FULL STACK OPTIMIZATION OF FRACTAL IMAGE COMPRESSION

Anton Brucherseifer, Sarah Moy de Vitry, Daniel Peter, Michael Seeber

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Fractal image compression is a form of lossy image compression that encodes images based on similarity within their sub-blocks. While it is a storage-efficient form of image compression, it suffers from having a slow encoding step. This bottleneck during compression can be specifically traced to the heavy calculations done during the search for self-similarity and makes fractal image compression a prime candidate for optimizations. In this work we explore different approaches to optimizing the computationally expensive encoding step by using generic methods as well as domain-specific improvements.

1. INTRODUCTION

Nowadays, we use image compression almost every day, even though we do not always notice. Most images on the web are compressed, for instance using JPEG. The reason for using compressed images, is that uncompressed images would need quite a lot of space since they store a value *for every pixel*. Like JPEG, fractal image compression is *lossy*, meaning that the compression step might change the images slightly, due to the information loss. Moreover, fractal image compression is asymmetric in the sense that compressing an image takes much longer than decompressing it. Due to this and the high time complexity, fractal image compression is not so widely used. In the following, we will present different ways of optimizing the compression step to make it more usable in the real-world.

Our Contribution. Starting from the GitHub Repository [1], we implemented our own version, which already resulted in large performance improvements. From there, we added various optimizations alike the methods that were learned during the Advanced Systems Lab class. After identifying the single performance critical function, we applied many different optimization techniques to it. To measure the performance of the different versions, we implemented a benchmarking infrastructure for counting the cycles of an execution. Combining this with the flop count for every version, we could easily evaluate the performance, and thus determine which versions were worth keeping and improving. In Section 3, we describe the successful optimizations in greater detail. After realizing that we could not improve the performance any further, we decided to add some methods that reduce the number of required optimizations. One way to achieve this was to introduce an adaptable *stride*, meaning that we can change the distance between blocks that are used for comparison. In addition, this method enables us to use a smaller stride, such that the blocks are overlapping, allowing to increase the compression quality. A more sophisticated way of decreasing the necessary number of operations, was to reduce the search space by precomputing the *entropy* of every block. Using this method, our algorithm only considers blocks with a low entropy for the comparisons, since they have a higher probability to match other blocks.

Related Work. The domain of lossy image compression has been hashed and rehashed with implementations ranging from discrete cosine transform (JPEG) [2] to wavelet compression (JPEG 2000) to fractal compression. Fractal image compression in its simplest form was introduced by Michael Barnsley in 1988 [3] and its advertised theoretical compression ratios made it an attractive option. However, as mentioned above, the drawback to fractal compression was the overhead of the compression step and more specifically the high number of mean squared error (MSE) computations. Most performance-related research in fractal compression has been aimed at optimizing this step and more specifically at reducing the search space to the most critical blocks and thereby minimizing the number of MSE computations performed. Tseng, Hsieh and Jeng in [4] explored visual-based particle swarm optimization, a form of search space reduction based on the edge type of sub-blocks. Other optimizations seek to reduce the search space on a per-block basis as in [5] where genetic algorithms reduce block candidates to only the most likely matches. The work on genetic algorithms is mostly centered around choosing the right parameters for the algorithm parameters to achieve a compromise between performance and image quality. An more intuitive metric for measuring use probability in blocks is entropy as explored by Hassaballah, Makky, and B. in [6]. Our work most closely reassembles a combination of low-level code optimization and search space reduction based on an entropy metric. To our knowledge, no other implementa-

tion available does optimization of the fractal image compression on the level of intrinsics, code optimization, and entropy-based search space reduction. Moreover, while papers have addressed the issue of entropy-based search space reduction, there is little work available on how to determine a meaning full threshold above which to reduce the search space. We concretely determine a threshold that maintains good image quality across a range of image types, all while significantly raising performance.

2. BACKGROUND ON THE ALGORITHM/APPLICATION

The high level idea behind fractal image compression is to work with self-similarities. Experimental results have shown that photographs usually have such self-similarities at different scales and that they can be used for compression [7]. This means the algorithm searches the image for patches that look similar at two different scales. In this section, we will explain the algorithm more precisely.

The Algorithm. In a first step, a copy of the input image is scaled down to half the size using bilinear interpolation. The original image and the down-scaled copy are then both split into non overlapping 8x8 pixel blocks.

In the next step, the algorithm finds the best way to approximate each original image block by transforming a block from the small image, see Figure 1. The transformations include element-wise multiplication and addition with a scalar contrast and brightness value respectively. Additionally we allow the transformations to possibly rotate and flip an image block, which often helps to find transformations that better approximate the original image block.

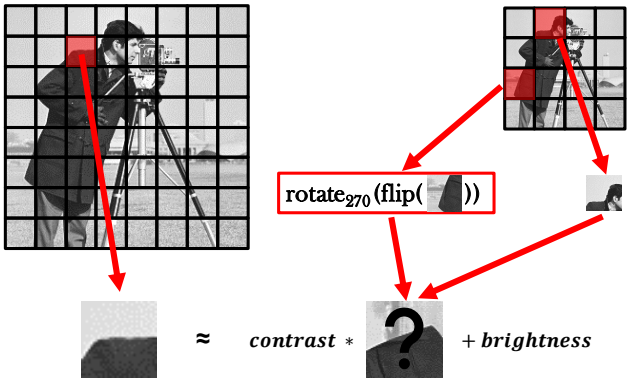


Fig. 1: An image block from the original image (left) can be approximated by transforming an image block from the small copy (right). Allowing rotations and flips may help to find better approximations.

To find such an optimal transformation for each original image block, we solve a least squares problem for each pair of blocks. Finally, all optimal transformations (i.e. scalar contrast and brightness values, information about how to rotate and flip, and the index of the block in the small copy) are written to the compression file. Starting from an arbitrary image, these transformations can then be applied iteratively to reconstruct the image as illustrated in Figure 2.

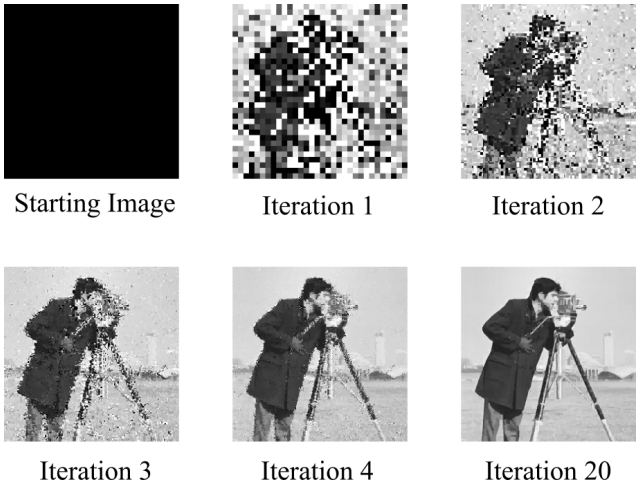


Fig. 2: Visualization of decompression iterations. The Cameraman image (256x256) was compressed using rotations, flips and stride=2.

Cost Analysis. Our algorithm has an asymptotic time complexity of $\mathcal{O}(m^2 \cdot n^2)$, where m and n are the width resp. height of the image. There are two sets of blocks, each containing $\mathcal{O}(m \cdot n)$ blocks of constant size. The high complexity is because for every block in the first set we must iterate over every block in the other set. The algorithm uses additions, multiplications, divisions and square roots. All of them have the same asymptotic complexity, however it is important to mention the absolute numbers. The counts for additions and multiplications are fairly similar at around $\frac{1}{8}m^2n^2$ each. The divisions are much less: $\frac{1}{1024}m^2n^2$. However they shall not be forgotten since they require more processor cycles to compute. The square root count is even lower at $\frac{1}{4096}m^2n^2$. These numbers are for the fast unoptimized version, i.e. before applying any optimizations, using the default stride and not using search space reduction with entropy. In the most optimized version the number of divisions was reduced by a factor of 4 while the square roots were omitted completely. We also tried to have even less divisions by precomputing the inverse of the denominator and using multiplications instead (strength reduction), however, this slightly decreased the performance.

3. OUR PROPOSED METHOD

Starting from Scratch. The main part of our work was to optimize and extend the fractal image compression al-

gorithm found in [1]. However, we did not want to constrain our optimization efforts by this existing implementation. We thus proceeded by implementing the algorithm from scratch which allowed for various major improvements compared to the existing code.

The implementation in [1] uses double precision floating point arithmetic for all computations. We observed that using single precision instead results in practically identical looking images. Therefore, all our computations are done using single precision numbers which allows for more efficient memory transfer as well as twice the number of computations per SIMD instruction.

As stated before, given the two sets of image blocks, the algorithm solves a least squares problem for each possible pairing of blocks. Consequently, each individual pixel block is used to solve $\mathcal{O}(m{\cdot}n)$ least squares problems. This allows to perform some computations only once and then reuse the results in each least squares problem. Computations that can be reused include the rotations and flips of pixel blocks, as well as some other computations needed to solve the least squares problem.

While our code is written in C++, the performance critical parts conform to standard C. We only make use of C++ specific features to implement elements of the infrastructure such as the loading of an input image and the writing of the final compression file. Compared to the object-oriented implementation in [1], it becomes easier to identify, analyze and optimize the performance critical parts in the code.

A square has eight symmetries that form the Group of Symmetries of a Square, see Figure 3. For fractal image compression it is desirable to allow as many of these symmetries as possible, since it often helps to find better self-similarities. However, the implementation in [1] only allows the four rotational symmetries. We thus extended our implementation such that it computes all eight similarities by allowing the image blocks to be rotated and/or flipped. While we provide optimized implementations for both versions[1], this report mainly focuses on analyzing the performance of the extended implementation.

In addition, we stumbled upon a bug in the implementation from GitHub [1]. For every image block, all four rotations are generated by applying a method, that rotates a block by 90°, the appropriate number of times. However, the implementation does a flip about the main diagonal (transposition) instead, which leads to some unwanted behaviour. In fact, the GitHub repository [1] is limited to only two symmetries (identity and diagonal flip) which leads to slightly worse looking images. This shows that it is important to validate the implementation of the rotations, we thus made sure that our rotations work as expected.

Scalar Replacement. The most basic optimization we implemented was scalar replacement. Surprisingly, some of the replacements we tried, while seemingly logical because they were used iteratively many times, resulted in worse performance. The biggest improvement for this step was when we used scalar replacement in the function[2] that computes the similarity measure between two blocks.

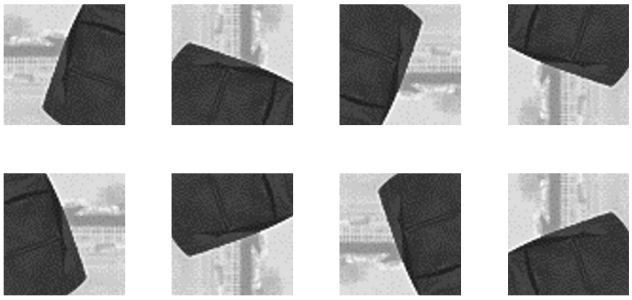


Fig. 3: The *group of symmetries of a square* (or *dihedral group of degree 4*) visualized by using a pixel block. The group consists of rotations by 0°, 90°, 180°, 270°, horizontal flip, flip about the other diagonal, vertical flip and flip about the main diagonal

Furthermore, we noted that the code in [1] computes the Euclidean Distance divided by the number of pixels in a block as a measure of similarity. We replaced this computation with the sum of squared errors (RSS) which avoids the costly square root and division. This works since we are only interested in finding the transformations that minimize the MSE, the actual value of the similarity measure is not relevant. While the gains in performance seems to be negligible, compared to the scalar replacement, we still decided to include it as it may help with the following optimizations.

Inlining. The compiler is able to inline functions at compile time in order to avoid the overhead of function calls. However, it may sometimes still be beneficial to manually inline functions since it may enable further optimizations.

In order to keep track of the optimal transformations, the measure of similarity (RSS) needs to be computed after every least squares problem. Our implementation uses a separate function to compute the RSS. We found that inlining this function permits some further optimizations. Most notably, the inlining allows to avoid the explicit writing and reading of the transformed block.

Instruction Level Parallelism. Superscalar processors are able to execute multiple instructions per cycle which is called instruction level parallelism (ILP). Unfortunately, compilers are usually not good at restructuring algorithms to increase ILP. It thus makes sense to unroll and restructure numerical functions manually in order to make ILP more explicit to increase the performance.

Our implementation has a nested loop to find the op-

[1] Executables `fic_compress_fast`, `fic_compress_flip_fast`

[2] In our code we refer to it as `difference_norm()`.

timal transformation for each pair of blocks. This allows to fully unroll the most inner loop where we solve a least square problem for every rotation and flip. The unrolled code can then be restructured such that impact of blocking dependencies is reduced, which leads to the desired increase in ILP.

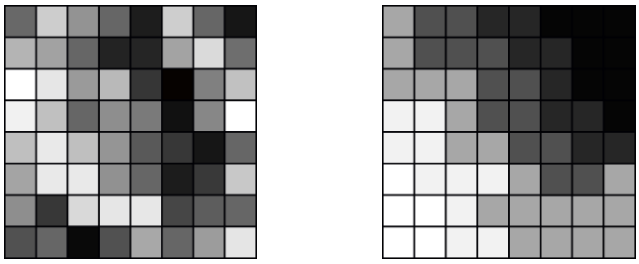
While testing our initial ILP optimization for the case with flip, we noted that the performance was highly dependent on the input size. In fact, we observed significant performance decreases whenever the input size (= height = width) was divisible by 256. This strange behaviour might be explained by conflict misses in cache which we only observe under certain conditions. Thus, we split the whole loop in half, such that it computes the non-flipped blocks first and then does the same for the flipped blocks. The resulting ILP optimization showed great performance for all tested input sizes.

Manual Vectorization. The next logical optimization step after ILP was to manually vectorize the code using AVX2 intrinsics. Having the ILP optimization at hand, made this a surprisingly straightforward task. This initial version was therefore just doing the calculations for a whole row (i.e. 8 pixels) in parallel. Due to this we still have to perform horizontal sums to combine the values, which is not really in the spirit of AVX2. It seemed obvious to us that we should try getting rid of the horizontal sum by reordering the memory layout beforehand. This way, we could use a simple load to get one SIMD vector to hold the 8 transformations of a pixel. In case of not using flip, a SIMD vector would store 2 pixels with 4 transformations each. Although the resulting code turned out to be really compact and followed the vectorization idea more closely, the performance was really disappointing. Our final AVX2 version therefore relies on the initial try, although we were able to push the performance further by optimizing the horizontal sum computation as well as splitting up the loops for the flip case.

4. EXTENSIONS

After implementing the optimizations introduced in the course, we decided to extend our solution to include two novel improvements. This lead us to add entropy-based search space reduction, reducing the runtime, and to allow variable stride steps, improving quality or decreasing the runtime depending on the image size.

Entropy-Based Search Space Reduction. Entropy as defined for images is a measure of the variance of pixel values among themselves. This measure of similarity is relevant to the fractal image compression algorithm because it is also the basis for the choice of which blocks to store. For example, the high-entropy block in Figure 4a is less likely to be chosen when iterating over the small blocks because the error is likely to be higher than the error of the low-entropy



(a) High entropy block (b) Low entropy block

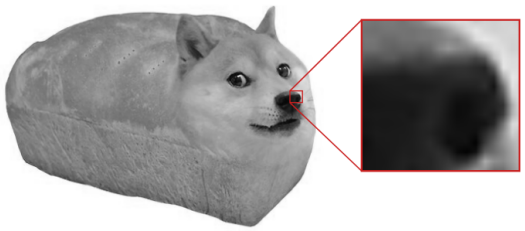
Fig. 4: Two 8x8 blocks with contrasting entropies

block in Figure 4b [6]. Considering, therefore, that certain blocks are unlikely to be used, it makes sense to remove them from the search space to avoid unnecessary comparisons.

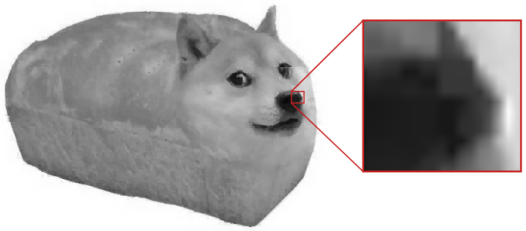
We implemented this extension by precomputing the entropy of every small block and defining a meaningful threshold above which blocks are discarded from the search space. The biggest challenge was choosing a meaningful threshold above which to discard blocks. A few failed attempts included discarding blocks with entropy higher than the image's mean entropy, higher than the mean entropy plus standard deviation, and higher than hard set bounds. Although these thresholds gave good results for certain images, they seemed unusable for images with little contrast, where there was low variance in the entropy. In the end, we defined the threshold as the limit above which we knew to be discarding a certain percentage of the total search blocks. There was only a noticeable decrease in quality after discarding more than 80 percent of the blocks, even for low contrast images. Figure 5b shows the result of normal fractal image compression on Figure 5a. Comparing Figure 5b and Figure 5c where the search space has been reduced by 50 percent, we can see that the same blocks are used. However, after an 80 percent reduction in Figure 5d it is visible that simpler blocks are being used.

Strides. Another way of reducing the number of block comparisons is to adapt the stride. The default method looks at neighboring blocks, i.e. it uses a stride of 16 [3]. For instance, if we now set the stride to 32, the number of blocks to be compared is decreased by a factor of 4. An example result of this can be viewed in Figure 6b. Similarly, this parameter also empowers us to increase the number of blocks. This turned out to be particularly useful for images with a low resolution, as it is less likely to find well-matching blocks within small images. One can compare the resulting images after compressing with different strides in Figure 6.

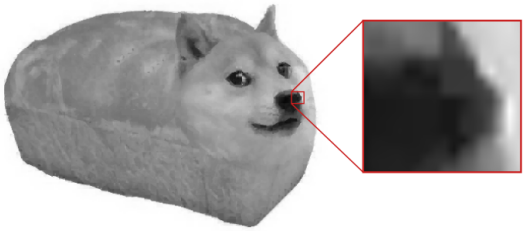
[3] 16x16 pixels is the block size of the down-scaled blocks before downscaling them.



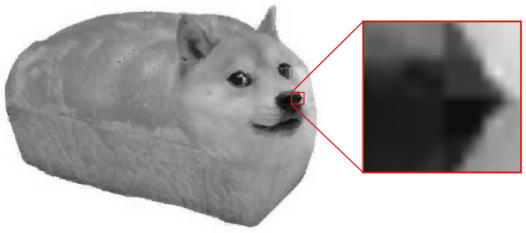
(a) Original image



(b) Block differences after normal compression

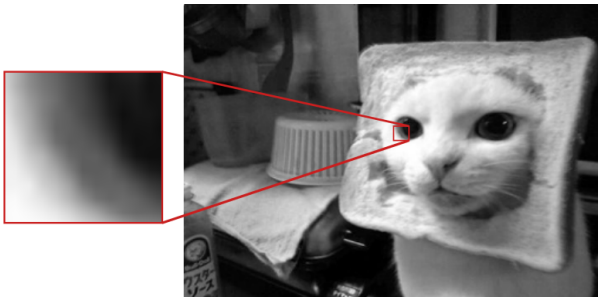


(c) Block differences after 50 percent reduction of search space

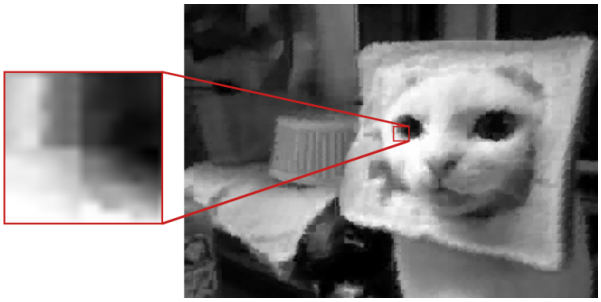


(d) Block differences after 80 percent reduction of search space

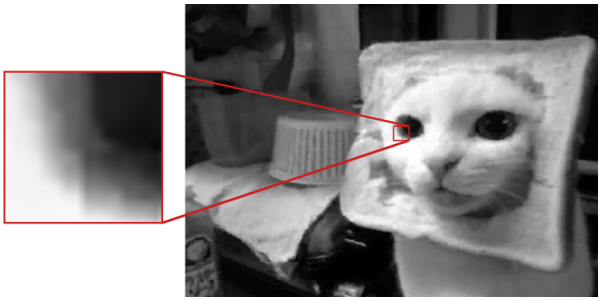
Fig. 5: Results of fractal image compression with entropy-based search space reduction for a 544x384 image



(a) Original image



(b) Block differences when using stride 32



(c) Block differences when using stride 2

Fig. 6: Results of fractal image compression with different strides on a 256 x 376 image

5. EXPERIMENTAL RESULTS

Experimental Setup. Throughout the project we tested our code on various CPU architectures, but for the plots we opted to go with a single machine for comparison reasons. The system information for the machine we ran the experiments on can be found in Table 1.

For GCC the flags that resulted in the best performance are `-O3 -ffast-math -march=native -funroll-loops`. For experiments done with ICC we relied on the same flags.

To streamline the benchmarking, we also created our own testing infrastructure. It consists of various python scripts that support the compiling, running and also plotting of the collected data. Furthermore they include support for various input parameters as well as a custom logging infrastructure, that allows to reload experiment data at a later point.

As in practice it is almost never possible to reach the theoretical memory bandwidth, we also included a more practical measured memory bandwidth in our roofline plots. We gathered the values for our testing machine with the STREAM benchmark described in [8].

CPU	Intel Core i7-6700K @ 4 GHz (Skylake)
Memory	16 GB DDR4 @ 2133 MHz (Dual Channel)
OS	Ubuntu 20.04 LTS
Compilers	GCC-9.3.0 and ICC-19.1

Table 1: System information

Performance Plot. With the proposed optimizations we get a significant speedup over our base implementation. However, it does not always make sense to compare the performance of the optimized methods to the base implementation. For this purpose we can use the fast unoptimized version as it uses the same approach as the optimized methods and thus has a similar number of flops.

In the performance plot shown in Figure 7 one can identify the number of flops per cycle for every function described in Section 3. The varying input size is to be understood as the width and the height of the image, i.e. the input size goes from 128x128 to 2048x2048. For this plot, we used a real image that had been down-scaled for every image size. The experiments were performed with the flip transformation.

We can see that manual vectorization and ILP perform best when the image size is not larger than 1024x1024. The inlining and scalar replacement versions on the other hand decrease already around 896x896 pixels. This phenomenon might seem strange at first, but it is exactly what we expect, as we will explain now. The amount of single precision floating point numbers needed for our performance critical

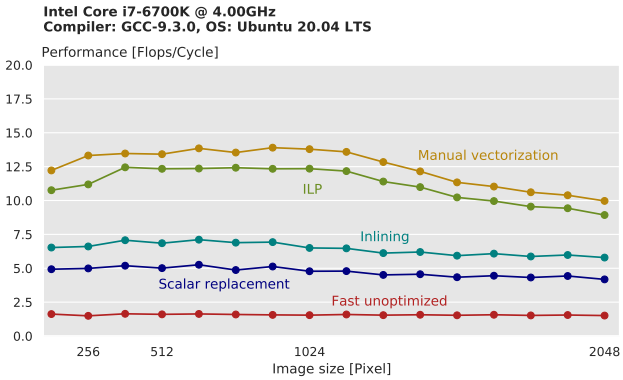


Fig. 7: Performance plot for different input sizes of the ape image depicted in Figure 11

part is

$$n^2 + \frac{trans}{4} \cdot n^2 + \frac{5}{64} \cdot n^2 \tag{1}$$

, where n is the side length of a square image in pixels and $trans$ is the number of transformation that are computed in the third nested loop. Hence, $trans = 8$ for the versions that compute the rotations including flip in the same loop (inlining and scalar replacement). As described in Section 3, we split the nested loop in half in the ILP and manual vectorization versions, meaning that the variable $trans$ gets reduced to 4. How do the different values for $trans$ relate to the performance decline at 896 resp. 1024? For this, we compute the maximum image size for which the L3 cache is large enough to avoid capacity misses. Our system has 8MB of L3 cache, meaning it can hold 2^{21} single precision numbers. Thus, we can define n_{max} as a function of $trans$:

$$n_{max}(trans) = \sqrt{\frac{64}{64 + trans \cdot 16 + 5} \cdot 2^{21}} \tag{2}$$

Plugging in the corresponding value for $trans$, we get the following threshold for manual vectorization:

$$n_{max}(4) \approx 1004. \tag{3}$$

For the versions that compute the whole chunk in the same loop we can fit only images of size 825x825 pixels in L3 cache:

$$n_{max}(8) \approx 825. \tag{4}$$

As we can see in the plot, the performance decreases monotonically between 1024 and 2048. We also ran the measurement for the image size 4096x4096 which resulted in a similar performance as for 2048x2048.

We tried ICC with the same optimization flags which resulted in worse performance as demonstrated in Figure 8. Note that we can clearly observe the performance drops for conflict misses if the image size is divisible by 256 for

the functions inlining and scalar replacement. This can also be seen in the GCC plot, although less pronounced. And as expected, the functions ILP and manual vectorization do not produce such a behaviour due to the cache optimization.

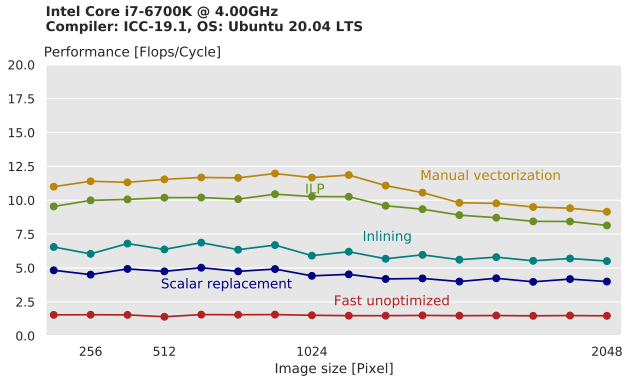


Fig. 8: ICC performance plot

Roofline Plot. For the roofline plot presented in Figure 9 we added every version of our performance critical function `find_optimal_mappings` except the ones that reduce the search space. As input for these tests we used random data, as the performance does not depend on the input. We used an input size of 512x512 pixels, which fits into Skylake's L3 cache of 8MB. It is therefore sufficient to load all required data only once from main memory. Hence, the computations are clearly compute-bound.

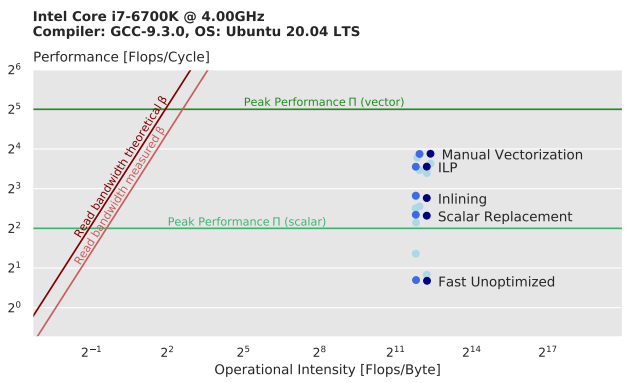


Fig. 9: roofline plot that includes all our version of `find_optimal_mapping` using default stride and no entropy-based search space reduction for image size 512x512

Since not all optimization attempts of our performance critical function were successful, we highlighted the ones discussed in Section 3. The lighter blue dots are without using flip and the darker blue dots include the flip transformation. The performance stays quite similar when adding the flip transformation. However, we can see a small performance decrease for inlining, scalar replacement and fast unoptimized. This cannot be observed for the manual vectorization and ILP since they split the whole loop in half such that it requires the same amount of cache than without the flip.

Time Measurements. Up to this point, we only measured the flops per cycle. As already stated, comparisons between the flops per cycle only make sense for similar flop counts. We will therefore measure the time it takes to compress an image of size 1024x1024 pixels using the following versions of our code:

- base: the implementation in [1], to which we added a bug fix as well as the ability to flip images
- fast unoptimized: as described in the first paragraph of Section 3. This corresponds to the slowest version in Figure 7.
- manual vectorization: This corresponds to the best version in Figure 7.
- manual vectorization + entropy: the same method as before, but with computing the entropy of every block and only using the blocks with an entropy in the lowest 10%

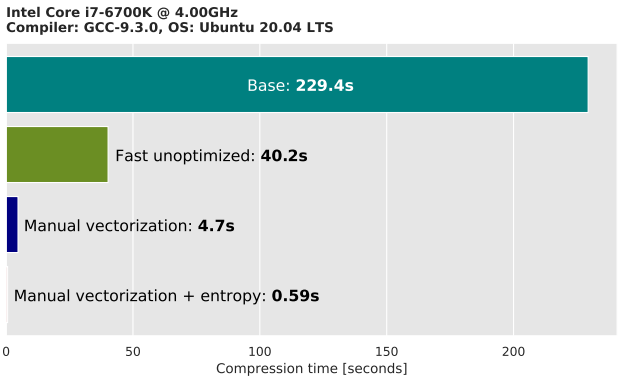


Fig. 10: Compression times for the Grumpy Cat image shown in Figures 12 to 16 (1024x1024)

Our algorithmic improvements by starting from scratch already yields a speedup of 5.7x The optimizations learned in Advanced Systems Lab gave us another 8.6x speedup. As we can see in Figure **??**, the quality stays the same up until this point. Finally, we reduced the search space with the entropy method, which decreased the runtime by another factor of 8 with nearly the same image quality. All together, we obtained a speedup of 389x!

6. CONCLUSION

In this work we have sought to improve the performance and reduce the runtime of fractal image compression using not only basic optimizations such as pre-computation and manual vectorization, but also search space reduction. There are three important points that can be taken away from our work on fractal image compression. First, although code available in open repositories is useful for simple code examples, it can often be optimized to achieve noticeably better runtimes by applying basic concepts covered in the Advanced Systems Lab course. Second, although there were excellent gains from doing basic optimizations, the best gains were obtained when combining these with optimizations that were based on precise knowledge of the fractal compression (entropy and stride). To conclude, we achieved a speedup of almost 400x for the compression step. This enormous unused optimization potential of an already good looking open source library deeply impressed us. Future work on fractal image compression could include using quad-tree decomposition to improve quality as well as parallelization and block-specific search space reduction to improve runtime.

7. CONTRIBUTIONS

In general, detailed information can be found in the commit log of our repository. [4]

Anton. For the optimizations of our performance critical function, I have implemented the following:

- reorder the memory layout for the manual vectorization version (As described in the last paragraph of Section 3, this attempt turned out to be slower than the original one.)
- implement flip for several versions
- split the loop for the ILP flip version

In addition, I also implemented the methods for evaluating correctness and performance for the performance critical function. This included:

- make sure that all performance critical part is in `find_optimal_mappings`
- keep track of the precise number of floating point operations for every version of `find_optimal_mappings`
- implement validator that checks whether optimizations have the same result as the reference version.
- create input data
- make more portable by allowing different command-line options

Daniel. My most notable contribution was the initial implementation of the following executables:

- base: `fic_compress` and `fic_compress_flip`, i.e. the simple rewriting of OOP code in [1] such that it conforms to the code style from the lecture. Bugfix and flip extensiongt.
- fast unoptimized: `fic_compress_fast` and `fic_compress_flip_fast`, see Section 3, Starting from Scratch. Is also used to plug in all other optimizations we discussed. Implementation of strides.
- `fic_decompress`. Due to the flip and stride extensions we cannot use the decompression in [1].

I also helped to optimize `find_optimal_mappings`:

- ILP (initial version), see Section 3.
- Inlining, see Section 3

Michael. My main contribution to the project can be found in the areas of vectorization using AVX2 intrinsics as well as in the analysis part. On the vectorization side I implemented the following:

- All manual vectorized versions of `find_optimal_mappings`. This includes noflip, flip as well as horizontal sum and reordered memory layout based variants.
- Our final best version for both flip and noflip. (`fic_compress_fast` and `fic_compress_flip_fast_avx`).

For the analysis part I worked on automating the whole benchmarking pipeline. This includes compiling, gathering data and and also plotting the collected measurements. More specifically:

- Make sure our executables output meaningful performance measures into a log.
- `performance.py` and `benchmark.py` which can compile our code, feed input to the executable and afterwards automatically create performance and roofline plots for the gathered data.

Sarah. The section I worked on most was the search space reduction extension of the project. I also worked on basic optimization of the `find_optimal_mappings` and `difference_norm` function (scalar replacement) and helped with the memory access analysis for the roofline plots.

[4] https://gitlab.inf.ethz.ch/COURSE-ASL2020/team022

8. APPENDIX

Fig. 11: The ape image we used for the performance plots.

Fig. 13: Base

Fig. 12: Original image

Fig. 14: Fast unoptimized

Fig. 15: Manual vectorization

Fig. 16: Manual vectorization + entropy image

9. REFERENCES

[1] Stan Zonov, "fractalmapping," https://github.com/mannyray/fractalMapping, 2017.

[2] Gregory K. Wallace, "The jpeg still picture compression standard," *IEEE Transactions on Consumer Electronics*, April 1991.

[3] Michael F. Barnsley, "A better way to compress images," *BYTE The Small Systems Journal*, 1988.

[4] Jyh-Horng Jeng Chun-Chieh Tseng, Jer-Guang Hsieh, "Fractal image compression using visual-based particle swarm optimization," *Image and Vision Computing*, August 2008.

[5] Boukelif Aoued Faraoun Mohamed, "Speeding up fractal image compression by genetic algorithms," *Multidimensional Systems and Signal Processing*, April 2005.

[6] Youssef B. Mahdy M. Hassaballah, M.M. Makky, "A fast fractal image compression method based entropy," *Computer Vision Center / Universitat Autonoma de Barcelona, Barcelona, Spain*, December 2004.

[7] Yuval Fischer, "Fractal image compression," *The San Diego Super Computer Center University of California, San Diego*, 1992.

[8] John D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," Tech. Rep., University of Virginia, Charlottesville, Virginia, 1991-2007, A continually updated technical report. http://www.cs.virginia.edu/stream/.