An Efficient Implementation Method of Fractal Image Compression on Dynamically Reconfigurable Architecture

Hidehisa Nagano, Akihiro Matsuura, and Akira Nagoya

NTT Communication Science Laboratories
2-4 Hikaridai, Seika-cho, Soraku-gun, Kyoto, 619-0237, JAPAN
{nagano, matsuura, nagoya}@cslab.kecl.ntt.co.jp

Abstract. This paper proposes a method for implementing fractal image compression on dynamically reconfigurable architecture. In the encoding of this compression, metric computations among image blocks are the most time consuming. In our method, processing elements (PEs) configured for each image block perform these computations in a pipeline manner. By configuring PEs, we can reduce the number of adders, which are the main computing elements, by half even in the worst case. This reduction increases the number of PEs that work in parallel. In addition, dynamic reconfigurability of hardware is employed to omit useless metric computations. Experimental results show that the resources for implementing the PEs are reduced to 60 to 70% and the omission of useless metric computations reduces the encoding time to 10 to 55%.

1 Introduction

Due to recent advances in programmable logic devices such as Field Programmable Gate Arrays (FPGAs), reconfigurable computing has become a realistic computing paradigm [1]. Some reconfigurable devices have begun to support dynamic and partial reconfiguration [2][3], and recent research has made use of such reconfigurability to accelerate solving problems [4]. In this paper, we focus on the acceleration of fractal image compression using dynamically reconfigurable computing.

Fractal image compression [5] is expected to be a promising compression method for digital images in terms of its high compression ratios and fidelity. However, due to the unacceptable encoding time, use of this compression method is limited to areas where the encoding time is not critical. Therefore, acceleration of the encoding is indispensable to a variety of uses. In fractal-based image encoding, an image is divided into sub-blocks and the root mean square metrics among them are computed. These metric computations carry the burden in encoding the entire image because numerous metric computations for the sub-blocks must be done.

In our method, processing elements (PEs) are devoted to the corresponding sub-blocks. Each PE computes the metrics between the corresponding sub-blocks and the candidate sub-blocks in the pipeline. In each step of the pipeline, one metric computation is executed on a PE. In addition, plural PEs implemented on the reconfigurable architecture concurrently perform the metric computations.

Our method has two advantages owing to the reconfigurability of the hardware. One advantage is the structure of PEs. Since each PE is reconfigured for a sub-block, a number of multiplications in the metric computations can be regarded as constant multiplications. This makes it possible to use shifts and adders instead of variable-by-variable general-purpose multipliers. By using constant multiplications, the number

of adders needed for PEs is reduced , even in the worst case, to half that of PEs in the straightforward implementation. This reduces the reconfigurable resources needed for the PE, and consequently increases the number of PEs that work in parallel. The second advantage is the omission of useless metric computations. We can accelerate the encoding by omitting metric computations that do not influence the fidelity of the decoded image. Such omission can be achieved by the partial reconfiguration of the hardware.

Experimental results show that the resources needed to implement the PEs can be reduced to approximately 60 to 70% of the straightforward implementation, even in the worst case. Results also show that omitting useless computations reduces the encoding time to 10 to 55%. Thus, our method utilizing reconfigurability significantly accelerates the encoding.

This paper is organized as follows. Sect. 2 briefly describes fractal image compression. In Sect. 3, we present a dynamically reconfigurable architecture of the encoder for fractal image compression. In Sect. 4, experimental results are shown and analyzed. Sect. 5 concludes this paper.

2 Fractal Image Compression

In this section, we describe the outline of fractal image compression. A fractal-based image encoding scheme was first promoted by M. Barnsley [6]. The fractal image compression for grey-scale images was developed by A. Jacquin. For a detailed survey and extensions of fractal image compression, we refer readers to a paper by Jacpuin [7].

2.1 Iterated Function System (IFS)

Here, we briefly present the underlying mathematical principles of fractal image compression based on a theory of IFS [8][9]. An IFS consists of a collection of contractive affine transformations $\{w_i : M \to M | 1 \leq i \leq m\}$, where w_i maps a metric space M to itself. A transformation w is said to be contractive when there is a constant number $s (0 \leq s < 1)$ and $d(w(\mu_1), w(\mu_2)) \leq s\, d(\mu_1, \mu_2)$ holds for any points $\mu_1, \mu_2 \in M$, where $d(\mu_1, \mu_2)$ denotes the metric between μ_1 and μ_2. The collection of transformations defines a map $W(S) = \bigcup_{i=1}^{m} w_i(S)$, where $S \subset M$. Note that the map W is applied not to the set of points in M but to the set of sub-sets in M. Let W be a contractive map on a compact metric space of images with the bounded intensity, then the Contractive Mapping Fixed-Point Theorem asserts that there is one special image x_W, called the attractor, with the following properties.

1. $W(x_W) = x_W$.
2. $x_W = \lim_{n \to \infty} W^n(S_0)$, which is independent of the choice of an initial image S_0.

Fractal image compression relies on this theorem. Suppose the attractor of a contractive map W is the original image to be compressed. From property 2, the original image is obtained from any initial image by applying W. The goal of fractal image compression is to find the contractive map W whose attractor is sufficiently close to the original image and to store the parameters of W instead of the intensity values.

2.2 Image Compression Using IFS

The fractal image compression algorithm we focus on is based on the quad-tree decomposition scheme [5][8]. This scheme was developed for grey-scale images.

Let $R_1, R_2, \ldots, R_p$ be non-overlapping sub-squares of $B \times B$ pixels in an image. These sub-squares are called *range blocks*. The union of the set of range blocks covers

the entire image. All $2B \times 2B$ sub-squares, $D_1, D_2, \ldots, D_q$, are called *domain blocks*, where all of the over-lapping sub-squares of this size are taken into account. The collection of all domain blocks is called *the domain pool*. The map W in Sect. 2.1 maps sub-sets of the domain pool to range blocks.

For each R_k, we search for the domain block that approximates R_k with the smallest metric out of the domain pool. A range block that is approximated by an acceptable metric is said to be *covered*. A range block for which acceptable domain blocks are not found is said to be *uncovered*. Note that before calculating metrics, domain blocks are resized to the same size as range blocks by averaging the intensity values of 2×2 pixels. Let the pixel intensities of the resized domain block be $a_1, a_2, \ldots, a_n$ and let those of the range block be $b_1, b_2, \ldots, b_n$. The metric $R(D_j, R_k)$ between a domain block D_j and a range block R_k is defined as $R(D_j, R_k) = \sum_{i=1}^{n} (s\, a_i + o - b_i)^2$, where s is the scaling factor and o is the offset factor. Differentiating R by s and o, s and o that minimize R are calculated as

$$s = \frac{n\sum_{i=1}^{n} a_i b_i - \sum_{i=1}^{n} a_i \sum_{i=1}^{n} b_i}{n\sum_{i=1}^{n} a_i^2 - (\sum_{i=1}^{n} a_i)^2}, \quad o = \frac{\sum_{i=1}^{n} b_i - s\sum_{i=1}^{n} a_i}{n}. \tag{1}$$

In this case, $R(D_j, R_k)$ is calculated as

$$R(D_j, R_k) = \frac{1}{n}(n\sum_{i=1}^{n} b_i^2 - (\sum_{i=1}^{n} b_i)^2) - \frac{s^2}{n}(n\sum_{i=1}^{n} a_i^2 - (\sum_{i=1}^{n} a_i)^2). \tag{2}$$

The entire fractal image compression scheme is shown in Fig. 1. We call this algorithm EXAUS. The parameter *tolerance* settles the compression ratio and the fidelity. If a range block is uncovered, that is, if *tolerance* is violated for all domain blocks, the range block is divided into four smaller range blocks and the best-match domains for these smaller range blocks are searched. In this case, domain blocks with half the size in the side length are evaluated. To increase the probability of finding good range-domain matches, the domain pool may include sub-squares obtained by rotating domain blocks $90°$, $180°$, $270°$ and $360°$ and flipping them vertically (eight transformations). These sub-blocks are also called domain blocks. After finding the best matches for all range blocks, s and o and the number specifying the best-match domain block are stored for each range block instead of their pixel intensities; as such, the encoding is completed. For s and o, the necessary bit widths to be preserved are empirically known to be 5 and 7, respectively. On the other hand, for decoding the image, the corresponding transformations are repeated. This iteration is started from any initial image. Iterations of 10 to 20 times are sufficient to decode an image with good fidelity.

With regard to the computing time, the iterations in the decoding are not so troublesome. However, the encoding time is often not acceptable due to the many iterations of complex metric computations. For example, in encoding a 256×256 pixel image, the number of range blocks of 8×8 pixels is 1024 and the number of domain blocks of 16×16 pixels is 58081 (not including the domain blocks obtained by rotating and flipping). Therefore, the total number of iterations is about 60 million. For reducing the encoding time, some schemes that try to reduce the number of range-domain comparisons have been proposed [7]. In these schemes, only range-domain comparisons among sub-blocks with the same characteristics are evaluated. Unfortunately, the encoding times using these schemes are still unacceptable.

In the next section, we propose a method for implementing fractal image compression using parallel computation on a reconfigurable hardware. Our method has two

advantageous features for reducing the encoding time. The first one concerns accelerating the inside of the innermost loop of EXAUS by using pipeline processing. In each step of the pipeline, one iteration is executed on a PE. Looking at the equations (1)–(2), we can observe that many computations, $\sum_{i=1}^{n} a_i$, $\sum_{i=1}^{n} a_i^2$, $(\sum_{i=1}^{n} a_i)^2$, $\sum_{i=1}^{n} b_i$, $\sum_{i=1}^{n} b_i^2$, $(\sum_{i=1}^{n} b_i)^2$, $(n\sum_{i=1}^{n} a_i^2 - (\sum_{i=1}^{n} a_i)^2)$ and $(n\sum_{i=1}^{n} b_i^2 - (\sum_{i=1}^{n} b_i)^2)$, can be performed in advance out of the innermost loop and have to be done only once for each domain block and range block. Therefore, inevitable computations inside of the innermost loop are other computations for s and R and comparisons. More precisely, the indispensable burden is the part $\sum_{i=1}^{n} a_i b_i$. The method presented in this paper lightens this burden by executing it in a pipeline. The second feature concerns reducing the number of iterations of metric computations. In EXAUS, the metric computations for all pairs of range and domain blocks are done. With regard to the fidelity of the decoded image, it is not necessary to evaluate all range-domain matches. Our scheme omits these useless computations. This omission is owing to the dynamic and partial reconfigurability of the hardware. In the next section, we describe these features in detail.

```
min_R = tolerance;
For (j = 1 ; j ≤ num_range ; j++) {
        For (k = 1 ; k ≤ num_domain ; k++) {
                compute s;
                if (0 ≤ s < 1.0)
                        if (R(D_j, R_k) < min_R)
                                { min_R = R(D_j, R_k); best_domain[j] = k; }
        }
        if (min_R == tolerance)
                set R_j uncovered and divide it into 4 smaller blocks;
}
```

Fig. 1. Pseudocode of EXAUS.

3 Encoder Architecture

In this section, we propose an encoder of fractal image compression implemented on dynamically reconfigurable hardware. In this paper we assume that the dynamically reconfigurable hardware is constructed from fine-grain programmable logic elements (LEs) with a register and programmable wires among them such as Xilinx XC6200 series FPGAs [2] or Atmel AT40K FPGAs [3]. We also suppose that these LEs and wires can be configured by setting configuration data dynamically and partially. Some kind of architectures of LEs have been proposed such as Look-Up-Table (LUT) based architectures, multiplexer based architectures and others. However, generalizing these LEs as only programmable logic function units with limited inputs and a output, for example, 3 or 4 inputs, we consider the encoder architecture independent from the architectures of LEs and realized as wired logic of LEs.

3.1 A Pipelined PE Network

Fig. 2 shows the overview of the proposed encoder implemented on dynamically reconfigurable hardware. The PE performs computations inside the innermost loop of EXAUS. Each PE is devoted to a given range block and finds the best-match domain among all domain blocks that are given through the data path in pipeline. In each step of the pipeline, one iteration of the inside of the innermost loop is executed on each PE. After finding the best match, the PE informs the best-match domain to the control unit. Then, the PE is reconfigured for another range block with the configuration data given

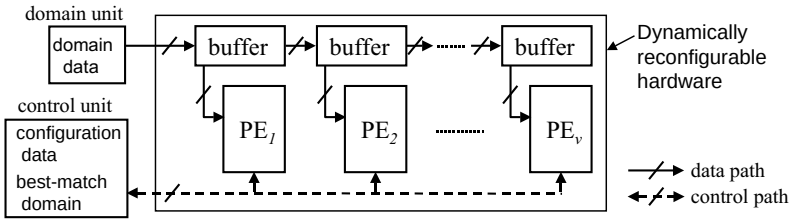

Fig. 2. Overview of the Encoder.

by the control unit. These PEs are reconfigured until no uncovered range block exists. All PEs are connected by the data path and perform the calculations in a pipeline manner while transferring the domain data. The domain unit inputs the domain data into the PE network until all range blocks are checked against all domain blocks. A buffer transfers domain data to the PE connected with it and the neighboring buffer through the data path without the pipeline stole during the reconfiguration of that PE.

3.2 Dynamically Reconfigurable PE

Here, we describe the PE architecture in detail. A PE performs the calculations inside of the innermost loop of EXAUS. As stated in Sect. 2.2, the computation of $\sum_{i=1}^{n} a_i b_i$ is the burden for computing R. Therefore, it is desirable that $\sum_{i=1}^{n} a_i b_i$ is calculated in parallel. The basic implementation is shown in Fig. 3. The parameters $a_1, a_2, \ldots, a_n$ are enter the PE in parallel and $\sum_{i=1}^{n} a_i b_i$ is calculated in the pipeline. The hardware component for calculating $\sum_{i=1}^{n} a_i b_i$ is denoted by P1. The hardware component for calculating s, o and R is denoted by P2. P2 finds the minimum R and stores the identity number of the domain block that provides it. These computations are also performed in the pipeline. In each step of the pipeline, one iteration of the inside of the innermost loop is completed on a PE. Precomputable parameters for the domain block, such as $\sum_{i=1}^{n} a_i$, $\sum_{i=1}^{n} a_i^2$ and others, are given to the PEs through the data path in parallel cooperating with the pipeline. The other precomputable parameters for the range block are stored in P2. In order to implement P1, we need n multipliers and $n-1$ adders. On the other hand, the computations performed in P2 are three multiplications, three comparisons, and one division. Since the hardware resources needed for P2 and the buffer are much smaller than those for P1, it is important to reduce the hardware resources for P1 for total resource reduction. By reducing the resources for a PE, we can implement many PEs at the same time and increase the number of range blocks processed in parallel.

Now, we describe an efficient implementation method using reconfigurability. Note that the multipliers in P1 in Fig. 3 are implemented with adders and shifts. Since shifts are implemented by wiring on reconfigurable devices, it is important to reduce the number of adders. Therefore, we implement P1 as follows. Let the integer b_i be presented as $b_{i,l} b_{i,l-1} \cdots b_{i,1}$ in the binary presentation, where $b_{i,j}$ is 0 or 1, and let the bit width of a_i's and b_i's be l. Then, multiplication $a_i b_i$ is presented as $a_i b_i = \{(a_i b_{i,l}) \ll (l-1)\} + \{(a_i b_{i,l-2}) \ll (l-2)\} + \cdots + \{a_i b_{i,1}\}$, where $a \ll j$ denotes the j-bit shift of a to the left. The multiplication is implemented by $l-1$ adders. In this case, the number of adders for computing $\sum_{i=1}^{n} a_i b_i$ at P1 is

$$(l-1)\,n + (n-1) = ln - 1\,. \tag{3}$$

On the other hand, in the case of reconfiguring a PE for a range block, we can make use of the fact that $b_{i,j}$ is a constant. Namely, we can get rid of the part $(a_i b_{i,j})$ if $b_{i,j}$ is 0.

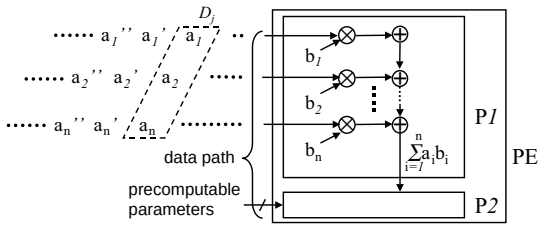

Fig. 3. PE architecture.

First, we rearrange $\sum_{i=1}^{n} a_i b_i$ as follows.

$$\begin{aligned}\sum_{i=1}^{n} a_i b_i = \quad & \{(a_1 b_{1,1} + a_2 b_{2,1} + \cdots + a_n b_{n,1})\} \\ & + \{(a_1 b_{1,2} + a_2 b_{2,2} + \cdots + a_n b_{n,2}) \ll 1\} \\ & \cdots \\ & + \{(a_1 b_{1,j} + a_2 b_{2,j} + \cdots + a_n b_{n,j}) \ll (j-1)\} \\ & \cdots \\ & + \{(a_1 b_{1,l} + a_2 b_{2,l} + \cdots + a_n b_{n,l}) \ll (l-1)\}. \end{aligned} \tag{4}$$

The term of the j-th row of (4),

$$(a_1 b_{1,j} + a_2 b_{2,j} + \cdots + a_n b_{n,j}), \tag{5}$$

can be computed in a different way by inverting $b_{i,j}$'s for all i, $1 \le i \le n$, as follows.

$$\sum_{i=1}^{n} a_i - (a_1 \overline{b_{1,j}} + a_2 \overline{b_{2,j}} + \cdots + a_n \overline{b_{n,j}}). \tag{6}$$

If the number of non-zero bits among $b_{1,j}, b_{2,j}, \ldots, b_{n,j}$ is more than $\frac{n}{2}$, we configure P1 using (6) instead of (5). This is done for all j. Then, the number of adders needed to implement P1 is

$$\frac{nl}{2} - 1 \tag{7}$$

in the worst case. Therefore, the number of adders is reduced at least by half. The worst case happens when the number of non-zero bits of $b_{1,j}, b_{2,j}, \ldots, b_{n,j}$ is $\frac{n}{2}$ or $\frac{n}{2}+1$ for all $j (1 \le j \le l)$. In practice, since b_i's $(1 \le i \le n)$ are intensities of a sub-block of a digital image, $b_{i,j}$'s $(1 \le i \le n)$ tend to be the same, especially when j is close to l. Therefore, the worst case rarely happens and the number of adders is expected to be much smaller than that of the worst case. In Sect. 4, the scheme's experimental results show the reduction of adders and resources for the PE.

$\sum_{i=1}^{n} a_i b_i$ discussed here can be implemented by distributed arithmetic (DA) using memories [10], if LEs are LUT based and can be used as memories. This implementation can be space efficient. However, the space efficiency is terribly affected by the number of inputs, M, of LEs (i.e., the number of content bits possible for a LUT). When M is 3, the DA implementation is not comparable even with the our worst case implementation for practical n and l. In addition, the DA implementation is limited to LUT based architectures. Therefore, we would not adopt the DA implementation.

3.3 A Compression Algorithm Using Dynamic Reconfiguration

Next, we describe the improvement of EXAUS. We use the dynamic partial reconfigurability of the pipelined PE network stated in Sect. 3.1. In this PE network, each PE can be reconfigured without the pipeline stole since buffers transfer the domain data. Therefore, all PEs do not have to be reconfigured at the same time. In EXAUS, all domain blocks are checked for each range block. However, by choosing the appropriate *tolerance* and finding only one range-domain match that accepts the *tolerance* for each range block, we can obtain a decoded image with sufficiently good fidelity.

Generally, the fidelity of the decoded image is evaluated with the signal-to-noise ratio (*SNR*) defined as $SNR = 10\log_{10}(dr(S)^2/d_e(S,S'))$ (dB). Here, $dr(S)$ is the dynamic range of the original image S and $d_e(S,S')$ is the metric between S and the decoded image S'. $d_e(S,S')$ is defined as $d_e(S,S') = \sum_{i=1}^{t}(\mu_i - \mu'_i)^2$, where μ_i's and μ'_i s are pixel intensities of S and S'. An *SNR* of approximately 28 to 32 dB is sufficient for fine fidelity.

Now, we present the improved algorithm that omits the exhaustive range-domain matches in Fig. 4. We call this algorithm OMIT. This improvement can be implemented on hardware by dynamically and partially reconfiguring each PE as soon as the first acceptable range-domain match is found.

```
For (j = 1 ; j ≤ num_range ; j++) {
        flag = 0;
        For (k = 1 ; k ≤ num_domain ; k++) {
                compute s;
                if (0 ≤ s < 1.0)
                        if (R(D_j, R_k) < tolerance)
                                { best_domain[j] = k; flag = 1; break; }
        }
        if (flag == 0)
                set R_j uncovered and divide it into 4 smaller blocks;
}
```

Fig. 4. Pseudocode of OMIT.

4 Experiments

4.1 Resources for PE

First, we evaluate the number of adders needed to implement a P1 and the resources for PEs. Table 1 shows the number of adders needed to implement P1 when the range size is 4×4 pixels and 8×8 pixels for two benchmark images, "Girl" (256×256 pixels, 8 b/p) of SIDBA and "Lenna" (480×512 pixels, 8 b/p). NO-RECONF denotes the number of adders needed to implement a P1 without using reconfigurability. WORST of RECONF

Table 1. Number of adders.

Range size (pixels)	NO-RECONF	RECONF		
		WORST	AVERAGE	
			Girl	Lenna
8×8	511	255 (49.9%)	150 (29.3%)	136 (26.6%)
4×4	127	63 (49.6%)	36 (28.3%)	29 (22.8%)

Table 2. Number of LEs for a PE

Range size (pixels)	NO-RECONF	RECONF		
		WORST	AVERAGE	
			Girl	Lenna
8×8	10532	6548 (62.2%)	4674 (44.4%)	4642 (44.1%)
4×4	3535	2591 (73.3%)	2109 (59.7%)	2057 (58.1%)

denotes the number of adders using reconfigurability in the worst case and its ratio to that of NO-RECONF. The values of WORST are calculated by (7) and those of NO-RECONF is calculated by (3). AVERAGE of RECONF denotes the number of adders in the average case using reconfigurability. Table 1 shows that the number of adders is significantly reduced by utilizing reconfigurability.

Table 2 shows the total number of LEs needed for the PE. Again, NO-RECONF denotes the number of LEs for a PE implemented without using reconfigurability. WORST of RECONF denotes the number of LEs in the worst case using reconfigurability and its ratio to the number of NO-RECONF. AVERAGE of RECONF denotes these average numbers using reconfigurability. Here, we assume that an LE is an LUT with four inputs and an output. Carry save adders are used for the adders for P1. The final summation and the carry vector provided by these carry save adders is added by a carry look ahead adder at the end. Therefore, one carry look ahead adder is implemented in P1. We evaluated the number of LEs by counting 1-bit full adders needed for carry save adders of P1 and assuming that a 1-bit full adder is implemented with two LEs. Altera FPGA mapping tool MAX+Plus II was used to evaluate the number of LEs for P2 and the carry look ahead adder. The number of LEs for P2 is 1556 when the range-size is 8×8, and 1343 when the range-size is 4×4.

Table 2 indicates that the value of WORST is approximately 60% of that of NO-RECONF when the range-size is 8×8. When the range-size is 4×4, the ratio is approximately 70%. These reductions are important for implementing many PEs under the resource limitation to accelerate the encoding. Although further investigations for dynamic allocation of PEs are needed, the average case suggests that dynamically changing the number of PEs and making full use of limited resources is promising.

4.2 Efficiency of The Improved Algorithm

Here, to confirm the efficiency of OMIT, we evaluate the number of iterations of the inside of the innermost loop by computer simulation. The numbers of iteration for EXAUS and OMIT are compared under the same *SNR*s for "Girl" and "Lenna." *SNR*s are evaluated with the decoded images obtained after 20 iterations of the mapping. The image was divided into 8×8-size range blocks at the beginning. Uncovered range blocks were divided into four 4×4-size range blocks. Domain blocks obtained by rotating and flipping are not included in the domain pool. We evaluated the summation of iterations for 8×8 and 4×4-size range blocks.

In each step in our pipelined PE network, a domain block is inputted to the network. Therefore, the number we evaluated indicates the encoding time when one PE is implemented. Fig. 5 shows the evaluated values for "Girl" and "Lenna." For "Girl," the values for OMIT are from 55% to 70% of EXAUS when the *SNR* is from 28 to 30 dB. For "Lenna," these ratios are from 10% to 30%. Note that for "Lenna," no 8×8-size range block is divided in EXAUS when the *SNR* is 30 dB. Therefore, the *SNR* does not fall to less than 30 dB, and the number of iterations cannot be reduced more. The ratios for "Lenna" are given by comparing the values of OMIT and the minimum value of

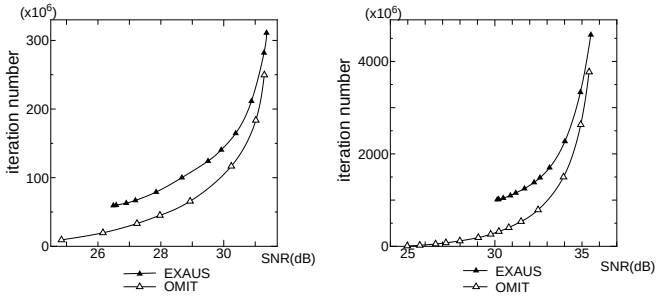

Fig. 5. Number of iterations

EXAUS. As a result, the improvement of the algorithm reduces the encoding times for these images to 10 to 55% with fine fidelity.

5 Conclusion

In this paper, we have proposed a method for implementing fractal image compression on a dynamically reconfigurable architecture. In our method, PEs are specialized for image blocks and concurrently perform the computations in a pipeline manner. Furthermore, plural PEs work in parallel. By using the reconfigurability, the reconfigurable resources needed for each PE is reduced to 60% to 70% of that of the straightforward implementation in the worst case. This reduction is quite important for implementing many PEs under the resource limitation in order to accelerate the encoding. In addition, an improved fractal image compression algorithm omitting useless computations, which can be achieved by dynamic and partial reconfigurability, can reduce the encoding time to 10 to 55%. As a result, by applying these utilization of the reconfigurability to the proposed pipelined PE network, the encoding time of the PE network can be significantly reduced under the reconfigurable resource limitations.

In future works, we plan to implement the presented encoder and confirm the effectiveness of the presented PE network in practice.

References

1. T. Miyazaki, "Reconfigurable Systems: A Survey," *Proc. of ASP-DAC '98*, pp. 447–457, Feb. 1998.
2. Xilinx, *XC6200 Field Programmable Gate Arrays*, Apr. 1997.
3. Atmel, *AT40K FPGAs*, Dec, 1997.
4. J. R. Koza, F. H. Bennett III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre, "Evolving Computer Programs using Rapidly Reconfigurable Field-Programmable Gate Arrays," *Proc. of FPGA '98*, pp. 209–219, Feb. 1998.
5. Y. Fisher (Ed.), *Fractal Image Compression: Theory and Application*, Springer, 1996.
6. M. F. Barnsley, V. Ervin, D. Hardin, and J. Lancaster, "Solution of an Inverse Problem for Fractals and Other Sets," *Proc. of Natl. Acad. Sci USA*, 83:1975–1977, Apr. 1986.
7. A. E. Jacquin, "Fractal Image Coding: A Review," *Proc. of the IEEE*, vol. 81, no. 10, pp. 1451–1465, Oct. 1993.
8. Y. Fisher, *Fractal Image Compression*, SIGGRAPH Course Notes, 1992.
9. M. F. Barnsley, *Fractals Everywhere*, AK Peters, 1993.
10. S. A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review," *IEEE ASSP Magazine*, pp. 4–19, July. 1989.