Parallel fractal compression for medical imaging

Jacob Toft Pedersen 19983275 jtp@cs.au.dk

Abstract—Fractal compression has high compression rates and a dimensionless compression scheme. Fractal compression has been deemed unfeasible as the compression requires intensive computation. This paper will examine the quality of fractal interpolation and introduce a parallel computation to show that the compression coefficients may be computed in reasonable time on a modern GPU.

Index Terms—parallel computation, GPU, medical imaging, fractal compression, fractal interpolation.

1 Introduction

Fractal compression is a lossy and dimensionless compression scheme. The compressed image may be decompressed at any desired resolution. When an image is decompressed at a higher resolution than the original, it is referred to as fractal interpolation.

The foundation of fractal compression is Iterated Function Systems (IFS). The compressed image is described by the affine transformations which has the original image as their attractor.

Fractal compression is an asymmetric compression scheme. Compressing is computational very hard, but decompressing is fast. The prohibitly expensive compression is a main reason for the limited success of fractal compressions.

In this article I will examine whether a parallel implementation of the compression scheme will offer a significant speedup.

1.1 Background

In this section the mathematical foundation of fractal compression will be presented, as well as the nomenclature used. The method for doing fractal compression is based on course notes from a SIGGRAPH workshop [1], and is inspired by Wocjcieh Walczak' master thesis [2]. The goal of this section is to give the reader an introduction to the terms and methods used. The reader is referred to [1] for further details.

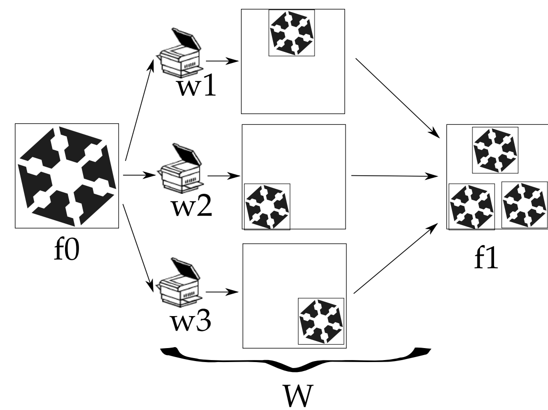

Fig. 1. How does IFS work

1.1.1 Iterated function Systems

IFS's consists of a collection W (2) of affine transformations $w_1, \dots, w_n$ (1). When W is applied to an input image f_0, the result is f_1. The composite operator $W^{\circ n}$ denotes the nth application (3). The process can be visualized by a collection of Xerox-machines, as illustrated in figure 1.

$$w_i \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \end{bmatrix} \quad (1)$$

$$W(\cdot) = \bigcup_{i=1}^{n} w_i(\cdot) \quad (2)$$

$$f_2 = W^{\circ 2}(f_0) = W(W(f_0)) \quad (3)$$

$$|W| \equiv f_\infty = \lim_{n \to \infty} W^{\circ n}(f_0) \quad (4)$$

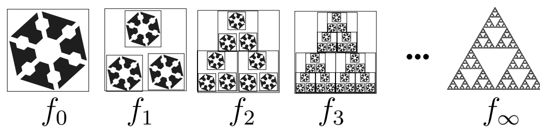

Fig. 2. The attractor $|W|$ of an IFS

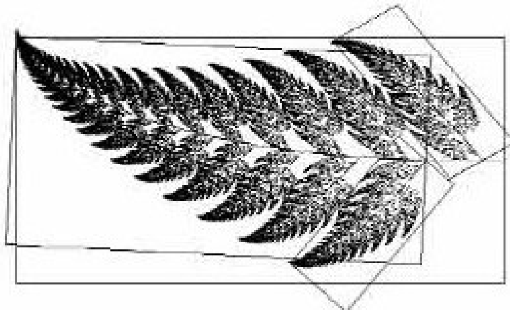

Fig. 3. Barnsleys fern (4 transformations)

An IFS can have an attractor $|W|$ (4). Figure 2 illustrates this, where the attractor is the Sierpinski triangle. The interesting observations that can be made are : f_∞ does not depend on f_0. Thus the Sierpinski triangle can be stored in infinite resolution with only 3 affine transformations. As figure 3 with *Barnslesy fern* shows, IFS's are not limited to abstract images.

The lure of fractal compression is this theoretical infinite compression rate. In practice there are limitations to this compression scheme.

First and foremost an IFS will not work for a normal photograph as the attractor is based on self similarity between the entire image and is parts. A photograph rarely has this property. Figure 4 illustrates that most images possess self similarity between parts of the image. Intuitively this most especially hold for medical images, where the branching in lungs or blood vessels exhibit a high degree of self similarity across scale and are considered as naturally occurring fractals (figure 5).

1.1.2 Partitioned Iterated function systems

A Partitioned IFS (PIFS) is an IFS were the transformations w_i are restricted to operate on specific subsets of the input, domain blocks. Applying a transformation w_i to a domain block results in a range block. (Figure 6)

In order to fractally compress an image, we have to identify the self similarity in the input

Fig. 4. Self similarity

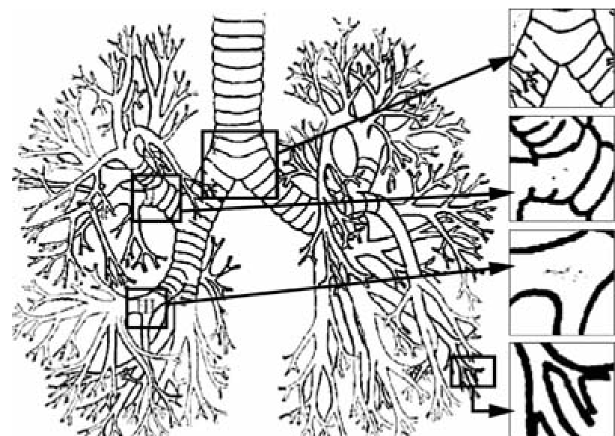

Fig. 5. Self similarity in the lung system

image, in such a way that we can express the image as a set of transformations W such that $|W|$ is our input image.

This is done by dividing the image into range blocks, and for each range block find a domain and a transformation, that maps the domain blocks onto the range blocks.

As we want to compress images and not just sets of points, we have to express the transformation of brightness as well. If we consider the brightness of a pixel as a function of its position $z = p(x, y)$ we can scale and offset this value with s_i and o_i respectively as part of the transformation w_i(5).

Thus a PIFS consists of transformations mapping domain blocks into range blocks, with

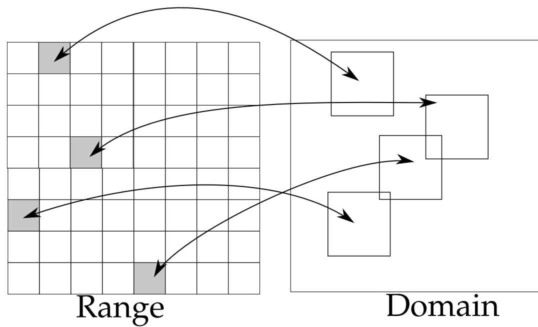

Fig. 6. Relation between range and domain

respect to position and pixel brightness.

$$w_i \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_i \\ f_i \\ o_i \end{bmatrix} \quad (5)$$

1.1.3 *Collage theorem*

An important theorem, which makes the search for w_i feasible, is the collage theorem. It states that given a IFS W with a contractivity factor s, and an image f, then

$$h(f, W(f)) \leq \epsilon \implies h(f, A) \leq \frac{\epsilon}{1-s} \quad (6)$$

where A is the attractor of the IFS and h is the Hausdorff metric. In layman's terms: if we have a set of transformations that approximates our desired attractor (the input image), then applying this transformation again will bring us even closer to the attractor[1].

In terms of searching for self similarity within an image, this limits the search to finding the best transformations of a domain block to a range block.

2 Implementation

In this section I will present the basic outline of a generic algorithm and show how this can be implemented on the GPU utilizing CUDA. Furthermore a CPU implementation and the process of decompressing the image will be discussed.

1. https://secure.wikimedia.org/wikipedia/en/wiki/Collage_theorem

2.1 Basic algorithm

There is an infinite number of possible transformations to examine. In order to limit this to a reasonable number, a set of restrictions on the type of transformations and the domains to considered has to be chosen.

The basic method consists of dividing the input image into disjunct square range blocks $\mathbf{R}$ ($n \times n$) and overlapping domain blocks $\mathbf{D}$ ($2n \times 2n$), and only consider a limited set of affine transformations.

The set of transformations to be considered are referred to as *symmetry operations* S_i. The symmetry operations consists of simple affine transformations with 90 degrees rotations and mirroring. These transformations has the property that they are easy to calculate and when they operate on squares and the result is a square. (see appendix A for the 8 operations used)

To map a domain, which is twice the size of a range block, onto a range block, it it necessary to have contracting transformation τ. As the relation between ranges and domains is fixed then τ is fixed. The reason for choosing domains with twice the width of the ranges is that τ can be implemented as a average of 4 pixels, corresponds to setting $a_i = d_i = .5$ in w_i.

For each domain (d_j) several *code blocks* are generated. A code block is a region with the same size as the range where a symmetry operation has been applied $S_j\tau(d_i)$. The set of all code blocks is the *codebook*.

2.1.1 *Generating the codebook*

The size of the codebook is a determining factor in the computational requirement of the algorithm much previous work has been put into reducing its size, whilst still getting a good compression [3]. However many of these techniques do not lend themselves to parallelization. It is not necessary to precalculate the codebook it can be done *on the fly*. The codebook used in this implementation are generated on the fly.

In general not all domains of the desired size are considered, it is common to restrict the search to a grid. The spacing in this grid is referred to as `D_STEP`. A algorithmic overview is provide in figure 7.

Fig. 7. Generating the code book

```
for x = 0 to image width step: D_STEP do
    for y = 0 to image width step: D_STEP do
        c ← [x, x + 2n][y, y + 2n]
        for i = 1 to 8 do
            add S_i(τ(c)) to the codebook
        end for
    end for
end for
```

Fig. 8. The main loop

```
for all d ∈ D do
    for all b ∈ R do
        for i = 1 to 8 do
            a ← S_i(τ(d))
            {s, o} ← solveLeastSquares(a, b)
            R = calculateResiduals(a, b, s, o)
            if R < r_j.R then
                r_j.{s, o} ← {s, o}
                r_j.domain ← d
                r_j.S ← i
            end if
        end for
    end for
end for
```

2.1.2 Least Squares

By utilizing the least squares method to compare a code block and a range block we get a metric for *a good match*, and we are able to calculate s and o, the scaling and offset of the pixel brightness.

The metric will be the residuals, if a code block a is to be transformed into range block b, we must minimize (7). The SIGGRAPH course notes [1] has a typo which was a source for some frustrations. In appendix E there are the correct formulas.

$$R = \sum_{i=1}^{n} (s \cdot a_i + o \cdot -b_i)^2 \qquad (7)$$

Shadow blocks is a range block where all pixels have the same color, their variance is zero. it is not necessary to compare them with a code block. It is sufficient to store their brightness. By the same measure we do not need to include code blocks with a zero variance.

2.1.3 The main loop

The main part of the algorithm is a brute force search for matches between code blocks and range blocks, (Figure 8).

Fig. 9. Simple parallel method

```
b ← r_thread_index
for all d ∈ D do
    for i = 1 to 8 do
        a ← S_i(τ(d))
        {s, o} ← solveLeastSquares(a, b)
        R = calculateResiduals(a, b, s, o)
        if R < b.R then
            b.{s, o} ← {s, o}
            b.domain ← d
            b.S ← i
        end if
    end for
end for
```

This is a $O(n^2)$ algorithm with a huge potential for parallelization.

After the algorithm has been run, the coefficients for the transformations must be stored. It is not necessary to store the full transformation matrix, τ is fixed and the translations can be deduced by the order of the range blocks. It is sufficient to store the domain block d associated with the transformation and which symmetry operation was used.

To examine the algorithm and get a feel of the expected run time, a rough CPU version were implemented. This was necessary as I was not able to locate any previous implementation[2]

At a first glance this revealed that compressing images $> 512 \times 512$, with reasonable compression would take more than 30 minutes.

2.2 Parallelization

The key observation when parallelizing the algorithm is that launching a thread for each range block will avoid write conflicts.

This leads to a simple parallel version, figure 9

A lossy and badly compressed image taking 1 minute on the CPU, would take .5 second with this naive kernel.[3]

The kernel utilizes CUDA's texture fetching to load range and domain data, it was optimized further as discussed in section 2.5.

2. I was not able to find any that I could compile with a recent compiler, without resorting to rewriting major parts of the code.

3. This does not constitute a fair comparison, but clearly illustrates that their is a huge speedup

2.3 Decompression

It was necessary to implement custom methods to decompress the images. It was a matter of iteratively applying the stored transformations. Further notes and source code are in appendix F.

2.4 Increasing compression quality

The basic method can compress images with a reasonable quality and the parallelized version can do it in reasonable time. However as the goal was to do fractal interpolation, it was not a satisfactory quality and the the output was crippled by blocky artifacts.

A proposed simple solution is to create ranges that overlap [1]. This was implemented by still using square ranges, however overlapping such that each pixel would get a contribution from 4 transformations. This introduced complexity in relation to handling borders. To remedy this the method implemented considers the input as having toroidal topology, in other words addresses are wrapped.

In section 3 there are experimental results comparing the quality of both methods.

2.5 Optimizing the kernels

Naively created kernels for the CUDA framework might boost performance compared to a CPU implementation, but to fully realize the potential of the GPU the code must be tailored to fit the hardware.

There are implementations for the two methods *simple* and *quad*[4], optimization efforts has focused on the quad version.

In order to focus on the algorithms and not calculating indexes, all input images are assumed to be square and have dimensions 2^i.

2.6 Setup

CPU	AMD64-3700 2GB ram
GXX	version : `4.4.5`
GPU	460GTX, 336 cores, 7 multiprocessors
NVCC	`release 3.2, V0.2.1221`
OS	Ubuntu Linux 32 bit

4. quad as it quadruples the number of ranges

2.7 Kernels

In the simple method only range block sums are precalculated and in quad version all possible sums are precalculated. The source code is in appendix G.

The two methods may superficial look very similar, however they use different indexing schemes and texture settings.

Two essential helper `_device_` functions are `get_pixel()` and `set_indices()`. They correspond to τ and S_i respectively.

`get_pixel()` uses textures and returns the average of 4 neighboring pixels. In the simple version this is done addressing in the range $[0, \text{img.w}[$ with a 2D texture. Care has to be taken when setting up textures. Somehow the initial setup resulted in *off-by-one* errors, after several rewrites of the kernels index calculations it was discovered that the texture setup was the guilty party.

The quad version uses address wrapping and initially this was implemented using a coordinate wrap function compiled to both host and device. CUDA has hardware support for address wrapping but this requires normalized addressing (in the range $[0, 1[$). Rewriting the kernel to use normalized addressing and utilizing the hardware gave a tremendous speedup of more than a factor 3.

`set_indices()` is written naively using a `switch statement` and pointer manipulation, it can be observed that when enabling optimizations the compiler are able to optimize the inlined functions. This can account for a factor two speed up.

The launch setup with grid and block dimensions is important to getting full utilization of the GPU. The kernels for the two methods however uses more than 22 registers[5], and this becomes the limiting factor in block size. In general a 16 by 16 block layout was used and it gave a perfectly adequate utilization of at least 60%.

Appendix C.1 contains some notes on how further optimizations we tested.

5. The precise number of registers used is highly dependent on the size of ranges, and other constant factors

2.8 Batches

When utilizing CUDA without a dedicated GPU there is a hard limit on the time a kernel may execute. The computations for these methods takes considerable time, this makes it impossible to encode images larger than 256×256 with high quality settings.

A solution is to divide the computation into batches. This was implemented for the quad method. At first the number of required comparisons between range blocks and domain blocks is analyzed, and a array of `launch_configs` are generated. Each batch contains a thread for every range block, however the `launch_configs` specifies which domains to consider.

Having divided the launches into batches enabled the use of CUDA streams. This enables kernels to execute concurrently in different streams and can in this way speed up the computations by utilizing all SMP most efficiently. This provides a great speed up for launches that have to be batched in order to be processed. Appendix G.2 contains the source code for this setup.

3 Results

In this section two types of results will be discussed. The quality of the images produced and the execution time of the kernels.

3.1 Metrics

Measuring the quality of a lossy compression scheme is a more subjective matter. A quantitative measure of the quality is the Peak Signal to Noise Ratio(PSNR)[6].I have used `imagemagick`'s[7] compare utility to calculate it. This measure can be used on images of the same size.

To examine the quality of a fractally interpolated images, visual inspection of artifacts seems to be the best way. Furthermore the results has been compared to cubic interpolation.

It has been visually observed that images with a low PSNR produce artifacts when decompressing at a larger than original size. The artifacts present in the original resolution does not disappear at at larger resolution.

Whether the PSNR in the original resolution is a valid metric for the quality of the fractal interpolation is unknown. But it must be be assumed that a high PSNR for the original image is indicative of a potential to perform well when used for fractal interpolation.

6. https://secure.wikimedia.org/wikipedia/en/wiki/Peak_signal-to-noise_ratio
7. http://www.imagemagick.org

Test number	RW	D_STEP
0	2	1
1	2	2
2	2	4
3	2	8
4	2	16
5	4	1
⋮	⋮	⋮
13	8	8
14	8	16

TABLE 1
Experiment I test setup

3.2 Parameters

There are several parameters which influence the speed and quality of these methods.

Two main factors was experimentally examined. The width of a range block(`RW`) and the spacing of domain blocks `D_STEP`.

If the goal was a high compression rate then another relevant parameter to consider would be the *shadow limit*, the variance at which the block is considered to be a shadow block.

3.3 Experiment I

Since there are many parameters and it is not evident how they relate to each other a test setup were devised to analyze the relation between `RW` and `D_STEP`. It measures the time to execute both the simple and quad kernel and then analyzes the PSNR the results can be seen figure 10, the x axis is the test number(Table 1).

Fractal compression is utilizes self similarity, hence the image used does influence the quality, to remedy this all test were executed on three different black and white x-ray images of the authors skull (see appendix B).

It can be observed the overall PSNR for quad is the best, and it is interesting that what intuitively should give the best PSNR, small values

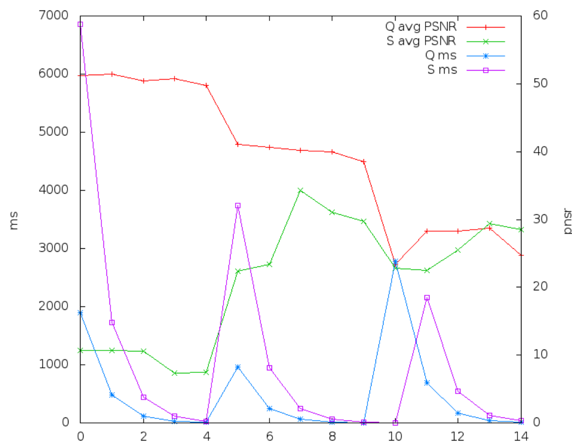

Fig. 10. Experiment I

for RW and D_STEP, does not work for the simple version and when RW is 2 the quad has a almost consistent quality. With respect to execution time quad is almost always slower, but both kernels is highly influenced by D_STEP which is expected as this determines the size of the codebook. It is entirely possible that the quad version suffers greatly as it creates more thread batches where the simple version does it all in one batch.

There are more plots in appendix B.

3.4 Experiment III

Interpolation with the goal to scale an image beyond its original size is of the selling points for fractal compression. Test image 1 (Figure 13) in 256 $\times$ 256 resolution were compressed using the quad method with RW and D_STEP set to 2, the image were decompressed 4096 $\times$ 4096. For comparison the same image were scaled using cubic interpolation[8]. Figure 11 shows a small section, with part of the top of the nose, 12 shows the same section with cubic interpolation.

The fractal interpolation shows blocky artifacts, however the cubic interpolation looks very blurred. The fractal interpolation, though blocky, seems to preserve details. This is considered a very good result, especially taking the problems with the implementation into account.

8. this was done with imagemagicks convert -filter cubic

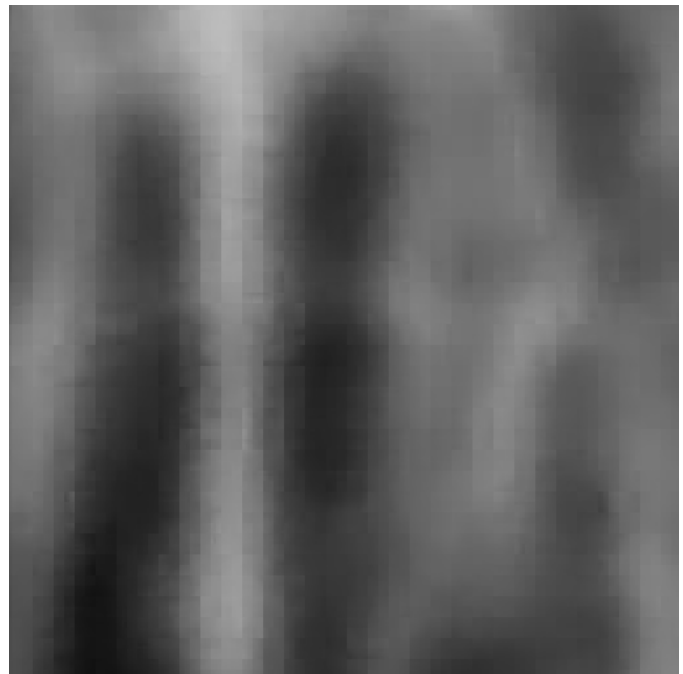

Fig. 11. Fractal interpolation 1600% zoom

Fig. 12. Cubic interpolation 1600% zoom

It is also noteworthy that the fractal compression is lossy, and in the perspective a comparison could have been made with a JPEG image that were scaled without interpolation.

3.5 Problems

The implementation does contain some numerical errors, when inspecting the output it is evident that in some cases the regression line is wrong. It seems to happen in instances where the variance of the code block is low, but not zero. However it depends on the value of the nominator when calculating the slope of the regression line. It has not been possible to figure out exactly the error is. It does only happen to a few (< 10 in a 256 × 256 image) it is noticeable as blocky artifacts. Furthermore is seems that the simple version has a another fault in the calculation of the regression line, which produces output with values `>1.0f`, the normalized version of the images do show that it works correctly, but the brightness is off.

The introduction of numerical flaws might be the reason why a `D_STEP` of one does not produce superior results, as more blocks are considered there is a higher likelihood that a wrong computation is introduced, and this computation will set the residual value to zero and no other blocks are considered from this point on.

4 Further work

The partitioning scheme used is very likely the reason behind the blocky output, it is believed that introducing a quad-map partitioning with overlapping ranges would be a better solution. The basic outline would be to partition the image into large regions, and whenever the residuals were above a certain threshold, the region should be split 4 smaller regions. When taking shadow blocks into account, it should be possible to get at variable `RW` and have a low `D_STEP`.

Whenever the regions are small enough this should be possible to do on the GPU. A 16 × 16 regions would be reasonable in order to let the GPU do the brunt of the work. Utilizing a parallel prefix sum algorithm it will be possible to divide the ranges of each iteration into those the are done and those that should be split for further processing in parallel. The data structures and indexing in the quad algorithm presented are prepared for this, unfortunately time did not permit the implementation.

Shadow blocks would be an interesting area to work with to increase the speed of the algorithm. Initial experiments showed that there were few shadow blocks, and thus the efforts were focused on overlapping ranges.It is suspected that a quadmap scheme will benefit greatly from the extended use of shadow blocks.

There is a large body of previous work were different partitioning schemes and fitness functions has been examined [3]. As they are from the last millennium they have mainly focused on sequential algorithms, it would be interesting to examine which, if any of strategies would benefit from parallelization.

With regards to getting full utilization of the GPU further experiments are needed to determine the optimal strategy for batching threads and using streams.

5 Conclusion

It has been clearly demonstrated that is not only possible but also a great advantage to parallelize fractal compression. The speed up is significant to such a degree that fractal compression should not be considered as an interesting technology without any practical use [4]. It has been experimentally showed that the method of fractal interpolation has a great potential and with a modern GPU the technology can indeed serve a useful purpose.

It is conceivable the there are applications where the asymmetrical nature with built in interpolation may serve as a benefit. The prospect of utilizing a GPU to parallelize the computations makes this a feasible scenario.

Jacob Toft Pedersen Computer Science, Aarhus.

ACKNOWLEDGMENTS

The author would like to thank Helle for putting up with his fractal obsessions.

Source code, image examples and a Linux binary file is available at http://cs.au.dk/~jtp/ppfrac.

I gave a light and lightning talk [4] at Open Space Aarhus http://osaa.dk on the subject, slides are available at http://cs.au.dk/~jtp/ppfrac/doc/fractal_compression.pdf.

REFERENCES

[1] Y. Fisher, "Fractal image compression," *SIGGRAPH*, 1992.
[2] W. Walczak, "Fractal compression of medical images," Master's thesis, Blekinge Institute of Technology, Sweden, 2008.
[3] D. Saupe and R. Hamazaoui, "A review of fractal image compression literature," *Computer Graphics*, vol. 28, pp. 268–276, 1994.
[4] J. T. Pedersen, "Fractal compression - a technology in search of a problem," jan 2011.

CONTENTS

1 Introduction 1

2 Implementation 3

3 Results 6

4 Further work 8

5 Conclusion 8

Biographies 8

References 9

Appendix A: Symmetry transformations 10

Appendix B: Experiment I 10

Appendix C: Running the program 10

Appendix D: Interesting, but not enough 12

Appendix E: Least squares 12

Appendix F: Decompressing an image 13

Appendix G: Source code 14

Appendix H: CUDA prof 18

LIST OF FIGURES

1 How does IFS work 1
2 The attractor $|W|$ of an IFS 2
3 Barnsleys fern (4 transformations) 2
4 Self similarity 2
5 Self similarity in the lung system . 2
6 Relation between range and domain 3
7 Generating the code book 4
8 The main loop 4
9 Simple parallel method 4
10 Experiment I 7
11 Fractal interpolation 1600% zoom . 7
12 Cubic interpolation 1600% zoom . 7
13 Test image 1 10
14 Test image 2 10
15 Test image 3 10
16 quad PSNR for the individual images 11
17 simple PSNR for the individual images 11
18 Convergence 12
19 The required output from the profiler 18
20 The cache misses 18

Fig. 13. Test image 1

Fig. 14. Test image 2

APPENDIX A
SYMMETRY TRANSFORMATIONS

- Identity
- Mirroring in:
  - x - axis
  - y - axis
  - the line y=x
  - the line y=-x,
- Rotating:
  - 90
  - 180
  - 270

APPENDIX B
EXPERIMENT I

APPENDIX C
RUNNING THE PROGRAM

The source code, test and sample images is available at http://cs.au.dk/~jtp/ppfrac/.

The `bin/data` directory contains some test images.

It was developed on a Linux machine and the makefile uses GCC and Linux conventions. It is linked against `freeImage`. There is a binary compiled with `RW`=2 and `D_STEP`=2.

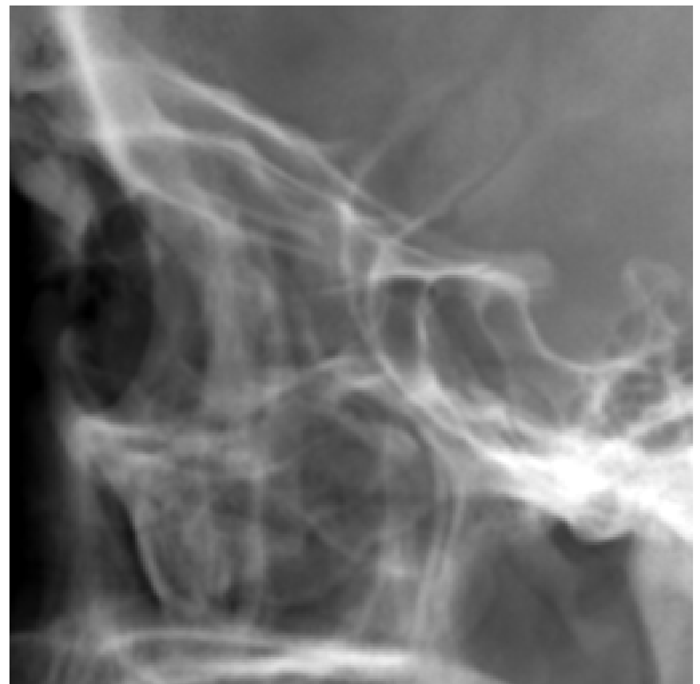

Fig. 15. Test image 3

To compile change the paths to CUDA and the SDK to match you settings(the first two lines of the make file).

then compile by :

```
$ make release=1 opt=1 info=1
```

The program is in the bin directory and can

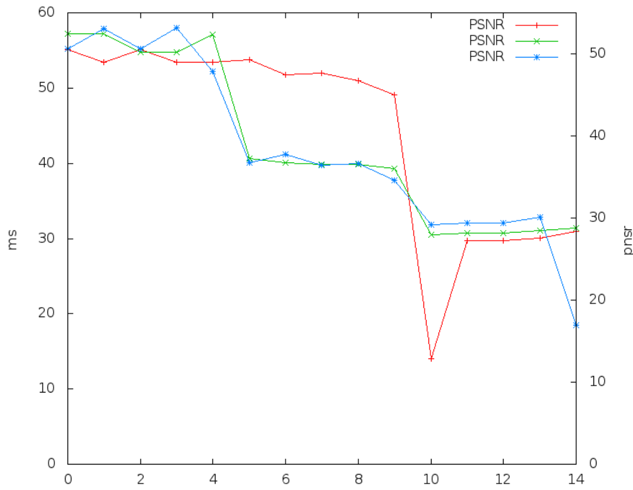

Fig. 16. quad PSNR for the individual images

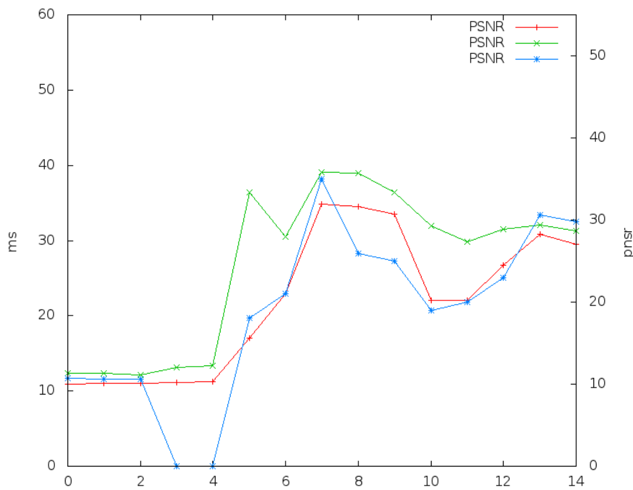

Fig. 17. simple PSNR for the individual images

be run with following options

```
c data/inputImage #simple
o data/inputImage #quad
```

To change `RW` and `D_STEP` edit `parameters.h` and recompile. This was test several different versions and still get the benefits from compiler optimizations and `#pragma unroll`.

When either of the compression kernels are run they create a data file `cuda_blocks.lst` which contains a ASCII representation of the transformations.

To decompress :

```
d inputImage cuda_blocks.lst 8 1
p inputImage cuda_blocks.lst 16 0
```

The first version uses the CPU to decode 8 iterations and save the intermediate images. The second version uses the GPU to decompress 16 iterations without intermediate images. (as mentioned the GPU version will have write-confilcts if output from the quad version is used).

All image output from the program will be saved in `bin/output`

There are test scripts in the root directory to run the test from Experiment I and III `run_test.sh` and `ex3.sh`.

C.1 Optimization log

The calculations of the sums needed for least squares can be precalculated (see appendix E.1), such that for a given code block everything but `ab_sum` is precalculated.

The main part of the loop collectively loads different permutations $S_i(\tau(d))$ into shared memory before calculating `ab_sum` it seemed like a very good place to remove some memory access by doing the permutation using registers and shared memory.

```
float tmp;
if (threadIdx.x < rw && threadIdx.y < rw) {
  tmp = domain[threadIdx.x + threadIdx.y * rw];
}
/* all are saved in local register */
__syncthreads();
if ( threadIdx.x < rw && threadIdx.y < rw) {
  int x = threadIdx.x; int y = threadIdx.y;
  set_indices(type, &x, &y, rw);
  domain[x + y * rw] = tmp;
}
__syncthreads();
```

This yielded a speed up by a factor of two, however it is not sound. It messes up the permutations, as it permutes the permutations and not the original. However it did give rise to the expectation that this would be a good place to optimize. It should be evident that it is possible to carry out the permutations in a sound manner, by clever ordering.

It seemed like a good idea to change the calculation of `ab_sum`, to use permuted addressing in the code block.

```
for (int j = 0 ; j < rw ; j++ )
  for (int i = 0 ; i < rw ; i++ )
   int x = i; int y = j;
   set_indices(type, &x, &y, rw);
   ab_sum += range_data[i + j *rw] * domain[x + y *rw];
```

This version does not use parallel computations and even though its a small loop and it does remove some synchronizations it is slower.

When examining the kernels with the profiler it became apparent that the kernel does not use much shared memory and there were a significant number of cache misses. CUDA uses the same memory area for L1 cache and shared memory, and by telling CUDA to prefer L1 cache the number of cache misses were reduced and it shaved a few percent of the execution time.

By making a copy of the original domain data and performing the permutations on this, the factor two speed up can be realized whilst still getting the benefit from reducing the shared memory requirements. But apparently it introduces concurrency errors, even though the necessary synchronizations are included. The experiments were carried out without this optimization to ensure correctness, as time did not permit further investigations into this behavior.

In appendix H there are output from the profiler.

APPENDIX D
INTERESTING, BUT NOT ENOUGH

D.1 Fractal interpolation

The focus on fractal interpolation meant that the file format used were unoptimized, thus it stores redundant information and includes the calculated residuals. All information are stored as normalized (in the range $[0,1[$) 10 digit floats using ASCII, if compression were the goal the format should of course be binary and use less digits or indexing. However truncating the digits may change the best match, and the residuals has to be recalculated with the truncated digits to ensure the best possible compression [2]

This would unnecessarily complicate the implementation and the ASCII data provides and good way to examine the output, through visual inspection or utilizing gnuplot. The ASCII output does have some flaws changing from base 2 to base 10 and only using 10 digits will introduce minor numerical flaws. They are small and will not affect the output.

It is suspected that the quality of the output is greatly influence by the fact that the file format used is png with indexed gray scale

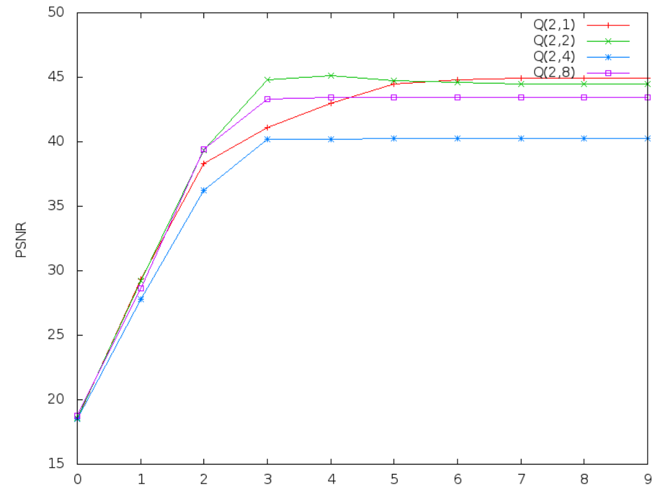

Fig. 18. Convergence

values and thus i limited to 256 different levels of grey.

D.2 Experiment II

In the previous experiments 10 iterations were used when decompressing an image. To examine the necessary number of iterations to reconstruct an image, the intermediate iterations were saved and compared with the reference image, figure 18 shows the result for a single image. Its interesting to note the best quality is with a D_STEP of 2, but that the results however degenerates after further iterations. In all it seems that after 8 iterations we have reached the attractor of the transformations. Since the attractor is what we need the setting D_STEP to 1 will give the best results.

APPENDIX E
LEAST SQUARES

In [1] they had a typo which I missed at first and it took a very long time to track down, as I just expected that their formulas were correct.

The misprint were that they used n^2 were it should just be n, this might be because they work with $n \times n$ blocks.

The goal is to minimize the following function

$$R = \sum_{i=1}^{n} (s \cdot a_i + o \cdot -b_i)^2 \quad (8)$$

Were a_i is the i'th pixel in the code block (from the domain) and b_i is the i'th pixel from the range we want to map to.

s and o defines the regression line and are the scaling and offset we will need to apply to transform pixel intensities when decompressing an image.

$$s = \frac{n\left(\sum_{i=1}^{n} a_i b_i\right) - \sum_{i=1}^{n} a_i \sum_{i=1}^{n} b_i}{n\sum_{i=1}^{n} a_i^2 - \left(\sum_{i=1}^{n} a_i\right)}$$

$$o = \frac{\sum_{i=1}^{n} b_i - s\sum_{i=1}^{n} a_i}{n}$$

If the variance:

$$n\sum_{i=1}^{n} a_i^2 - \left(\sum_{i=1}^{n} a_i\right)^2 = 0$$

of the domain block is zero, then the slope of the regression line is zero and the offset is a simple average.

$$s = 0$$
$$o = \frac{\sum_{i=1}^{n} b_i}{n}$$

The variance of the range block can be use to determine it it is a *shadow block*. If there is no variance, it will only map to domain blocks with no variance. We do not need to search for a domain block for a shadow block, if we do do this then we do not need to process code blocks with zero variance.

We can calculate the residuals using (8) but we do have all the numbers precalculated and it is:

$$R = \frac{\sum_{i=1}^{n} b_i^2 + s\left(s\sum_{i=1}^{n} a_i^2 - 2\left(\sum_{i=1}^{n} a_i b_i\right) + 2\cdot o\sum_{i=1}^{n} a_i\right) + o\left(o\cdot n - 2\sum_{i=1}^{n} b_i\right)}{n}$$

E.1 Implementing

When calculating the least squares we need to calculate several sums, in the code they are

a_sum	$\sum_{i=1}^{n} a_i$
a2_sum	$\sum_{i=1}^{n} a_i^2$
b_sum	$\sum_{i=1}^{n} b_i$
b2_sum	$\sum_{i=1}^{n} b_i^2$
ab_sum	$\sum_{i=1}^{n} (a_i b_i)$

APPENDIX F
DECOMPRESSING AN IMAGE

Unfortunately their is not a standard file format for fractally compressed images, thus to be able to verify and analyze the compression algorithm it was necessary to implement decompression.

Fortunately there is a naive and straightforward method for this. Load an input image f and transformations, apply the transformations to get f_1, continue and apply the transformations to f_1 to get f_2 etc. It will only take a small number of iterations before, the attractor is reached.

This method can be parallelized, there is source code for both methods. The CPU version does not present write conflicts when confronted with overlapping ranges, and is actually fastest, as the parallelized version is not optimized and transfers the data to and from the host for each iteration. However it does have the benefit that it is able to use the same texture functions as the compression kernels which is useful to verify the computations.

APPENDIX G
SOURCE CODE

This appendix contains the source code

```
typedef struct {
  float x,y;    /* of domain */
  float rx, ry;
  float rw;
  float s, o, rms;
  char type;
} cuda_range;


typedef struct {
  int w, h;
  float *data;
  FIBITMAP *fi;
} image;


typedef struct {
  int start_x,start_y, stop_x, stop_y;
} launch_config;
```

Listing 1. Structs Text

```
__global__
void encode_kernel(float *out, const int stride, cuda_range *blocks,
     const int step) {
  int rx, ry, dx, dy;
  rx = blockIdx.x * blockDim.x  + threadIdx.x;
  ry = blockIdx.y * blockDim.y  + threadIdx.y;

  const float n = RW*RW;

  __shared__ float domain[(int) n];

  float range_data[(int) n];
  float min_rms = blocks[rx + ry * (stride/RW)].rms;

  /* load and precalculate */
  float b2_sum = 0, b_sum = 0;
  /* int idx = rx * RW + ry * RW * stride; */
  #pragma unroll
  for (int j = 0 ; j < RW ; j++ ) {
    #pragma unroll
    for (int i = 0 ; i < RW ; i++ ) {
      range_data[i + j * RW] = tex2D(img_tex, i + RW*rx, j + RW*ry);
      b_sum += range_data[i + j*RW];
      b2_sum += range_data[i + j *RW] * range_data[i + j *RW];
    }
  }

  const float shaddow_limit = .00001f;

  const char shaddow = abs(n*b2_sum - b_sum*b_sum) < shaddow_limit;

  if (shaddow) {
    blocks[rx + ry * (stride/RW)].type = SHADDOW;
    blocks[rx + ry * (stride/RW)].y = 0;
    blocks[rx + ry * (stride/RW)].x = 0;
    blocks[rx + ry * (stride/RW)].rms = 0;
    blocks[rx + ry * (stride/RW)].s = 0;
    blocks[rx + ry * (stride/RW)].o = b_sum/n;
    min_rms = 0;
  }

  for (dy = 0 ; dy < stride-RW ; dy += step ) {
    for (dx = 0 ; dx < stride-RW ; dx += step ) {
      char type;
      //#pragma unroll
      for (type=0; type < 8 ; type++) {
 if (threadIdx.x < RW && threadIdx.y < RW)
   domain[threadIdx.x + threadIdx.y * RW] = get_pixel(dx, dy, type,
        threadIdx.x, threadIdx.y, RW);
 __syncthreads();

 /* calculate here */
 float ab_sum =0;
 float a_sum = 0, a2_sum = 0;
#pragma unroll
 for( int i= 0 ; i < (RW*RW) ; i++) {
   a_sum +=  domain[i];
   a2_sum +=  domain[i] *  domain[i];
   ab_sum += range_data[i] * domain[i];
 }


 float s_denom = n * a2_sum - (a_sum*a_sum);
 float s_nom = n*ab_sum - a_sum *b_sum;

 float s, o = 0;
 float rms;
 if ( abs(s_denom) < shaddow_limit ) {
   s = 0;
   o = b_sum/n;
   rms = b2_sum + o * (o * n*n - 2 * b_sum);
 } else {
   s = s_nom/s_denom;
   o = (b_sum - s * a_sum)/n;
   rms = b2_sum + s * ( s * a2_sum - 2 * ab_sum + 2 * o * a_sum) + o
        * (o * n*n - 2 * b_sum);
 }

 rms /= n;
 if ( rms >=0  && rms < min_rms  ) {
   min_rms = rms;
   blocks[rx + ry * (stride/RW)].type = (s==0) ? SHADDOW : type;
   blocks[rx + ry * (stride/RW)].y = (float)(dy)/(float)stride;
   blocks[rx + ry * (stride/RW)].x = (float)(dx)/(float)stride;
   blocks[rx + ry * (stride/RW)].rms = rms;
   blocks[rx + ry * (stride/RW)].s = s;
   blocks[rx + ry * (stride/RW)].o = o;
 }
      }
    }
  }
}
```

Listing 2. Simple kernel Text

```
__global__
void q_quad_kernel(const int stride, cuda_range *blocks, const
     launch_config *p_cfg, const int cfg_idx) {

  const int x = blockIdx.x * blockDim.x + threadIdx.x;
  const int y = blockIdx.y * blockDim.y + threadIdx.y;
  int idx =  x +  gridDim.x * y * blockDim.x;
  const float n = RW*RW;
  /* const int rx = blocks[idx].rx * stride; */
  /* const int ry = blocks[idx].ry * stride; */
  const float f_step = 1.0f / (float)stride;
  const launch_config *cfg = &(p_cfg[cfg_idx]);

  __shared__ float domain[RW*RW];

  float range_data[RW*RW];
  float min_rms = blocks[idx].rms;

  /* load and precalculate */
  float b2_sum = 0, b_sum = 0;

  #pragma unroll
  for (int j = 0 ; j < RW ; j++ ) {
    #pragma unroll
    for (int i = 0 ; i < RW ; i++ ) {
      float pixel = tex2D(img_tex, blocks[idx].rx + i*f_step, blocks[
           idx].ry + j*f_step);
      range_data[i + j * RW] = pixel;
      b_sum += pixel;
      b2_sum += pixel * pixel;
    }
  }

  const float shaddow_limit = .001f;
  const char shaddow = abs(n*b2_sum - b_sum*b_sum) < shaddow_limit;

  if (shaddow) {
    blocks[idx].type = SHADDOW;
    blocks[idx].y = blocks[idx].ry;
    blocks[idx].x = blocks[idx].rx;
    blocks[idx].rms = 0;
    blocks[idx].s = 0;
    blocks[idx].o = .25f*b_sum/n;
    min_rms = 0;
  }

  for (int dy = cfg->start_y  ; dy < cfg->stop_y  ; dy += D_STEP ) {
    for (int dx = cfg->start_x  ; dx < cfg->stop_x  ; dx += D_STEP ) {
      if (threadIdx.x < RW && threadIdx.y < RW) {
 domain[threadIdx.x + threadIdx.y * RW]
   = get_quad_pixel(dx, dy, 0, threadIdx.x, threadIdx.y, RW, stride
        );
      }
      __syncthreads();

      float a_sum = 0, a2_sum = 0;
      for( int i= 0 ; i < n ; i++) {
 a_sum +=  domain[i];
 a2_sum +=  domain[i] *  domain[i];
      }

      for (char type=0; type < 8 ; ) {
 /* calculate here */
 float ab_sum =0;
 for( int i= 0 ; i < n ; i++) {
   ab_sum += range_data[i] * domain[i];
 }

   float s_denom = n * a2_sum - (a_sum*a_sum);
   float s, o = 0;
```

```
    float rms;
    if ( abs(s_denom) < shaddow_limit ) {
      s = 0;
      o = b_sum/n;
      rms = b2_sum + o * (o * n - 2 * b_sum);
    } else {
      float s_nom = n*ab_sum - a_sum *b_sum;
      s = s_nom/s_denom;
      o = (b_sum - s * a_sum)/n;
      rms = b2_sum + s * ( s * a2_sum - 2 * ab_sum + 2 * o * a_sum) +
            o * (o * n - 2 * b_sum);
    }

    rms /= n;
    if ( rms >=0  && rms < min_rms   ) {
      min_rms = rms;
      blocks[idx].type = (s==0) ? SHADDOW : type;
      blocks[idx].y = (float)(dy)/(float)stride;
      blocks[idx].x = (float)(dx)/(float)stride;
      blocks[idx].rms = rms;
    blocks[idx].s = s*.25f;
    blocks[idx].o = o*.25f;
    }
  type +=1;
  if (type < TYPES && threadIdx.x < RW && threadIdx.y < RW) {
    domain[threadIdx.x + threadIdx.y * RW]
      = get_quad_pixel(dx, dy, type, threadIdx.x, threadIdx.y, RW,
            stride);
  }
  __syncthreads();
        }
      }
    }
}
```

Listing 3. Q kernel

```
void apply_cuda_tranforms(float *img, float* out, cuda_range *blocks,
       int block_cnt, int stride) {

  for(int block = 0; block < block_cnt ; block++) {
    cuda_range b = blocks[block];
    int rw = b.rw * stride;  /* only square images!! */
    /* get/write data for this block */
    for(int j = 0; j < rw; j ++) {
      for(int i = 0; i < rw; i ++) {
  int x,y;
  x = i;
  y = j;
  float pixel;
  /* torus world  */
  int rx = ((int) (b.rx *stride + i));
  int ry = ((int) (b.ry *stride + j));

  rx = wrap_coordinate(rx, stride);
  ry = wrap_coordinate(ry, stride);

  int idx = rx + ry * stride;
  if (idx < 0 || idx >= stride * stride) {
    printf("aaargh %d, %d \n", rx, ry);
  }

  if (b.type == SHADDOW) {
    /* shaddow block */
    pixel = b.o;
  } else {
    set_indices(b.type, &x, &y, rw);
    pixel = get_pixel_no_tex(b.x*stride, b.y*stride, x, y, img,
          stride);
    pixel *= b.s;
    pixel += b.o;
  }
  out[idx] += pixel;
        }
      }
    }
}

extern "C"
float *cpu_decode(image *img, const char *filename, int iterations,
       char debug) {
  cuda_range *blocks;
  int block_cnt, it, size;
  float *out;
  char buf[512];
  size = img->w *img->h;

  out = (float*) malloc(sizeof(float) * size);
  clear_data(out, size);

  blocks = load_blocks(filename, &block_cnt);

  for (it = 0; it < iterations; it++) {
    apply_cuda_tranforms(img->data, out, blocks, block_cnt, img->w);
    swap_buffers(&(img->data), &out);
    if (debug) {
      /* save image */
      sprintf(buf, "iteration%02d.png", it);
      save_image(buf, img);
      sprintf(buf, "normalized_iteration%02d.png", it);
      save_normalized_data(buf, img);
    }

    clear_data(out, size);
    printf("%d of %d\n", it, iterations);
  }
  sprintf(buf, "normalized_final_output.png", it);
  save_normalized_data(buf, img);
  return img->data;
}
```

Listing 4. CPU decompress

```
/* one thread pr pixel, one block pr range_block */
__global__
void decode2_kernel(float *out, int stride, cuda_range *blocks) {
  int range_idx,x,y,idx;
  float pixel;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;

  idx = x + gridDim.x * y * blockDim.x;
  __shared__ cuda_range range;
  range_idx = blockIdx.x + gridDim.x * blockIdx.y;
  if (threadIdx.y == 0 && threadIdx.y == 0) {
    range = blocks[range_idx];
  }
  __syncthreads();

  pixel = get_pixel( (range.x * stride) , (range.y * stride), range.
        type, threadIdx.x, threadIdx.y, blockDim.x);
  pixel *= range.s;
  pixel += range.o;

  idx = x + y * stride;
  out[idx] =  pixel;
}
```

Listing 5. GPU decompress

G.1 Device functions

```
extern "C"
__device__
__host__
void set_indices(char type, int *i, int *j, int range_width) {
  int x,y, offset;
  offset = range_width-1;
  x = *i;
  y = *j;

  switch(type) {
  case IDENTITY:
    *i = x;
    *j = y;
    break;
  case ROT90:
    *i = y;
    *j = offset - x;
    break;
  case ROT180:
    *i = offset - x;
    *j = offset - y;
    break;
  case ROT270:
    *i = offset - y;
    *j = x;
    break;
  case MIRROR_Y:
    *i = offset - x;
    *j = y;
    break;
  case MIRROR_X:
    *i = x;
    *j = offset - y;
    break;
  case MIRROR_XY:
    *i = y;
    *j = x;
    break;
  case MIRROR_YX:
    *i = offset - y;
    *j = offset - x;
    break;

  default:
    break;
  }
}
```

```
__device__
__host__
int wrap_coordinate(int p, int stride) {
  if (p < 0) return p+stride;
  if (p >= stride) return p-stride;
  return p;
}

__device__
float get_pixel(int dx, int dy, char type, int x, int y, int
      range_width) {
  float tmp;
  set_indices(type, &x, &y, range_width);
  tmp=  tex2D(img_tex,     dx + 2*x,      dy + 2* y);
  tmp+= tex2D(img_tex, 1 + dx +2*x,       dy + 2* y);
  tmp+= tex2D(img_tex,     dx + 2*x,  1 + dy + 2* y);
  tmp+= tex2D(img_tex, 1 + dx +2*x,   1 + dy + 2* y);
  tmp*= .25f;
  return tmp;
}

#define N(X) ((X)/(float) stride)
__device__
float get_quad_pixel(int dx, int dy, char type, int x, int y, int
      range_width, int stride) {
  float tmp;
  set_indices(type, &x, &y, range_width);

  tmp=  tex2D(img_tex,    N(dx + 2*x),      N(dy + 2* y));
  tmp+= tex2D(img_tex, N(1 + dx +2*x),      N(dy + 2* y));
  tmp+= tex2D(img_tex,    N(dx + 2*x),  N(1 + dy + 2* y));
  tmp+= tex2D(img_tex, N(1 + dx +2*x),  N(1 + dy + 2* y));
  tmp*= .25f;
  return tmp;
}

__global__
void locate_shaddow_blocks(cuda_range *blocks, int stride, float *out
      ) {
  int x, y, idx;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  idx = x +  gridDim.x * y * blockDim.x;
  cuda_range range = blocks[idx];
  float S=0, S2 = 0, n = 0;
  int w = (int) (range.rw * stride);
  int rx = (int) (range.rx * stride);
  int ry = (int) (range.ry * stride);

  for (int i = 0 ; i < w ; i++ ) {
    for (int j = 0 ; j < w ; j++ ) {
      float pixel = tex2D(img_tex,(float)(rx + i), (float)(ry + j));
      S += pixel;
      S2 += pixel * pixel;
      n+=1.0;
    }
  }

  float var = abs(n * S2 - S*S);
  const char shaddow = var < 1e-6;

  for (int i = 0 ; i < range.rw * stride ; i++ ) {
    for (int j = 0 ; j < range.rw * stride ; j++ ) {
      int oidx = rx + ry * stride;
      out[oidx + i + j *stride] = shaddow ? 1 : 0;
    }
  }

  if ( shaddow ) { /* a almost zero variance */
    blocks[idx].rms = 0;
    blocks[idx].s = 0;
    blocks[idx].o = S/n;
    blocks[idx].x  = 0;
    blocks[idx].y = 0;
  }
}


__device__
void clear_block(cuda_range *b) {
  b->rms = 999;
  b->x = -42;
  b->y = -42;
  b->s = -42;
  b->o = -42;
  b->type = -42;
}


__global__
void init_range_blocks(cuda_range *blocks, int w, int rw) {
  int x, y, idx;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  idx = x +  gridDim.x * y * blockDim.x;
  clear_block(&blocks[idx]);
  blocks[idx].rx = (float)x / ((float)(w)); /* cast a lot make sure
        it is float computations/ */
  blocks[idx].ry = (float)y / ((float)(w)); /* cast a lot make sure
        it is float computations/ */
  blocks[idx].rw = 1.0f / (float)(w);
}


__global__
void init_quad_blocks(cuda_range *blocks, int w, int rw) {
  int x, y, idx;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  idx = x +  gridDim.x * y * blockDim.x;

  for (int j = 0 ; j < 2 ; j++ ) {
    for (int i = 0 ; i < 2 ; i++ ) {
      int tmp = 4*idx + i + j *2;
      clear_block(&blocks[tmp]);
      float rw = 1.0f / (float)(w);
      blocks[tmp].rw = rw;
      blocks[tmp].rx = (float)x / ((float)(w));
      blocks[tmp].ry = (float)y / ((float)(w));

      blocks[tmp].rx += rw * .5 * i;
      blocks[tmp].ry += rw * .5 * j;
    }
  }
}


__global__
void test_texture(float *data ) {
  int x, y, idx;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  idx = x +  gridDim.x * y * blockDim.x;
  const float offset = -10.0f; //gridDim.x * blockDim.x;
  data[idx] = tex2D(img_tex, x + offset, y + offset);
}


__global__
void test_normalized_texture(float *data ) {
  int x, y, idx;
  x = blockIdx.x * blockDim.x + threadIdx.x;
  y = blockIdx.y * blockDim.y + threadIdx.y;
  idx = x +  gridDim.x * y * blockDim.x;
  const float max_val = 256.0f;
  const float offset = (gridDim.x * blockDim.x)/2.0f;
  data[idx] = tex2D(img_tex, (x + offset)/max_val, (y + offset)/
        max_val);
}
```

Listing 6. Interesting device function

G.2 Thread batching

```
extern "C"
float *cuda_opt_encode(image *img, char config) {
  cuda_range *h_blocks, *d_blocks;
  float *h_data, *d_out;
  int texture_size, block_cnt, n;
  cudaArray *a;
  cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();

  // Set texture parameters
  img_tex.normalized = true;  /* OBS */
  img_tex.addressMode[0] = cudaAddressModeWrap;
  img_tex.addressMode[1] = cudaAddressModeWrap;
  img_tex.filterMode = cudaFilterModePoint;

  /* allocate memory */
  /* copy to device code removed */
  /* init data blocks */
  reset_quad_blocks(d_blocks, n, RW);
  /* reset_blocks(d_blocks, n, RW); */
  CUT_CHECK_ERROR("Kablam!");

  /* optimize share memory / l1 */
  // Runtime API
  // cudaFuncCachePreferShared: shared memory is 48 KB
  // cudaFuncCachePreferL1: shared memory is 16 KB
  // cudaFuncCachePreferNone: no preference
  cudaFuncSetCacheConfig(q_quad_kernel, cudaFuncCachePreferL1);


  float time = 0;
  const int tilesize = 16;
  int grid_cnt = block_cnt/ (tilesize * tilesize);
  block_size.x = tilesize;
  block_size.y = tilesize;
  block_size.z = 1;
```

```
  grid_size.x = sqrt(grid_cnt);
  grid_size.y = sqrt(grid_cnt);
  grid_size.z = 1;


  long long domains = (img->w/D_STEP) *  (img->w/D_STEP);
  printf("total_comparisons = %lld \n", domains * block_cnt );
  int batchsize = 1e9;

  /* hver batch laver STEP^2*block_cnt comparisons */
  int STEP = sqrt(batchsize/block_cnt);
  printf("Step = %d\n", STEP);

  /* make sure the dimensions fit .. */
  int batches=0;
  for (int x = 0; x < img->w ; x += STEP)
    for (int y = 0; y < img->w ; y += STEP)
      batches++;
  printf("Need %d batches\n", batches);

  launch_config *h_cfg;
  launch_config *d_cfg;

  cutilSafeCall( cudaMalloc( (void **) &d_cfg, batches * sizeof(
       launch_config)));
  CUT_CHECK_ERROR("Kablam!");
  h_cfg = (launch_config*) malloc(batches * sizeof(launch_config));

  int idx = 0;
  for (int x = 0; x < img->w ; x += STEP) {
    for (int y = 0; y < img->w ; y += STEP) {
      h_cfg[idx].start_x = x;
      h_cfg[idx].start_y = y;
      h_cfg[idx].stop_x = x + STEP;
      h_cfg[idx].stop_y = y + STEP;
      h_cfg[idx].stop_x = min(h_cfg[idx].stop_x, img->w);
      h_cfg[idx].stop_y = min(h_cfg[idx].stop_y, img->w);
      idx++;
    }
  }

  /* copy the configuration to the device */
  cutilSafeCall(cudaMemcpy(d_cfg, h_cfg, batches * sizeof(
       launch_config), cudaMemcpyHostToDevice));
  CUT_CHECK_ERROR("Kablam!");


  /* create the streams */
  const int stream_cnt = 4;
  cudaStream_t streams[stream_cnt];
  for(int i = 0; i < stream_cnt; i++) {
    cudaStreamCreate(&streams[i]);
  }

  /* launch the streams */
  for(int i = 0; i < batches; i++) {
    int stream_idx =  i % stream_cnt;
    q_quad_kernel<<<grid_size, block_size, 0, streams[stream_idx]>>>(
         img->w, d_blocks, d_cfg, i);
  }
  CUT_CHECK_ERROR("Kablam!");
  cudaThreadSynchronize();
  printf("%f ms to execute kernels on image \n",  time);

  /* release the streams */
  for(int i = 0; i < stream_cnt; i++) {
    cudaStreamDestroy(streams[i]);
  }
  CUT_CHECK_ERROR("Kablam!");

  return  h_data ;
}
```

Listing 7. Batching threads

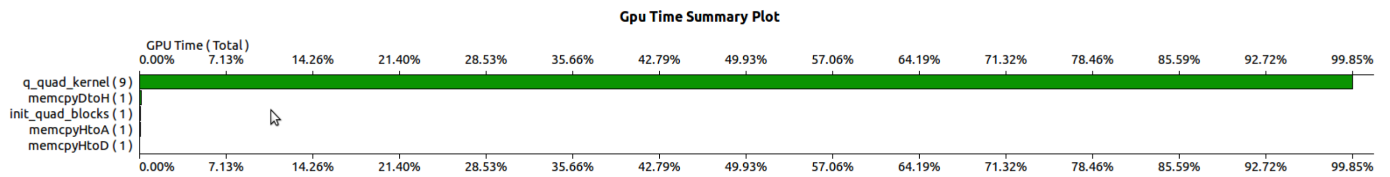

Fig. 19. The required output from the profiler

	GPU Timestamp	Method	GPU Time	l1 local load hit Type:SM Run:2	l1 local load miss Type:SM Run:3	l1 local store hit Type:SM Run:3	divergent branch Type:SM Run:4	l1 local store miss Type:SM Run:4	l1 global load hit Type:SM Run:4
1	-0.671875	memcpyHtoA	80						
2	7557.47	access	14	0	0	0	0	0	0
3	7922.02	init_quad_blocks	152	0	0	0	0	0	0
4	8574.05	memcpyHtoD	0						
5	8611.42	q_quad_kernel	4	155208867	38715167	147084285	89221	10098850	557079
6	4.12636e+06	memcpyDtoH	88						
7	4.12681e+06	memcpyDtoH	196						

Fig. 20. The cache misses

APPENDIX H
CUDA PROF

Figure 20 clearly demonstrates that it is a wise decision to focus optimization efforts on one kernel.