FRACTAL IMAGE COMPRESSION OPTIMIZED ON MODERN X86 ARCHITECTURES

Stefano Boschetto, Gianluca Lain, Alessia Paccagnella, Andrea Ziani

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Fractal compression is a lossy compression scheme that leverages the fact that within an image there are a lot of redundancies. These repeated patterns can replace similar areas in the image, resulting in a reduction of the total image size. However, this algorithm requires a mutual comparison between image parts through a prolonged and computationally expensive process. In this paper, we propose a fast implementation of fractal compression, which leverages Intel's AVX2 intrinsics and improves the locality of data by reducing cache misses. We present different optimized versions of the algorithm, showing the performance gain that we can achieve with each of them. In particular, our best optimization achieves, on average, a performance speedup of 3.4x compared to the naive implementation, resulting in a runtime speedup of 10 times.

1. INTRODUCTION

Within the last few decades, the development of new capacious and fast storage systems partially solved any issue related to storing vast amounts of data. However, moving large quantities of data across the network still poses some problems: the download of large contents (for example, high-quality pictures) will be slower and more expensive than the download of small contents. This fact worsens the user experience, considering users want to have pictures in the greatest detail possible, but at the same time, they do not want to wait long loading times. For these reasons, compression algorithms have been and are still applied today to reduce the amount of data stored and transferred across the network. With regards to images, the focus of the compression algorithms is to obtain an optimal trade-off between quality and compression ratio.

Among the many compression schemes developed, one algorithm which has been studied and extensively discussed in the past is fractal compression. First promoted by M. Barnsley at [1], fractal compression is a lossy compression scheme based on fractals that leverages the fact that image data is highly correlated, i.e., there are many redundancies in image data that compression algorithms can remove without affecting the image quality too much. Furthermore, the human eye is not perfect: it can tolerate small errors in images and is less sensitive to specific components of an image than others. By taking advantage of these two factors - physiology of the human visual system and the nature of image data - image data can be compressed significantly with acceptable reconstructed image quality.

On the one hand, an advantage of this scheme is that it provides a high compression ratio maintaining a decent amount of detail and quality of the image, especially for large images with repetitive content as indicated in [2]. On the other hand, as explained in [3, 4], this algorithm requires a mutual comparison between image parts in a very computationally expensive and slow encoding process.

Contributions: In this paper, we propose a fast implementation of a fractal compression algorithm featuring quadtree decomposition, tuned on Intel's modern x86 architecture using SSE3 and AVX2 intrinsics. Our improvements decrease the execution time of the algorithm of 10 times compared to a naive `C/C++` implementation.

Organization: In Section 2, we overview the necessary background of fractal compression, and give an analysis of the operations involved. In Section 3, we describe step by step the optimization methods we used to reach a 10x runtime speedup. In Section 4, we present the results achieved by running our optimized algorithm using an Intel Skylake processor. In Section 5, we conclude the research summarizing the contributions of our work.

2. BACKGROUND

In this section, we define the main idea behind fractal compression, explaining in detail the naive compression algorithm we optimize. Then, we present a cost analysis in terms of arithmetic operations.

2.1. Fractal compression algorithm

Fractal compression is a technique that leverages the property that small parts of an image, after appropriate affine

transformations, are similar to other parts of the same image. The resulting set of transformed parts does not form an exact copy of the original image. The applied transformations could introduce some distortions and aberrations in the new reconstruction of the original picture. Such distortions are unavoidable in lossy high-ratio image compression. The objective of an ideal compression algorithm is to minimize them without affecting the achieved compression ratio. The fractal compression algorithm described below is a slightly modified version of the *Fractal block-based coding* described by Jacquin, Fisher and Öztürk in [5, 6, 7].

Initialization: as shown in Figure 1, to encode an image *F* of size $W \cdot H$, we divide *F* into non overlapping squared blocks of dimensions BxB called *range blocks*. We define this image as *Range*. We then create a new image, called *Domain*, by down-sampling *F* by a *scaling factor* S_F (reduction by pixel averaging), obtaining a new image of size $\frac{W}{S_F} \cdot \frac{H}{S_F}$. The Domain is divided into blocks of dimensions BxB, called *domain blocks*.

Codebook creation: we apply a set of affine transformations to every domain block, we perform nearest-neighbor interpolation to the transformed blocks, and we add the results to a new structure called *Codebook*. The transformations used are rotations, translations, shears, and scalings. At the end of this step the Codebook consists of $\frac{\frac{W}{S_F} \cdot \frac{H}{S_F}}{B \cdot B} \cdot N_T$ blocks of size BxB, where N_T is the total number of transformations.

Compression procedure: for each range block R_i we search for a matching block C_j among all blocks in the Codebook, such that R_i is similar to C_j. We evaluate the similarity between R_i and C_j by calculating a distortion measure, namely the Squared Euclidean Distance d:

$$d(R_i, C_j) = \sum_{k=0}^{B} \sum_{w=0}^{B} (R_{i,kw} - C_{j,kw})^2$$

The smaller the distortion, the more alike the two blocks are. We choose the block C_j that has lower d, as long as it is lower than some specified tolerance γ. If the best calculated d is greater than γ, we apply a partitioning technique called **quadtree**. In a quadtree partition, the range block R_i is split up into four equally sized sub-blocks, each of size $\frac{B}{2} \cdot \frac{B}{2}$. For each of the newly considered blocks, we repeatedly search over the blocks (of equal size) of the Codebook, to find a d lower than γ. This process recursively repeats until all blocks of the Range match with a block of the Codebook. To encode the entire image, we have to store the following information: the Domain, the lists of transformations, and one mapping for every R_i to the matching

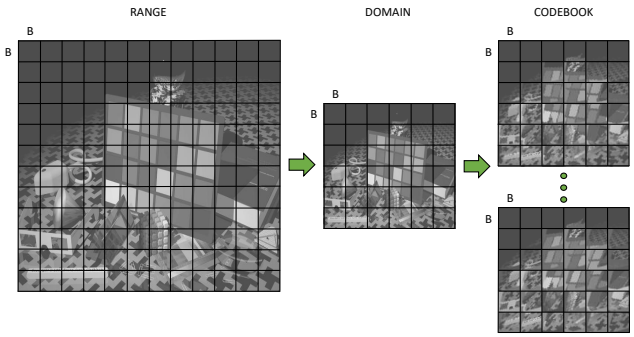

Fig. 1. Visual representation of the structures involved in the encoding phase. From left to right, we can see Range, Domain, and Codebook divided into blocks of size BxB.

C_j. The mapping contains the coordinates of the block D_w of the Domain corresponding to the chosen block C_j, and the transformation T_k used to derive C_j from D_w, so that $C_j = T_k(D_w)$.

Decompression procedure we can reconstruct the original Range of size $H \cdot W$ starting from the stored Domain of size $\frac{W}{S_F} \cdot \frac{H}{S_F}$. Each block R_i can be reconstructed by applying the corresponding stored mapping: $R_i = T_k(D_w)$. All the information to perform this operation is stored in the encoding phase.

2.2. Cost analysis

The number of operations of our algorithm does not depend uniquely on the image size, but also the image content. By analyzing the algorithm, we can state that it uses both floating-point and integer operations: the Codebook creation and the decompression consist mainly of floating-point operations, while the compression operates only with integers. We can define the cost measure $C(N)$, based on the parameter $N = H = W$, as follows:

$$C(N) = (i_{add}(N) + i_{sub}(N) + i_{mul}(N) + fp_{add}(N) + fp_{mul}(N) + fp_{div}(N))$$

where i_{add}, i_{sub}, i_{mul},fp_{add},fp_{sub},fp_{mul}, fp_{div} is equal to the cost of the corresponding operation multiplied by the number of times the algorithm performs that operation.

We can define the cost for the algorithm more precisely by giving an upper bound on the number of operations. Since the number of operations depends on the image content, the upper bound considers that the *quadtree* search is performed until the maximum depth level for every range block. We can divide the costs into two main categories:

Creation of the Codebook:

$$C(N) = \sum_{t=0}^{N_T} \sum_{y=0}^{N} \sum_{x=0}^{N} (6 \cdot C_{fp\_add} + 6 \cdot C_{fp\_mul} + 2 \cdot C_{fp\_div}) \quad (1)$$

or more intuitively: for every pixel of the Range, we perform six fp_add, six fp_mul and two fp_div for every affine transformation.

Compression:

$$C(N) = \sum_{i=0}^{2} \sum_{y_R=0}^{\frac{2^i N}{BH}} \sum_{x_R=0}^{\frac{2^i N}{BW}} \sum_{y_C=0}^{\frac{2^i N \cdot N_T}{BH}} \sum_{x_C=0}^{\frac{2^i N}{BW}} \frac{32}{2^i} (C_{i\_add} + C_{i\_mul} + C_{i\_sub}) \quad (2)$$

or more intuitively: for every range block, we perform $\frac{32}{2^i}$ int_add, int_sub and int_mul for every block of the Codebook. The sum over i describes the maximum *quadtree* level: from a maximum size block of 32x32 to a minimum of 8x8.

2.3. Roofline Plot

Before presenting the optimization steps, it is useful to visualize the *Roofline plot*, to understand whether the algorithm is memory or compute-bound. For our reasoning, we assume the worst-case scenario: the range blocks are always in the cache (since they are only read once and periodically reused), while the Codebook never fits entirely in the cache. To understand the calculations that we are going to observe to outline the plot, we consider only the *compression* phase described in Equation 2. This assumption is justified as it contributes to the vast majority of the computational effort of the entire algorithm. For a codebook block of size BxB bytes, we perform $3 \cdot B \cdot B$ operations (since on every pixel, we make 1 add, 1 mul and 1 sub) to compare it with a range block. This set of operations gives us a lower bound on the operational intensity I of 3 ops/byte (since every pixel weighs 1 byte).

To perform the benchmarks, we operate on hardware based on Intel Skylake architecture. We consider only ports with integer capabilities and, as described in [8], this architecture presents 4 scalar integer ports and 3 SIMD integer ports. To draw the *Roofline plot*, we need the peak performances π for scalar and SIMD operations. The maximum number of scalar operations is 4 ops/cycle since we have 4 ports. On the other hand, the maximum number for SIMD operations is $32 \cdot 2 + 2 = 66$ ops/cycle, since we have 2 V_ALU that perform 32 ops/cycle (as the operation that performs more ops/cycle based on its throughput is `_mm256_subs_epu8`) and 2 S_ALU that execute 1 ops/cycle.

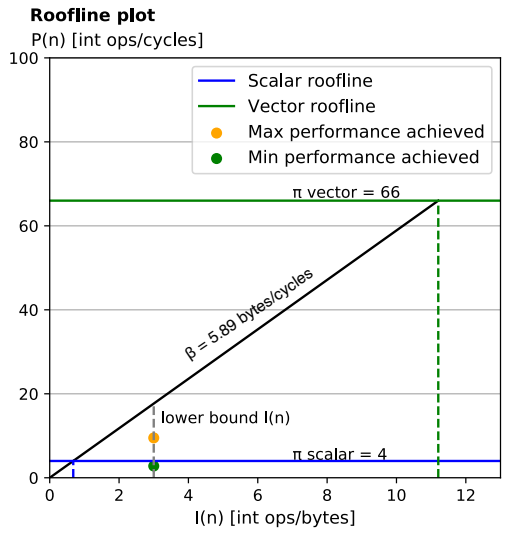

Fig. 2. Roofline plot for scalar and vector peak performance. Includes the maximum performance achieved using SIMD operation, and the minimum performance of the scalar algorithm.

Therefore we can define $\pi_{\text{scalar}} = 4$ and $\pi_{\text{SIMD}} = 66$ ops/cycle. Finally, to complete the *Roofline plot*, we found the bandwidth of our system: 15.89 GB/s @ 2.7GHz leading to a bandwidth $\beta = 5.89$ bytes/cycle. The *Roofline plot* is showed in Figure 2.

3. PROPOSED OPTIMIZATION METHOD

In this section, we delineate which are the computationally expensive parts of the algorithm, and we describe step by step, the optimization methods we used to reach 10x runtime speedup. We describe all the optimizations we used to improve the performance by dividing them into non-SIMD, SIMD, and cache optimizations.

Assumptions: Our implementation takes some assumptions into consideration.

First, every input image *F* is in the Bitmap format (BMP). *F* is an 8-bit greyscale image, so we work with only one color channel. We examine square input images, where height and width are divisible by 256.

We consider a limited set of range block sizes: starting from a maximum of 32x32 pixels, to a minimum, with *quadtree* partitioning, of 8x8 pixels.

Fractal compression exploits affine transformations by applying them to the domain blocks for the creation of the Codebook. Our choice consisted of four rotations, four translations, four shears, and four scaling, as they empirically gave us a good trade-off between computational burden,

compression ratio, and image quality. Our choice of optimizations consists of three different groups, each of them achieving the same final result. We describe all the different optimization steps in the next paragraphs.

3.1. Basic optimization (non-SIMD)

By profiling our base implementation, we detected that a few functions were responsible for 99% of our software execution time. We can divide our implementation into two predominant categories in terms of execution time: compression and Codebook creation. On the one hand, we can summarize the compression with the following functions:

- *compress(·)* which iterates over all range blocks and calls *quadtree* recursively on every of them;
- *quadtree(·)* which calls *euclidean_distance* on a pair of range and domain blocks. If the resulting distance is less than a fixed threshold tolerance, it calls itself recursively on blocks of sizes $\frac{B}{2}$;
- *euclidean_distance(·)* which takes as input two blocks of the same size and performs the Euclidean distance as described in Section 2.

On the other hand, the Codebook generation predominantly executes the function:

- *affine_transform(·)* which applies an affine transformation to a block of pixels (pixel transposition via coordinate-matrix multiplication).

Compression operations primarily consist of searching a best-similarity match between blocks from Range and Codebook. We compare every block of the Range with each block of the Codebook, with fixed considered block sizes (8x8, 16x16, and 32x32). Due to this and the fact that the Codebook and Range grow linearly with the input size (Bytes), the computational cost of compression grows proportionally to the square of the input size.
We applied several basic optimizations in this phase, and we achieved decent results compared to the baseline algorithm. Since the cost of compression operations are predominant, our primary objective in this phase was to reduce the redundancies and bottlenecks associated with this part of the algorithm. In all the functions, we simplified conditional statements and expressions, precalculating loop invariants, and merging conditional branches that were redundant. Then, we checked the consistencies among variable types to reduce (or entirely avoid) expensive type conversions. As a secondary objective, we aimed at reducing the overall reliance on immutable data and return-by-copy, allowing a sharp reduction of expensive memory system calls, e.g., *malloc, free*. To achieve this objective, we replaced expensive dynamic memory allocation with fixed-size arrays. Moreover, we prefer modifying data *by-reference* instead of creating local copies of data that we will subsequently assign to the old structures. We achieve better performance by avoiding system calls at the cost of the application's memory footprint. Finally, we applied loop unrolling in the functions where operations in the inner loops were reused among iterations. In this way, we could precalculate and save operations outside the inner loop, reducing unnecessary operations. Combined with this optimization, we also applied scalar replacement. Thus, we saved in local variables the data that was fetched several times from memory. Finally, we transformed the function *quadtree* from a recursive implementation to an iterative one that leverages queues.

3.2. SIMD Optimization

The computation of Euclidean distances between pixel blocks was an optimal candidate for a refactoring focused on vector instructions. The homogeneous and regular structure of the Euclidean distance computation allows extensive use of AVX2 instructions, in particular in the computation of:

- **Pixel distances**: loading 8 pixels from the range block, and 8 pixels from the codebook block per iteration allows us to perform 8 subtractions, with a single AVX2 instruction `_mm256_sub_epi32`.
- **Distances squaring**: pixel distances can be squared using the AVX2 instruction `_mm256_mullo_epi32` by multiplying the distances by themselves. In this way, we can perform 8 squaring with a single instruction. The final squared distances are then added to a vector using the instruction `_mm256_add_epi32`.

Despite performing many more operations simultaneously, loading into the AVX registers the same range block several times (once every Euclidean distance) caps the possible performance gain. This limitation is given by that, in the worst-case scenario, the same block is divided into sub-blocks by the *quadtree* partitioning, and each sub-block is loaded into the registers as many times as the number of Euclidean distance performed. The description of how we improved this SIMD optimization, taking care of the data movements from memory, is described in the next paragraph.

3.3. Cache optimization on SIMD

Caching effectiveness on our unoptimized implementation is highly dependent on input image size. If the image is small enough that the entire Codebook fits in cache, then even with a naive implementation, we can achieve a respectable cache hit rate. Indeed, the most data-intensive operation in our algorithm is the similarity check between

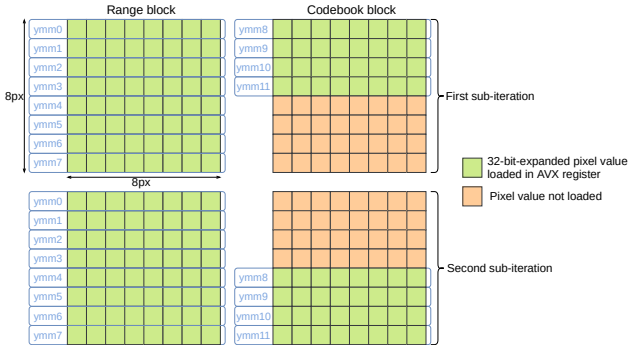

Fig. 3. Visual representation of the memory-optimized similarity comparison between one 8x8 range block and one 8x8 codebook block.

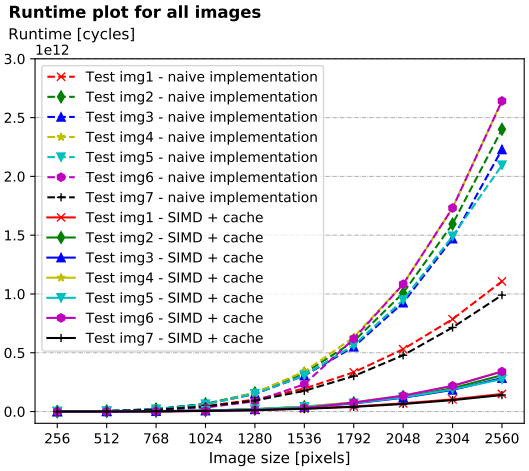

Fig. 4. Runtime plot obtained by running the naive implementation and the SIMD with cache optimization on all the test images from the resolution 256x256 up to 2560x2560.

every block R_i from the Range R and every block C_j from the Codebook C:

$$\forall R_i \in R, C_j \in C \text{ similarity}(R_i, C_j)$$

Such operation achieves a reasonable cache hit rate if every block from the Codebook is transferred from memory to the cache only once for every range block similarity computation. On modern processors, this is almost always the case since even L1 cache is about 128KB, which comfortably fits a row of codebook blocks and prevents their premature eviction.

Our proposed SIMD/cache optimization adaptively loads pixel vectors in xmm/ymm registers, and appropriately selects which distance measuring operations are most efficient for the current depth of *quadtree* and the resulting block size. We implemented three variations of Euclidean distance, one for each considered block size (32x32, 16x16 and 8x8). For the two largest sizes, we used respectively 128 bit `xmm`, and 256 bit `ymm` vectors to load 16 and 32 pixel values per iteration. For the smaller 8x8 blocks, we unrolled the block distance computation and preemptively loaded 8 vectors of 8 32-bit-expanded pixel values (range block) before iterating over all codebook blocks and loading half-block per sub-iteration (see Figure 3).

A small adaptation in our memory allocation procedure for Codebook and Range allowed the sole use of fast aligned vector-load instructions (`_mm256_load_epi64` and `_mm_load_epi64`). Allocating 32-Byte aligned and oversized memory areas for Codebook and Range improved our cache-optimized implementation performance by 3.5% and lowered its execution time by 3.2%.

We applied a different cache optimization to the Codebook creation procedure. In the naive algorithm, we apply a transformation to all domain blocks before moving to the next transformation. This iteration order leads to a high cache miss rate because every block must be loaded once per transformation. Thus, we decided to swap the iterations order: we apply all the affine transformations to a cached domain block before moving on to the next one. This improvement helps only if the Domain cannot fit entirely in the L1 cache. Conversely, the naive algorithm would lead to the same number of cache movements of the optimized version. If the Domain is larger than L1 (for an input image size of 2048), the optimized version always keeps each block in L1, while the naive algorithm has to swap between L1 and L2 levels.

4. EXPERIMENTAL RESULTS

In this section, we describe the different classes of experiments we performed, giving empirical results obtained in our benchmarking architecture.

4.1. Benchmarking architecture

We use the following platform to run the benchmarks: **Intel Core i7-6820HQ @ 2.70GHz** with respectively **256KB, 1MB and 8MB of L1, L2 and L3 cache**. We perform all tests on Arch Linux running on Linux kernel 5.6.12.

We used two compilers: GCC 9.3.0 and ICC 19.1.1.219. We created an infrastructure that compares hashes of the resulting images to the hash of the baseline algorithm, proving the correctness of the optimized versions. All tests are run on executables compiled with flags `-O3 -march=native`. We created a benchmark infrastructure that measures execution time automatically in both CPU cycles and milliseconds. We measured cache misses with the help of Linux `perf` utility. All benchmarks consider only the compression procedure, which includes the Codebook creation. To

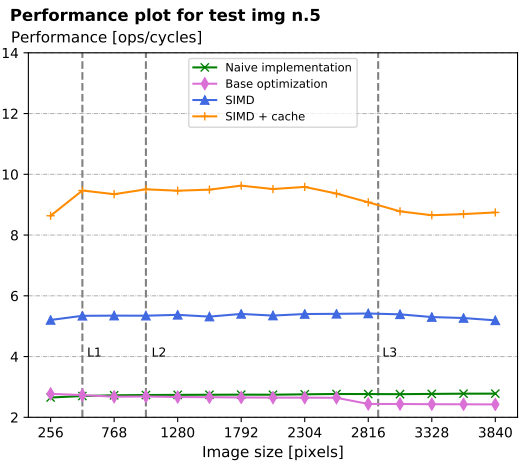

Fig. 5. Performance plot obtained running all the different versions of the algorithm for the average image from the resolution 256x256 up to 3840x3840. Dotted gray lines indicate levels of cache L1, L2, and L3.

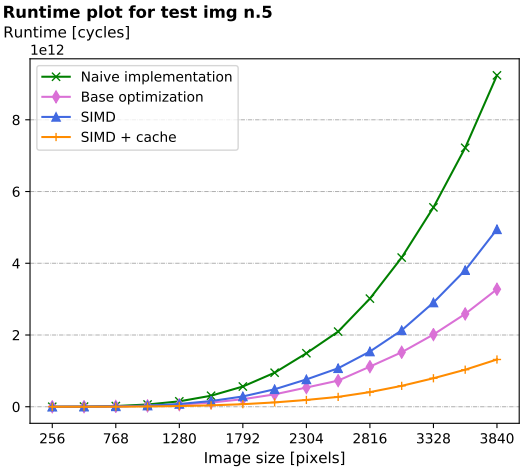

Fig. 6. Runtime plot obtained running all the different versions of the algorithm for the average image from the resolution 256x256 up to 3840x3840.

test the algorithm, we use a set of 7 publicly available images that are common in the field of compression benchmarking [9]. For each image, we created a copy at 15 different resolutions, such that $N = 256 * i \ \ \forall i = \{1, .., 15\}$, where $N = H = W$ as discussed before.

4.2. Runtime and performance plots

As we explained before, the number of operations in the algorithm (due to *quadtree partitioning*) depends on the image size and the image content. In Figure 4, we show the runtime plot over the seven images of our test set at ten different resolutions. As we can see the difference is substantial across all the images even after we applied our improvements. We analyzed the images and we found out that images 1 and 7 (the ones with lowest runtime) have approximately 30% of the image as solid background and contains repetitive patterns. Thus, it is easier for the *quadtree partitioning* to maintain a large block size, since one block can encode all the similar blocks of the background. On the other hand, images 2 and 6 contain harder pattern to encode, such as the fur of an animal, grass, trees and elements with high contrast. For this reason, from our test set, we decided to pick an "average" image to plot all the subsequent results. The image chosen is number 5.

In Figure 5, we show the performance plot referred to this image. Observing the Roofline plot in Figure 2, we can understand, based on our empirical lower bound on operational intensity, that the algorithm is compute-bound for scalar operations, while memory-bound leveraging SIMD instructions.

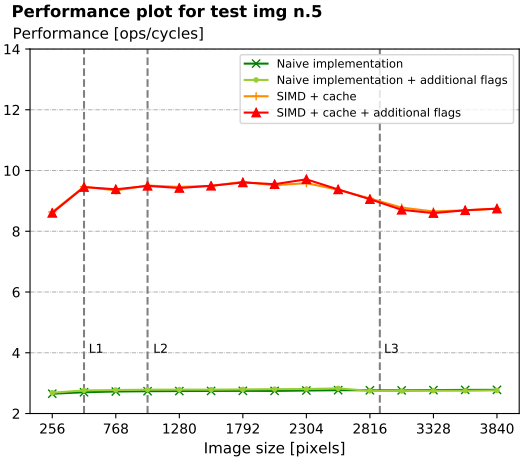

Fig. 7. Performance comparison among the naive implementation and the SIMD with cache optimization compiled with and without additional flags. Dotted gray lines indicate levels of cache L1, L2, and L3.

For this reason, we expect that the SIMD optimization without introducing improvements in data movements would not bring substantial benefits, as it is memory-bound. As we can see from the Figure 5, the SIMD version reaches a performance of approximately only 5 ops/cycle.
Contrary, our cache optimized version achieves a performance of almost 10 ops/cycles, which is 3.4 times more than the naive implementation. We are far from reaching the peak performance of 66 ops/cycle since our application is memory-bound and highly dependant on sequential operations (with intrinsically lower peak performance). It is also interesting to notice that the performance decreases as

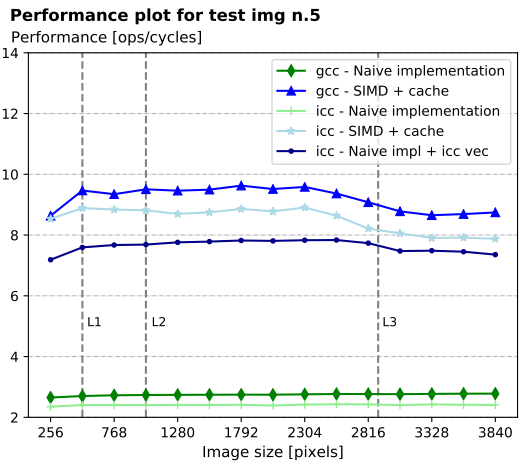

Fig. 8. Performance comparison among the naive implementation and the SIMD with cache optimization compiled with GCC and ICC. Dotted gray lines indicate levels of cache L1, L2, and L3.

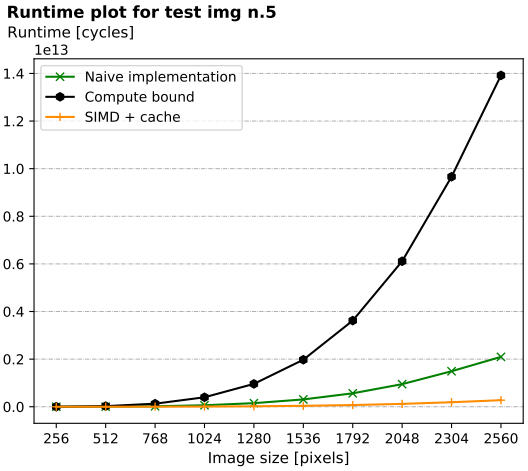

Fig. 9. Runtime plot obtained running the naive algorithm, the compute-bound algorithm, and the SIMD with cache optimization for the average image from the resolution 256x256 up to 2560x2560.

the Codebook cannot fit in the cache, and it has to be loaded from the RAM. This happens when the image size is close to 2896x2896 pixels since the Codebook size is approximately 8MB (equal to the L3 cache of our system). Indeed, as long as the Codebook fits into the level L1, we see that the performance increases because it can reuse the blocks that reside in the fastest cache. When the Codebook is large enough not to fit any more in L1, the performance slightly decreases or remains pseudo constant. Conversely, when the image size is close to 2896, the Codebook does not fit in the cache, and the performance drops.

4.3. Basic optimizations help the runtime but not the performance

As may be noticed in Figure 5 and 6, the basic optimizations do not increase the performance, but they significantly reduce the runtime of the algorithm. This behavior is caused by the optimization we introduced in the function *euclidean distance*, which interrupts the calculation if the partial distance computed is already higher than the best distance found so far. The improvement over runtime is distinctly visible on the runtime plot in Figure 6.

4.4. More compiler flags do not help performance

As explained in Section 4.1, we compiled all benchmark with flags `-O3 -march=native`. We included other different flags to the compilation and we can see the results in Figure 7. We included the flags `-funroll-all-loops -ffast-math -funsafe-math-optimizations -ftree-vectorize`. However we did not experienced significant changes compared to the previous version. We compared also the runtime between the two versions and also in that scenario we did not see any improvement. Thus, we decided to keep only the flags `-O3 -march=native` for all subsequent benchmarks.

4.5. GCC vs ICC

We compiled the code of every version of the algorithm using two different compilers: GCC and ICC. For both of them, we applied compilation flags `-O3 -march=native`. When we run the scalar code, we added `-no-vec` and `-fno-tree-vectorize` respectively to disable auto vectorization.
As evident from the plot in Figure 8, both the scalar and the vector optimizations have better runtime if compiled with GCC. Also, we tried to automatically vectorize the code of the naive implementation with the ICC compiler, and we discovered that this produces a runtime not far from our cache optimized implementation compiled with ICC, especially for large images. This result is most likely since we vectorize a single function while compiling with ICC, the whole program undergoes the automatic vectorization.

4.6. New implementation without saving the Codebook

Instead, every time it reads a domain block, it derives on the fly all the transformed versions of that block. Then, the considered range block is compared to the transformed blocks just computed. This approach brings the operational intensity to a value of $14 * 16 + 3 = 227$ ops/byte since, for every pixel in a domain block, we perform 14 operations (as

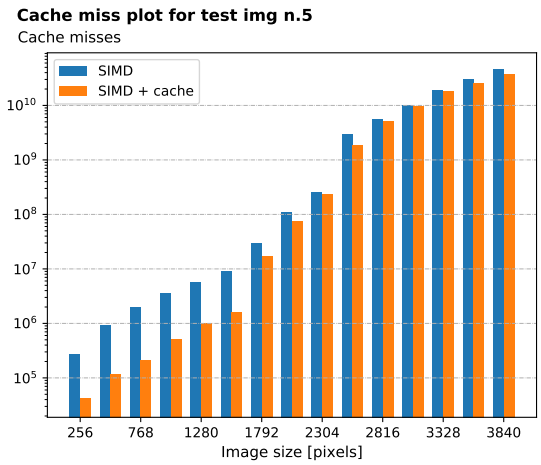

Fig. 10. Comparison between the number of cache misses before and after the cache improvement on the SIMD optimization.

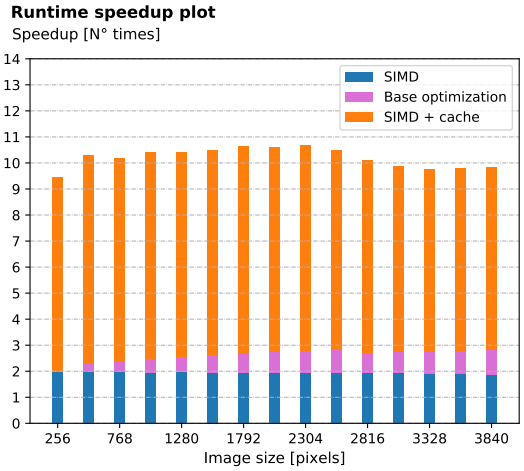

Fig. 11. Contribution in runtime speedup of each optimization on the average image (test image n.5) from the resolution 256x256 up to 3840x3840.

described in Equation 1) for a total of 16 transformations, plus the 3 operations described in Equation 2. This lower bound on the operational intensity brings the algorithm on the compute-bound area. Figure 9 shows the comparison of the runtime between the new version described above and the version using SIMD and cache optimization. Note that the new version has all the optimizations contained in the other one, and the only change is the process just explained. As we can see, the new version is awfully slow, and on average, it is 50 times slower than SIMD with cache optimization. Therefore, we discarded this idea, keeping the old version of the algorithm to try the other improvements.

4.7. SIMD needs cache optimizations

As we may see from Figure 5 and 6, the simple usage of vectorization does not significantly improve the performance, and the runtime is even worse than applying basic optimizations to the naive algorithm. This behavior is caused by the overhead introduced by moving data from memory to cache, and from the cache to the AVX registers. Applying the memory optimization described in Section 3.3, we notably reduced the number of cache misses, as shown in Figure 10. Also, the performance increased up to a maximum of almost 10 ops/cycle, as displayed in Figure 5. Furthermore, the histogram in Figure 11 shows how the different optimization techniques we used contribute to speed up the runtime. We can notice that the combination of SIMD operations with memory optimizations allowed us to achieve a speedup of 10x from the naive algorithm on the average image.

4.8. Optimization merging is sublinear

The last version of the algorithm we produced consists of the merge of all the different optimizations we described. As mentioned in the previous paragraphs, our bottleneck is the computation of the Euclidean distance, and our algorithm is massively memory-bound. Thus, every optimization not directly involved with that part generates improvements that might not increase the overall performance. As we have no other memory optimization outside the SIMD with cache optimization, combining all the improvements does not change the bottleneck. For this reason, the merge of all the optimizations does not linearly grow the performance leading to approximately the same results as our SIMD version with cache optimization.

5. CONCLUSIONS

In this paper, we presented a high-performance implementation of fractal compression with adaptive quadtree partitioning and Euclidean distance similarity measurement. We showed how a combination of SIMD vector instructions and cache optimizations could lead to a performance improvement of 3.4x on average compared to a naive implementation, with a maximum achieved performance of 10 ops/cycle. Cache optimizations brought the majority of the performance improvement since our version of the algorithm is massively memory-bound. Further improvements in the implementation efficiency are possible but necessitate a more modern architecture supporting AVX-512 instructions. Such an extension of the instruction set and register space would allow us to use the memory optimization, evidenced in Section 3.3, even for block sizes of 32x32 and 16x16 pixels.

6. CONTRIBUTIONS OF TEAM MEMBERS

For sake of simplicity, we denote the optimization versions as follows:

- **Basic optimization** → Variant 1
- **SIMD optimization** → Variant 2
- **Cache optimization** → Variant 3
- **Compute bound version** → Variant 4
- **GCC, ICC and compiler flags** → Variant 5

Stefano Boschetto: Unrolling and scalar replacement in function *euclidean_distance* Variant 1 to minimize conditional loops. Pipeline improvements by operation reordering in function *euclidean_distance* Variant 3 to reduce vector unpacking operations. Implementation of strategy pattern in *similarity* and *euclidean_distance* Variant 3 to adaptively handle different block sizes and reduce XMM/YMM registers flush/reloads. Caching and memory usage improvements by reducing memory allocations in function *similarity* Variant 2. Evidence-based GCC flag selection in Variant 5.

Gianluca Lain: worked on function *create_codebook* of Version 1. Worked with Andrea on Variant 1, converting the recursive function *quadtree* to an iterative one (queue based). Worked with Andrea on Variant 3 to create the Codebook in a cache optimized way. Created the new version of the algorithm of Variant 4, trying with both scalar and SIMD operations. Worked with Alessia on function *nearest_neighbors* of Variant 1, adding loop unrolling and simplifying expressions and conditional checks. Worked with Andrea to remove `malloc` on all functions of Version 1 and replacing them with fixed size array.

Alessia Paccagnella: Worked with Gianluca on function *nearest_neighbors* in Variant 1, doing loop unrolling, merging redundant conditional checks and eliminating dynamic allocations. Worked on the function *euclidean_distance* in Variant 1 to avoid variables dependencies. Worked with Stefano on the vectorization of the function *euclidean_distance* for the different cases of range blocks of size 32x32 and 16x16 in Variant 3.

Andrea Ziani: Worked on function *affine_transform* of Variant 1 performing scalar replacement, loop unrolling and removing operation redundancy. Worked with Gianluca on Variant 1 to convert the recursive function *quadtree* to an iterative one using a queue, and to replace dynamic memory re-allocations with static arrays. Worked on Variant2 to vectorize the function *euclidean_distance* without cache optimization. Worked with Gianluca on Variant 3, create the Codebook in a cache optimized way. Worked on Variant 5 trying to compile with 2 compilers (icc and gcc) and different flags every optimization we made.

7. REFERENCES

[1] Michael Barnsley, *Fractals Everywhere*, vol. 97, 01 1989.

[2] Dietmar Saupe, Raouf Hamzaoui, and Hannes Hartenstein, "Fractal image compression - an introductory overview," in *SIGGRAPH 1997*, 1997.

[3] Guojun Lu, "Fractal image compression," *Signal Process. Image Commun.*, vol. 5, pp. 327–343, 1993.

[4] Texas Instrument Europe, *Introduction to Fractal Image Compression*, 1997.

[5] Arnaud E Jacquin, "A novel fractal block-coding technique for digital images," in *International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 1990, pp. 2225–2228.

[6] Yuval Fisher, "Fractal image compression," *Fractals*, vol. 2, no. 03, pp. 347–361, 1994.

[7] Aydın Öztürk and Cengiz Güngör, "A fast fractal image compression algorithm based on a simple similarity measure," *Information System Sciences*, vol. 36, pp. 159–178, 01 2011.

[8] Agner Fog, "Instruction tables," Tech. Rep., 08 2019.

[9] Niels Fröhling Pete Fraser Tony Story Wayne J. Cosshall David Coffin Bruce Lindbloom Axel Becker, Thomas Richter, "Image compression benchmark image set," http://imagecompression.info/, 2015.