FAST FRACTAL IMAGE COMPRESSION

Jinank Jain, Milan Pandurov, Anton Permenev, Manuel Rodríguez

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Fractal compression is a lossy compression algorithm for images based on fractals,using the fact that a fractal is made up of a union of several copies of itself and tries to find such patterns in the encoding image. Finding such patterns in the image is an expensive task as one must compare every possible image segment with the complete image and therefore traversing it numerous times. This paper presents one approach of making such computation faster by extending a straightforward implementation with numerous optimization techniques such as loop unrolling, single static assignment, scalar replacement, code vectorization using Intel AVX2 intrinsics and reducing cache and data TLB misses by improving temporal locality in the computations. This paper also analyzes the effects of different optimizations techniques and their mutual relationship on the overall compression runtime. On average we were able to obtain a speedup of approximately 14x.

1. INTRODUCTION

With advance of Internet and the need of massive data storage, compression schemes have become an important part of every computing process. Storing images in less memory leads to a direct reduction in storage cost and faster data transmissions. Fractal image compression [1]-[2] is one such scheme which provides high compression ratio, thus improving runtime of encoding process could be beneficial.

One example of commonly used compression algorithms is Joint Photographic Experts Group (JPEG) whose encoding is based on Discrete Cosine Transform (DCT) which is applied on localized parts of the image. On the other hand, fractal compression requires mutual comparisons between image parts in encoding process, which is more computationally expensive work.

Fractal Compression was first promoted by M.Barnsley at [3] and since then it has found usage in several academic works. Implementation on which this paper is based on was developed by Alex Kennberg [4]. Most successful implementation is done by Microsoft corporation in their Encarta digital multimedia encyclopedia.

2. BACKGROUND

Basic idea in fractal image compression is to divide the whole image into two types of square blocks namely range and domain blocks. After this division objective is to find a domain block that covers range block as closely as possible when contractive mapping is applied to it. Contractive mapping is affine transformation that also shrinks domain block, domain blocks in our implementation are always 4 times larger than range blocks. For every given range block we iterate through all domain blocks, and find the best matching transformation that also satisfies constraint of given threshold value. If no match has been found, initial range and consequently all domain blocks will be divided into 4 smaller parts and same procedure will be repeated. Every time a satisfactory transformation that converts domain to range block has been found, properties of that transformation are stored in a resulting list. This resulting list of transformation properties is compressed image. In order to decompress the image, list of transformation properties is applied to an arbitrary bitmap through several phases. Resulting bitmap is decompressed image.

2.1. Cost & Complexity Analysis

Since number of operations algorithm performs is not only dependent on image size, but on the image content as well, it was hard to define cost measure as a simple function of image features. Therefore we had to calculate cost measures for every test image individually by instrumenting code to count arithmetic operations empirically. As a result we found out that the majority of algorithm's operations consisted of integer and floating point additions and multiplications, therefore we defined cost measure as:

$$C(n) = (int_{add}(n), int_{mul}(n), fp_{add}(n), fp_{mul}(n)) \quad (1)$$

It has to be noted that our algorithm uses a negligible amount of floating point and integer divisions, as compared to the number of additions and multiplications, therefore they are not included in cost analysis.

The algorithm starts by dividing image into a fixed number of what is defined as range blocks and then tries to find matches for them in domain blocks. If no match for certain range block is found it gets divided into 4 smaller parts and process is repeated. Since initial range block can be divided into smaller parts only fixed number of times we can say that number of comparisons is always constant in worst case. Complexity thus only depends on the image size or more precisely block size, formally complexity is $\mathcal{O}(n^2)$ where n is height of the square image.

2.2. Roofline Plot Analysis

Before jumping into optimizations it is important to quickly analyze where algorithm lies in roofline plot, more specifically to see if algorithm is memory or compute bound. We assumed worst case scenario where image can't fit in cache and during the process of comparing range with domain blocks every domain block has to be transferred from RAM. We assume that comparing range block is always in cache as it is constantly reused. By inspecting the code we see that for domain block of size b bytes we perform around 237 b operations in order to compare it with range block, which gave us lower bound on operational intensity of 237 ops/bytes as shown in Fig. 1

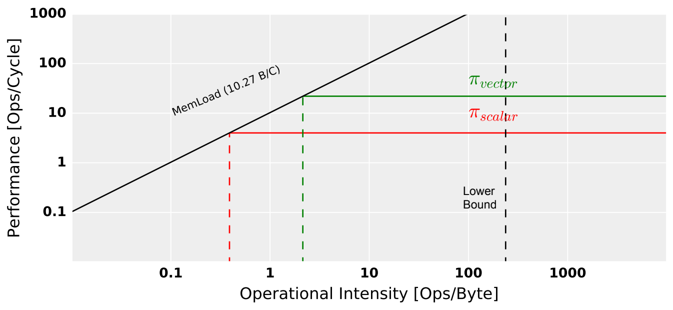

Fig. 1. Roofline Plot with adjusted peak performance value due to operation imbalance

Looking at the number of operations gathered empirically from method described in Sec.2.1 we concluded that there is a constant ratio between number of multiplications and additions (3:1). Throughput for addition and multiplications is 3 and 1 ops/cycle respectively. So we simplified our calculations and replaced all multiplications with appropriate number of additions.

In order to calculate the roof we have to understand the architecture of benchmarking hardware (Intel Skylake CPU). It has 4 relevant ports and out of those 4 ports 2 ports can perform vectorized instructions. Since we are dealing with 32-bit integers number of operations that can be performed by those 2 ports would be 16(=8*2). Remaining two ports can only perform non-vectorized additions. So total number of additions that could be done in one cycle would be 6(=3*2) ops/cycle. So the max performance that could be achieved would be 22(=16+6)ops/cycle.

3. PROPOSED METHOD

There are several assumptions that are made in order to simplify implementation. Firstly we are restricting our implementation to work with squared images of sizes which are multiplies of 32 pixels. Secondly our implementation works only with scenarios where domain blocks are always 4 times larger then domain blocks.

Optimizations are divided into 4 groups, where each group is using different methodology for achieving the final goal. These 4 groups are later combined in layered approach as described below.

3.1. Compiler Optimizations

As a first step in improving program's runtime, we decided to explore what compiler has to offer. As algorithm utilizes lot of mathematical operations we included `ffast-math` `funsafe-math-optimizations` flags as algorithm is not sensitive for small errors in calculations. Also code consists of many nested loops and we therefore used flags `funroll-all-loops` and `ftree-vectorize` to let compiler vectorize original code as much as it can.

3.2. Basic Optimizations

Since frequently invoked functions in original code were grouped in different libraries, compiler wasn't able to inline them automatically. One major refactoring that we did as part of basic optimizations was moving those functions in single header file and making them inline. By doing this we wanted to avoid function call overhead and also allow compiler to enable other optimizations.

As said before, result of encoding phase is a list of transformation properties. In the original code those transformations were stored in a linked list with dynamic memory allocation. Since frequently invoking `malloc` carries big overhead, we replaced linked list with a static allocated array. Size of array was set to be maximum theoretical size and in this way we increased memory footprint of the application in order to get better performance.

At the end we did scalar replacements and simplified array index calculations by moving all the loop invariants outside the loops.

3.3. SIMD Optimizations

In our code there are three major functions which are affected by SIMD optimizations: `get_average_pixel`, `get_error` and `get_scale_factor`. Every function is blocked into three different sizes 32x32, 16x16, 8x8 and the left over. Since we are dealing with images which have 8-bit integers but all these functions require accumulation of result into 32-bit integer, we have to pad every 8-bit integer to make it a 32-bit integer which restricts us from exploiting the full power of vector instructions. Thus we can get maximum speed up to 8x. Now let's analyze these functions individually.

`get_average_pixel`

$$average\_pixel = \frac{\sum_{i=0}^{size}\sum_{j=0}^{size} p_{ij}}{size * size} \tag{2}$$

Since the size of blocks is large, the sum of all pixels will overflow 16-bit integers. Thus 32-bit integers becomes the next obvious choice which is good enough to hold the value of all the expected block size. We therefore have to extend 8-bit integers to make them 32-bit integers.

`get_error`

$$error = \frac{\sum_{i=0}^{size}\sum_{j=0}^{size} scale * (p_{ij} - p'_{ij})}{size * size} \tag{3}$$

The structure of squared error difference between range and domain block perfectly matched with formulation of Fused Multiply-Add(FMA). It shares the same overflow issue, so we have to pad 8-bit integers to make it 32-bit integers.

`get_scale_factor`

$$scale\_factor = \frac{\sum_{i=0}^{size}\sum_{j=0}^{size} (p_{ij} * p'_{ij})}{\sum_{i=0}^{size}\sum_{j=0}^{size} (p_{ij} * p_{ij})} \tag{4}$$

From the above formula, it can be observed that numerator and denominator could be vectorized individually with the help of some accumulators. But it shares the same fate of integer overflow thus needs some padding.

3.4. Cache Optimizations

As we saw in section 2, finding best match for a given range block requires traversing through all domain blocks of an image in order to conduct comparisons. For large images, that can't fit in cache, all domain blocks loaded will be evicted from it by the succeeding domain blocks and never be reused. Moreover this will happen when trying to find a match for every range block. At first glance solution for improving temporal locality - reusing loaded domain blocks - would be to compare them not only with one range block but with several range blocks in batch. This approach would theoretically decrease data movement between RAM and cache by a factor proportional to number of range blocks in a batch and therefore improve runtime. Unfortunately that is not the case, main problem comes from a fact that some range blocks won't find a match while others will. Bookkeeping which range blocks are matched and which still have to be matched in a batch requires help of additional data structures, introduces additional operations and complex branching in code. More precisely figure 2 shows one possible way of matching initial 3 range blocks over 4 iterations. Matched blocks in one iteration are marked by green color, while unmatched (red blocks) get divided into 4 smaller parts and they are compared again in next iteration. As this approach didn't give expected speedup, in fact mentioned overhead was too large that it created slowdown, we had to look for other approach to reduce data movement.

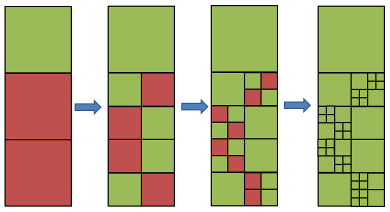
Fig. 2. Finding matches for range blocks organized in batch

By analyzing algorithm's behavior, we concluded that for majority of pictures at every level of recursion (except last) around half of range blocks get matched. Other half gets divided into 4 smaller parts so every time block hasn't been matched image is being traversed at least 4 additional times. Leveraging this fact we modified algorithm, so it uses loaded domain block to look for matches not only for one range block, but also for its 4 immediate sub blocks in advance. Fig. 3 shows how comparisons are performed between range blocks (blue blocks labeled with numbers) and domain blocks (green blocks labeled with letters). Instead of only comparing 1 with A we also compare elements {2,3,4,5} with all elements from {B,C,D,E}. If there is no match for 1 and A we don't have to traverse image 4 more times as we already have those results, also we expect not to find a match for half of the blocks. In case we find a match between block 1 and A additional pre-computations that we performed are wasted. After this we repeat the same step for all smaller unmatched blocks again.

This approach also brings several implicit benefits in reducing operation count. When calculating average pixel for block (i.e. block 1), we have average pixel values for sub blocks (B,C,D and E) as intermediate values so we don't need to compute them individually again. Also generating a permutation of a larger block implicitly permutes smaller sub blocks inside it, figure 4 shows how rotating a block by

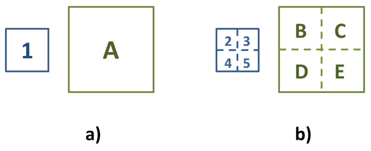

Fig. 3. Comparing range block and domain block. a) comparing bigger blocks b) pre-computing smaller blocks

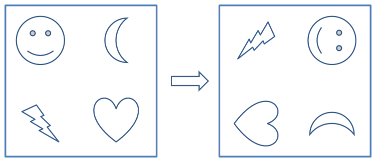

Fig. 4. Rotating a block by 90 degrees right

90 degrees also rotates its sub-blocks, therefore only rotating bigger block is sufficient. Its also worth mentioning that while iterating through domain blocks pre computation is done only while match for larger block hasn't been found. As soon as the match has been found, we stop computing matches for smaller blocks.

4. EXPERIMENTAL RESULTS

Platform used for benchmarking has the following specification: **Intel Core i7-6700 @ 3.40 GHz** with **32 KB L1 cache**, **256 KB L2 cache** and **8MB L3 cache**. Two different compilers were used: GNU C compiler (version 4.8) and Intel C compiler (version 14.0.1). For calculating runtime in cycles, RDTSC counter has been used. For measuring number of cache and TLB misses data from `perf` was used. Measurements are only related to encoding process.

In order to verify the correctness of our implementation, we used perceptual image hashing [5] to compare final outputs and also size of compressed image.

4.1. Runtime and Performance Plots

From the Fig. 5 following conclusions can be made. Firstly, since program is compute bound, expected behavior was to see a significant speedup from SIMD optimizations. Secondly, cache optimizations became significant only on large image size. Thirdly, anomalies with image sizes 2048 and 4096 are caused by way that vectorization is implemented (refer to Sec.3.3). In the end, we were able to achieve a speed up of around 14x. In order to calculate performance, code was instrumented according to observations described in Sec. 2.1. After calculating number of operations for every image, runtime data presented in Sec.4.1 was used to calculate performance. Looking at the performance plot in Fig. 6 following observation can be made: theoretical peak

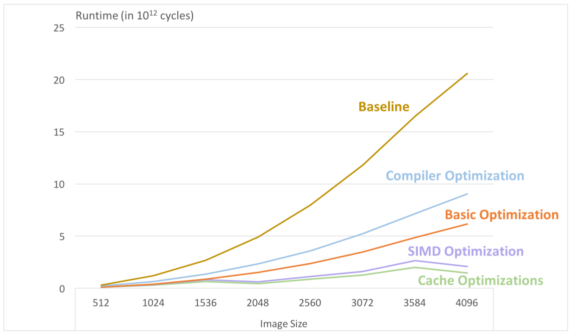

Fig. 5. Runtime plots for various optimizations schemes

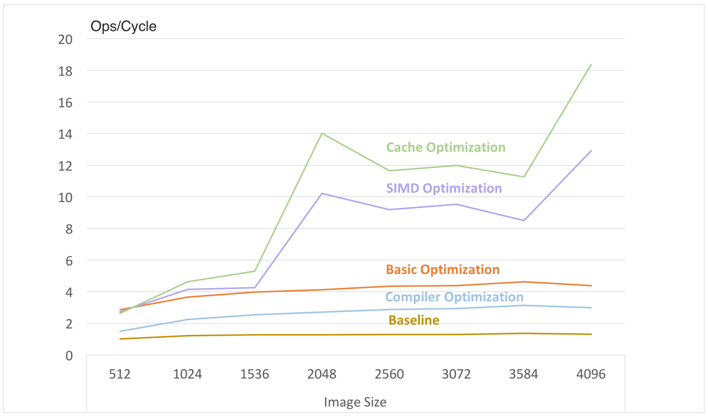

Fig. 6. Performance plots for various optimizations schemes

performance of 22 ops/cycle wasn't reached. Reason for that mainly lies in a fact that for calculating runtime all operations (both floating point and integer) were used together, but in reality floating points are more expensive than integer operations. Another reason for achieved slightly lower values for peak performance can be described by how vectorization is implemented, mainly there will be some leftovers which won't be processed by vectorized code (described in 3.3).

4.2. ICC vs GCC

Two different compilers were used in benchmarking as noted in 4. Results showed that `icc` was slower than `gcc` in all scenarios, shown in Fig.7. In order to make comparison fair we used equivalent compiler flags: `-O3 -march=native -funroll-all-loops -ffast-math -funsafe-math-optimizations -ftree-vectorize` for `gcc` and `-O3 -march=CORE-AVX2 -axCORE-AVX2 -xCORE-AVX2 -parallel -finline-functions -ip -fp-model fast -unroll -opt-prefetch=4` for `icc`.

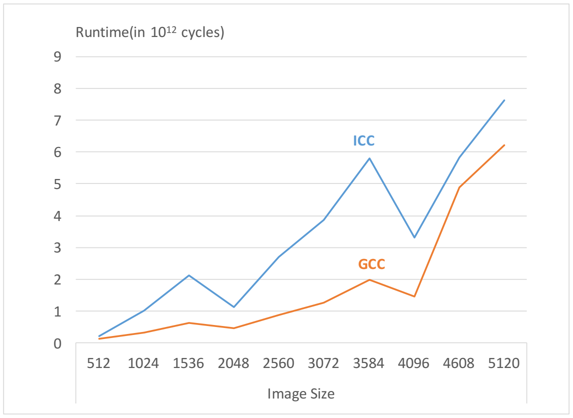

Fig. 7. ICC vs GCC Compiler

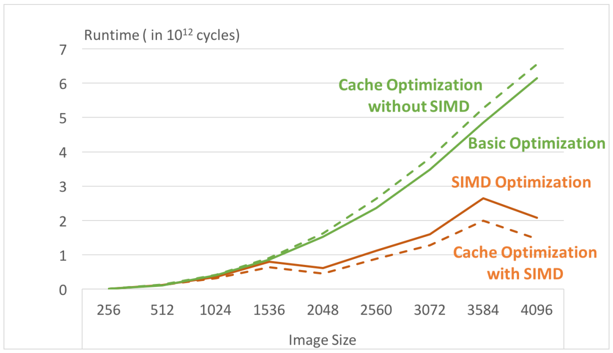

Fig. 8. Cache Optimizations needs help of SIMD Optimizations

4.3. Cache Optimization needs help of SIMD

As we have seen in the previous section, cache optimization reduces data movement at expense off performing wasted computations. Main idea of this approach is that performing wasted computation can be made faster than just waiting for data to come from RAM. If performing wasted computations is not fast enough, cache optimization will create slowdown. This can be clearly seen in Fig. 8 where cache optimization only gives benefit when combined with vectorized code. Scalar implementation gives slowdown as it can't compute wasted computations faster than only moving relevant data from RAM. In order to get a better understanding about how cache optimization helped in improving runtime, we did function profiling for the versions with and without cache optimizations. Mainly we instrumented code to measure cycles for 3 most frequently used functions, as they contribute majority of programs total runtime. Two out of these three functions are performing dense calculations on data (getting scale factor, calculating error) and they contribute to majority of programs total operation count. Third function is doing matrix permutations and it barely contains any numerical operations, but in contrast in non cache optimized code it is taking roughly same number of cycles as the other two functions as seen in Fig. 9. When cache optimizations are included, functions for calculating scale factor and error are spending more cycles, as they are doing some wasted computations. On the other hand matrix permutation execution time has been drastically reduced mainly because that function is being invoked less times, shown in Fig. 10. This lead to increase of performance in cache optimized version of the program.

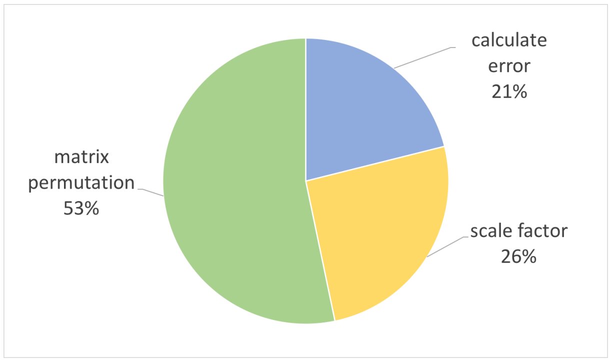

Fig. 9. Function Profiling for SIMD Version

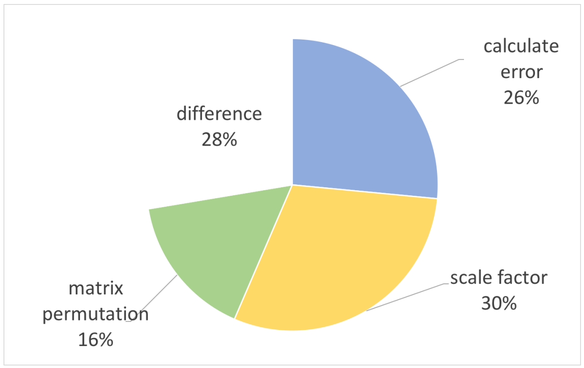

Fig. 10. Function Profiling for Cache Optimized Version.

4.4. Speed-Up Decomposition

If we break up speedup in components that its consisted from, we can make few interesting remarks, Fig. 11. Mainly adding appropriate compiler flags will give consistent speedup of around 2x over all image sizes. Applying general optimizations, described in 3.2, doesn't give huge speedup, but it rather serves as a base point for other optimizations to build upon. Applying SIMD optimizations (3.3) only brings benefit when working with larger images, as vectorized instructions can't be fully utilized on small images. Similar to SIMD optimizations, cache optimizations are starting to be beneficial only when working with larger images.

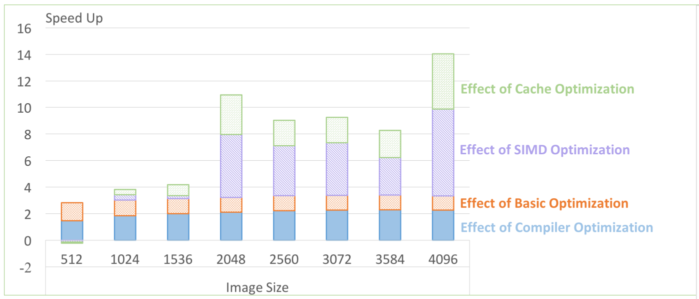

Fig. 11. Speed-Up Decomposition from various optimization

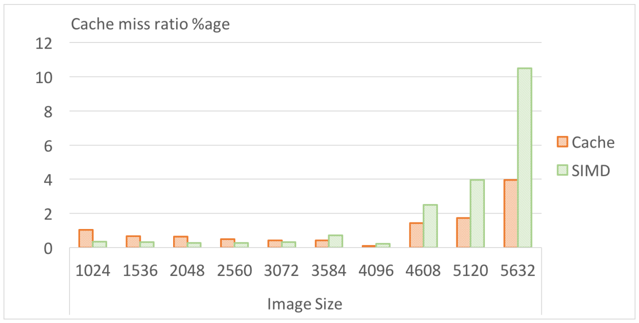

Fig. 12. Reduction in cache miss ratio in Cache Optimized from SIMD Version

4.5. Reduction in Cache miss ratio

While looking for a match for one range block, one must traverse complete image and load domain blocks for comparison. In this scenario we assume that one range block will always reside in cache (as it is constantly being reused in comparison process) and domain blocks are being loaded and evicted from cache when complete image can't fit in cache. We also assume that cache can always hold one range and one domain block. With our implementation of cache optimizations, we are expecting to see reduction of cache misses up to 5 times depending on the image content. Reduction of cache misses can be seen in Fig. 12. It is worth mentioning that in worst case proposed approach won't reduce cache misses, for example this is a case in single color images because finding a match for every range block takes exactly one traversal over image.

4.6. Reduction in D-TLB misses

For large images spreading across many page tables, cache optimization also helps in reducing TLB misses. If we take an example of image size 4096x4096 pixels where every row fills one page table, we need 4096 TLB entries (one per row) for working dataset. Unfortunately number of TLB entries is much smaller and they will be completely flushed every time image is traversed. In cache optimized version,

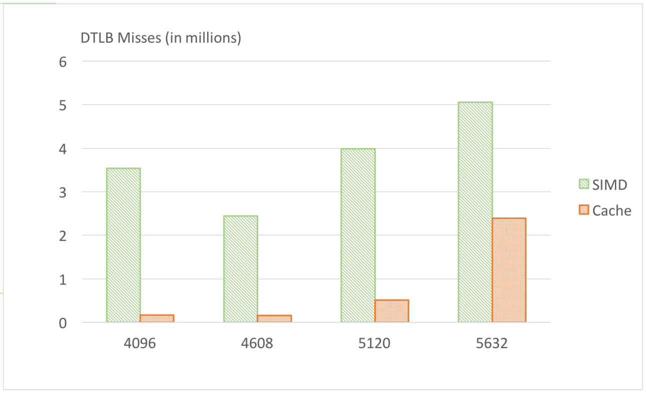

Fig. 13. Reduction in D-TLB misses in Cache Optimized from SIMD Version

number of TLB misses is significantly reduced as we are comparing 4 domain blocks more every time image is traversed and therefore reusing loaded page tables. By analyzing how cache optimization is implemented, we can expect to get reduction of TLB misses up to 5 times depending on image content, as it can be seen in Fig. 13.

5. CONCLUSIONS

We successfully applied all optimizations techniques learned in course and found a blend of them that allowed us to achieved speedup of 14x on average and performance of 18 ops/cycle (which is slightly under theoretical peak performance). Our approach of compensating cache misses with wasted computations gave promising speedup when combined with other optimization methods.

6. REFERENCES

[1] E.W. Jacobs R.D. Boss, *Fractals-Based Image Compression*, NOSC Technical Report 1315, 1989.

[2] Texas Instruments Europe, *An Introduction to Fractal Image Compression*, 1997.

[3] Barnsley M., *Fractals Everywhere*, Academic Press. San Diego, 1989.

[4] *Fractal Image Compression Implementation*, http://www.kennberg.com/projects/projects/fractals/.

[5] *Perceptual Image Hashing*, http://www.phash.org/.