a worse performance as it only allows one add and shift in a cycle as each iteration depends on the previous. The `assign_codewords` loop was pipelined and achieved an initiation interval of 1 and an iteration latency of 3. This was achieved with the array partitioning as the shifting and addition operations on the `first_codeword` could occur in one cycle, removing the dependency between the loops.

4.2.9 Other Optimisations

Many of the loops involved initialising array values or copying arrays. These are simple operations that do not take much time and result in a lot of “dead time” in each cycle. To reduce the amount of “dead time”, the target clock period was varied from 6ns to 10ns to find the optimal clock period for better operation chaining. A target clock period of 8ns resulted in the best performance and any clock period below that did not meet the timing requirements.

4.3 Future Works

This project can be extended to accelerating the compression algorithm using the directives available in Vivado HLS. The algorithms also have an opportunity to be implemented in software to observe the performance between hardware and software solutions using different file sizes. The character encodings inside the compressed file could also be reduced to allow smaller sized files to be compressed further.

5 Project Management

5.1 Project History

Week	History
6	Choose and research HE and CHE. Began project plan and project plan presentation.
7	Finished project plan and project plan presentation. Revised project plan from feedback received to be clear about objectives before submitting. Test and fixed bugs of CHE code given from Chapter 11.
8	Began compression and decompression algorithm. Began different optimisation methods for CHE. Found out that compression and decompression must both be run on Vivado HLS for decompression to be successful because of compiler bugs.
9	Fixed bugs and cleaned up compression/decompression code. Finished optimising CHE. Began final report and presentation.
10	Finished final report, final presentation, and recorded video demonstration.

5.2 Teamwork

The tasks were divided as follows, with everyone successfully completing their tasks on schedule with the initial timeline set by the Gantt Chart:

- Quynh Boi Phan: Decompression algorithm
- Khang Trinh: Compression algorithm
- Jasmine Young: CHE acceleration and optimisation.

Using Facebook Messenger, members consistently communicated their progress on the project, and if a problem was encountered help was given almost immediately. Video calls were done on Microsoft Teams for the screen sharing features available. Everyone was able to view and solve the problems encountered together, and with the help of the *Control* feature, members were able to change parts of the code without having to describe where the changes should be made.

Code was shared using Github and everyone created their own branch to avoid tampering with each other's work. Files related to the report and presentation were created in Google Docs and Google Slides for collaborative effort and viewing the latest update of the files.

5.3 Reflection

There were many challenges faced in this project, however as a group, we managed to effectively work together in order to achieve the objective of compressing a file using the CHE algorithm. Furthermore, we were able to optimise the CHE algorithm to minimise the performance time of the overall system.

A lot of time was needed to have an in-depth knowledge of the CHE algorithm and the CHC generated. In return, this allowed a solution for the compression and decompression algorithm to be made easily. It also gave us an idea on the functions that can be optimised in the CHE algorithm. We also encountered many bugs that caused the file to not be properly encoded for compression, consequently causing the decompressed file to differ from the original input file. This however allowed the opportunity of learning and utilising the HLS debugging tools to help fix the code.

Therefore, through great teamwork, we managed to meet the scope of the project while expanding our knowledge on the CHE process and subsequently compressing a text file using CHE.

6 Reference

Kastner, R., Matai, J. and Neuendorffer, S., 2018, *Parallel Programming for FPGAs*, pp.207 - 235.