Journal of Real-Time Image Processing (2020) 17:1375–1387
https://doi.org/10.1007/s11554-019-00894-7

ORIGINAL RESEARCH PAPER

CUDA implementation of fractal image compression

Abir Al Sideiri[1,2,3] · Nasser Alzeidi[2] · Mayyada Al Hammoshi[4] · Munesh Singh Chauhan[5] · Ghaliya AlFarsi[1]

Received: 29 November 2018 / Accepted: 24 June 2019 / Published online: 3 July 2019

Abstract
Fractal coding is a lossy image compression technique, which encodes the image in a way that would require less storage space using the self-similar nature of the image. The main drawback of fractal compression is the high encoding time. This is due to the hard task of finding all fractals during the partition step and the search for the best match of fractals. Lately, GPUs (Graphical Processing Unit) have been exploited to implement fractal image compression algorithms due to their high computational power. The prime aim of this paper is to design and implement a parallel version of the Fisher classification scheme using CUDA to exploit the computational power available in the GPUs. Fisher classification scheme is used to reduce the encoding time of fractal images by limiting the search for the best match of fractals. Encoding time, compression ratio and peak signal-to-noise ratio was used as metrics to assess the correctness and the performance of the developed algorithm. Eight images with different sizes (512×512, 1024×1024 and 2048×2048) have been used for the experiments. The conducted experiments showed that a speedup of $6.4\times$ was achieved in some images using NVIDIA GeForce GT 660 M GPU.

Keywords Fractal image compression · Quad-tree partitioning · GPU · Parallel processing · CUDA

1 Introduction

Due to the advances in information systems and technologies, there is an essential need for efficient data storage and fast data transmission. Digital images possess the characteristic of being data intensive [1, 2]. Thus, storing these images in less memory leads to a direct reduction in data transmissions and storage costs. Therefore, data compression has always been an active area of research to offer solutions for these critical issues.

There are two general categories of data compression methods, namely lossless and lossy methods. A lossless method will produce an image that is identical to the original image when decompressed. On the other hand, a lossy method will produce an image that closely resembles the original image. The main drawback of lossless methods is that they cannot achieve very high compression ratios. Hence, lossy methods are mostly used for image compression applications because the losses of very minor graphic details are not critical. Nonetheless, for certain applications, lossless compression is a necessity such as the compression of text files or executable codes [1]. One example of the lossy image compression methods currently available is the method of fractal image compression, developed by Michael Barnsley and his associates [1]. Fractal image compression

✉ Abir Al Sideiri
abir@buc.edu.om

Nasser Alzeidi
alzidi@squ.edu.om

Mayyada Al Hammoshi
mhammoshi@viu.edu

Munesh Singh Chauhan
munesh.sal@cas.edu.om

Ghaliya AlFarsi
galfarsi@buc.edu.om

1 Department of Information Technology, Buraimi University College, Al-Buraimi, Oman

2 Department of Computer Science, Sultan Qaboos University, Muscat, Oman

3 Department of Systems and Networks, Universiti Tenaga National (UniTen), Kajang, Malaysia

4 School of Computer Information System, Virginia International University, Fairfax, VA, USA

5 Department of Information Technology, College of Applied Science, Salalah, Oman

methods seek for a way to represent images in terms of iterated functions that describe how parts of an image are self-similar to other parts. An encoding stores information about an image which can always be decoded at a prescribed level of details, in spite of the size of the decoded image and without the regular scaling artefacts such as pixilation. The size of the resulting encoding is based on the encoding algorithm's ability to achieve the self-similarity of the image, theoretically leading to more efficient encodings.

Despite its advantages like fast decoding, resolution independence and high compression ratio, fractal image compression requires enormous execution time as searching all domain blocks for a matching range block is very time consuming. Researchers proposed different schemes to improve the searching process such as classification methods. A classification method is a pre-process executed before the search phase and aims to reduce the size of the domain pool [3–5]. The most commonly used classification scheme is the one proposed by Fisher [3]. Nonetheless, the encoding phase still suffers from extensive time. New trends have been studied for speeding up the encoding phase including different parallel processing schemes [3].

The rapid increase in the peak performance of the GPUs has motivated researchers to study the use of GPUs for fractal image coding [4–9]. In [10] and [11], the authors presented the systolic architecture approach where each range block is compared with a different domain blocks at any given step. Domain blocks are shifted to the next processor for further comparison step in the pipeline once the comparison step is completed. When all domain blocks have passed through the pipeline, the comparison step is complete for all range blocks.

The majority of parallel algorithms that have been developed tend to use some forms of dynamic load balancing as in [11]. This approach involves a master process and a large number of slave processes. In [7, 12], the master process divides the image blocks amongst the slaves equally. Therefore, each slave is responsible for its own subset of the overall domain pool. Then the master transmits range blocks to any idle slaves and waits for them to return the best match from their subset of the domain pool. Depending on whether this is a satisfactory match or not, this range block may or may not be sent to another processor. The master process ensures that all slaves are kept busy until the task is completed.

An Open MP parallelization for Fisher's fractal image encoding method was proposed in [13, 14]. The parallel execution procedure is applied to partition the data involved in range. Therefore, several slice areas are formed, and a multicore thread is responsible for searching the fractal codes at every slice region of every range block involved in range block. The final fractal codes are combined with the ones of each slice region [13, 14].

The authors in [15] implemented adaptive fractal image coding algorithm on GPUs using CUDA (Compute Unified Device Architecture) to achieve a given quality of a compressed image. The original adaptive fractal image coding algorithm consists of three loops which include no data dependences. This eased parallelizing the coding algorithm by distributing the loop elements on multiple computing cores [15]. The proposed algorithm utilized a vector of pointers to range blocks whose quality is less than the required level to improve the efficiency of the parallelism on the GPU environment. An improved parallel algorithm is described in [16]. Whilst original implementation carries out the computation by 1 thread, the improved version calculates it by 16 threads. Despite that the number of threads in a thread block can increase to enhance the occupancy; the communication between the threads within the thread block may increase the overhead.

A parallel model using CUDA has been presented to parallelized fractal encoding algorithm in [17]. This model states that the parallelization can be done in levels. First, the range blocks can be processed in parallel to find the best matching domain block. Second, whilst finding the scaling and offset, all pixel values in a block can be processed in parallel. This model considers each domain/range block as a corresponding to a block of threads on the CUDA model; the processing of one pixel values from the domain block is handled by a single thread along with the corresponding single pixel values in the range block.

A new parallel implementation for fractal image compression for medical imaging using GPU platform was presented in [18]. The implementation launches a thread for each range block. Also, this model utilizes CUDA's texture fetching to load range and domain data.

In this research work, we propose a CUDA parallel implementation for one of the most commonly used classification schemes in fractal image compression (i.e. Fisher scheme). The validity and the performance of the proposed scheme have been empirically assessed using NVIDIA GeForce GT660 M GPU. To the best of our knowledge, this is the first attempt to provide CUDA implementation for this classification scheme. The rest of this paper is organized as follows. Section 2 provides basic preliminary information whilst Sect. 3 presents the proposed parallel scheme. The experimental evaluation of the scheme is outlined in Sect. 4. Section 5 concludes the paper and presents some future directions.

2 Preliminaries

The basic scheme of fractal image compression is to partition a given image into non-overlapping blocks, called range blocks and overlapped blocks of double size of range blocks

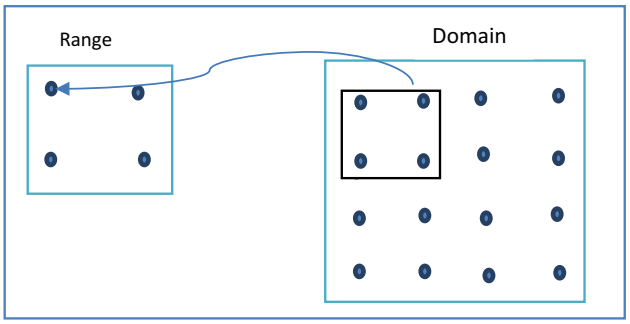


Fig. 1 Range vs. domain size

called domains [3]. The range is obtained from the quad-tree partitioning (a method to partition an image into quadrants) of the original image that needs to be compressed. The domains, on the other hand, are obtained from the domain library (also called domain pool). The first step is to reduce the domain size to that of the range size, since the domain is double the range size (in other words, the number of pixels in the given domain is four times the number of pixels that is in the range as shown in Fig. 1). After the averaging, for each range block, the scheme exhaustively searches the domain pool for a best-matched domain block using the root mean square (RMS) error function to determine how closely a transformed domain block is approximating a corresponding range block. If the RMS value is minimum (or not greater than the maximum threshold), the mapping (w) is saved. The mapping consists of the quantized value scaling (contrast s) and offset (brightness o) of the affine transformations (a group of transformations such as scaling, rotation, and brightness) [3] plus the quad-tree partition levels. Otherwise, a new domain is taken, and the same process is repeated again to compare it with the range. The following subsections provide some basic information about fractal image compression.

2.1 GPUs and CUDA

A CUDA compatible device is a set of multiprocessor cores, which are able to execute a large number of threads concurrently. Each multiprocessor has a single instruction multiple thread (SIMT) architecture (i.e. means that each processor of multiprocessor executes different threads, but all the threads run the same instructions) that operates on different data based on its thread ID, at any clock cycle [16]. A CUDA compatible device has its own memory (DRAM) which is divided into three different types: global memory, constant memory and texture memory. These memories can be read from or written to by the host CPU and they all are persistent throughout the life of the application. The multiprocessors have on-chip memories such as registers, shared memory, constant cache and texture cache. Since these memories are on chip, they are very fast as compared to off-chip memories such as DRAM (global memory). The registers (32-bit) are the fastest available but only support a limited amount of space (32–64 KB). Shared memory is limited to 48 KB and it is shared by all processors. A constant cache speeds up reads from the constant memory. Similarly, texture cache speeds up read from the texture memory [16]. Figure 2 illustrates the CUDA memory model.

The scalable programming model provides for the parallel execution of certain portion of an application on a device called Kernel. Kernel Function is executed on the device by many threads running on different processors of the multiprocessors. A thread block is a batch of threads which use shared memory to synchronize their execution. Each thread block executes on one multiprocessor and is limited only with 512 threads. A group of thread blocks of same size and dimensions execute the same kernel batched together into a grid of blocks. A grid consists of one or maybe two-dimensional array of blocks. A block is further one-, two- or three-dimensional array of threads as illustrated in Fig. 3. In CUDA, a thread is the basic unit of processing. Threads are organized into warps of 32 threads, and then executing all threads in a warp in parallel. CUDA includes C/C++ software development tools to help programmers to combine host code with the device code. Tto do that, CUDA programming requires a single program (i.e. kernel) written in C/C++ with some extension. CUDA *nvcc* compiler is used to compile the source code containing these extensions [19].

2.2 Range and domain blocks

A range block consists of fixed-size partitions of the image based on quad tree. A quad-tree partitioning is a

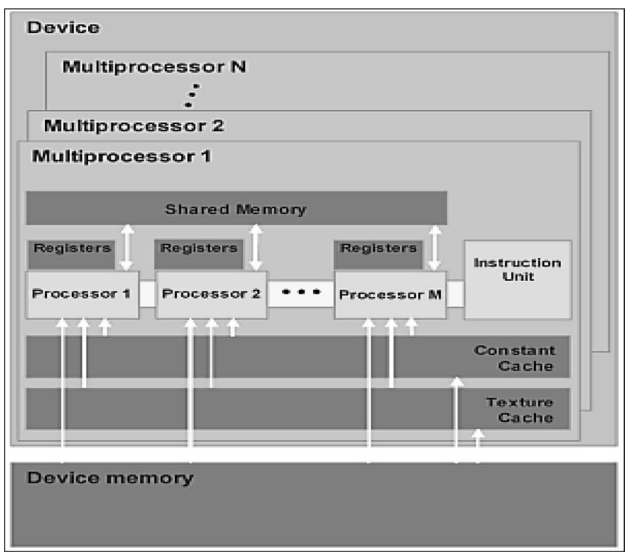


Fig. 2 CUDA Memory Model [16]

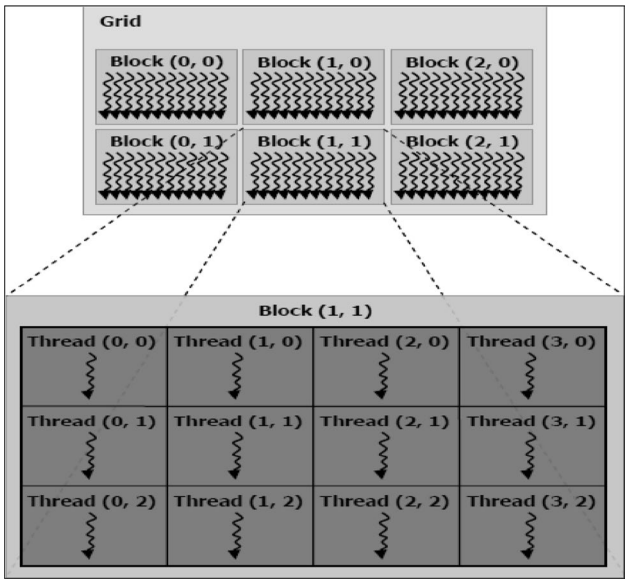


Fig. 3 CUDA thread execution model [16]

tree representation of an image in which each range node (square portion of the image) contains four sub-nodes (four quadrants of the square). The initial image is the root of the tree. In this method, maximum and minimum depth of the quad tree is set and a maximum allowable error to govern domain-range match needs to be set before partitioning. The next step is to partition an image into four square ranges of the same size until the maximum depth is met. An optimum matching domain block will be searched using an error function such as RMS for each range block on the level. RMS computed as RMS $= \sqrt{\min E(R,D)/n}$, where E is the distance between the range (R) and transformed domain (D), n is the number of pixels in the range (R). If such pair is found, the best matching range block with the domain block will be stored, and further partitioning is ceased. Otherwise, the quad tree is partitioned again, and the process is repeated to find the best match. This process continues till the maximum depth is met [20]. Figures 4 and 5 illustrate the image partitioning into ranges and domains, respectively.

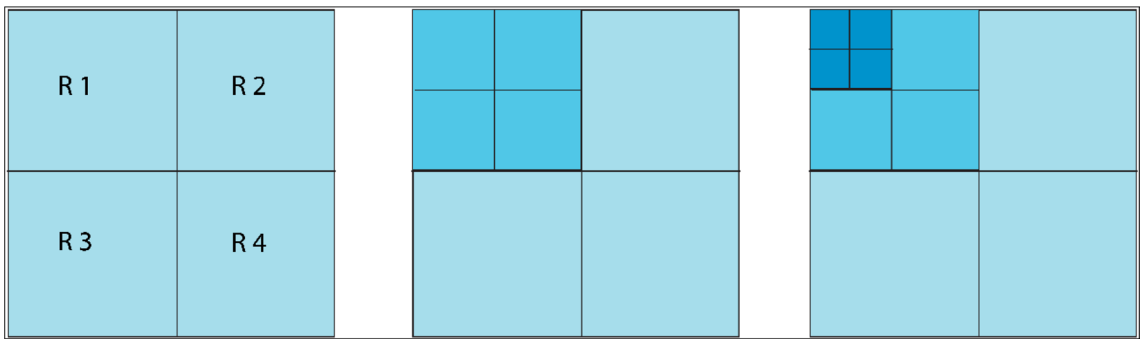
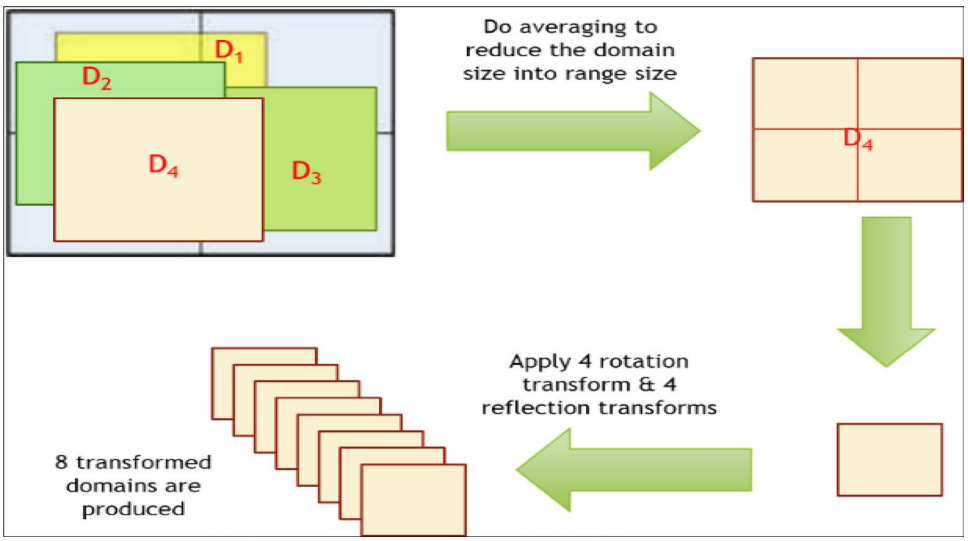


Fig. 4 Range partitioning

Fig. 5 Domain partitioning



The domain pool is crucial to an efficient representation. In the basic fractal image coding, there are a lot of domain blocks that represent the domain pool. As a result, searching through a large pool for every range block is a very time-consuming process. So, if the image size is 256×256, the number of ranges, if the range size is 4×4, will be $(256\times256)/(4\times4)=4096$. Whereas the numbers of domains which are twice the range size will be $(256-8+1)\times(256-8\times1)=62{,}001$ and 8 symmetric transformations need to be applied for each of these 62,001 domain blocks (i.e. 4 rotation directions and 4 mirror directions) resulting in a total of 496,008 domain blocks. Each of the 4096 range blocks needs to be compared with every transformed domain block. This is why most researchers have concentrated on making the search process faster in reducing complexity of computation [20].

2.3 Fisher classification scheme

There are several approaches proposed to reduce the time to search for domain-range match in FIC. The most popular approach is the classification scheme. Classification scheme is part of a pre-process executed before the search phase which aims to reduce the size of the domain pool and, hence, reducing the encoding time [3–5]. The Fisher classification scheme is the most widely used classification scheme. All the domains in the domain pool are classified before encoding phase and this will avoid reclassification of the domains later. A potential range block is classified during the encoding and only the domains with the same classification are compared with the range. According to Fisher [3], the classification of the blocks in the domain pool is done as following:

1. The sub-image (called Domain D) is divided into four quadrants.
2. For each quadrant, average of pixels is calculated separately (A_1, A_2, A_3, A_4).
3. Then the domain D is re-organized in three classes as shown below:

$$A_1 \geq A_2 \geq A_3 \geq A_4$$
$$A_1 \geq A_2 \geq A_4 \geq A_3$$
$$A_1 \geq A_4 \geq A_2 \geq A_3.$$

4. Each class can further be organized in 4! ways as described below, so all the classes will have $3\times24=72$ classes. Consider Class 1: $A_1 \geq A_2 \geq A_3 \geq A_4$, as one fixed domain state. For each quadrant i, also variance is calculated with the following formula: $V_i = \sum_{j=1}^{n} \left(r_j^i\right)^2 - A_i^2$, where n is the number of pixels in the quadrant and $V_i = \sum_{j=1}^{n} \left(r_j^i\right)^2 - A_i^2$. So for Class 1 domain, four calculated variances for each of its quadrants V_1, V_2, V_3, V_4 can be placed in four factorial ways. Meaning that, V_1 can be placed in any one of the four quadrants (i.e. V_2 can be placed in any one of the remaining three quadrants, V_3 in any one of the remaining two quadrants, V_4 in the last remaining quadrant). So $4\times3\times2\times1=24$ ways, from the 3 total classes, thus $3\times24=72$ different classes.

2.4 The sequential FIC algorithm

Figure 6 illustrates the steps of compression algorithm based on Fisher scheme. The basis for the encoding procedure as stated before is: an image is divided into parts that are resembled by other parts in the same image after some

- ❖ Read the image to be compressed
- ❖ Domain partitioning and initial classification:
  - ➢ Split the image into domains
    - ▪ Each domain block:
      - Averaged to be reduced to range size
      - Rotated and flips (8 new domains are produced)
  - ➢ domains with same size are classified
- ❖ Quad tree
  - ➢ Partition same image into four quadrants to the maximum depth recursively
- ❖ Range_Domain comparison
  - ▪ Pick up a range
  - ▪ Pick up the first domain from domain pool with same size
    - Calculate best RMS error from the domain:
      - ♦ If *RMS* meets tol^1 levels (tolerance value), output the results the file-system
    - If no domain matches, then we split the current range and proceed
    - If we cannot split further, then we pick up the best domain that we have found from the list of domains

Fig. 6 Fisher scheme for fractal image compression (sequential algorithm). [1]tol: is the tolerance level which is a loose target for the final RMS error of the encoded image. In this algorithm it is stated as 1

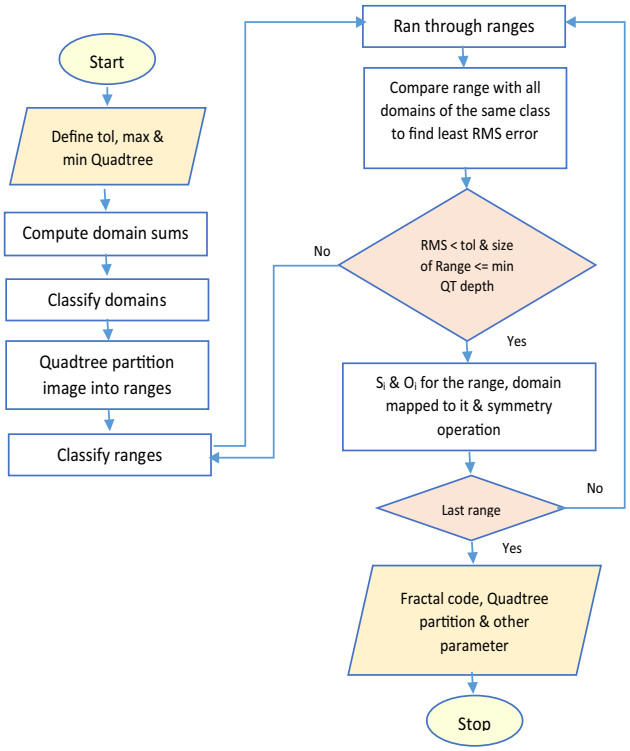


Fig. 7 Flow chart of Fisher scheme (sequential algorithm)

scaling. The result is a set of transformations which, when iterated from any initial image, have a fixed point of approximation (attractor). A flow chart of the sequential algorithm is illustrated in Fig. 7. RMS is an error function, determining how closely a transformed domain blocks approximate to a corresponding range block. Precisely, the error between two equal-sized collections of pixels is the sum of the errors between each corresponding pairs of pixels. Each domain block is compared against each range block and the error between the range block and the transformed domain block is calculated [3].

3 The CUDA parallel scheme

As mentioned earlier, the problem of fractal image coding is the large encoding time. In this section, we propose an approach towards parallelizing the encoding step to reduce the overall compression time. The basic Fractal Image Compression algorithm serves as a good candidate for parallelization because of the huge amount of computation involved in finding the best match between the domain block for each range block [17]. When complemented with Fisher classification scheme, the FIC algorithm also would require careful memory optimization. Figures 8 and 9 illustrate the proposed parallel compression algorithm based on Fisher classification scheme. The main difference between

- Read the image to be compressed
- Domain partitioning and initial classification:
  - Split the image into domains
    - Each domain block:
      - Averaged to be reduced to range size
      - Rotated and flips (8 new domains are produced)
  - domains with same size are classified
- Quad tree
  - Partition same image into four quadrants (ranges)
  - Quad Tree is linearized by formula $4n + 1$
- Launch kernels for all nodes of the quad tree at the same time
  - A reduction process is done
- Range Domain comparison
  - Pick up a range
  - Pick up the first domain from domain pool with same size
    - Calculate best *RMS* error from the domain:
      - If RMS meets *tol* levels, then output to memory buffer the quad transformations
      - Otherwise output to the memory buffer the fact that we are recursively going down
- Do a gather operation of the data that was output in the memory buffer recursively
  - Check what the parallel version outputted and write it in the file

Fig. 8 Fisher scheme for fractal image compression (parallel algorithm)

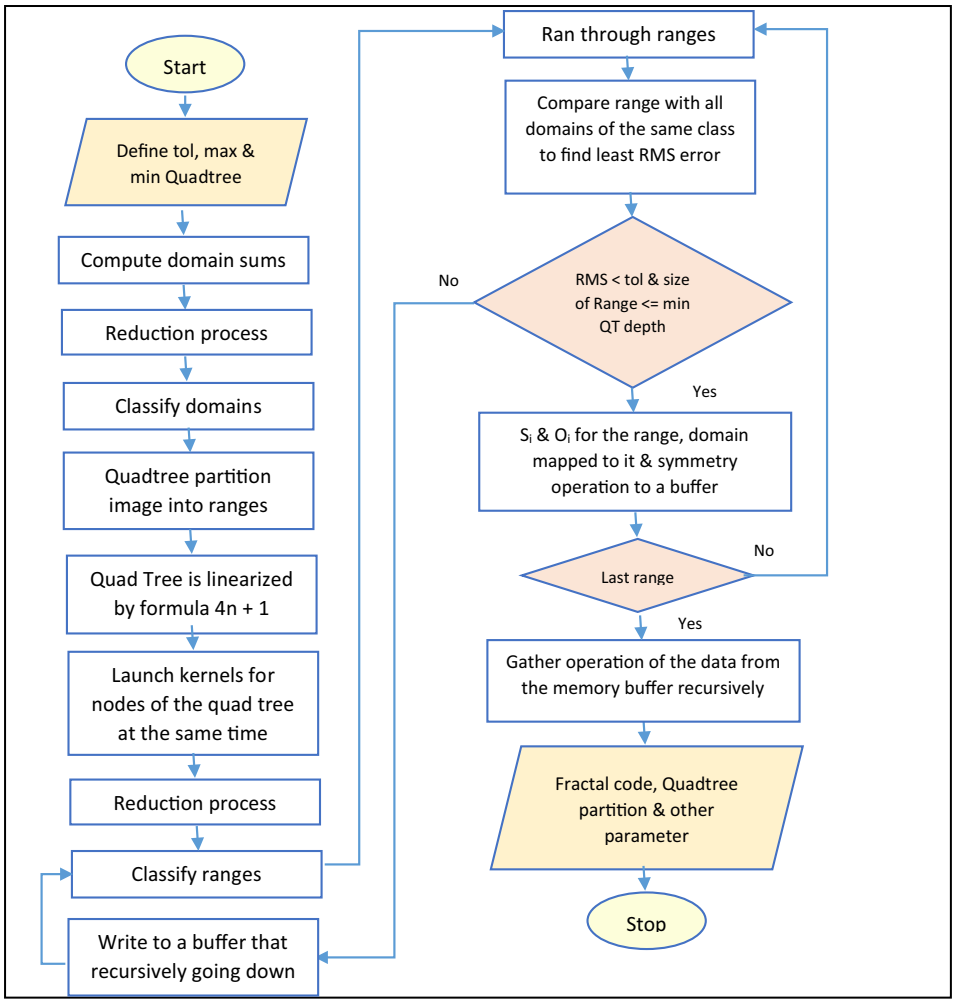


Fig. 9 Flow chart of proposed parallel algorithm

the sequential algorithm and the parallel algorithm is that in quad-tree traversal the decision to split is based on the RMS value. Whereas in parallel CUDA, all results are stored in the memory and these stored results are subsequently parsed and output as compression results to the file system. The algorithm uses tree-like structure, data-dependent instructions and divergent branches. These three components are explained in more details in the following subsections.

3.1 Mapping tree structure to parallel architecture

The quad tree is linearized by formula $(4n+1)$ due to the memory address mappings. For example, if a node has address i, then its four children have indices $4 \times i+1$, $4 \times i+2$, $4 \times i+3$, and $4 \times i+4$ as shown in Fig. 10. In this way, the tree is mapped to a linear buffer. When the quad-tree recursion is done in parallel, every node or leaf outputs into a linear buffer. Then the linear buffer with a recursion is gathered after the parallel process finishes. These buffers are runtime memory buffers which are initialized dynamically and are part of heap area of the runtime memory.

3.2 Using parallel reduction

Reduction is a well-known operation that has relevance in many engineering applications. It involves applying an operator on a range of values residing in a linear memory or vector and accumulating the final result in a single location that, in most cases, is the first element of the vector as described in Fig. 11. For the reduction to work with high performance on a GPU, the input vector size should be large [21]. In our proposed scheme, the parallel algorithm does the reduction operations when it iterates on pixels of an image and accumulates results. As shown in fig. 4.3, when a quad tree is traversed initially (level 0), the image size (node) is quite large 1024×1024. At the next level (level 1), quad tree operates on 4 images of size 512×512 and accumulates the sum of all pixels. On second level, 16 images 256×255, on third level 64 images of size 128×128 and so on. If kernels are launched on every level of the tree, then at the first level there will be 4 cores working on sizes of $512 \times 512 = 262{,}144$ and at the last level, will be 64 cores working on block of 16×16 each. Clearly, the quad tree is slow in the beginning

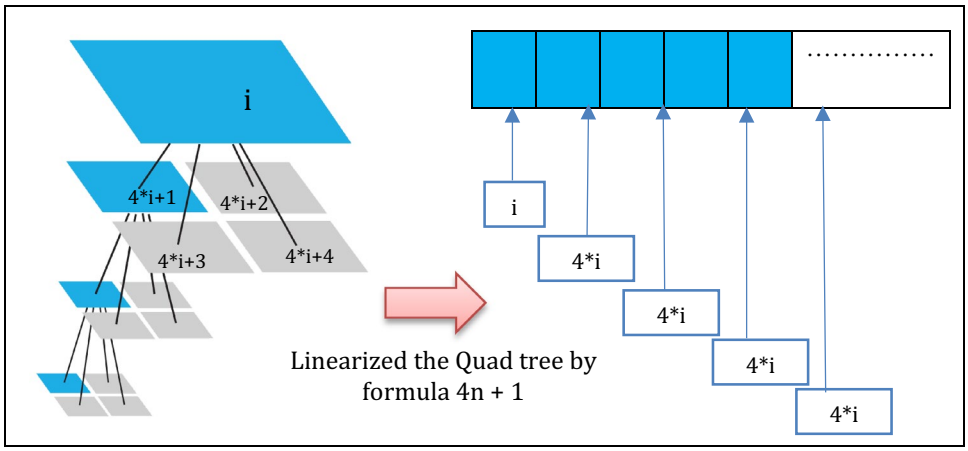


Fig. 10 Linear quad tree

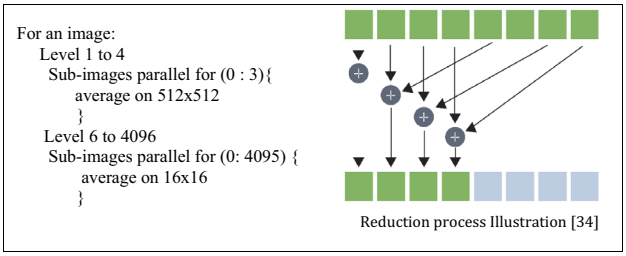


Fig. 11 Reduction process

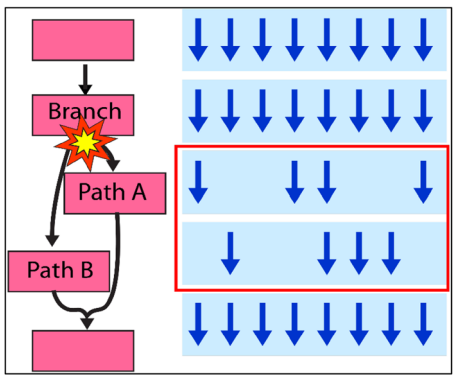


Fig. 12 Branch divergence [21]

in terms of cores utilized and it is irregular. To regularize it a bit, the following have to be done: at the lower levels of tree traversals, a reduction is done for accumulation. For example, spawn one kernel over a set of 512 × 512 pixels with many threads; it accumulates results in the shared memory (of the GPU device) and outputs the result in the main memory. At the higher levels, normal fork join model is done; spawn kernels over the small sets of size 16×16 and 32×32.

3.3 Removing divergent branches

The GPU architecture runs threads in groups of 32 threads, called warps, which are executed in a SIMD fashion [9]. All threads within a warp must execute the same instruction at any given time with different data sets. Branch divergence, as shown in Fig. 12, occurs when threads inside warps branches to different execution paths. This could lead to a situation where each branch had thousands of instructions for comparing domains with ranges in sequential implementation and, consequently, adversely affecting the performance. Branch divergence occurrences can be minimized by reducing long divergent paths to smaller ones [21]. In our implementation, this issue of thread divergence was considerably reduced by changing algebraic expressions through re-arrangement of the loops in parallel implementation. In the original code, there were seven loops and they lead to divergent paths and different steps, but in the modification, they are put as common structures.

4 Experimental evaluation

We have done few improvements to the implementation of the fractal image compression algorithm based on Fisher classification scheme. First, a class was developed to handle the loading of images from the file system. Free Image [22] library is chosen to load the images. Free Image is an open source library written in C++. This library uses a unique Free Image IO structure to load different types of images. Second, another class was developed to validate the loaded image. It makes sure that the loaded image is a square 8-bit gray scale image and of dimensions 2i, otherwise, an error message will be generated.

Once the validated image is supplied to the algorithm, two tasks are performed; the first one is the classification of domains for the picture and the second one is the quadtree partitioning. The classification is done by splitting the image into sub-images (e.g. an image of size 1024 is split

into sub images of sizes 1024×1024, 512×512, 256×256, 128×128, 64×64, 32×32, 16×16). In every split, we calculated its sum and squared sum. This will be used for calculating the averages and variance. As mentioned before, Fisher algorithm classifies the domains into 72 classes based on average and variance values [3].

The quad-tree decomposition is used to partition the image into ranges that are classified into classes. Each range is then compared with domains of the same class. If there is a match, the transformation will be output to the matching domain. Otherwise, the range will be partitioned into four quadrants. If the range size is 16×16, it will not be partitioned further, the closest domain will be considered as a match. The output from the compression phase is a compressed image consisting of the following:

- For each range, the corresponding mapped domain (only positions).
- Affine transformation values for each range.
- The final quad-tree partition of the image, each level is represented by 1 bit.
- The symmetry orientation operations for mapping each range to a domain.

The following points were considered before the algorithm is programmed in CUDA:

- Memory allocation is slow in CUDA, the *cuda Malloc Managed* function (same as Malloc in C/C++) costs around 2–3 ms [22]. This problem is fixed by allocating one big buffer at the beginning of the algorithm and subsequently allocating from it. Finally, it is freed at the end of the algorithm. Besides, fast memory allocation on the CPU side, significant amount of *cuda Memcpy* (CPU to GPU and vice versa expensive memory transfers) are avoided.
- Launching CUDA kernel is slow. In the implementation, this problem is fixed by launching as few kernels as possible. As mentioned before, the reduction is only done for lower level (larger ranges blocks).
- Normally, 2D data are fetched as arrays. GPUs have texture cache, which is separated from the L1 cache (were stack and shared memory goes). Tto utilize it, the 2D arrays should be seen as textures to read from and as surfaces to write on. Initial domain images obtained from domain partitioning of FIC algorithm, are scaled down in a kernel function implementation that averages 2×2 texture. It works with textures, not with c arrays of pixels. Using textures utilizes the L2 cache of the hardware for read-only memory. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance.

4.1 Hardware and software setup

The implementation of sequential algorithm has been programmed using C/C++ programming language, whereas the parallel algorithm has been programed using mix of C/C++ and CUDA programming languages. The programming has been done using Microsoft Visual Studio 2013 and CUDA Toolkit 6.5. The experiments were conducted in a stand-alone machine with the following specification:

- Operating System: Windows 7 Professional 64 bit.
- Processor: Intel(R) Core(TM) i5 Quad CPU (four processing cores).
- Memory: 6 GB RAM NVIDIA GeForce GT 660 M GPU with 960 cores.

The GPU configuration used for the experiments is shown in Table 1. The following images with different sizes are used for the experiments:

- Barbara and Lena: 512×512.
- Lamp_post, London and Satellite1: 1024×1024.
- River, Lake and Satellite2: 2048×2048.

Each one of these images is loaded to both sequential and parallel programmes. Three metrics were then used to assess the validity of the algorithm and its performance. These metrics are:

- The encoding time is the time taken to compress the image.
- The compression ratio (CR) is the ratio between the size of the source file and the size of the compressed file. This

Table 1 GPU configuration

CUDA driver version/runtime version	6.5/5.5
CUDA capability major/minor version number	3.0
Threads per warp	32
Max warps per multiprocessor	64
Max thread blocks per multiprocessor	16
Max threads per multiprocessor	2048
Maximum thread block size	1024
Registers per multiprocessor	65,536
Max registers per thread block	65,536
Max registers per thread	63
Shared memory per multiprocessor (bytes)	49,152
Max shared memory per block	49,152
Register allocation unit size	256
Register allocation granularity	Warp
Shared memory allocation unit size	256
Warp allocation granularity	4

evaluates the effectiveness of compression algorithm using file size. Lowering the compression ratio does not increase the reconstructed image quality significantly [3].

- Peak signal-to-noise ratio (PSNR) measures the difference between the original (without noise) and reconstructed image. This is most widely used to measure the quality/distortion for an image and it is calculated after the decoding phase [2]. PSNR measures the distortion of the image using the following equation: $\text{PSNR} = 10\log_{10}\left(\frac{256^2}{\text{MSE}}\right)$ MSE values between two images are calculated as: $\text{MSE} = \frac{1}{MN}\sum_{y=1}^{M}\sum_{x=1}^{N}\left[I(x,y) - I'(x,y)\right]^2$, where $I(x,y)$ is $I'(x,y)$. The original image, $I'(x,y)$ is the approximated version (which is actually the decoded image) and M, N are the dimensions of the images. It is worth mentioning at this stage that we have also implemented the FIC decoding algorithm. The decoding phase starts with an image (black image) of the same size as the original image, applies the transformations and finally an image close to the original image is produced. This is necessary to calculate PSNR.

4.2 Validation of the parallel algorithm

It is very important to validate the correctness of the proposed parallel implementation. Therefore, the compression ratio (CR) and peak signal-to-noise ratio (PSNR) were calculated for all test images for both sequential and parallel algorithms. Table 2 shows the results of the compression ratio for each test image from both sequential and parallel implementations. As it can be seen from table, the compression ratio of both sequential and parallel implementations for each image is the same. Similarly, Table 3 shows the PSNR results for each test image for both sequential and parallel implementations. The results of PSNR for both sequential and parallel implementations for each image are the same. The above table clearly demonstrates the validity of the parallel implementation as both metrics produce same results for both the parallel and the sequential implementations.

Table 2 Compression ratio results from sequential and parallel algorithm

Image	Image size	Sequential	Parallel
Lena	512×512	4.68	4.68
Barbara	512×512	4.68	4.68
Lamp_posts	1024×1024	18.79	18.79
London	1024×1024	20.41	20.41
Satellite1	1024×1024	18.89	18.89
River	2048×2048	17.58	17.58
Lake	2048×2048	18.18	18.18
Satellite2	2048×2048	18.16	18.16

Table 3 PSNR results from sequential and parallel algorithm

Image	Image size	Sequential	Parallel
Lena	512×512	39.6	39.6
Barbara	512×512	34.14	34.14
Lamp_posts	1024×1024	31.12	31.12
London	1024×1024	36.31	36.31
Satellite1	1024×1024	29.47	29.47
River	2048×2048	35.02	35.02
Lake	2048×2048	40.57	40.57
Satellite2	2048×2048	39.42	39.42

4.3 Encoding time

Encoding time is calculated for each test image for both sequential and parallel implementations. The experiments were carried out with the size of the range block of 16×16 as mentioned before. This is the optimal range block size as shown in [5], [21]. The following table shows the speedup for each test image. The speedup is calculated as:

$$\text{Speedup} = \text{sequential time(ms)}/\text{parallel time(ms)}.$$

The results clearly show that the performance gain for parallel image coding. The kernel utilized CUDA's texture fetching to load range and domain data that is cached. Therefore, a texture fetch costs one device memory read only on a cache miss; otherwise, it just costs one read from the texture cache. Also, using parallel reduction for larger range blocks help speeding up the process. Functions are reworked to do reduction for the lower levels of the quad tree (big sizes). This involved using shared memory to exchange data between threads within the same thread block which involves writing data to shared memory, synchronizing, and then reading the data back from shared memory. The experiments were done using Kepler having shuffle instruction (SHFL) which enables a thread to directly read a register from another thread in the same warp (32 threads) and that allows threads in a warp to collectively exchange or broadcast data. Batching kernel parameters in big buffers to save *cuda Memcpy* when launching kernels and removal of divergent branches in the compare function all help in speeding up the encoding time.

From Table 4 and Fig. 13, several tests were run of the same program with same images and the average results show that the speedup is increasing from 1.3× for 512×512 image size, to around 5× for 1024×1024 image size. For images of size 2048×2048, the algorithm achieves speedup up to 6.4× in some image like satellite images due to the

Table 4 Average of encoding time results from sequential and parallel algorithm

Image	Image size	Sequential	Parallel	Speedup
Lena	512×512	470.97	364.83	1.3×
Barbara	512×512	475.69	366.67	1.3×
Lamp_posts	1024×1024	1029.55	208.965	4.9×
London	1024×1024	1100.23	242.197	4.5×
Satellite1	1024×1024	891.093	180.059	4.9×
River	2048×2048	24,295.7	3950.68	6.1
Lake	2048×2048	22267.9	4001.63	5.6×
Satellite2	2048×2048	22948.3	3596.61	6.4×

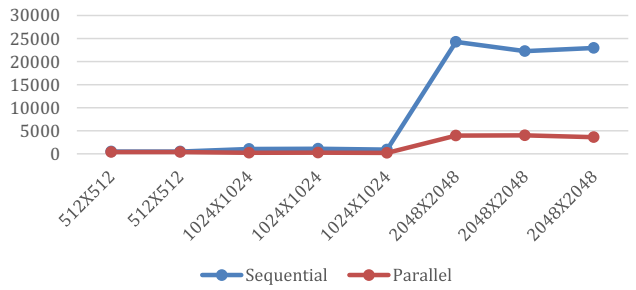


Fig. 13 Average of encoding time results from sequential and parallel algorithm

SIMD fashion where there should be huge amount of data needed to be done on the GPU.

4.4 GPU occupancy

GPU occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM [23] and it is calculated using CUDA Occupancy Calculator provided by NVidia [24]. In Figs. 14 and 15, we see that our program maximizes GPU occupancy without causing register spillage. This indicates that our kernel launches are at near-maximum efficiency.

Figure 14 shows how varying the block size whereas holding different parameters constant would influence the tenancy. The red triangle shows the current number of threads per block and the current upper limit of active warps. Note that the quantity of active warps is not the number of warps per block (that is threads per block divided by warp size, rounded up). Figure 15 shows how varying the record count whereas holding different parameters constant would influence the occupancy.

Figure 16 shows the achieved occupancy for every SM. The values reported are the average across all warp schedulers for the duration of the kernel execution.

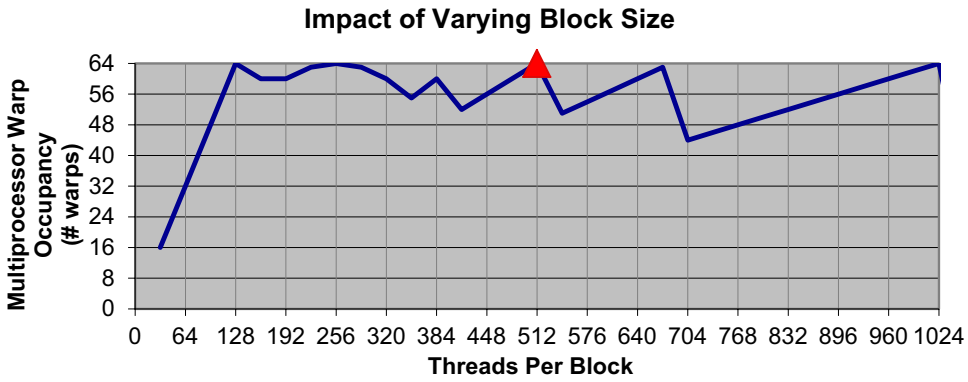


Fig. 14 Estimated occupancy as a function of threads per block

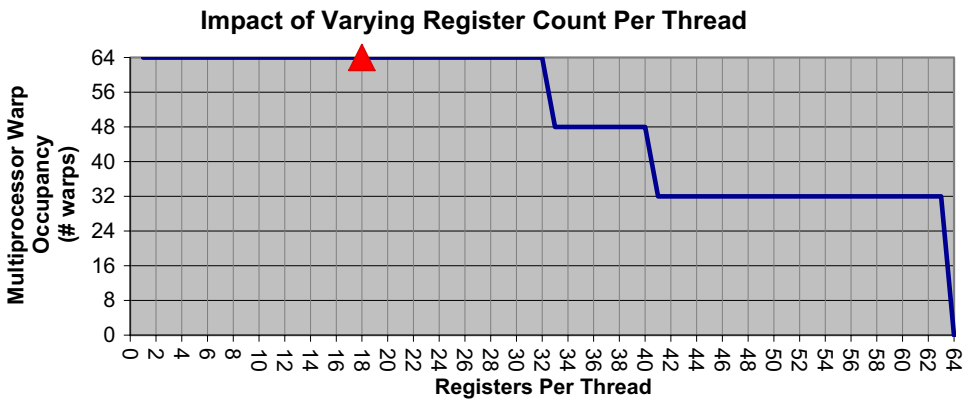


Fig. 15 Estimated occupancy as a function of registers per thread

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	16	64	4
Registers	(Warp limit per SM due to per-warp reg count)	16	84	5
Shared Memory (Bytes)		0	49152	16

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	**4**	**16**	**64**
Limited by Registers per Multiprocessor	5		
Limited by Shared Memory per Multiprocessor	16		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

Fig. 16 The CUDA occupancy calculator

5 Conclusions

In this research, we have proposed a new parallel implementation for fractal image compression based on Fisher classification scheme using CUDA. The implementation is based on three main components; linearizing the tree structure to fit the parallel architecture, getting read of the divergent branches and using parallel reduction to reduce the number of launched kernels. The performance of the proposed implementation has been evaluated on NVIDIA GeForce GT 660 M GPU using CUDA. Eight images with different sizes were used for the experiments with block of size 16×16 pixels. The compression ratio and peak signal to noise ratio results prove the validity of the parallel implementation. The encoding time for parallel implementation is better compared to the sequential implementation and speedups of up to 6.4 are recorded with some test images. The results also demonstrate that the parallel algorithm was able to achieve better performance when the image size is large due to the SIMD mode of execution.

The contribution of this chapter is to parallelize Fisher algorithms with six different images on one GPU. For future work, this parallel algorithm can be run on different GPUs and comparison can be done between the proposed one and the others. Also, the decoding process was implemented but the time was not considered in this research. It will be considered after running the same algorithm on different GPUs and then check the results and compare. Also, this implementation will be extended to exploit CUDA dynamic parallelism to launch kernels from the GPU, not only the CPU. This is expected to further decrease the encoding time.

Funding This work was financially supported by The Research Council/Sultanate of Oman, Grant ORG/ICT/10/003.

References

1. Sashikala, Y.M., Arunodhayan, S.S.: A survey of compression techniques. Int. J. Recent Technol. Eng. **2**(1), 152–156 (2013)
2. Liu, D., Jimack, P.K.: A survey of parallel algorithms for fractal image compression. J. Algorithms Comput. Technol. **1**, 171–186 (2007)
3. Fisher, Y.: Fractal Image Compression Theory and Application. Springer, Berlin (1995)
4. Wu, X., Jackson, D.J., Chen, H.-C.: A fast fractal image encoding method based on intelligent search of standard deviation. Comput. Electr. Eng. **31**(6), 402–421 (2005)
5. El-Khamy, S., Khedr, M., Al-Kabbany, A.: Efficient fractal image coding using adaptive domain pool reduction technique. In: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim) (2007)
6. Kaur, M., Kaur, G.: A survey of lossless and lossy image compression Techniques. IJARCSSE Int. J. Adv. Res. Comput. Sci. Softw. Eng. **3**(2), 323–326 (2013)
7. Jackson, T.B.: A parallel fractal image compression algorithm for hypercube multiprocessors. In: 27th Southern Symposium on System Theory (1995)
8. Kulkarni, M.V., Kulkarni, D.B.: Parallel computing using CUDA-GPU in fractal video coding introduction. In: Video & Image Processing, p. 2008, (2008)
9. Park, I.K.: Design and performance evaluation of image processing algorithms on GPUs. IEEE Trans. Parallel Distrib. Syst. **22**, 91–104 (2011)
10. Lee, S., Omachi, S., Aso, H.: A parallel architecture for quadtree-based fractal image coding. In: Proceedings of the International Conference on Parallel Processing, vol. 2000, pp. 15–22 (2000)
11. Zalan, B.: Maximal processor utilization in parallel quadtree-based fractal image compression on MIMD architectures, vol. XLIX, no. 2 (2004)
12. Thao, N.T.: Local search fractal image compression for fast integrated implementation. IEEE Int. Symp. Circuits Syst. ISCAS **2**, 1333–1336 (1997)
13. Hua Cao, X.-J.G.: OpenMP parallelization of jacquin fractal image encoding. In: International Conference on E-Product E-Service and E-Entertainment (ICEEE) (2010)
14. Hua Cao, X.-Q.G.: Implement research of fractal image encoding based on open MP parallelization model. In: Presented at the International Conference Electric Information and Control Engineering (ICEICE) (2011)

15. Wakatani, A.: Improvement of adaptive fractal image coding on GPUs. In: IEEE International Conference on Consumer Electronics (ICCE) (2012)
16. Wakatani, A.: Preliminary implementation of two parallel program for fractal image coding on GPUs. In: Presented at the IEEE International Conference Consumer Electronics (ICCE) (2011)
17. Khan., A.N.S.: Parallelization of fractal image compression over CUDA. In International Conference on Trends in Information, Telecommunication and Computing (2013)
18. Haque, Md.E., Al Kaisan, A., Saniat, M.R., Rahman, A.: GPU accelerated fractal image compression for medical imaging in parallel computing platform. Comput. Vis. Pattern Recognit. (2014). arXiv preprint arXiv:1404.0774
19. Bohong Liu, Y.Y.: An improved fractal image coding based on the quad tree. In: IEEE 3rd International Congress on Image and Signal Processing (2010)
20. Yu, H., Li, L., Liu, D., Zhai, H., Dong, X., Based on quadtree fractal image compression improved algorithm for research. In: 2010 International Conference on E-Product E-Service and E-Entertainment. IEEE, pp. 1–3 (2010)
21. Harris, M.: Optimizing Parallel Reduction in CUDA, NNVIDIA Developer Technology. NVIDIA Developer Technology. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. Accessed 03 Jan 2016
22. Media, S.: Freeimage. https://sourceforge.net/projects/. Accessed 03 Jan 2016
23. NVIDIA Corporation: Achieved Occupancy. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm
24. CUDA Occupancy Calculator. http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

Abir Al Sideiri Abir Al Sideiri is a Master in Computer Science at Sultan Qaboos University in Oman. She is a lecturer in Information Technology department in ALBuraimi University College, Oman. Her research interests are in the area of parallel computing on high performance computing, date-intensive computing, image analysis and Big data analysis.

Nasser Alzeidi Nasser Alzeidi is a PhD holder in Computer Science, University of Glasgow in UK; he is an Assistant Professor, Department of Computer Science, College of Science, Sultan Qaboos University. He is currently an Assistant Professor of Computer Science and the director of the Centre for Information Systems at Sultan Qaboos University, Oman. His research interests include performance evaluation of communication systems, wireless networks, interconnection networks, System on Chip architectures and parallel and distributed computing. He is a member of the IEEE.

Mayyada Al Hammoshi Mayyada Al Hammoshi is a PhD holder in Computer Science from Mosul University in Iraq; she is a Professor and a program chair at the School of Computer Information Systems at Virginia International University, VA, USA. Her research interests include cybersecurity, wireless networking, cryptography and distributed systems. She is an editorial board member in many international journals and conferences and an IEEE member.

Munesh Singh Chauhan Munesh Singh Chauhan has a PhD in Computer Science from Pacific Paher University, India with a research focus in GPU multicore computing. Currently, he is working as a Lecturer in Information Technology department at College of Applied Sciences, Salalah, Oman. His research interests involve leveraging parallel multicores to accelerate high compute intensive applications. He has worked on the application of parallel machines in deep learning, large graph analytics, flight route chartering for drones and fractal image compression.

Ghaliya AlFarsi Ghaliya AlFarsi is a Master in Computer Science at Sohar University in Oman. She is a Lecturer in Information Technology department in ALBuraimi University College, Oman. Most of her publications were indexed under Scopus. Her current research interests include e-learning, m-learning, technology adoption and acceptance, academic performance and image processing.