PERFORMANCE OPTIMIZED FRACTAL IMAGE COMPRESSION WITH QUADTREE PARTITIONING

Jonas Hansen, Pascal Huber, Fabian Wüthrich

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Fractal image compression is a lossy image compression method, which yields good compression ratios and quality at the cost of high encoding times. We implemented the algorithm from scratch using quadtree partitioning and exhaustive search for self-similarity in the image.

This baseline implementation was analyzed using the roofline model and profiling to identify bottlenecks. With these insights we implemented precomputations and removed complex data structures to improve performance. In addition, we optimized the code for instruction level parallelism for a better pipeline utilization. Then we vectorized the code using AVX2 intrinsics and changed the memory layout to avoid expensive gathering instructions. At last, different compiler flags for GCC and ICC were tested.

The optimized version without vectorization led to a 4x increase in performance whereas a speedup of up to 8x was observed with SIMD. Additionally, the runtime of the compression was reduced significantly.

1. INTRODUCTION

Human perception relies heavily upon visual information, particularly images. This medium is very powerful in gaining and conveying *ideas*, memories, and emotions. No one would doubt the saying "A picture is worth a thousand words". Especially with the widespread use of the internet and the advent of social media the ability to share and store images efficiently is crucial.

A practical way to store images without occupying too much space is to use lossy image compression which decreases the image quality in places where a high-quality image is not perceivable or where an image of lower quality is not disruptive. JPEG, whose basic building block is the discrete cosine transform, is undoubtedly the most widely used lossy compression scheme. With a good compression quality and fast compression times, JPEG hits a sweet spot for many practical applications.

Fractal image compression is another method for lossy image compression. The main idea is to compress an image by exposing self-similarity which can be observed in many places in nature (e.g. fir cones or romanesco broccoli). Contrary to JPEG, the underlying theoretical construct of a fractal compression scheme is that of an iterated function system (IFS).

This approach yields great compression results in terms of compression size and quality [1]. However, it is computationally expensive to encode images because the algorithm involves an exhaustive search over different regions of the image with many numerical computations to find self-similarity. Optimizing these computations is therefore crucial for an efficient implementation of the algorithm.

Related work. The most widely known practical fractal compression scheme was developed and patented by Michael Barnsley and Alan Sloan in 1987. They published a paper about their work in 1989 [2]. Barnsley's graduate student Arnaud Jacquin was the first who implemented a practical version of it in 1992 in his PhD thesis [3]. Numerous improvements and variations have then been developed to this original approach, e.g. archetype classification ([4], [5]) which decreases the exhaustive search space for self-similarity. Yuval Fisher published a book in 1995 with a detailed description of various fractal schemes and an elaborate list of optimizations [1].

Contribution. Based on the explanations of Fisher in [1] and an open-source implementation written in Python [6] and C++ [7], we implemented our own fractal image compression scheme with quadtree partitioning and exhaustive self-similarity search. We proceeded with improving the runtime of the compression using several performance optimizations and vector intrinsics.

2. BACKGROUND

To introduce the algorithm, we first present the essential parts of the underlying mathematical theory, show how it can be used to compress images and what practical considerations must be made for the implementation. A cost analysis can be found at the end of this section. Throughout this chapter, we use the notation and results from Fisher [1].

Iterated function systems (IFS). Fractal image compression builds on the theory of iterated function systems.

For completeness, we provide a short introduction to this rich topic.

Definition 1 (Contractivity of Functions)**.** *A function f on a metric space X with metric d is contractive, when there exists some $0 \leq k < 1$ such that for all $x, y \in X$: $d(f(x), f(y)) \leq k \cdot d(x, y)$.*

As an informal example, when f is a contractive function on $\mathbb{R}$, then mapping two numbers brings them *closer* together, i.e. their distance gets smaller.

Definition 2 (Iterated Function System)**.** *An iterated function system (IFS) is a set of n functions*

$$\{f_i : X \to X | i = 1, \ldots, n\}$$

where each f_i is contractive and X is a metric space.

Definition 3 (Hutchinson Operator)**.** *The Hutchinson operator for an IFS is the function*

$$F\colon 2^X \to 2^X$$
$$x \mapsto \bigcup_{i=1}^{n} f_i(x)$$

With $F^{\circ n}(x)$, we denote the iterative application of F n times on its input x, e.g. $F^{\circ 2}(x) = F(F(x))$.

Definition 4 (Attractor of an IFS)**.** *Let $A \in X$. When $F(A) = A$, then A is called an attractor.*

Hutchinson used the following theorem in his work [8] to build up the essential theoretical parts of how fractal image compression works.

Theorem 1 (Contractive Mapping Fixed-Point Theorem)**.** *For an IFS on a compact set X it holds that:*

1. *There always exists a unique attractor $A \in X$*
2. *For any nonempty set $S_0 \subset X$ it holds that $A = \lim_{n\to\infty} F^{\circ n}(S_0)$*

From IFS to image compression. The key idea in fractal image compression is that one computes a set of contractive functions $w_i \in \boldsymbol{W}$ (in this context called transformations) on the image. From theorem 1, we know that some unique attractor A exists. If the transformations $\boldsymbol{W}$ are chosen in a way such that the attractor A is the image to be compressed, one only needs to store the transformations. For decompression, one can iteratively apply all transformations on any initial starting image, which then converges towards the original image.

A transformation w_i is defined on a contiguous source region (called domain block) and a contiguous target region (range block) of the image. The transformation maps the pixels of the domain block to the pixels of the range block and then applies a brightness and saturation adjustment.

The transformations can then be computed with the following steps.

1. Partition the image into range blocks $\boldsymbol{R}$ and domain blocks $\boldsymbol{D}$.
2. For each range block $R_i \in \boldsymbol{R}$, find a transformation w_i with domain block $D_i \in \boldsymbol{D}$ such that $w_i(D_i)$ approximates the pixel values of block R_i best.

Finding the best transformation by comparing all domain/range block pairs is called exhaustive block mapping. More elaborate versions like archetype classification restrict possible domain block candidates for a given range block to speed up compression [4].

Transformations. A grayscale image can be interpreted as a function

$$f\colon I^2 \to I$$
$$(x, y) \mapsto z = f(x, y)$$

where I is the interval $[0, 1]$. Specifically, an image is a set of three-tuples $(x, y, z) \in I^3$, where x, y are positions and z is the grayscale value. The image of a range (or domain) block R_i is then defined by $f \cap (R_i \times I)$. A transformation w_i from the image of a domain block D_i to the image of a range block R_i can then be described as a linear affine transformation

$$w_i(D_i) = \bigcup_{(x,y,z)\in f\cap(D_i\times I)} \begin{pmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e_i \\ f_i \\ o_i \end{pmatrix}$$

where $a_i, b_i, c_i, d_i, e_i, f_i$ map *location* (which can involve rotations) and s_i, o_i correspond to contrast and brightness adjustments. To ensure that w_i is contractive, D_i has to be larger than R_i (in terms of region sizes) and $0 \leq s_i < 1$.

Now we define a metric to compare two images (or two blocks). Usually, the root mean square error (RMS) metric is used:

$$d_{RMS}(f, g) = \sqrt{\int_{I^2} (f(x, y) - g(x, y))dxdy}$$

For a range block R_i and domain block D_i we now seek the best transformation w_i:

$$w_i = \arg\min_{w} \left(d_{RMS}(w(D_i), f \cap (R_i \times I))\right) \qquad (1)$$

Calculating w_i is a minimization problem, because we may choose s_i and o_i in order to minimize the error.

The set of all transformations is a partitioned iterated function system (PIFS). The only difference to an IFS is that the transformations are restricted to *blocks* (partitions) of the image. However, the underlying theory still holds [1].

Computing Transformations. In practice, an image consists of pixels, where each pixel has a grayscale value between 0 and 1 and integer coordinates.

The minimization problem in equation (1) can be solved using least squares regression, as done in [6]. Fisher uses an analytical solution in [1] to compute the adjustments directly.

Let $b_1, ..., b_n$ be the n pixel values of the range block R_i and $a_1, ..., a_n$ be the n pixel values of the downsampled and rotated domain block. We then seek s (saturation) and o (brightness) such that

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

is minimal. An analytical solution for s, o, R is:

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

$$o = \frac{1}{n} \left(\sum_{i=1}^{n} b_i - s \sum_{i=1}^{n} a_i \right) \tag{3}$$

$$R = \frac{1}{n} \left[\sum_{i=1}^{n} b_i^2 + s \left(s \sum_{i=1}^{n} a_i^2 - 2 \sum_{i=1}^{n} a_i b_i + 2o \sum_{i=1}^{n} a_i \right) + o \left(no - 2 \sum_{i=1}^{n} b_i \right) \right] \tag{4}$$

Computing these values is the core numerical challenge of the algorithm.

Image Partitioning. There are many ways to partition an image into range and domain blocks. We require that the range blocks cover the full image and do not overlap, otherwise the decompressed image would have uncovered regions. The domain blocks represent the search space for self-similarity and may overlap. Increasing the domain block pool $\boldsymbol{D}$ may lead to superior transformations but can increase the compression time significantly. Blocks can be of various sizes and shapes (quadratic, rectangular, etc.), but domain blocks must be larger than range blocks to satisfy the contractivity requirement. For example, the code in [6], uses quadratic range blocks of size $s \times s$ and domain blocks of size $k \cdot s \times k \cdot s$ for some integer constant $k > 1$.

Quadtree Partitioning. In practice, quadtree partitioning is considered a reasonable reference point for more advanced partitioning schemes [1]. It dynamically adapts range and domain block sizes by a predefined error threshold ϵ. We start with some initial (quadratic) range blocks of size $s \times s$, and domain blocks of size $2s \times 2s$. When there is a range block for which all domain blocks exceed the threshold ϵ, we partition the range block into 4 smaller blocks of size $s/2 \times s/2$ and try to cover them again with domain blocks of size $s \times s$. This scheme has the effect that (potentially larger) homogeneous regions of the image are covered with few transformations while image regions with details are covered with transformations that map smaller blocks. Usually, one defines also a maximum depth of the quadtree as described in [1].

Cost analysis. In this course, we focus on floating point operations and ignore all integer calculations (e.g. index computation). We count only floating point multiplications, divisions, additions, and subtractions because no special mathematical functions are used. The cost metric is defined as

$$C = N_{mult} + N_{div} + N_{add} + N_{sub} \tag{5}$$

The cost of the algorithm does not only depend on the size but also on the content of an image. To obtain a rough estimate of the runtime, we provide a pessimistic upper bound on the cost of the algorithm. We assume that no transformation will yield an error below the threshold and thus the only termination criteria is the maximum quadtree depth. In practice, the cost is (much) lower assuming a reasonable error threshold ϵ.

Let m be the maximum quadtree depth and let s be the width of the square input image. We assume that we start quadtree with range blocks of size $\frac{s}{2} \times \frac{s}{2}$ and one domain block of size $s \times s$. For quadtree depth $i \in \{1, \ldots, m\}$, let $R^{(i)}$ be the set of range blocks, $D^{(i)}$ the set of domain blocks and n_i the number of pixels of one range block. Note that the number of pixels of a domain block is then always $4 \cdot n_i$ due to quadtree partitioning and $n_i = \frac{s^2}{|R^{(i)}|} = \frac{s^2}{2^{2i}}$. For a given domain block and range block in quadtree depth i, let α_i denote the cost to compute four transformations w_i^0, w_i^{90}, w_i^{180}, w_i^{270} and their errors. Each transformation w_i^γ represents a rotation of the domain block by γ degrees.

To calculate α_i, we need to compute all sums for brightness (3), contrast (2) and error (4) and apply the formulas. Besides, we need to downscale and rotate the domain block.

Downscaling one domain block can be done in $4 \cdot n_i$ flops because we have to aggregate the average of four pixels of the domain block into one pixel which requires 4 flops for every square of four pixels.

The sum $\sum_{j=1}^{n_i} a_j b_j$ has to be computed for each rotation which requires $2 \cdot n_i - 1$ flops per rotations. The results of the remaining sums ($\sum_{j=1}^{n_i} a_j$, $\sum_{j=1}^{n_i} a_j^2$, $\sum_{j=1}^{n_i} b_j$,$\sum_{j=1}^{n_i} b_j^2$) are identical for all four rotations and and require a total of $6 \cdot n_i - 4$ floating point operations. Applying the formulas for brightness, contrast and the error involves 26 flops using the precomputed sums. Therefore,

$$\begin{aligned} \alpha_i &= 4 \cdot n_i + 4 \cdot (2 \cdot n_i - 1) + (6 \cdot n_i - 4) + 26 \\ &= 18 \cdot n_i + 18 \\ &= 18 \cdot \frac{s^2}{2^{2i}} + 18 \end{aligned}$$

Thus, we get the following upper bound on the cost:

$$
\begin{aligned}
C &\leq \sum_{i=1}^{m} |R^{(i)}| \cdot |D^{(i)}| \cdot \alpha_i \\
&= \sum_{i=1}^{m} 2^{2i} \cdot 2^{2i-2} \cdot (18 \cdot \frac{s^2}{2^{2i}} + 18) \\
&= \frac{6}{5} \cdot (4^m - 1) \cdot (4^{m+1} + 5 \cdot s^2 + 4)
\end{aligned}
$$

We see that the cost of the algorithm is exponential in the quadtree depth m and polynomial in image width s. For the roofline plot and the other benchmarks, we determined the cost empirically by instrumenting the code.

3. METHODOLOGY

This section defines the scope of our implementation, describes our initial baseline implementation, analyzes its bottlenecks and outlines the steps which were followed to increase performance.

Scope. In our project, we focus on grayscale images of size $s \times s$, where s is a power of two. To simplify vectorization, we also ensure that the range blocks do not get smaller than 4×4 pixels. Furthermore, exhaustive block mapping to search for suitable transformations is used. Our focus is solely on compression, which is the performance bottleneck in fractal image compression.

Baseline implementation. The baseline is written from scratch in C. Algorithm 1 illustrates the compression of the algorithm, whose inputs are the image (row-wise array of doubles) of size $s \times s$, the max quadtree depth m and the error threshold ϵ.

The function `partition(image,s)` partitions the image into contiguous non-overlapping blocks of size $s \times s$. The function `quad(`R_i`,s)` takes a range block of size $s \times s$ and partitions it into 4 smaller range blocks. The function `compute(image,`R_i`,`D_i`)` computes a transformation with its resulting RMS according to section 2. This function also downscales the domain block to the size of the range block, rotates it three times (90, 180 and 270 degrees) and returns the best of the four possible transformations.

As opposed to algorithms in [1] and [7], this algorithm does not implement quadtree partitioning recursively but iteratively. Our optimized versions (see later) depend heavily on precomputations of domain blocks (especially downscaling). At every new quadtree depth c, we build a new domain block pool $D^{(c)}$ whose domain block sizes are adapted to the new range block sizes. In an iterative approach, we can perform precomputations on $D^{(c)}$, process all domain/range block pairs and then discard the precomputations. Using a recursive approach, all precomputations of all domain block pools must be kept in memory to avoid recomputing the precomputations.

Algorithm 1 Compression of an $s \times s$ image with iterative quadtree partitioning

Input: `image`, ϵ, m
Output: $\boldsymbol{W}$ (set of computed transformations)

$\boldsymbol{W} \leftarrow \{\}$
$\boldsymbol{R} \leftarrow$ `partition(image,` $s/2$`)`
for $c = 1..m$ **do**
  $\boldsymbol{D} \leftarrow$ `partition(image,` $s/2^{c-1}$`)`
  for $R_i \in \boldsymbol{R}$ **do**
    $err_i \leftarrow \infty, w_i \leftarrow$ NULL
    for $D_i \in \boldsymbol{D}$ **do**
      $w_x, err_x \leftarrow$ `compute(image,` D_i, R_i`)`
      if $err_x < err_i$ **then**
        $w_i \leftarrow w_x$
        $err_i \leftarrow err_x$
  $\boldsymbol{R} \leftarrow \boldsymbol{R} \backslash R_i$
  if $err_i > \epsilon$ **then**
    $\boldsymbol{R} \leftarrow \boldsymbol{R} \cup$ `quad(`$R_i, s/2^c$`)`
  else
    $\boldsymbol{W} \leftarrow \boldsymbol{W} \cup \{w_i\}$

Roofline. We used the roofline model [9] to see whether the algorithm is memory or compute bound. The peak performance π was determined by counting the dispatch ports specified in Intel's *Optimization Reference Manual* [10]. We distinguish between scalar and vectorized peak performance. The bandwidth β was measured with the *STREAM* benchmark [11]. We verified both the peak performance and the bandwidth with the *Empirical Roofline Tool* [12], which confirmed the values.

For the program model, we need to measure the work W, the runtime T and the data move movement Q of the algorithm. Due to the dynamic nature of quadtree, the model does not only depend on the image size but also on the content of the image itself. Therefore, we define the three quantities as $W = W(s, image)$, $T = T(s, image)$ and $Q = Q(s, image)$, where s and $image$ are defined as in algorithm 1. We measured W by instrumenting the code according to the cost metric in equation (5). For measuring T, we used the `RDTSC` instruction available on all x86 architecture and disabled Turbo Boost to guarantee an integer runtime. The measurement of Q is the most challenging and requires performance counters which are not supported by our hardware. A correct, yet too pessimistic lower bound for the bandwidth is $Q \geq 8 \cdot s^2$ because we have to load the image (at least) once from memory. One of our group members developed a tool to simulate a program's memory accesses throughout the course, supporting multi-level caches [13].

We simulated our scalar-optimized code with this tool for the hardware we run our benchmarks on, which should give us a tighter lower bound for Q. This simulation is still too pessimistic for the non-optimized code but more

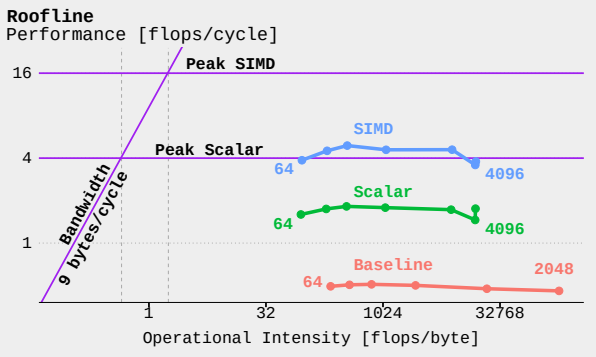

Fig. 1: Roofline plot for the baseline and various performance optimizations

accurate for our scalar-optimized and vectorized code. Our experiments showed that for small images ($\leq 512 \times 512$), our theoretical bound holds well. It is no surprise that for larger images ($\geq 1024 \times 1024$) the theoretical bound is way too pessimistic because these images do not entirely fit into the L3 cache of our hardware anymore, which leads to more traffic between main memory and cache.

To place the baseline in the roofline plot, we calculated the operational intensity $I = \frac{W}{Q}$ and the performance $P = \frac{W}{T}$ of our implementation. Figure 1 shows the roofline plot for a specific image of different sizes. One can see that the baseline is inherently compute bound and we have good potential for performance improvement as the baseline runs only at 12.5% of scalar peak performance. Next, we use profiling to find the performance bottlenecks of the code.

Hotspots. A major performance blocker is the exhaustive search for block matches. In our project, we focus solely on performance, whereas using a better search strategy is an algorithmic improvement and thus out of scope in the project.

When inspecting the code of the baseline, it is apparent that the majority of the work is done when calculating the contrast, brightness and error of each range/domain block pair (equations 2, 3 and 4). We used *Valgrind* [14] to profile the code and the profiling report confirmed our assumption. Especially, the sums over all pixels of a block were identified as hotspots so we started our optimizations there.

Scalar Optimizations. As a first optimization, we moved the calculation of the sums $\sum_{i=1}^{n} a_i$, $\sum_{i=1}^{n} a_i^2$, $\sum_{i=1}^{n} b_i$ and $\sum_{i=1}^{n} b_i^2$ out of the nested loop (see algorithm 1 line 5 and 7). The values of these sums change only if the algorithm advances to the next quadtree depth so we can easily precompute them in each quadtree iteration and reuse them when we compare each domain block with a range block.

The baseline implementation focuses on a clear and understandable code style. Therefore, it uses a queue to keep track of the remaining range blocks and several structs to model different concepts of the algorithm (e.g. the image and domain/range blocks). More precisely, a block is a struct with integer coordinates x, y and a size s. Both the queue and the usage of structs are not optimal for performance so we replaced them with simple variables and arrays. With this change, we went from *explicit* to *implicit* blocks. A block at quadtree depth m can be uniquely identified by an index i. Its size s is depending on m, and its coordinates x, y can be computed from index i. The QUAD function then reduces to index computations. These changes made the code harder to understand but eliminated pointer chasing and placed the data better in memory which improves locality.

After the removal of the queue and several structs, we continued to inline several functions, e.g. mapping indices in blocks to indices in the image. The inlining revealed many small optimizations that were not immediately obvious.

Another profiling of the optimized version of the code showed that the computation of $\sum_{i=1}^{n} a_i b_i$, which is used by the equations (2) and (4), is the remaining performance bottleneck. Precomputation of this sum is not possible because the term depends on values from specific range and rotated domain block pixels. The baseline implementation rotated the domain block by creating 3 copies of it, one for each rotation (90, 180 and 270 degrees). We removed this explicit rotation to reduce memory reallocation and instead traversing the domain block differently. As an example, when a domain block consists of pixels a_1, a_2, a_3, a_4, then the access pattern of the 180 degree rotated version of that block is a_4, a_3, a_2, a_1. The fusion of these three implicit rotations into one loop allowed us to compute traversal indices more efficiently. We furthermore exploited instruction level parallelism (ILP) to calculate the aforementioned computations more efficiently by factoring out the nested mathematical expressions into single line multiplications, additions and FMAs.

Finally, we did some experiments with different traversal methods for the domain/range blocks. Like blocked matrix-matrix-multiplication, we traversed blocks which are closer together in the image to exploit better temporal locality. With the insights from the roofline analysis, we abandoned any further experiments in that direction, because no big performance gains could be observed.

SIMD vectorization. Two different strategies to implement SIMD with AVX2 intrinsics were followed. One approach was to perform the computations of brightness, contrast and the corresponding error for all four rotations for each domain/range block pair at once. Another approach was to compute those values for four different domain blocks at a time in addition to processing four pixels at once when computing the sums in equations (2), (3) and (4).

A main challenge in implementing vectorization was due to the fact of the implicit domain block rotation by 90 and 270 degrees. Processing them resulted in expensive gather in-

structions (e.g. `_mm256_i64gather_pd`). Their performance drawback was so significant that we couldn't achieve any performance speedup compared to the scalar optimized code.

We proceeded by not considering 90 and 270 degree rotations for our transformations. This led to the speedup we originally expected when implementing SIMD instructions. One implication of this change was that image compression quality gets worse. However, we found that quality was still practically good because the algorithm then simply advances to further quadtree depths when quality requirements can not be satisfied.

A second attempt without removing any rotations was to withdraw from the idea of always implicitly rotating domain blocks. Instead, we rotate the range block once by 90 degrees explicitly while traversing the domain blocks in a manner such that we can represent all four rotations. Rotating the range block by 90 degrees instead of the domain block is more efficient because the number of range blocks in deeper quadtree levels tend to get smaller than the number of domain blocks (see algorithm 1), leading to less memory copying. The change had the effect that compression quality remained the same without extensive use of costly gather instructions.

Another important step was to decrease the amount of other costly instructions such as permutations or blends across lanes (e.g. `_mm256_permute4x64`).

4. RESULTS

In the first part of this section, we describe the benchmark infrastructure and the images we used for the measurements. Then we analyze how different compilers and flags affect the performance of our code. In the end, we discuss each optimization and compare them in different plots.

Experimental setup. All benchmarks and tests were conducted on an Intel Core i7-8650U processor with *Intel Turbo Boost* disabled, running at 1.9 GHz. The CPU has a 4×32 KB 8-way associative L1 cache, a 4×256 KB 4-way associative L2 cache and 4×2 MB 16-way associative L3 cache [10].

We used the peak signal-to-noise ratio (PSNR) to ensure that our baseline implementation compresses the image correctly. This metric is widely applied to compare an image with its compressed version. Typical values for the PSNR range from 25 to 50 dB (higher is better). The same metric was used to verify that the optimizations produced the same result as the baseline.

The PSNR of the compressed image depends on the maximum depth m of the quadtree and the error threshold ϵ. We set $m = 7$ and $\epsilon = 300$ for all our experiments. These parameters produced an image of good quality in a reasonable amount of time.

Fig. 2: Decompressed Image

We chose to benchmark our algorithm with a challenging image depicting a lioness with its cub [15]. While some parts of the image, for example the background, are easy to compress, other parts such as the fur contain lots of details, which are harder to compress. Figure 2 shows the output of the vectorized code with $\epsilon = 100$ and 3 decompression iterations.

Compiler Flags. Several benchmarks were conducted comparing the achieved performance using different compiler flags for the *GNU Compiler Collection (gcc)* version 9.3.0 and the *Intel C++ Compiler (icc)* version 19.1.1.217. For all tests `-march=native` was set.

For both the scalar and the vectorized versions the flag `-O1` increased performance significantly. While `-O2` did improve the scalar implementation (figure 3), it did not make a difference for the vectorized version (figure 4). Neither `-O3` nor `-Ofast` were able increase performance for both code versions.

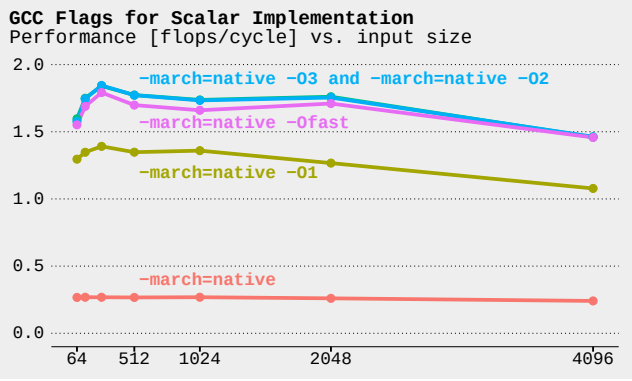

Fig. 3: Performance of the four major GCC optimization flags for the scalar implementation

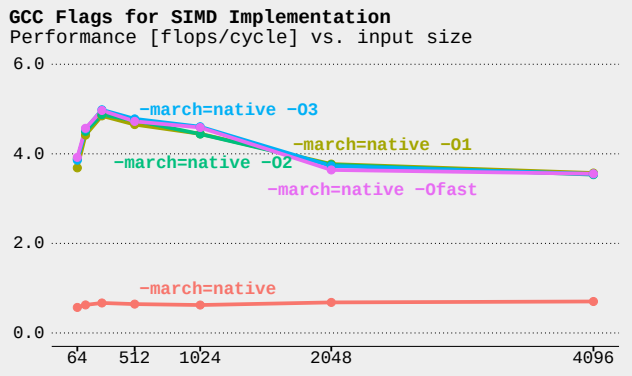

Fig. 4: Performance of the four major GCC optimization flags for the vectorized implementation

Intel's compiler was not able to outperform gcc but at least for the vectorized version it managed to keep up for the larger images as shown in figure 5.

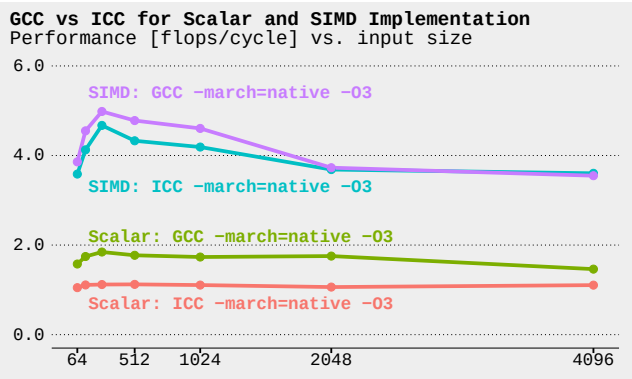

Fig. 5: Performance comparison between GCC and ICC with the best scalar and SIMD implementation

Because the Intel compiler with various compiler flags did not lead to improvements, we used gcc with the flags `-march=native -O3` which performed best for all the upcoming benchmarks.

Performance. The plot in figure 6 shows the performance of our major implementations. As a first result we observe that our baseline implementation achieves an equal or slightly better performance than the open-source reference C++ implementation [7]. Values for large images are not shown in the plot because the measurements took too much time.

The optimized scalar implementation which includes precomputations, a better memory layout and ILP is four times faster than the baseline. The optimizations become even more apparent when we compare the runtime as in figure 7. The plot shows that the runtime decreased by several orders of magnitude and we were finally able to compress larger images in a reasonable amount of time.

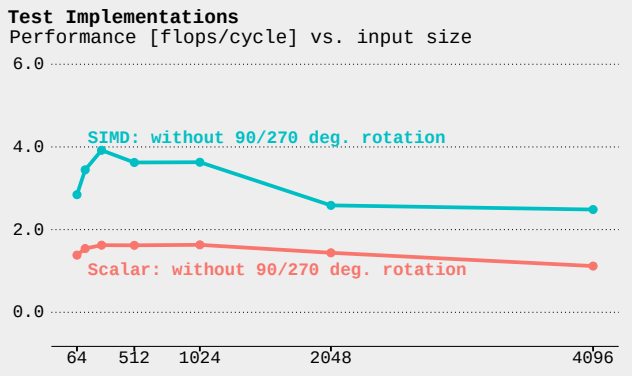

Fig. 8: Performance Plot without 90/270 degree rotations

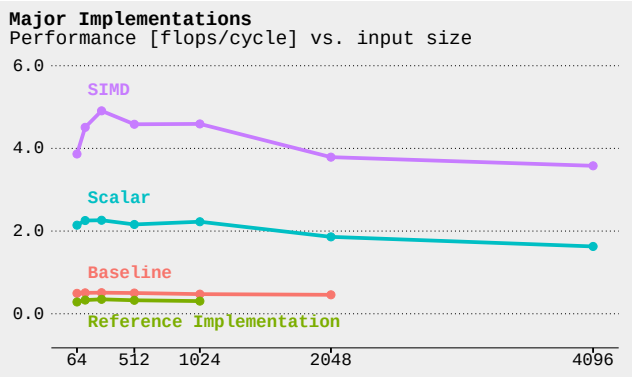

Fig. 6: Performance of our three major implementations and a reference implementation from GitHub written in C++

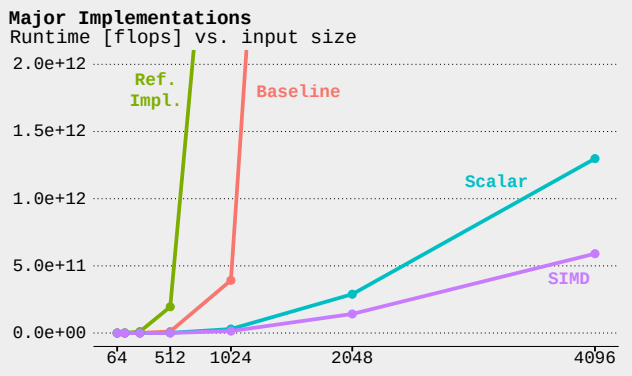

Fig. 7: Runtime of our three major implementations and a reference implementation from GitHub written in C++

As described in section 3 the initial attempt in vectorizing the code did not lead to the expected performance improvements. In fact, it performed just slightly better than the scalar optimized code. We suspected that the gathering instructions, which are necessary for the rotations, are responsible for poor performance. To verify this assumption,

we removed the column-wise access to the image i.e. 90/270 degree rotations from the scalar optimized and the vectorized version. The result of this experiment is shown in figure 8 and one can clearly see how the vectorized code outperforms the scalar optimized version.

After this experiment, we improved the column-wise access by rotating the range block instead of the domain block for the 90/270 degree rotations. The improved vectorized implementation has a performance roughly eight times as high as the baseline and about twice the performance of the scalar optimized version. The runtime was also reduced especially for large images.

5. CONCLUSIONS

With scalar optimizations, our implementation gains a performance speedup of roughly 4x, whereas the vectorized implementation has a performance speedup of roughly 8x compared with our straightforward implementation. It is important to mention that the runtime was decreased significantly and tangibly. As an example, compressing an image with 2048×2048 pixels with the baseline implementation takes about 2 hours, whereas the vectorized implementation needs 2.5 minutes.

Our fastest implementation with a performance of 4 flops/cycle is still 4 times below the theoretical peak performance of 16 flops/cycle. As the algorithm in this form is not memory bound, further performance improvements can be expected.

To furthermore boost performance one would also need to apply algorithmic changes. Using exhaustive block mapping with a rather large domain block pool (e.g. with four rotations) is a significant performance bottleneck which does not necessarily lead to better compression results.

We consider our optimizations to be applicable in more advanced and mature fractal image compression schemes.

6. CONTRIBUTIONS OF TEAM MEMBERS

Jonas. Focused on the baseline implementation and the following scalar optimizations (ILP, removal of structs, removal of pointer chasing and decreasing of memory reallocation). Helped in prototyping SIMD (especially the four-domain-blocks-a-time approach) and contributed to the solution which increased SIMD performance regarding 90 and 270 degree rotations.
Wrote a tool to simulate cache access in order to determine tighter memory bounds in the roofline plot.
Experimented with block-wise domain/range block traversal.

Pascal. Added argument processing, file I/O and some parts in the brightness and contrast computation of the baseline implementation. Added test data and prepared performance and runtime analysis. Did bottleneck analysis and contributed to the scalar optimizations (e.g. merging loops, more efficient precomputations). Contributed to the SIMD implementation (e.g. eliminate expensive instructions, scaling blocks, version without rotation). Tested compiler flags including ICC.

Fabian. Added rotations to baseline. Created roofline plot and compared baseline to reference implementation. ILP and index optimizations for $\sum_{i=1}^{n} a_i b_i$ calculation.

Implemented parts of four-domain-blocks-a-time SIMD (tricky part was error calculation with `blendv` and mask) and participated in bringing the two SIMD approaches together. Helped increasing performance regarding 90 and 270 degree rotations.

Janis. Changed baseline from C++ to C and started with the implementation of a fast queue. Due to personal problems he withdrew from the project.

7. REFERENCES

[1] Y. Fisher, *Fractal image compression: theory and application*. Springer Science & Business Media, 1995.

[2] M. F. Barnsley and A. D. Sloan, "Fractal image compression," 1989.

[3] A. E. Jacquin, "A fractal theory of iterated markov operators with applications to digital image coding." 1990.

[4] E. Jacobs, Y. Fisher, and R. Boss, "Image compression: A study of the iterated transform method," *Signal Processing*, vol. 29, no. 3, pp. 251–263, 1992.

[5] R. Boss and E. Jacobs, "Studies of iterated transform image compression and its application to color and dted," NAVAL OCEAN SYSTEMS CENTER SAN DIEGO CA, Tech. Rep., 1991.

[6] P. Vigier. (2020, Jun.) Fractal Image Compression. [Online]. Available: https://github.com/pvigier/fractal-image-compression

[7] A. Kennberg. (2020, Jun.) Fractal Compression. [Online]. Available: https://github.com/kennberg/fractal-compression

[8] J. E. Hutchinson, "Fractals and self similarity," *Indiana University Mathematics Journal*, vol. 30, no. 5, pp. 713–747, 1981.

[9] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the roofline model," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 76–85.

[10] *64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, 2016.

[11] J. D. McCalpin. (2020, Jun.) STREAM benchmark. [Online]. Available: https://www.cs.virginia.edu/stream/

[12] C. Yang *et al.* (2020, Jun.) Empirical Roofline Tool (ERT). [Online]. Available: https://crd.lbl.gov/departments/computer-science/par/research/roofline/software/ert/

[13] J. Hansen. (2020, Jun.) Cache simulator. [Online]. Available: https://github.com/vl0w/cache-simulator

[14] J. Weidendorfer *et al.* (2020, Jun.) Valgrind. [Online]. Available: https://valgrind.org/

[15] (2017) Lions. [Online]. Available: https://pxhere.com/en/photo/915683