JOURNAL OF SOFT COMPUTING AND DATA MINING
e-ISSN: 2716-621X

JSCDM

Vol. 6 No. 1 (2025) 138-149
https://publisher.uthm.edu.my/ojs/index.php/jscdm

Efficient Implementation of Searchless Fractal Image Compression on Low-cost FPGA for Real-Time Encoding

Abdul-Malik H. Y. Saad[1]*, Zhia Hao Chai[2], Nayef A. M. Alduais[3], Mohammed Sultan Mohammed[4], Antar S. H. Abdul-Qawy[5], Abdullah B. Nasser[6], Waheed Ali H. M. Ghanem[7], Hisham Haider Yusef Sa'ad[8]

[1] *College of Engineering, University of Buraimi, Al Buraimi, 512, OMAN*
[2] *Division of Electronic and Computer Engineering, Faculty of Electrical Engineering, Universiti Teknologi Malaysia (UTM), 81310 JB, Johor, MALAYSIA*
[3] *Faculty of Computer Science and Information Technology (FSKTM), Universiti Tun Hussein Onn Malaysia (UTHM), 86400, Parit Raja, Batu Pahat, Johor, MALAYSIA*
[4] *Department of Electrical and Computer Engineering, Curtin University Malaysia, Miri 98009, Sarawak, MALAYSIA*
[5] *Department of Mathematics and Computer Science, Faculty of Science, Abdulrahman Al-Sumait University, Zanzibar, TANZANIA*
[6] *School of Technology and Innovation, University of Vaasa, 65200 Vaasa, FINLAND*
[7] *Faculty of Computer Science and Mathematics, Universiti Malaysia Terengganu, 21030 Kuala Terengganu, MALAYSIA*
[8] *Department of Artificial Intelligence, Al-Razi University, Sana'a, YEMEN*

*Corresponding Author: abdulmalik.h@uob.edu.om; eng.abdulmalik@gmail.com
DOI: https://doi.org/10.30880/jscdm.2025.06.01.010

Article Info

Received: 8 December 2024
Accepted: 17 April 2025
Available online: 30 June 2025

Keywords

Fractal image compression, quadtree partitioning, searchless scheme, real-time image compression, FPGA

Abstract

Self-similarity within images is utilized by Fractal Image Compression (FIC) method to provide potential benefit of high compression ratio, good recovery performance, flexible recovery resolution, and fast decoding process. Nevertheless, FIC has a significant drawback—its encoding process is highly time-consuming, particularly when a full-search approach is employed. To address this issue, searchless-based approaches have been developed and implemented in hardware to enable real-time encoding. Despite their advantages, existing hardware designs often require significant hardware resources, making them unsuitable for low-cost FPGA platforms. In this paper, a real-time and hardware-effecient architecture is proposed for encoding images with searchless-based FIC method, optimized for low resource utilization. The proposed design encoded the image blocks in different sizes based on quadtree approach. The design was synthesized and implemented on a low-cost FPGA, with circuit-level optimizations using parallelism and pipelining to enable real-time image encoding. Its performance was quantitatively assessed based on encoding time, hardware resource utilization, peak signal-to-noise ratio (PSNR), and compression ratio. With an operating frequency of 50 MHz, a run time of 9.65 ms can be achieved for an 512×512×8 image, which is equivalent to 103 images which can be encoded per second. The design consumes less than half of the total device resource in the low-cost Cyclone V SoC FPGA. The average PSNR and compression ratio achieved are 32 dB and 14:1 respectively.

1. Introduction

Nowadays, with the explosive growth of digital technology and data, data compression has become more and more important. Without data compression, the raw data is extremely huge, and it poses significant challenges for storing, archiving, and transferring. Data compression helps to minimize the storage requirement and transmission bandwidth utilization, and ultimately reduces the storage and transmission cost. In this context, data compression is widely applied in numerous types of applications, ranging from medical, security and surveillance, multimedia broadcasting, military, computer vision and many more.

Over the years, numerous compression techniques have been developed. Each of the compression techniques has its own strengths and weaknesses. The choice of which technique to apply depends on the desired application and requirements of the end user. In image compression, FIC algorithm, originally introduced by Barnsley et al. [1], is a distinctive compression method that exploits the self-similarity inherent in many natural images. FIC transforms an image into a set of fractals that can represent the original image and requires less storage space to be stored. By exploring the self-similarity within the image, FIC can achieve a high compression ratio with superior recovery performance, especially for images with repeating patterns and complex textures [2]. In this context, FIC is highly efficient for compressing aerial photography and satellite imagery [3]. Besides, due to the popularity of FIC in digital archiving, FIC has also found numerous applications in image processing such as image restoration [4], medical image classification [5], image watermarking [6], face recognition [7], and character recognition [8]. In addition, FIC offers multi-resolution features where the image can be retrieved or reconstructed in any resolution, thus providing zoom-in and out features.

In spite of the attractive features, FIC suffers from one major drawback that is the high computational complexity of the fractal encoding algorithm. In order to find the fractals representing the original image, the conventional full-search-based fractal encoding process requires intensive searching and computation [9, 10]. Consequently, it causes unacceptably long encoding time and ultimately poses significant challenges for the real-time implementation of FIC. To overcome the drawback, many researchers have developed and proposed various optimization strategies, such as parallel processing, classification of image blocks, partitioning of image blocks, etc. [11-14]. For instance, Hernandez-Lopez F.J. and Muñiz-Pérez O. [15] implemented Quadtree-based fractal image compression in parallel using multi-core CPU and GPU. They have achieved 15x and 25x speed-up respectively compared the serial implementation. However, achieving an encoding time below 1 second for a 512×512 image remained a challenge. Using the same approach, Asati R., Raghuwanshi M.M. and Singh, K.R. [16] introduced a multithreading-based parallelism technique implementation on a multi-core processor, reducing compression time by approximately 2.5x compared to sequential implementation. Meanwhile Abood B.S., Akkar H.A., and Al-Safi A.S. [17] used different approach, i.e. starting the search for the best matching domain block from the closest points to the range blocks and expands until finds an acceptable match. For 512x512 grayscale Lena image, they can reduce the search time by 87% and achieved an encoding time of 0.36 second. Overall, most of the optimization efforts are focusing on reducing the brute searching process or perform parallel processing using Multi-core CPU or GPU, thereby reducing the computational burden and time. However, the reduction achieved in the encoding time is still not sufficient for real-time image compression applications.

One of the useful optimization efforts is the hardware implementation of searchless-based fractal image compression, proposed by Jackson et al. [18]. The searchless-based design proposed by Jackson et al. can overcome the issues of traditional full-search-based FIC algorithm while still maintaining the quality of reconstructed image and compression ratio as good as or even better than the traditional full-search-based FIC. However, Jackson et al.'s design has high hardware resource usage and requires preprocessing, making it unsuitable for low-cost FPGAs. However, it has the potential to be improved with the algorithm optimization and additional hardware pipeline.

In Summary, the searchless-based FIC method presents a feasible solution by eliminating the exhaustive search phase, but its hardware complexity remains a barrier to widespread implementation on cost-effective platforms. This paper addresses these challenges by proposing a hardware-efficient architecture tailored for real-time operation on low-cost FPGAs, bridging the gap between high-performance compression and practical hardware deployment. Thus, the searchless-based algorithm proposed by Jackson et al. is studied and optimized. Thereafter, the corresponding hardware architecture is designed and implemented.

2. Fractal Image Compression (FIC)

2.1 Full-Search FIC

FIC is a lossy image compression technique that makes use of the fractal properties or self-similarity of an image. In this context, fractal refers to a geometric pattern that can be split into smaller parts, each of which is similar to the whole [19]. The fractal nature of the image implies that small portions of the image can resemble bigger portions of the image.

The idea of FIC defines how an image can be represented by its fractals using the Iterated Function System (IFS). IFS refer to a set of affine transformations that satisfy the Contractive-Mapping-Fixed-Point Theorem. Later, Jacquin [20] developed the first practical FIC algorithm based on the concept of Partitioned-Iterated-Function-System (PIFS), thereby making FIC feasible for digital image compression.

In general, fractal image compression involves the process of encoding an image into fractal coefficients and decoding the fractal coefficients at some other time to reconstruct back the original image. Fractal encoding aims to find a set of transformations while fractal decoding will then iteratively apply the transformations on an initial arbitrary image to reconstruct the original image. According on the Collage Theorem and Fixed-Point Theorem [1], repeatedly applying the defined transformations to any initial image will eventually lead to convergence at a fixed point (or attractor), which serves as an approximation of the original image.

Based on the contemporary Partitioned Iterated Function System (PIFS) [20], fractal encoding process can be specified into three stages, namely image partitioning, sub-images matching, and fractal parameter storing.

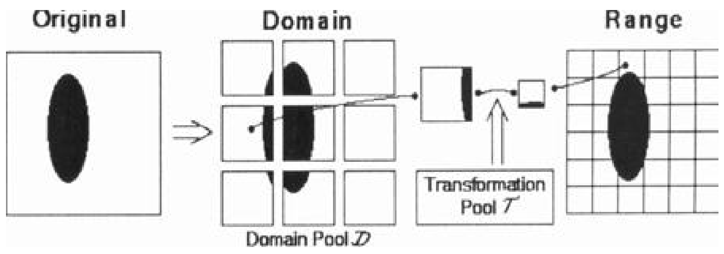

Fig. 1 *Fractal encoding process [21]*

Figure 1 depicts the fractal encoding process as described in [21]. In this approach, the image is partitioned into two distinct sets of blocks: range blocks (R), which are smaller and non-overlapping, and domain blocks (D), which are larger and may overlap. Typically, each domain block is twice the size of a range block in both width and height (i.e., D=2Rx2R) [22]. To match a range block (R) with a domain block (D), the domain block must first be contracted to match the size of the range block. The algorithm then applies various transformations to each domain block in the domain pool, searching for the best match to the range block based on an optimal transformation. The matching distortion or matching error can be determined using various error metrics such as Mean-Squared-Error (MSE) and Sum-of-Absolute-Differences (SAD. Once the suitable matching is found where the matching error is minimum, the corresponding transform parameters and best-match domain location or coordinates are encoded and stored as Fractal Code (FC).

In the domain-range matching process, the transformation applied to the domain blocks can be specified into isometric transformation and scaling transformation. In general, for hardware simplicity, only scaling transformation would be considered, where the transformation can be expressed in general as follows:

$$T(D) = s \cdot D + g \tag{1}$$

where T denotes for transformation, s is the luminance-scale parameter, D is the contracted domain block, and g is the luminance offset parameter. The scale factor s should be constrained to lie in the range between -1 and 1 to ensure that the transformation is contractive to comply with the Fixed-Point Theorem [1]. After that, the matching between the transformed domain blocks and the corresponding range blocks is performed. In general, the matching error can be determined and computed using MSE where the computation can be expressed as follows:

$$MSE(R,D) = \frac{1}{N}\sum_{i=1}^{N}\left(R_i - T(D_i)\right)^2 = \frac{1}{N}\sum_{i=1}^{N}(R_i - sD_i - g)^2 \tag{2}$$

where N represents the total number of pixels in the image block; Ri and Di are the ith pixels values for the range block and the contracted domain block, respectively. With differentiation, the optimum s and g values that give the minimum MSE can be obtained using the following equations [18, 23]:

$$s = \frac{N\sum_{i=1}^{N} RiDi - \sum_{i=1}^{N} Ri \sum_{i=1}^{N} Di}{N\sum_{i=1}^{N} Di^2 - \sum_{i=1}^{N} Di \sum_{i=1}^{N} Di} \tag{3}$$

$$g = r_{avg} - s \cdot d_{avg} \tag{4}$$

where r_{avg} and d_{avg} are the mean pixel value of the range and domain block, respectively.

For the previous equations, Tong & Pi [24] argued and proved that the correlation between s and g is large, thereby storing the parameters is not efficient. In addition, obtaining the optimal values of s based on equations (3) poses challenge to the hardware implementation due to its computational complexity. To overcome it, the luminance offset parameter, g, can be replaced by the range mean value, r_{avg}, by substituting equation (4) into (1):

$$T(D) = s \cdot (D - d_{avg}) + r_{avg} \tag{5}$$

Therefore, the error metric, MSE, become:

$$MSE(R, D) = \frac{1}{N}\sum_{i=1}^{N}(R_i - T(D_i))^2 = \frac{1}{N}\sum_{i=1}^{N}\left((R_i - r_{avg}) - s \cdot (D_i - d_{avg})\right)^2 \tag{6}$$

This scheme reduces the correlation between s and g, and also reduces the computational complexity. In the practical application, a set of quantized $\{sk\}$ can be applied for the simplicity of storing and computing.

On the other hand, the fractal decoding process operates iteratively, beginning with an arbitrary initial image and updating it with the fractal functions until the convergence is attained where the reconstructed image closely resembles the original image. For each range block, the reconstruction is performed as follows:

$$R' = s \cdot (D - d_{avg}) + r_{avg} \tag{7}$$

where R' represents the reconstructed range block, D is the best-match domain block, s and g are the luminance scaling and offset values encoded and stored respectively. The newly reconstructed range image is then utilized as the domain image for the next iteration, and this process continuing until convergence is achieved.

2.2 Searchless FIC

Jackson et al. [18] found that the domain blocks form from a spatially local neighborhood area of range blocks tends to have a high match probability. Thus, the authors proposed and presented a full-quadtree searchless FIC algorithm, partially derived from another author, Furao & Hasegawa [25]. In the algorithm proposed by Jackson et al., the domain-range match is performed between a range block and a single domain block constructed from the neighboring area of that range block. Besides, quadtree partitioning, as illustrated in Fig. 2, is employed during the domain-range matching process. If a suitable match is not found, the algorithm recursively divides the range block into four smaller blocks for further evaluation.

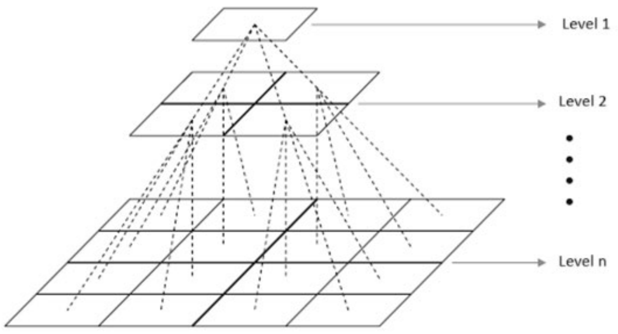

Fig. 2 *Quadtree partitioning structure*

The pseudocode for the searchless-based encoding algorithm is shown in Fig. 3. In the encoding process, the entire image is first partitioned into N×N non-overlapping range blocks. Then for each range block, the matching between the range block and neighboring contracted domain is performed. If the matching error is less than or equal to a user-defined threshold, the corresponding fractal parameters are stored for that range block, and the process continues to the next block. However, if the matching error exceeds the threshold, the range block is divided into four smaller blocks using quadtree partitioning, and the matching operation is repeated for each of these sub-blocks. This recursive process continues down to lower levels of the quadtree until an acceptable match is found or the end of the quadtree is reached.

In the algorithm, no domain location parameter is required to be stored as a part of the fractal code because there is no domain search. Therefore, the quadtree level can extend efficiently from 32x32 to 2x2 blocks, and even 1x1 blocks when necessary [18]. Apart from that, in the error matching process, the error metric described in equation (6) is applied due to the hardware simplicity. The algorithm determines the error value within a domain-range match for sk = [0, 0.25, 0.50, 0.75] and selects the sk value corresponding to the minimum error if the match is found where the calculated error is less than threshold. Besides, in the error matching process, an adaptive tolerance value is adopted based on the quadtree level. The user-specified tolerance value is set for the lowest-level, and the tolerance value decays by a factor for the higher subsequent levels in the quadtree [18].

```
Initialize quadtree level and list of range blocks covering
the image.
For each range block $R_i$:
  Call FindDomainMatched($R_i$)

FindDomainMatched ($R_i$) {
  If ($R_i$ is 1x1):
    Store MSB of $R_i$ as codebook
    Return

  Find matching domain block
  Perform domain-range matching

  If (error > threshold):
    Subdivide $R_i$ into 4 smaller blocks $R_{i0}$...$R_{i3}$
    For each $R_{ij}$:
      Call FindDomainMatched ($R_{ij}$)
  Else:
    Store $R_i$ with fractal parameters in codebook
  Return
}
```

Fig. 3 *Searchless encoding algorithm pseudocode*

2.3 Algorithm Optimization

The Full Quadtree Searchless Algorithm proposed by Jackson et al. [18] posed a drawback where the entire image is required to be buffered and preprocessed before the subsequent compression due to the sophisticated domain block construction. This is not beneficial for the hardware implementation. A minor modification can be done in the domain-range match process where the spatial locations of domain and range block can be redefined as shown in Fig. 4. The domain block is contracted to be the same size as the range block using a 2×2 averaging mask, which averages and shrinks 4 non-overlapping pixels into 1 pixel. The four range blocks, R1, R2, R3, R4, are compared to the same domain block, D. The domain construction from the local neighboring area of range blocks based on Fig. 4 could also have a high domain-range match probability.

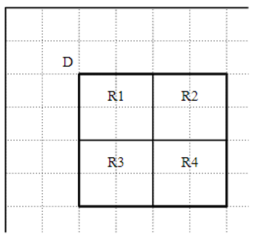

Fig. 4 *The redefined locations of domain and range block*

The modification is beneficial for the hardware implementation where the algorithm only needs to buffer a small portion of image at a time for processing instead of buffering the entire image at once. Consequently, it avoids the preprocessing operation as in the work proposed by Jackson et al. [18]. Besides, the proposed algorithm also avoids the additional requirement and operation for the treatment of the boundary blocks (i.e., boundary padding) due to the redefined domain location. Furthermore, in the modified algorithm, the parallel processing on domain-range matching and quadtree partitioning remained feasible for the hardware architecture as in the work proposed by Jackson et al.

With the modification, the general encoding process can be illustrated in Fig. 5. Basically, 4 image range blocks are consecutively read in and encoded into the fractal code. In the encoding process, the domain block is constructed based on Fig. 4. Then, domain-range matching is performed for each of the 4 range blocks. Quadtree partitioning is performed, if necessary, until the suitable match is found or the end of the quadtree is reached. The corresponding fractal code for the 4 range blocks is updated and stored. The process is repeated until the last image block is encoded.

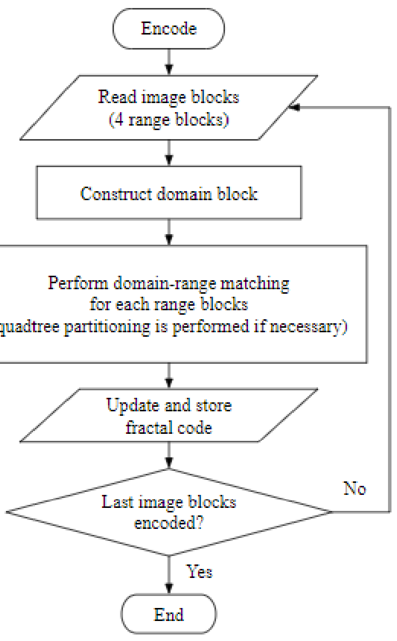

Fig. 5 *General encoding process after modification*

2.4 Image Compression Performance Metrics

In general, decompression comes after compression to reconstruct the original data. In image compression, especially for lossy compression, PSNR is the common metric that measures the difference between the original image and reconstructed image. PSNR can be computed as follows:

$$PSNR\ (dB) = 10\ log_{10} \frac{255}{\frac{1}{MN}\Sigma(f - f')} \tag{8}$$

where f is the original image, f' is the reconstructed images, M and N is the image size. The higher the PSNR value, the better the quality of the reconstructed image. Besides, Compression Ratio (CR) is also one of the important metrics that measures the compression factor and efficiency. Compression Ratio can be generally expressed as follows:

$$CR = \frac{size\ of\ original\ image}{size\ of\ compressed\ image} \tag{9}$$

3. Hardware Implementation

3.1 Overview

A hardware design which is capable of implementing the encoding process illustrated in Fig. 5 is successfully implemented. The hardware architecture exploits parallelism as much as possible to minimize the iterative computation of the algorithm. Operation chain and pipelining architecture are deployed in the design to achieve higher operating frequency and minimum resource usage.

3.2 Macro Architecture

The macro architecture of the hardware design is illustrated in Fig. 6. The architecture consists of Domain and Range Register, Domain and Range Mean Computation, Domain-Range Error Computation, Domain-Range Error Matching. The overall process includes buffering range blocks, constructing domain blocks, computing means, computing match error, and finally matching and outputting fractal codes.

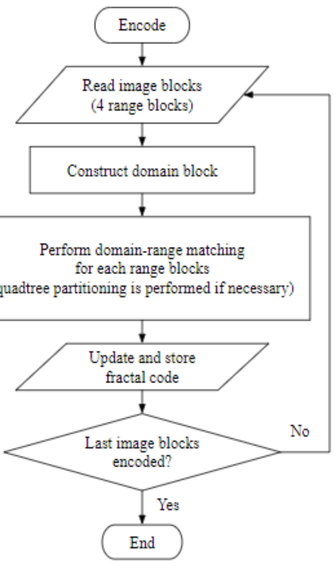
Encode
Read image blocks (4 range blocks)
Construct domain block
Perform domain-range matching for each range blocks (quadtree partitioning is performed if necessary)
Update and store fractal code
Last image blocks encoded?
No
Yes
End

Fig. 6 *Macro architecture*

For design input, four range blocks are buffered to be processed to fractal codes. The input stream width is 4 bytes or 32 bits in which 4 pixels are input and buffered at a clock cycle. To simplify and facilitate the domain and range mean computation, the image pixel data is accessed in a zig-zag manner instead of a row-major manner as illustrated in Fig. 7. On the other hand, the output stream consists of 10-bits fractal code and a valid indicator. Overall, the macro architecture can be divided into 3 modules, which are described in the following sections.

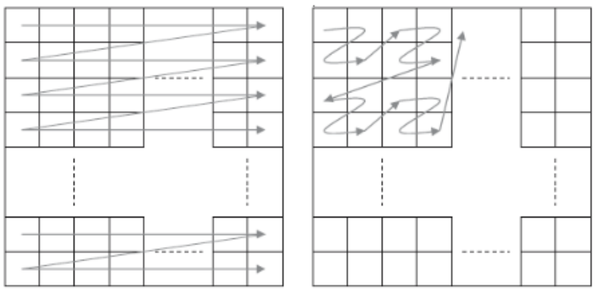

Fig. 7 *Row-major access vs Zig-zag access [18]*

3.3 Module 1: Range Buffering & Domain Construction & Mean Computation

Module 1 includes the process of buffering range blocks, constructing domain blocks, and computing domain and range means. The data path of module 1, shown in Fig. 8, consists of range register, domain register, mean registers, accumulators, and combinational logic. For each clock cycle, range buffering and domain construction are performed simultaneously. During the process, the domain and range mean computation is pipelined and carried out simultaneously using the accumulating architecture and effective control logic. Overall, Module 1 requires 67 clock cycles in which the range buffering and domain construction require 64 clock cycles and the pipelined mean computation requires extra 3 clock cycles. Upon finishing the process, the range data, domain data, and the domain and range means for all quadtree levels are ready to be used for the match error computation in the next Module 2.

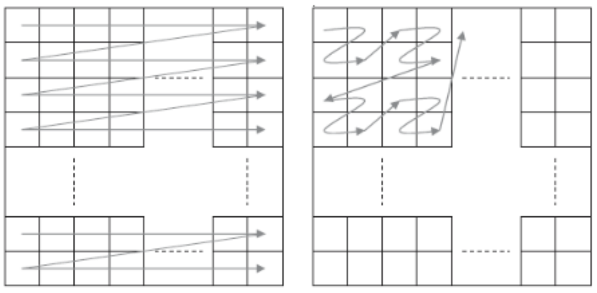

Fig. 8 *Datapath of module 1*

3.4 Module 2: Domain-Range Error Computation

In Module 2, the match error between domain block and range block is computed based on the equation (6) described previously, as illustrated in the equation below:

$$Error = 2 * \frac{\Sigma|sk * (Di - D\mu) - (Ri - R\mu)|}{N \times N} \tag{10}$$

where Di and Ri are ith pixel value of domain and range block respectively; Du and Ru are the calculated domain and range means respectively; N×N is the number of pixel blocks at the quadtree level (i.e., 2×2, 4×4, 8×8); and sk is the quantized s value where sk = [0, 0.25, 0.50, 0.75]. To simplify and facilitate the hardware computation, the equation (10) is further derived to avoid the signed operation and expand the absolute computation:

$$Error = 2 * \frac{\Sigma|Ai - Bi|}{N \times N} = 2 * \frac{\Sigma(Li - Si)}{N \times N} = 2 * \frac{\Sigma(Ei)}{N \times N} \tag{11}$$

where $Ai = sk * Di + R\mu$, $Bi = sk * D\mu + Ri$, $Li = max(Ai, Bi)$ and $Si = min(Ai, Bi)$.

The corresponding hardware unit, which performs the error computation as described in equation (11) for a single pixel, is illustrated in Fig. 9.

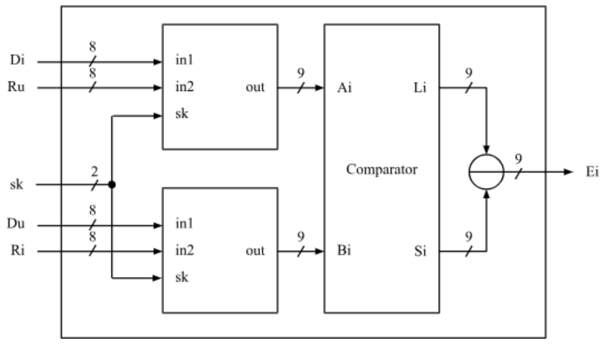

Fig. 9 *Pixel error unit*

The datapath of module 2, shown in Fig. 10, implements the match error computation in parallel and pipeline for all the upper quadtree levels (i.e., 2×2, 4×4, 8×8), except for 1×1 since the pixel data will be directly stored as fractal code and no error matching is needed when reaching level 1x1. The implementation process will undergo 4 iterations for applying sk = [0.75, 0.50, 0.25, 0] subsequently in computing the match error and only the minimum error with its corresponding sk value are stored. Each iteration takes 5 clock cycles to compute the error for all quadtree levels in which the process is carried out in parallel and pipeline. Therefore, Module 2 takes 20 clock cycles to complete the process of computing match error. Upon finishing the process, the error and its corresponding sk value for all quadtree levels are ready to be used for the domain-range error matching in next Module 3.

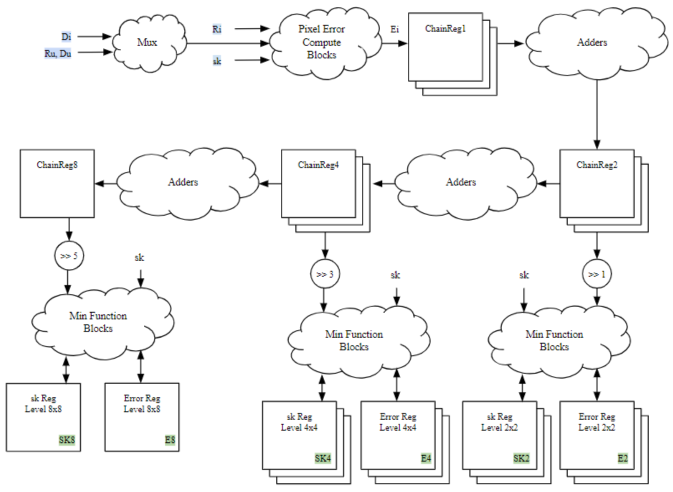

Fig. 10 *Datapath of module 2*

3.5 Module 3: Domain-Range Error Matching

Module 3 primarily comprises control logic and comparison units, and therefore it does not require a large datapath. In Module 3, the readily available errors from Module 2 are used to compare with the threshold value set to find the match. The quadtree matching process is the same as illustrated in the full quadtree searchless algorithm proposed by Jackson et al. [18]. A valid signal is set to '1' when the match is found and the corresponding fractal codes (sk, ru, quadtree level) are to be stored into memory. The overall process consumes 4 to 384 clock cycles.

3.6 Implementation

All the three modules are integrated into a single top module to form the overall architecture. The top module is capable of processing four 8×8 range blocks and producing its corresponding fractal code. To entirely process a 512×512 grayscale image, all the 8x8 range blocks are fed into the design with 32×32 iterations. For experimental and verification purposes, internal ram is used for inputting image pixels and collecting fractal code. The compression ratio is computed based on the collected fractal code size without employing any additional lossless encoding. The generated fractal code is used to reconstruct the image via software decoding process, and the reconstructed image quality is determined against the original image using PSNR as illustrated in equation (8).

The entire design is written in Verilog HDL, and compiled using the Intel Quartus Prime Lite design software, and simulated using the ModelSim simulator, and implemented on a DE1-SoC board equipped with Cyclone V SoC FPGA.

4. Result and Discussion

In the hardware design, the first module, which involves range buffering, domain construction, domain and range mean computation, requires 67 clock cycles to implement. With effective pipelining implemented, the first module does not require many clock cycles to perform the tasks. Next, the second module requires 20 clock cycles to compute the match error for all quadtree levels. Due to the effective parallelism and pipelining achieved, the matching error computation can be done in very few clock cycles. Lastly, the third module requires a maximum of 384 clock cycles to implement the quadtree error matching and outputting. Therefore, 471 clock cycles are required to process four 8×8 range blocks to the corresponding fractals. Overall, the design requires a total of 482,304 clock cycles to fully process a 512×512×8 image.

The design consumes 37% of the total device logic cells (11,785 / 32,070 logic modules used). A total of 20,340 logic units and 7,439 register memory units are required to implement the design in the selected device. From the timing report generated in the full compilation by the Quartus Prime, the design can operate at a maximum frequency of 70.86 MHz. In order to ensure stable hardware operation and to match the device's clock, the design is implemented and operated at 50MHz. With this clock rate, the design is capable of encoding an 512×512×8 image in 482,304/50M = 9.65 ms. With this encoding time, around 103 images can be encoded per second. Therefore, the design throughput for processing a 512×512×8 image is 25.91 MB/s.

For thoroughness of the investigation, the compression performance of the proposed design is assessed using three standard grayscale images with size of 512×512. Fig. 11 shows the image compression results. The average PSNR and compression ratio achieved for the three restored images are 32.90 dB and 14.73 respectively which is fairly high for lossy compression.

The performance comparison between the proposed design, the design proposed by Jackson et al. [18], and the GPU-based implementation introduced by Hernandez-Lopez F.J. and Muñiz-Pérez O. [15] is shown in Table I. In the proposed design, an image size of 512×512×8 is considered, operating at a frequency of 50 MHz. With 482,304 clock cycles, the execution time amounted to 9.65 ms, achieving a throughput of 25.91 MB/s. The logic and memory units required are 27,779. The image tested is a grayscale Lenna image, resulting in a peak signal-to-noise ratio (PSNR) of 32.71 dB and a compression ratio of 14.70. In comparison, Jackson et al. employed the same image size and a slightly lower operating frequency of 32.05 MHz. Their design required 268,032 clock cycles, leading to an execution time of 8.36 ms and a throughput of 29.90 MB/s. The logic and memory units used were higher, totalling 60,256. The image tested also involved a grayscale Lenna image, resulting in a PSNR of 33.12 dB and a compression ratio of 12.77. Overall, while the execution time, throughput, PSNR and compression ratio showed relatively similar results between the two designs, the proposed design demonstrated better performance in terms of logic and memory utilization. When compared to the GPU-based parallel implementation [15], the proposed design significantly outperforms it in all evaluated metrics, except for compression ratio, including execution time and PSNR, highlighting its suitability for real-time, resource-constrained applications.

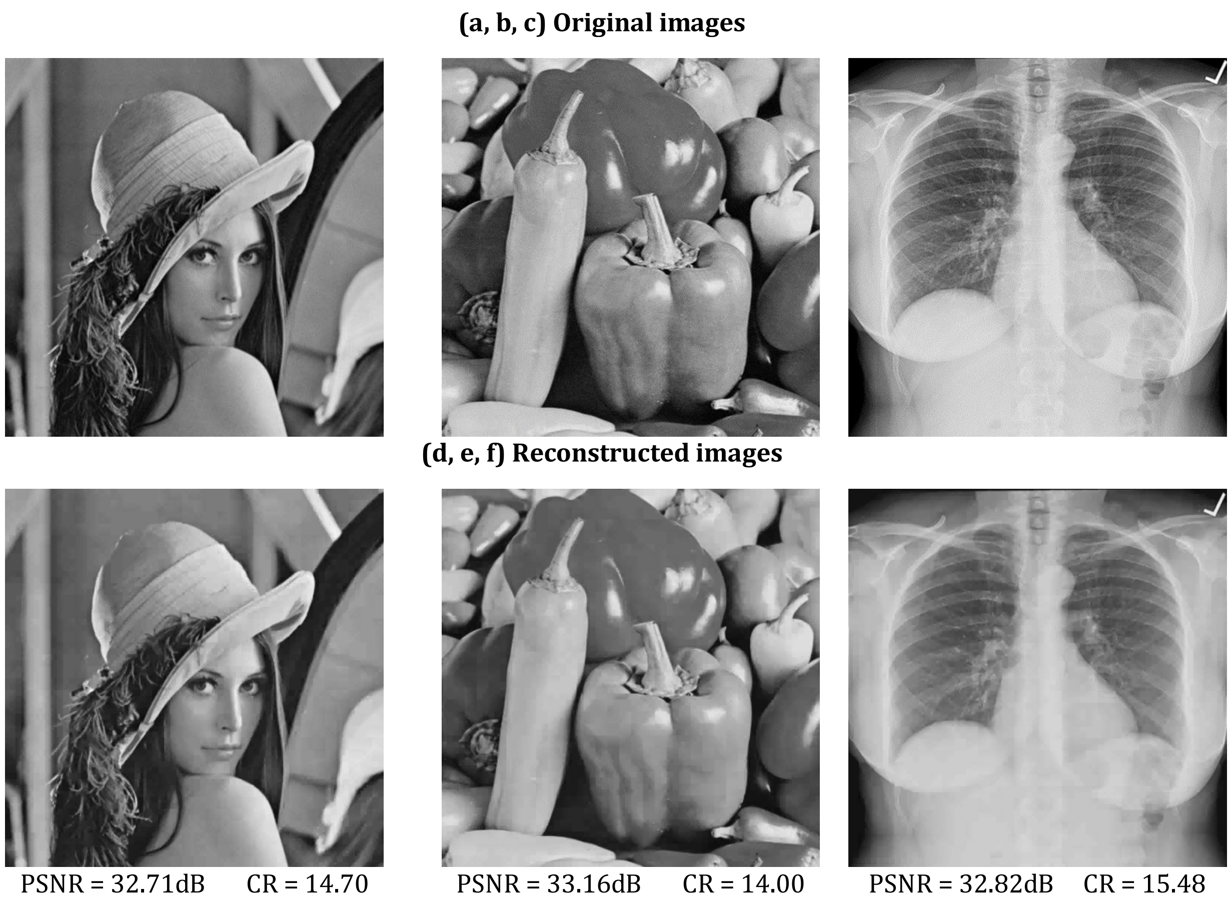
(a, b, c) Original images
(d, e, f) Reconstructed images
PSNR = 32.71dB CR = 14.70
PSNR = 33.16dB CR = 14.00
PSNR = 32.82dB CR = 15.48

Fig. 11 *(a, b, c) The Original Images; (d, e, f) The Compressed Images of (a, b, c) and the Results*

Table 1 *Comparison between proposed design and selected design*

	Proposed	**Jackson et al. [18]**	**Hernandez-Lopez F.J. and Muñiz-Pérez O. [15]**
Platform	Cyclone V SoC FPGA	APEX20K FPGA	Nvidia TITAN RTX GPU
Operating Frequency	50 MHz	32.05 MHz	1.77 GHz
Clock Cycles	482,304	268,032	-
Execution Time	9.65 ms	8.36 ms	1.16 s
Throughput	25.91 MB/s	29.90 MB/s	-
Logic and Memory Units	27,779	60,256	-
Image Test	512×512×8 Lenna	512×512×8 Lenna	512×512×8 Boat
PSNR	32.71 dB	33.12 dB	25.38
Compression Ratio	14.70	12.77	38.6

5. Conclusion

Fractal Image Compression (FIC) is useful in numerous applications because of its potential benefit of high compression ratio, good recovery performance, flexible recovery resolution, and fast decoding process. However, FIC suffers from a major issue that is slow encoding process especially when a full search scheme is implemented. To enable real-time performance, a searchless-based fractal algorithm was implemented on hardware. The design was optimized at the circuit level using parallelism and pipelining techniques to achieve real-time encoding while remaining suitable for implementation on a low-cost FPGA. The proposed architecture successfully encoded a 512×512×8 grayscale image in 9.65 ms, with a processing rate of approximately 103 images per second. Additionally, the design utilized less than half of the total available device resources, demonstrating its efficiency. Future research efforts can focus on increasing parallelism and pipelined operation among the modules in hardware to increase throughput. Additional research may also focus on implementing binary-tree portioning algorithm (known also as Horizontal-Vertical (HV)) instead of quadtree algorithm as the former gives better image compression size flexibility and even better performance.

Acknowledgement

The authors would like to thank Intel Cooperation for providing support through the DE1-SoC development board under Innovate Malaysia Design Competition (IMDC) 2023 program.

Conflict of Interest

Authors declare that there is no conflict of interests regarding the publication of the paper.

Author Contribution

The authors confirm contribution to the paper as follows: ***study conception and design:*** *Abdul-Malik H. Y. Saad, Zhia Hao Chai;* ***data collection:*** *Zhia Hao Chai;* ***analysis and interpretation of results:*** *all authors;* ***draft manuscript preparation:*** *Abdul-Malik H. Y. Saad, Zhia Hao Chai, Nayef A. M. Alduais. All authors reviewed the results and approved the final version of the manuscript.*

References

[1] Barnsley, M., & Hurd, L. (1993). Fractal Image Compression, Wellesley, MA: AK Peters, Ltd.

[2] Zhurba, A. O. (2024). Study of fractal image compression method with the purpose of improving compression quality. System technologies, 4(153), 24-33.

[3] Ghosh, S. K., Mukhopadhyay, J., Chowdary, V. M., & Jeyaram, A. (2002). Relative Fractal Coding and its Application in Satellite Image Compression. ICVGIP.

[4] Yokoyama, T., & Watanabe, T. (2007). DR image and fractal correlogram: A new image feature representation based on fractal codes and its application to image retrieval. International Conference on Multimedia Modeling, 428–439.

[5] Bocchi, L., Coppini, G., Nori, J., & Valli, G. (2004). Detection of single and clustered microcalcifications in mammograms using fractals models and neural networks. Medical Engineering & Physics, 26(4), 303–312.

[6] Kiani, S., & Moghaddam, M. E. (2011). A multi-purpose digital image watermarking using fractal block coding. Journal of Systems and Software, 84(9), 1550–1562.

[7] Ebrahimpour-Komleh, H., Chandran, V., & Sridharan, S. (2001). Face recognition using fractal codes. Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205), 3, 58–61.

[8] Mozaffari, S., Faez, K., & Ziaratban, M. (2005). Character representation and recognition using quad tree-based fractal encoding scheme. Eighth International Conference on Document Analysis and Recognition (ICDAR'05), 819–823.

[9] Garg, R., & Gupta, R. (2024). Comparative analysis of approaches to optimize fractal image compression. In Intelligent Fractal-Based Image Analysis (pp. 167-193). Academic Press.

[10] Breesam, A. M., Hussein, Y. M., & Mohammed, M. J. (2024, February). Fractal compression of digital image processing. In AIP Conference Proceedings (Vol. 3051, No. 1). AIP Publishing.

[11] Gupta, R., Mehrotra, D., & Tyagi, R. K. (2020). Computational complexity of fractal image compression algorithm. IET Image Processing, 14(17), 4425–4434.

[12] Saad, A. M. H., & Abdullah, M. Z. (2016, October). Real-time implementation of fractal image compression in low cost FPGA. In 2016 IEEE International Conference on Imaging Systems and Techniques (IST) (pp. 13-18). IEEE.

[13] Saad, A. M. H., Abdullah, M. Z., Nayef, A. A., & Abdul-Qawy, A. S. H. (2020, August). An Improved Full-search Fractal Image Compression Method with Dynamic Search Approach. In 2020 10th IEEE International Conference on Control System, Computing and Engineering (ICCSCE) (pp. 15-18). IEEE.

[14] Saad, A. M. H., Abdullah, M. Z., Alduais, N. A. M., Abdul-Qawy, A. S. H., Nasser, A. B., Ghanem, W. A. H., & Sa'd, A. H. Y. (2022). Deep Pipeline Architecture for Fast Fractal Color Image Compression Utilizing Inter-Color Correlation. IEEE Access, 10, 110444-110458.

[15] Hernandez-Lopez, F. J., & Muñiz-Pérez, O. (2022). Parallel fractal image compression using quadtree partition with task and dynamic parallelism. Journal of Real-Time Image Processing, 19(2), 391-402.

[16] Asati, R., Raghuwanshi, M. M., & Singh, K. R. (2022). Fractal Image Coding-Based Image Compression Using Multithreaded Parallelization. In Information and Communication Technology for Competitive Strategies (ICTCS 2021) ICT: Applications and Social Interfaces (pp. 559-569). Singapore: Springer Nature Singapore.

[17] Abood, B. S. Z., Akkar, H. A., & Al-Safi, A. S. (2023). Fast Full-Search Algorithm of Fractal Image Compression for Acceleration Image Processing. Science Journal of University of Zakho, 11(1), 119-126.

[18] Jackson, D. J., Ren, H., Wu, X., & Ricks, K. G. (2007). A hardware architecture for real-time image compression using a searchless fractal image coding method. Journal of Real-Time Image Processing, 1(3), 225–237.

[19] Martin, C. E., & Curtis, S. A. (2013). Fractal image compression. Journal of Functional Programming, 23(6), 629–657.

[20] Jacquin, A. E. (1992). Image coding based on a fractal theory of iterated contractive image transformations. IEEE Transactions on Image Processing, 1(1), 18–30.

[21] Galabov, M. (2003). Fractal image compression. CompSysTech, 320–326.

[22] Saad, A. M. H., Abdullah, M. Z., Alduais, N. A. M., & Sa'ad, H. H. Y. (2020). Impact of spatial dynamic search with matching threshold strategy on fractal image compression algorithm performance: study. IEEE Access, 8, 52687-52699.

[23] Saad, A.-M. H. Y., & Abdullah, M. Z. (2018). High-speed fractal image compression featuring deep data pipelining strategy. IEEE Access, 6, 71389–71403.

[24] Tong, C.-S., & Pi, M. (2001). Fast fractal image encoding based on adaptive search. IEEE Transactions on Image Processing, 10(9), 1269–1277.

[25] Furao, S., & Hasegawa, O. (2004). A fast no search fractal image coding method. Signal Processing: Image Communication, 19(5), 393–404.