Journal of Real-Time Image Processing (2024) 21:84
https://doi.org/10.1007/s11554-024-01467-z

RESEARCH

Real-time lossless image compression by dynamic Huffman coding hardware implementation

Duc Khai Lam[1,2]

Received: 30 January 2024 / Accepted: 20 April 2024 / Published online: 7 May 2024

Abstract
Over the decades, implementing information technology (IT) has become increasingly common, equating to an increasing amount of data that needs to be stored, creating a massive challenge in data storage. Using a large storage capacity can solve the problem of the file size. However, this method is costly in terms of both capacity and bandwidth. One possible method is data compression, which significantly reduces the file size. With the development of IT and increasing computing capacity, data compression is becoming more and more widespread in many fields, such as broadcast television, aircraft, computer transmission, and medical imaging. In this work, we introduce an image compression algorithm based on the Huffman coding algorithm and use linear techniques to increase image compression efficiency. Besides, we replace 8-bit pixel-by-pixel compression by dividing one pixel into two 4-bit halves to save hardware capacity (because only 4-bit for each input) and optimize run time (because the number of different inputs is less). The goal is to reduce the image's complexity, increase the data's repetition rate, reduce the compression time, and increase the image compression efficiency. A hardware accelerator is designed and implemented on the Virtex-7 VC707 FPGA to make it work in real-time. The achieved average compression ratio is 3,467. Hardware design achieves a maximum frequency of 125 MHz.

Keywords Dynamic Huffman coding · Linear prediction · Real time · FPGA

1 Introduction

Image compression is an essential area of research in the field of data compression. Today, with social platform development, image and video data sizes are also increasing. Therefore, image and video compressions are being widely researched and developed because these are practical applications. Image compression methods are divided into two categories: lossy compression and lossless compression [13]. Both compression methods have advantages and disadvantages and are practically applied in many fields. Figure 1 shows the results of two methods of image compression. There are several image compression algorithms:

Lempel–Ziv–Welch (LZW) [15]: They are also effective for highly similar data sets, complex for large samples, and not very duplicated.

Shannon–Fano coding [7]: The descending probability sort method can better handle complex files, but the compression ratio is not good because it has not yet created a tree optimal binary.

Huffman coding [14]: The Huffman coding algorithm compresses data in a lossless form. This algorithm must browse the entire file before compressing and saving binary tree information for decompression.

In this work, we propose a dynamic Huffman coding to handle the limitations of the Huffman coding algorithm. Besides, we use it in conjunction with the linear prediction method to simplify the compressed data. In addition, we split pixels to make data simpler. It helps us to reduce the compression time and increase the compression rate. Finally, we propose the hardware design architecture to implement the proposed dynamic Huffman coding to make it work in real time. Our goal is to successfully implement the algorithm on the hardware platform, using Virtex-7 VC707 FPGA,

✉ Duc Khai Lam
khaild@uit.edu.vn

[1] University of Information Technology, Ho Chi Minh City, Vietnam

[2] Vietnam National University, Ho Chi Minh City, Vietnam

Fig. 1 Lossless and lossy compressed images

running at 400 MHz with frequency and achieve an average compression ratio of 2.5.

The rest of this work is organized as follows: Sect. 2 introduces the background of this work. Section 3 presents the proposed algorithm used in this work. Section 4 describes the hardware architecture of the proposed algorithm. Section 5 shows the experimental results of the proposed algorithm. Finally, Sect. 6 gives the conclusions of this work.

2 Background

2.1 Linear prediction

Prediction is a core part of compression methods. Because this method can minimize most of the redundant space between pixels [12], which improves the compression ratio, choosing a tool to optimize the redundancy between pixels is essential.

The linear prediction method is used to remove the redundancy between the pixels of the image, the predicted value is rounded to the nearest integer, so the error value prediction (the difference between the original value and the value after prediction) is also an integer [10]. This ensures that the reconstructed image is the same as the original image. The block diagrams of the linear predictive encoding and decoding system are shown in Fig. 2a and b.

The input image is first passed through the predictor, where the neighboring pixels are calculated according to the prediction template. The resulting values are then rounded to the nearest integer. Finally, the error value is obtained by calculating the difference between the original pixel value and the predicted value. The prediction model of the method is shown in Table 1

The top left and bottom right pixel values of the input image are retained. The pixel values at the other location are calculated to obtain the predicted value $P_{i,j}$, as shown in Eq. 1. Then, the predicted value is rounded, as shown in Eq. 2, and the error value $E_{i,j}$ is calculated based on the results of the original pixel value and the predicted value, as shown in Eq. 3.

$$P_{i,j} = \frac{1}{5}(I_{i-1,j-1} + I_{i-1,j} + I_{i-1,j+1} + I_{i,j-1} + I_{i,j+1}) \quad (1)$$

Table 1 Prediction linear model

$I_{i-1,j-1}$	$I_{i-1,j}$	$I_{i-1,j+1}$
$I_{i,j-1}$	$I_{i,j}$	$I_{i,j+1}$

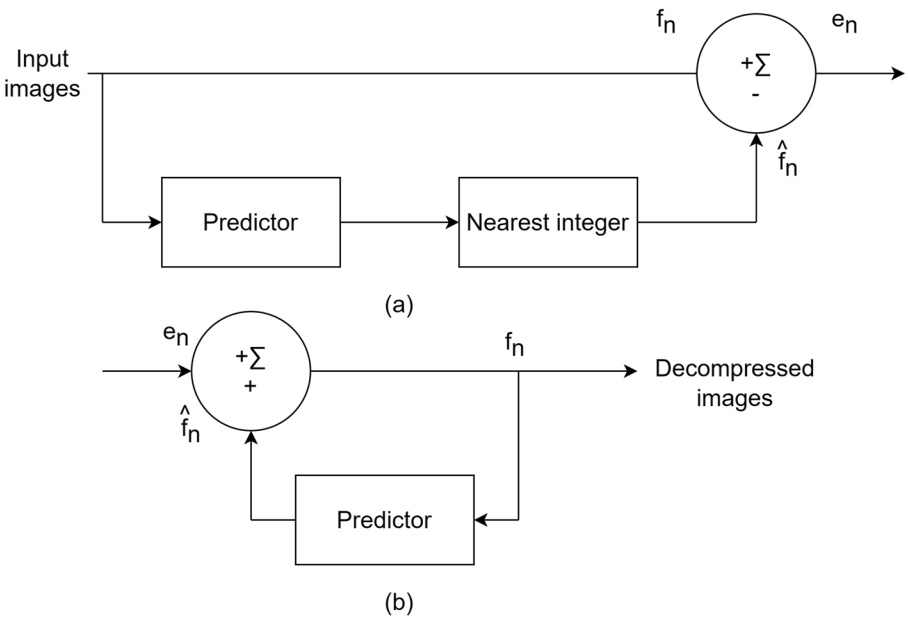

Fig. 2 Linear prediction algorithm

$$\overline{P_{i,j}} = round(P_{i,j}) \tag{2}$$

$$E_{i,j} = I_{i,j} - \overline{P_{i,j}} \tag{3}$$

3 Proposed method

3.1 Dynamic Huffman coding

The Huffman Coding algorithm [14] uses the probability distribution of the alphabet to develop codes for symbols based on a binary tree.

Huffman coding allows the creation of an optimized binary tree, significantly reducing the number of bits used per pixel, and improving the compression ratio. However, it requires knowing the input data in advance, calculating the probabilities, and arranging the data. In addition, the most significant disadvantage of Huffman Coding is that it has to save the tree to use for the decompression process.

Therefore, our proposed algorithm is to use a dynamic Huffman coding algorithm. It is an algorithm developed based on Huffman coding to create a dynamic binary tree. Instead of knowing the whole source data to compress like the Huffman coding, the dynamic Huffman coding compresses each symbol of the source data when it is being transmitted without having advanced knowledge of the source data. The compressed data are decompressed after receiving each symbol instead of receiving the entire data and calculating the total number of characters to decompress like the Huffman coding algorithm. Therefore, the dynamic Huffman coding can encode and decode the data in real time.

Figure 3 shows how to implement the dynamic Huffman coding algorithm. It includes five steps:

Step 1: The new input symbol is checked to determine whether it is the first symbol in the dynamic binary tree. If "Yes", a left side Not Yet Transferred (NYT) node and a right side symbol node are born from the root node or the last (parent) NYT node. The NYT node is called the child node, and its weight is "0" right after being born. The symbol node is unique, corresponding to each symbol, and its weight is initiated by "1" right after being born. If "No", the existing node of that symbol is tracked in the dynamic binary tree, then its weight is added up by "1".

Step 2: The compressed code is extracted by the codes of the paths from the root node to the parent NYT node and the binary codes of the input symbol.

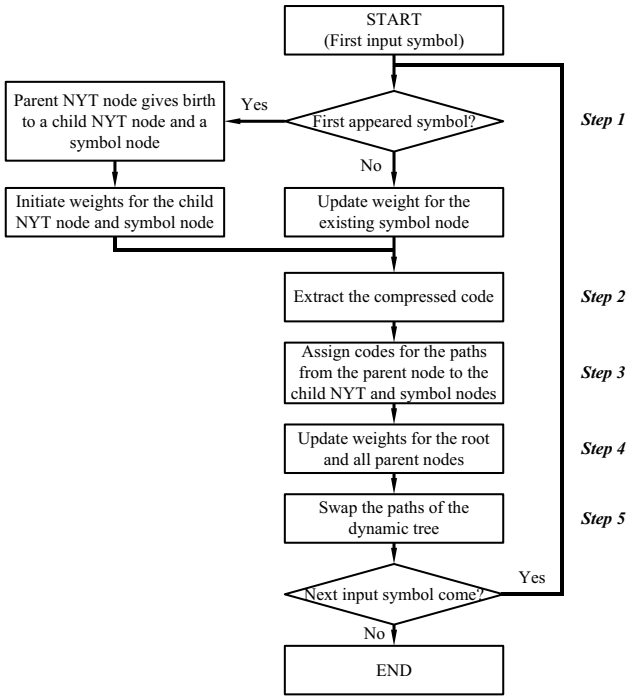

Fig. 3 Dynamic Huffman coding algorithm

Step 3: Assign codes "0" and "1" for the path from the parent node to the child NYT node and the symbol node, respectively.

Step 4: Update the weights of the root and parent nodes in the dynamic binary tree. The weight of the parent node is the sum of its child NYT node and symbol node. The weights are updated from the latest parent node to the root node.

Step 5: Swap the paths of the dynamic binary tree. The weights of the NYT node and symbol node of the same parent nodes are compared. The node with the higher weight is swapped to the right side along with its paths.

Figure 4 shows an example of the superiority of the Dynamic Huffman coding over the conventional Huffman coding. This example shows the progress in using Dynamic Huffman coding to compress the source data "ABBA". Each symbol is presented by 4 bits in hexa format. When the first symbol "A" is input, the compressed code of "A" is extracted by the hexa code of "A" (1010). Then, the symbol "B" is input, and the compressed code of "B" is extracted by the path code from its parent node to the root node (0) and its hexa code (1010). Then, another symbol, "B", is input; since "B" has been input before, the compressed code of "B" is extracted only by the path code from its parent node to the root node (0). At this time, the weight of the left child node is bigger than that of the right one, and then these nodes and their paths are swapped. Another symbol, "A", is input; since "A"

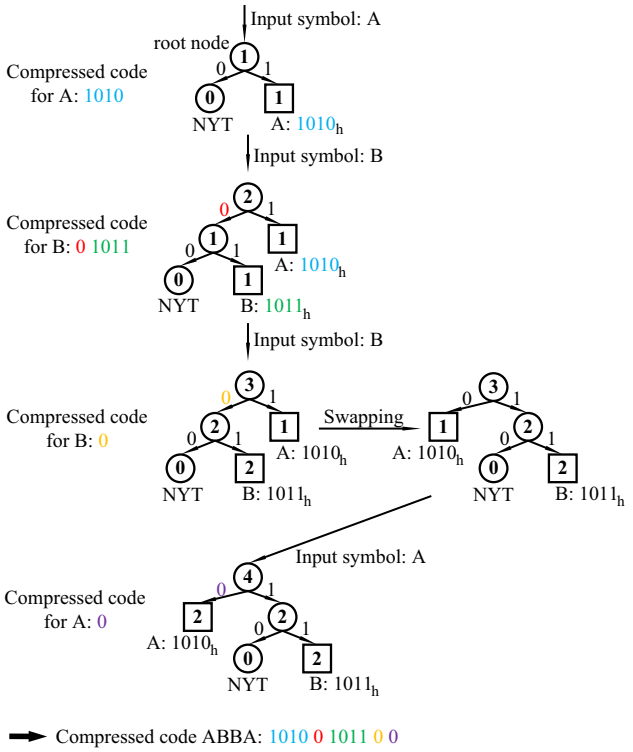

Fig. 4 Example of using Dynamic Huffman coding to compress the source data "ABBA"

has also been input before, the compressed code of "A" is extracted only by the path code from its parent node to the root node (0). Finally, the compressed codes for the source data "ABBA" are "1010 0 1011 0 0".

From the algorithm in Fig. 3 and the example in Fig. 4, the compressed code of each symbol of the source data is extracted immediately after each incoming symbol. Therefore, the advantages of dynamic Huffman coding, when compared to Huffman coding, are that, first, data can be compressed and decompressed in real-time while the data are being transmitted between the transmitter and the receiver. Second, if data is changed while compressing is processed, only the changed symbols are re-compressed instead of re-compressed entire source data like the Huffman coding. Third, it reduces the time needed to compress and reduces data storage memory since it does not need to browse the entire source data before compressing.

3.2 Pixel code to Hexa code

The idea is based on a dynamic binary tree to compress pixels, where color values are represented differently with 256 values. Therefore, a binary tree implementation would take 256 nodes to represent the pixel values in the tree, making the binary tree path costly for each input data. The solution to this problem is to convert the input data type to hex code

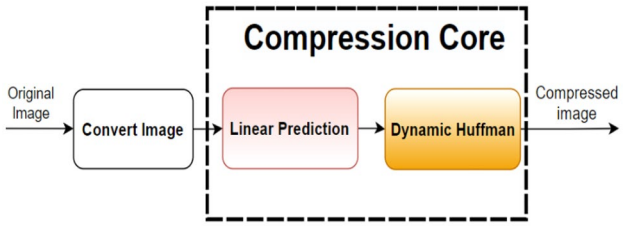

Fig. 5 Overview of image compression system

and then process each character instead of going through a whole pixel.

With this method, we only create a binary tree with a maximum of 16 nodes containing data (equivalent to 16 hex characters) and 16 NYT nodes representing the path of the binary tree at each node. In addition, the hardware implementation will reduce the area significantly. With the implementation, it only takes 32 registers to store for each node instead of the original 256 registers. Besides, the tree traversal will take a long time when the number of nodes in the tree is too large if the tree with 256 nodes is used. Therefore, the input processing minimizes the execution time of the compression system.

4 Proposed hardware architecture

4.1 System overview

The system overview is depicted in Fig. 5. The input image has different dimensions (256 x 256 or 512 x 512). We converted the original gray image to a data type stored in Hexadecimal code on Python programming language software, with each stored pixel represented by 8 bits. After that, the Compression Core block compresses the image into a binary bit string - this process is done on the hardware system. The results obtained from the compression process are decompressed and converted back to the original image by the software. To evaluating the proposed system, we analyze, calculate, and synthesize compression parameters and compare them with the original image.

4.2 Hardware architecture details

The Compression Core block has a detailed architecture, as shown in Fig. 6, with different sub-blocks to easily implement and optimize the design.

First, the Linear Prediction block performs a linear prediction process to process the input from 8 bits to 4 bits by calculating the error value based on the neighboring pixel values. The results of the Linear block will become the input for the Check Memory, Check Tree, and Tree Register File

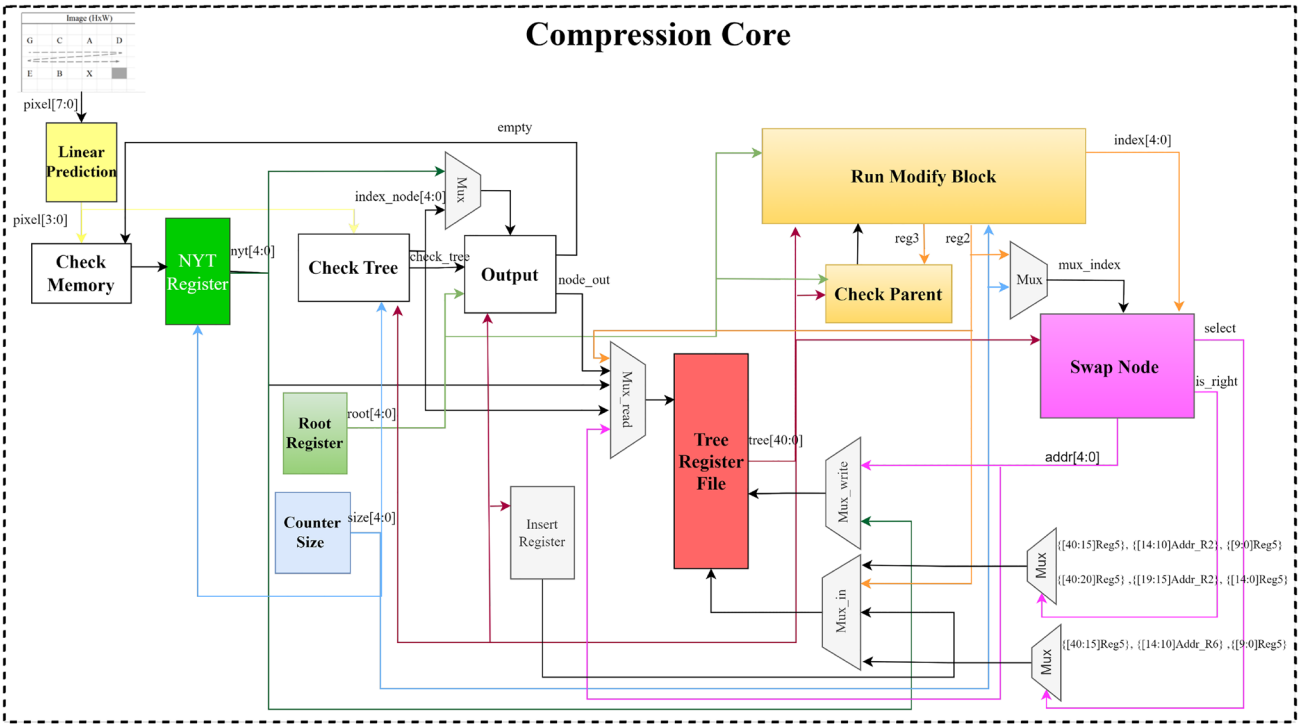

Fig. 6 Description of compression core hardware architecture

blocks. The Linear Prediction block helps reduce the number of input bits from 8 bits to 4 bits. The input of the linear block will range from 0 to F. Normally, storing pixels will take 256 registers to store the value for each node. However, with the linear block transformation, we will only need 32 registers to store nodes in the tree, including 16 value nodes and 16 NYT nodes. We see a big difference compared to 256 nodes to store pixel values.

The Check Memory block is responsible for checking whether the value under consideration is in the binary tree, with the input being the address value received from the output of the Linear Prediction block. In case it is already in the tree, this block is designed to use only 16 binary bits to store the conditions of 16 data nodes in the tree. The bit containing the value 1 means that the node has appeared in the binary tree. Otherwise, the node has never appeared in the tree. This design helps save maximum system area. The Check Tree block checks the position of that value in the tree by detecting each node in the tree (from the root node to the leaf node) that satisfies the condition that the node value in the tree matches the pixel value [3:0] taken from the Linear block, the results of the Check Tree block will be the condition to execute the Output Block in case the Check Memory block gives output equal to 1 (the node already exists in the tree). This block uses only 16 binary bits to store the conditions of 16 data nodes in the tree. The bit containing the value 1 means that the node has appeared in the binary tree. Otherwise, the node has never appeared in the tree. This design helps save maximum system area.

Then, the Output block is responsible for outputting the path of the node already in the tree (in case it already exists) or the path of the current NYT node (in case it doesn't exist). The results of the Output Block are a string binary, which includes the path of the node under consideration and the binary value of the pixel [3:0]. Designing the two Check tree and Output blocks to be independent of each other helps the design to execute the pipeline, reducing system execution time.

Next, Insert Register, considered a temporary register, is used to store node values in the tree temporarily after the node has complete data. This register is responsible for writing the value to the Tree Register File block if that value does not exist. The tree register file is designed specifically to use only 32 registers for storage. Each register includes value fields with specific meanings used to compare nodes in the tree with each other. In addition, tree register files can be read and written independently.

Finally, the Run Modify block is responsible for updating the binary tree. The Run Modify checks to see if the node in the tree satisfies the conditions of the binary tree (The farther from the root node, the smaller the weight must be. The left cannot be greater than the weight of the right node). Then, through the Swap node block, a swap of the node under consideration and the node in the tree is performed.

Table 2 Synthesis results of each block

Block	Clock (ns)	Frequency (MHz)
Linear Prediction	2.5 ns	400 MHz
Check Tree	2.5 ns	400 MHz
Output Block	2.5 ns	400 MHz
Run Modify Block	2.5 ns	400 MHz
Check Parent	3 ns	333 MHz
Tree	3 ns	333 MHz
Swap Node	3 ns	333 MHz
Compresstion Core	3 ns	125 MHz

Table 2 shows the synthesized results of each block based on the proposed architecture on board the Virtex-7 VC707 FPGA. The duty cycle is the minimum time for each block to perform the computation. Based on the duty cycle, each block will have different operating frequencies. The remaining blocks in the architecture are Register or Register File, so the operation parameters will be based on the Compression Core block. The Compression Core block has an operating cycle based on the largest operating cycle on each module. The operating cycle for the entire system is 8 ns, with an operating frequency of 400 MHz.

5 Experimental results

5.1 Evaluation methodology

The evaluation method of the proposed algorithm is shown in Fig. 7. The input image is converted to pixels using a text file in Python language. The hardware compression system is implemented using the hardware description language (Verilog). The hardware resources and the operating frequency of our compression system are obtained by the synthesis and simulation on the Vivado Embedded Development Kit (EDK) platform. After performing image compression on the hardware system, the results are obtained in binary code. We analyzed the results and compared this with the compression system on the software. If correct, these results are processed by the software decompression system. The image is rebuilt to the original image after performing the decompression. In this work, we use an image data set consisting of images with different sizes (256 x 256, 512 x 512), with the endings.tif, and.pgm provided by the author in [1, 2].

5.2 Results on the software implementation

The compression ratio (CR) measures the relative reduction in the size of the data representation produced by the compression algorithm. Compression ratio is defined as the ratio between the size before compression and the size after compression. Equation 4 shows how to calculate Compression Rate, CR is Compression Rate, $Size_{original}$ is the size of the images original, $Size_{encode}$ is the size of images after encode.

$$CR = \frac{Size_{original}}{Size_{encode}} \tag{4}$$

The Bits per pixel (BPP) is the total number of bits used to encode a pixel in the image. The smaller the number, the simpler the image, and the better the algorithm. Equation 5 shows how to calculate value bits per pixel, BPP is bits per pixel, TOB is the total of bits of images, and TOP is the total of pixels of images.

$$BPP = \frac{TOB}{TOP} \tag{5}$$

We use Visual Studio platform to simulate the results of the system. Figure 8 shows the testing images with the resolutions of 512x512 and 256x256.

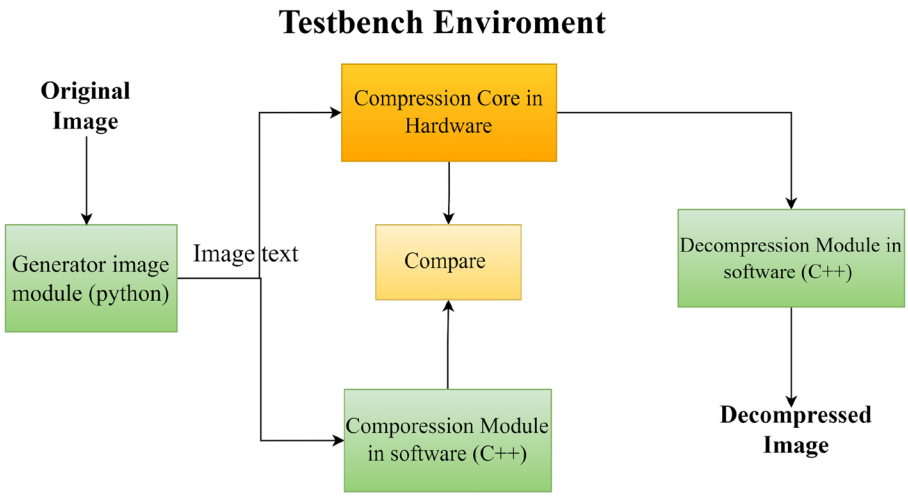

Fig. 7 Method evaluate results of proposed algorithm

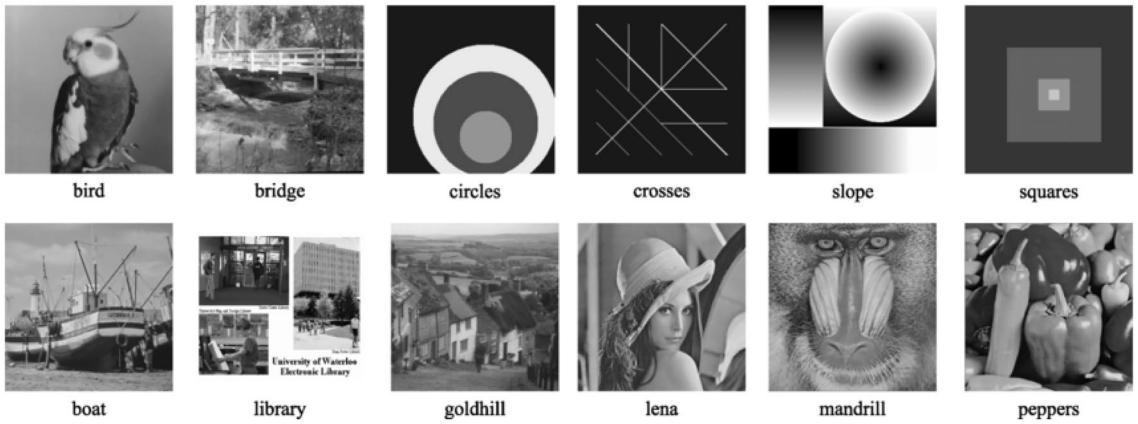

Fig. 8 12 different images from the Waterloo image compression benchmark

Table 3 Results on software implementation with dataset Waterloo

Images	Resolution	CR	BPP
Bird	256 × 256	2.55	3.14
Bridge	256 × 256	1.74	4.6
Circles	256 × 256	7.28	1.1
Crosses	256 × 256	6.46	1.24
Slope	256 × 256	3.72	2.15
Squares	256 × 256	7.9	1.01
Boat	512 × 512	2.14	3.74
Library	464 × 352	1.73	4.64
Goldhill2	512 × 512	2.07	3.87
Lena2	512 × 512	2.18	3.68
Mandrill	512 × 512	1.67	4.80
Peppers	512 × 512	2.16	3.72
Average		3.467	3.14

Table 3 describes the results of the simulation of the compression system on the software. The parameters are calculated based on the compression ratio and BPP of the dataset Waterloo [10] with the image extension.tif.

With this dataset, the proposed method achieves an average compression ratio of 3.467 and an average BPP of 3.14. The results show that the algorithm is highly effective for images with large sizes and high complexity.

Figure 9 depicts a dataset of type Medical Images taken from Kaggle [1], a dataset with images with a resolution of 512 x512 in PNG images. The evaluation methodology is shown in Table 4. For medical images that are highly complex and require high accuracy, the algorithm achieves quite high efficiency. The average compression ratio is 2.064, and bits per pixel is 4.0444.

Through simulation results from two datasets, we found that the compression ratio is significantly improved for the Medical image set. The bits per pixel ratio of the Medical image set is better than that of the Waterloo image dataset. In addition, resolution greatly affects the compression ratio of the image.

5.3 Results on the hardware implementation

We compared CR, BPP, and run-time results between the technique using the Linear prediction combined with the full pixel input dynamic Huffman (full pixel) and the technique using the Linear prediction combined with the half pixel input dynamic Huffman (half pixel). Comparison results are shown in Table 5, with dataset in Fig. 10 with size images 512 × 384.

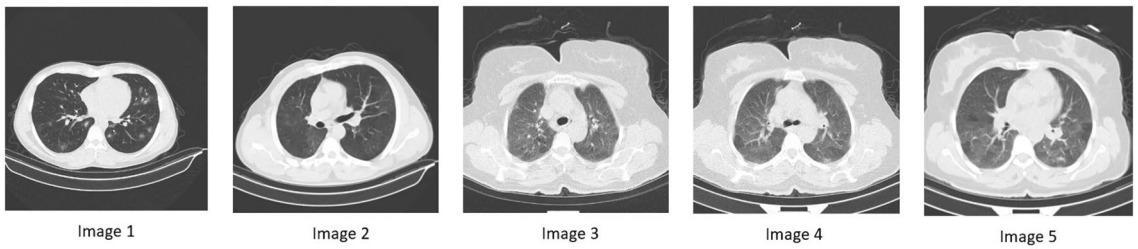

Fig. 9 Five images from dataset media CT Covid

Table 4 Results on the software implementation with medical images

Images	Type	Resolution	CR	BPP
Image 1	png	512x384	2.63	3.04
Image 2	png	512x384	2.57	3.11
Image 3	png	512x384	1.69	4.71
Image 4	png	512x384	1.70	4.73
Image 5	png	512x384	1.73	4.63
Average			2.064	4.0444

The results show that the half-pixel technique is better than the conventional full-pixel technique. The average compression ratio parameter of half a pixel is 1.046, which is better than that of the full pixel is 1.03. The BPP value is also better, with half pixel 7.69 compared to 7.9 for full pixel. Especially with pixel division, the number of different inputs will be less, leading to a shorter average running time. The average running time for half pixels is 13.8 s and 44.1 s for full pixels.

5.4 Compare results with other techniques

Table 6 describes the results of comparing the BPP of this proposed method with the algorithm proposed [10] and other algorithms such as Huffman, Arithmetic Coding, JPEG 2000, JPEG-LS, CALIC [11], HEVC RExt, 7-Zip and the Integer Wavelet Transform (IWT) - Huffman coding with data set from the Waterloo images in Fig. 8.

As shown in Table 6, the BBP results of the proposed technique are better than those of other techniques in images with many textures, such as Bridge, and Mandrill. With image Bridge, the BPP of the proposed algorithm is 4.6, and the BPP average of another algorithm is 6.3. With image Mandrill, the BPP of the proposed algorithm is 4.8, and the BPP average of another algorithm is 6.35. Additionally, the 7-Zip algorithm works well on simple images with less complex textures.

Table 7 describes the hardware implementation results of the proposed algorithm with related articles. The results show that the proposed algorithm runs with a cycle time shorter than other works, 0.5 times faster than the conventional Huffman Coding algorithm, and 0.25 times faster than the algorithm used in [5, 9]. In addition, the number of combinational circuit blocks (LUTs) used by the algorithm proposed is lower than the Huffman Coding [8] and LZW [6] algorithms. With the results shown in Table 7, the LZW algorithm [6] has the smallest and best operating cycle compared to the algorithm proposed by the group and other articles. Regarding other parameters, some works do not mention them, so we could not compare each parameter of the article in detail with the proposed system.

Table 8 describes the hardware implementation results of the the proposed architecture with those in other works using the same Vivado design platform. The hardware implementation results show that the compression ratio in the proposed system has a better operating cycle than that in the article [3], the cycle of the proposed method is 8 ns, and the cycle of the method in [3] is 9.081 ns. In addition, the hardware usage parameters in the proposed method are more optimal than those of the Modified color transformation (MCT/SCF) method in terms of the number of Flip Flop used. We use 160 LUTs, 400 Register, and 1 Block Ram in the proposed design.

Table 5 Compare between the half pixel input technique and full pixel input technique

		CR		BPP		Run time		Type
Images	Size	Half pixel	Full pixel	Half pixel	Full pixel	Full pixel	Half pixel	
Image 1	512 × 384	1.04	1.02	7.76	7.89	13 s	40 s	bmp
Image 2	512 × 384	1.11	1.08	7.21	7.92	15 s	45 s	bmp
Image 3	512 × 384	1.04	1.04	7.74	7.90	14 s	47 s	bmp
Image 4	512 × 384	1.05	1.04	7.67	7.95	13 s	40 s	bmp
Image 5	512 × 384	1.03	1.02	7.87	7.79	14 s	45 s	bmp
Image 6	512 × 384	1.01	1.01	7.91	7.97	14 s	44 s	bmp
Average		1.046	1.03	7.69	7.9	13.8	44.1	

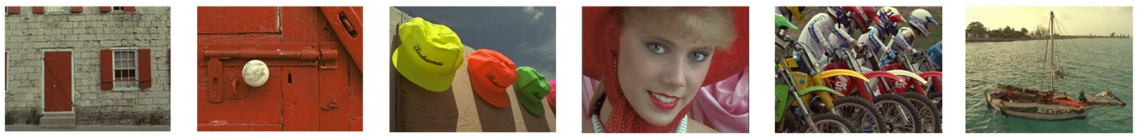

Fig. 10 Dataset bmp images from Kaggle

Table 6 Compression results in bpp of the Waterloo image set

Images	Size	Huff.	Arith. Coding	JPEG 2000	JPEG -LS	CALIC	HEVC Rext	7-Zip	IWT +Huff.[10]	Prop.
Bird	256 × 256	6.80	6.77	3.14	3.47	3.32	3.42	4.23	2.86	3.14
Bridge	256 × 256	7.69	7.67	5.91	5.79	5.54	5.71	6.32	4.49	4.60
Circles	256 × 256	1.85	1.78	1.26	0.15	0.14	0.15	0.11	1.34	1.1
Crosses	256 × 256	1.00	0.19	1.43	0.39	0.37	0.38	0.18	2.03	1.24
Slope	256 × 256	7.54	7.52	1.06	1.57	1.50	1.55	1.70	1.67	2.15
Squares	256 × 256	1.35	1.08	0.25	0.08	0.08	0.08	0.05	0.70	1.01
Boat	512 × 512	7.15	7.12	4.10	4.25	4.07	4.19	5.29	3.52	3.74
Library	464 × 352	5.87	5.85	5.83	5.10	4.88	5.03	4.25	5.17	4.64
Goldchill	512 × 512	7.50	7.48	4.65	4.71	4.51	4.64	5.60	3.80	3.87
Lena2	512 × 512	7.49	7.45	4.02	4.24	4.06	4.18	5.52	3.25	3.68
Mandrill	512 × 512	7.38	7.36	6.02	6.04	5.78	5.95	6.39	5.04	4.80
Peppers	512 × 512	7.60	7.57	4.40	4.49	4.30	4.43	5.55	3.51	3.72
Average		5.7662	5.65	3.51	3.36	3.22	3.31	3.77	3.15	3.14

Table 7 Results compare hardware parameters with other algorithms

Algorithm	Cycle	Frequency	Luts	Register	Block ram	Board FPGA
Static Huffman Coding [8]	12.5 ns	80 MHz	854	187	NA	Altera Flex10K20RC240
AHDB + PDLZW [9]	10 ns	100 MHz	NA	NA	NA	NA
LZW [6]	3.5 ns	280.17 MHz	307	278	13	Virtex 7 VC707
PNG (LZ77 + Huffman) [5]	10 ns	100 MHz	NA	NA	NA	NA
LZ77 [4]	10.2 ns	109 MHZ	NA	NA	NA	Virtex 5
Proposed	8 ns	125 MHz	160	400	1	Virtex 7 VC707

Table 8 Results compare hardware parameter simulate on vivado tools

Algorithm	Cycle	Frequency	Luts	Register	Block ram	Board FPGA
MCT [3]	9.081 ns	164.447 MHz	976	1046	None	Sparten3e
SC [3]	7.237 ns	128.179 MHz	208	416	None	Sparten3e
Proposed	8 ns	125 MHz	160	400	1	Virtex 7 VC707

6 Conclusion

In this work, we present the real-time image compression techniques based on Dynamic Huffman Coding addition Prediction Linear to improve compression rate and values Bits Per Pixel. The average compression ratio was 2.5, and the average bits per pixel achieved 3.467. The proposed hardware design uses the linear prediction input processing technique and pixel halving to optimize hardware capacity. The operating frequency of the proposed hardware implemented on board Virtex-7 VC707 FPGA is 125 MHz. Hardware use resources include 160 LUTs, 400 Registers, and 1 Block RAM.

Acknowledgements This research was supported by The VNUHCM-University of Information Technology's Scientific Research Support Fund.

Author Contributions Duc Khai Lam defined the methodology and designed, wrote, reviewed, edited, and revised the paper.

Data Availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. Hayden gunraj, dataset covid x ct- a large-scale chest ct dataset for covid-19 detection. https://www.kaggle.com/datasets/hgunraj/covidxct
2. Srachejack , dataset tid2013. https://www.kaggle.com/datasets/srachejack/tid2013

3. Al-Shebani, Q., Premaratne, P., Vial, P.J., McAndrew, D.J., Halloran, B.: Co-simulation method for hardware/software evaluation using xilinx system generator: a case study on image compression algorithms for capsule endoscopy. In: 2018 12th International Conference on Signal Processing and Communication Systems (ICSPCS), pp. 1–4. IEEE (2018)
4. Gao, Y., Ye, H., Wang, J., Lai, J.: Fpga bitstream compression and decompression based on lz77 algorithm and bmc technique. In: 2015 IEEE 11th International Conference on ASIC (ASICON), pp. 1–4. IEEE (2015)
5. Huang, S., Zheng, T.: Hardware design for accelerating png decode. In: 2008 IEEE International Conference on Electron Devices and Solid-State Circuits, pp. 1–4. IEEE (2008)
6. Kagawa, H., Ito, Y., Nakano, K.: Throughput-optimal hardware implementation of lzw decompression on the fpga. In: 2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW), pp. 78–83. IEEE (2019)
7. Katti, R.S., Ghosh, A.: Security using shannon-fano-elias codes. In: 2009 IEEE International Symposium on Circuits and Systems, pp. 2689–2692 (2009). https://doi.org/10.1109/ISCAS.2009.5118356
8. Lee, T., Park, J.: Design and implementation of static huffman encoding hardware using a parallel shifting algorithm. IEEE Trans. Nucl. Sci. **51**(5), 2073–2080 (2004)
9. Lin, M.B., Lee, J.F., Jan, G.E.: A lossless data compression and decompression algorithm and its hardware architecture. IEEE Trans. VLSI Syst. **14**(9), 925–936 (2006)
10. Liu, X., An, P., Chen, Y., Huang, X.: An improved lossless image compression algorithm based on huffman coding. Multimed. Tools Appl. **81**(4), 4781–4795 (2022)
11. Lone, M.R.: A high speed and memory efficient algorithm for perceptually-lossless volumetric medical image compression. Journal of King Saud University - Computer and Information Sciences **34**(6, Part A), 2964–2974 (2022). https://doi.org/10.1016/j.jksuci.2020.04.014.https://www.sciencedirect.com/science/article/pii/S1319157820303499
12. Matsuda, I., Shirai, N., Itoh, S.: Lossless coding using predictors and arithmetic code optimized for each image. In: Visual Content Processing and Representation: 8th International Workshop, VLBV 2003, Madrid, Spain, September 18-19, 2003. Proceedings 8, pp. 199–207. Springer (2003)
13. Satone, K., Deshmukh, A., Ulhe, P.: A review of image compression techniques. In: 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA), vol. 1, pp. 97–101 (2017). https://doi.org/10.1109/ICECA.2017.8203651
14. Sharma, M., et al.: Compression using huffman coding. IJCSNS Int. J. Comput. Sci. Netw. Secur. **10**(5), 133–141 (2010)
15. Sun, M.Y., Xie, Y.H., Tang, X.A., Sun, M.Y.: Image compression based on classification row by row and lzw encoding. In: 2008 Congress on Image and Signal Processing, vol. 1, pp. 617–621 (2008). https://doi.org/10.1109/CISP.2008.302