Toward Real Time Fractal Image Compression Using Graphics Hardware

Ugo Erra

ISISLab - Dipartimento di Informatica ed Appl. “R.M. Capocelli”,
Università degli Studi di Salerno, 84081 Baronissi, Italy
Email: `ugoerr@dia.unisa.it`

Abstract. In this paper, we present a parallel fractal image compression using the programmable graphics hardware. The main problem of fractal compression is the very high computing time needed to encode images. Our implementation exploits SIMD architecture and inherent parallelism of recently graphic boards to speed-up baseline approach of fractal encoding. The results we present are achieved on cheap and widely available graphics boards.

1 Introduction

Fractal compression is a lossy compression method introduced by Barnsley and Sloan [1] for compactly encoding images. The main idea of fractal compression is to exploit local self-similarity in images. This permits a self-referential description of image data to be yielded.

The general approach is firstly to subdivide the image using a fixed partition in simple case or adaptive partition in an advanced approach, and then to find the best matching image portion for each part. This searching phase is known to be the most time consuming part and numerous strategies have been presented to speed-up encoding. On the other hand, fractal image compression offers interesting features like fast decoding, independent-resolution and good image quality at low bit-rates which is useful for off-line applications.

Today’s GPUs (graphics processing units) have high-bandwidth memories and more floating-point units. One of the most recently presented GPUs, the NVIDIA 7800, has peak performance of 165 Gflops and memory bandwidth of 38.4 GB/sec. Recently, all this computational power has become cheap and widely available. As side effects, several researches has began to exploit GPUs for general purpose applications such as scientific computation, database operations, matrix multiplications and many more as shows in [2].

This paper presents a novel approach to perform fractal compression on programmable graphics hardware; to our knowledge, this is the first application that uses the GPU for image compression. Using programmable capabilities of the GPUs, we exploit the large amount of inherent parallelism and memory bandwidth to perform fast pairing search between portions of the image. As a result, we show that GPUs are effective co-processors for fractal compression.

2 Fractal compression

The basic idea of fractal compression is to find similarities between larger and smaller portions of an image. This is accomplished partitioning the original image into blocks of fixed size, called *range* and creating a shape codebook from the original image of double size of the range, called *domain.* Range blocks partition the image so that every pixel is included while the domain blocks can be overlapped and/or to not contain every pixel. We give below the baseline approch, the mathematical theory about these principles can be found on [3].

Given a range block R we must find a domain D from codebook such that $R \approx sD + o\mathbf{1}$ where s and o are called *scaling* and *offset* respectively. These values define the optimal transformation by which we can encode an image portion using another part. The encoder must scan all the codebook to find optimal D, s, and o. The domain block must be shrunk by pixel averaging to match the size of range block.

Given the two blocks R and D with n pixel intensities, $r_1, \ldots, r_n$ and $d_1, \ldots, d_n$, the quantity to minimize is $\sum_{i=1}^{n} (s \cdot d_i + o - r_i)^2$ where coefficients s and o are given by

$$s = \frac{n\left(\sum_{i=1}^{n} d_i r_i\right) - \left(\sum_{i=1}^{n} d_i\right)\left(\sum_{i=1}^{n} r_i\right)}{n\sum_{i=1}^{n} d_i^2 - \left(\sum_{i=1}^{n} d_i\right)^2} \qquad o = \frac{1}{n}\left(\sum_{i=1}^{n} r_i - s\sum_{i=1}^{n} d_i\right) \tag{1}$$

The values s, o, and the position of domain block D are the encoded values for range R. The steps of the baseline encoder with fixed block are the following:

1. *Range blocks R_i.* Given a fixed size (4×4, 8×8, and so on) create a set of range blocks overlapping the entire image.
2. *Shape codebook D_i.* The shape codebook is created in two steps:
   (a) Using a step size of l pixel horizontally and vertically create a set of domain blocks which are double the range size.
   (b) Shrink the domain blocks by averaging four pixel to match range size.
3. *The search.* For each range block R an optimal approximation $R \approx sD + o\mathbf{1}$ is computed in the following steps:
   (a) For each domain block D_i compute $R \approx sD_i + o\mathbf{1}$ using formulas (1).
   (b) Among all codebook D_i output the code for current range $[k, s, o]$ such that the error $R \approx sD_k + o\mathbf{1}$ is minimum.

Related works. Fractal compression allows fast decompression but has long encoding times. The most time consuming part is the domain blocks searching from each range. In [4] Beaumont adopts a search strategy using an outward spiral starting from the coordinate of range and halts when a necessary condition has been reached. This strategy reduces encoding time but image quality could suffer due to the overlook of some possible optimal pairing. Categorized search proposed by Boss, Fisher and Jacobs [5, 6] and features vector methods proposed by Saupe [7] are efficient classification techniques. They reduce the encoding complexity using a classification of the domain codebook block in such way that for each range the search is essentially more efficient.

The use of general purpose high performance architecture has been used to accelerate the encoding phase without a decrease of image quality. Related work has been done concerning the encoding phase on SIMD architecture. In [8] massively parallel processing approach has been used on an APE100/Quadrics SIMD machine. For testing, they used 512 floating point processors, offering a peak power of 25.6 GFLOPS. They are able to compress a gray level image of 512×512 using a scalar quantization techniques in about 2 seconds.

3 Programmable graphics hardware

Today, GPUs are fundamentally programmable stream processors [9]. In this computational model stream are collections of data requiring similar computation. Every object in the stream is processed by the some function called kernel. GPUs has a screen-space stream engine called fragment processor. The fragment processor supports a fully orthogonal instruction set optimized for 4-component vector processing. Furthermore, as stream architecture, the fragment processor exploits spatial parallelism; it runs the same fragment program for each pixel.

This processor presents limits and advantages. For each incoming pixel the fragment program is invoked at a specific location and returns the final value in the same location as output. That is not possible to write in a different location or to do scattering. Instead the kernel can do gathering using textures as lookup table to read precomputed values.

The textures can be used as lookup tables during computation though access to them is restricted to write-only or read-only. Floating-point texture vectors can be of two, three of four components. Each fragment can fetch a component vector as input from one or more textures and returns a vector components. This feature and the fact that fragment processor has enormous throughput make fragment engine well suited to fractal compression.

4 Mapping fractal compression on the GPU

In order to exploit the specialized nature of the GPU and its restricted programming model we must map the fractal compression as a streaming computation. The goal is to perform pairings test between range and domain exploiting parallel architecture of the GPU and high bandwidth access to pixels. The entire process uses a gray level image as input data and returns the textures T_{POS} with the position of optimal domain blocks and T_{SO} with scaling/offset coefficients as outputs.

The underlying idea is to use a producer/consumer scheme. The producer gathers from the domain pool a block which is broadcasted to all consumers that are the ranges. Each range stores the current domain as soon as it appears as the best pairing block. The entire process continues until all domain blocks have been consumed. In this scenario, a pixel appears as a single floating-point processor responsible for only one range. Then, the GPU mimics a computational grid rendering a sized-range rectangular upon which performs parallel pairing test among all the ranges for a given domain.

4.1 Data Structures Organization

Fractal compression is implemented as fragments programs. These programs are executed via multi-pass rendering of a screen-sized rectangle where each pixel is an encoding range. Notice that during encoding the same range will be paired among all other domains and from another point of view the same domain will be paired among all ranges. Then, for each range block and for each domain block we precompute all the related quantities that remain constant during the entire encoding. These constant values are the summations of the scaling and offset formulas in 1 and will be stored in the lookup textures T_R and T_D using one 32-bit component for T_R and two 16-bit component for T_D.

In order to exploit SIMD parallelism and efficient bandwidth, during the encoding, the source image is stored compactly into texture. An RGBA texture which uses 32-bit per component is capable to store up to 256 bit per pixel. Usually, for a gray level image, 8-bit per pixel are necessary. Thanks to the specialized instruction `pack` we are able to store up to 16 pixels into a single RGBA pixel's texture. Using this representation of the image it is possible to read 16 pixels simultaneously in a single texture access followed by a `unpack` instruction.

4.2 Fractal compression kernels

The entire flow diagram for the streaming fractal compression is illustrated in Figure 1. The following sections detail the implementation of each kernel and the read/write textures access. In the following, the kernels always take as input the compact version of source image.

Ranges summation. This kernel precomputes the range summation using the technique described in the previous section. It takes the original image as input and returns a texture T_R as output. This buffer is a previously declared one 32-bit color component texture. The size of this buffer is 1/4 of the input image if we choose a range block size of 4×4, or 1/8 for a range block of 8×8, and so on.

Domains summation. The kernel precomputes the domains summations. It takes the original image as input and returns the texture T_D as output. This buffer is a previously declared two 16-bit color components texture.

Stream range generator. This is the only party performed in the CPU and it is not a computational stage. It serves as a start-up routine to generate a stream of fragment programs. It draws a texture of size T_R to force the "Pairing Test" fragment program execution for each range. Furthermore, it passes to the next stage the position of the current domain and the entire group of pixels belongs to the current domain as parameters. This permits to store an entire domain into registers of fragment processor avoiding continuous texture fetches.

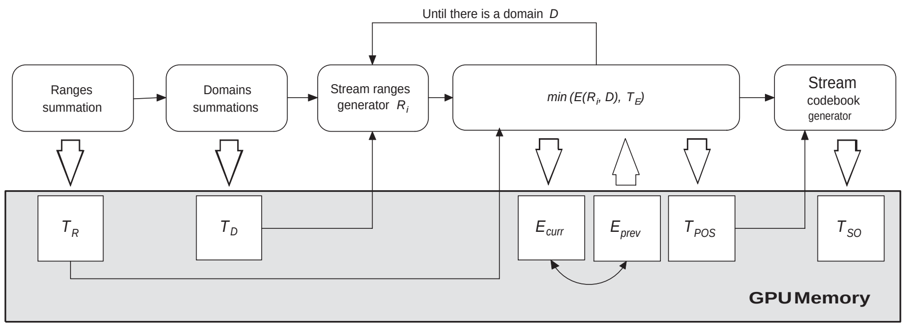

Fig. 1. A streaming fractal compressor.

Pairing test. This kernel performs all possible tests for optimal pairing. At each rendering pass this kernel has T_R as input textures, coordinate of current domain and pixels' domain from the previous stage. It returns a sized range texture T_{POS} with the coordinates of current optimal blocks as output. More precisely, due to hardware limits which not allow read/write access on the same texture, pairing tests are not performed for a given range R among all domains D_i. We must invert pairing tests as follows: given a domain D performs pairing tests among all ranges R_i updating minimal errors for each range. This schema use a double buffer to read previous errors in texture T_{prev} and to store current errors in texture T_{curr}. Thus, given a domain block D at each rendering pass each fragment program computes and stores the following value: $T_{curr} = min\left(E\left(R_i, D\right), T_{prev}\right)$. Before the next rendering pass the errors textures T_{curr} and T_{prev} are swapped and another domain block is passed as input.

Stream codebook generator. The last stage computes and writes into texture T_{SO} final scaling s and offset o. The kernel has textures T_{POS} as input and returns a texture T_{SO} with optimal coefficients. This operation is performed here in order to avoid useless write operations and therefore optimizes the amount of memory bandwidth required for texture accesses.

5 Experimental Results

In order to compare the amount of pairing tests that GPU is capable to perform, we implemented heavy brute force algorithm on GPU as well as on CPU. We have experimented on a Pentium IV based machine with 3.2GHz processor speed and 1GB of RAM. The used graphics card was a GeForce FX 6800, with 128MB of video memory, core speed of 300MHz and memory speed of 800MHz. We used OpenGL Cg shading language to implement our GPU-based compressor.

The test image has a resolution of 256×256 pixels. Choosing a range size of 4×4 pixels we obtain 64×64 ranges. The domain blocks must be twice the size of range blocks and using a step of one pixel to scan the image, the domain pool contains $(128 - 4 + 1)^2 = 15,625$ elements. In total, using a heavy brute force strategy $4,096 \times 15,625 = 64,000,000$ possible pairings require testing.

The CPU version takes about 280 seconds to perform all pairing test whereas the GPU version takes about 1 second. Then, the amount of paring test that the GPU is capable to perform is about 64 millions per second whereas the CPU performs about 220 thousands paring test per second. These results arise considering fractal compression a random-memory-access intensive problem. The memory bandwidth together arithmetic intensity advantages GPU over CPU. Moreover, our work shows its advantages when compared to expensive parallel architecture as for instance in [8] which uses 512 floating-point processors with performance comparable to our GPU implementation.

6 Conclusions and further work

Fractal image compression is well suited for parallel system due to its high computation complexity and regular algorithmic structure. Today, we think that graphics board offers substantial computational power to take full advantages of the baseline approach. We are going to investigate two scenarios. The former is to further exploit the graphics hardware to obtain more efficient compression schema and taking into account also color image. The latter is to use GPU as an efficient co-processor. The general purpose architecture of the CPU permits to excel on arranging data. The GPU could be used as an efficient pairing engine while the CPU arranges pairings tests.

Today, the consumers for less than $500 can buy an off-the-shelf graphics card with performances comparable to SIMD parallel machines that cost hundreds of thousands of dollars several years ago. Furthermore, as the GPU consumes less power than a high-end CPU, it is evident how using the graphics card can extend the life-time of an existing computer system. In conclusion, we think that GPUs offer the great opportunity to take full advantages of the fractal compression on consumer desktop personal computers.

References

1. Barnsley, M.F., Sloan, A.: Chaotic compression. Computer Graphics World (1987)
2. GPGPU. (Website) http://www.gpgpu.com.
3. Yuval, F.: Fractal Image Compression - Theory and Application. Springer-Verlag, New York (1994)
4. Beaumont, J.M.: Image data compression using fractal techniques. British Telecom Technol. Journal **9** (1991) 93–109
5. Jacobs, E.W., Fisher, Y., Boss, R.D.: Image compression: A study of the iterated transform method. Signal Processing **29** (1992) 251–263
6. Yuval, F.: Fractal image compression. Fractals: Complex Geometry, Patterns, and Scaling in Nature and Society **2** (1994) 347–361
7. Saupe, D.: Accelerating fractal image compression by multi-dimensional nearest neighbor search. In Storer, J.A., Cohn, M., eds.: Proceedings DCC'95 Data Compression Conference, IEEE Computer Society Press (1995)
8. Palazzari, P., Coli, M., Lulli, G.: Massively parallel processing approach to fractal image compression with near-optimal coefficient quantization. J. Syst. Archit. **45** (1999) 765–779
9. Venkatasubramanian, S.: The graphics card as a stream computer. In: SIGMOD-DIMACS Workshop on Management and Processing of Data Streams. (2003)