A Comparative Analysis of Parallel Image Compression Algorithms

Audrey Bongalon (bongalon@usc.edu), Padmanabha Kavasseri (kavasser@usc.edu),
Prasanna Rajagopalan (pr00946@usc.edu), Rishil Shah (rishilsa@usc.edu)

***Abstract*—A comparison of various parallelized approaches to optimize image compression time using Fractal Compression and Run Length Encoding Algorithm using CUDA and OpenMP.**

Keywords—Parallel Image Compression; Fractal Image Compression; Run Length Encoding (RLE); CUDA Programming; OpenMP;

I. Introduction

Digital images are usually represented as a 2D matrix of pixels. This matrix can become quite large, so the data is often compressed into a smaller file for storage or transmission. Compression algorithms can be categorized as lossless or lossy. With lossless compression, decompressing the file results in an exact copy of the original data, whereas with lossy compression, some loss in data precision is sacrificed to achieve a smaller file size, even if the decompressed file is slightly different from the original.

Our project focused on two image compression algorithms: fractal compression and run length encoding (RLE). These are lossy and lossless, respectively. These algorithms can be time consuming, especially for larger images. By introducing parallelism through multi-core GPUs, we were able to demonstrate significant reduction in their runtimes.

Since our university course focused extensively on multiple techniques for parallelizing algorithms involving integer matrices, we found its implementation in image compression. This was accomplished by recognizing which steps in the algorithms have scope for parallelization. Following their implementation, an analysis was conducted by comparing the compression times between different algorithm implementations in both their sequential and parallel forms.

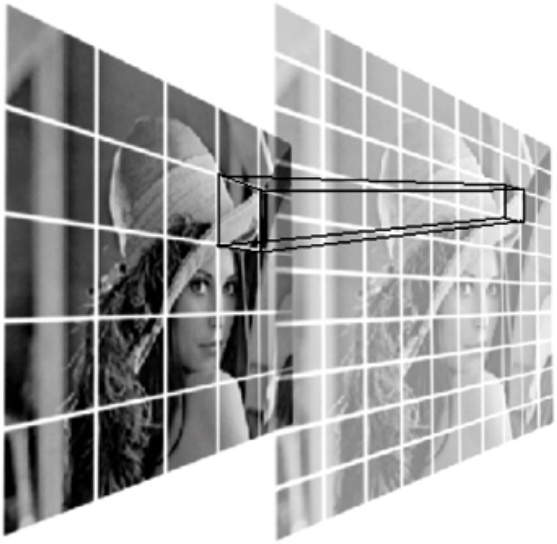
Following is a brief introduction of the algorithms we have optimized in this project:

A. Fractal Compression Algorithm

Fractal Compression Algorithm relies on finding similarities in an image by splitting it into block sized fragments and then looking for Affine transformations from those blocks in smaller-range block partitions. The steps that are required to perform fractal compression of an image are shown below.

Stage 1: Downsizing: The first step of fractal image compression involves partitioning the image into two sets:

- The first set of image blocks are known as range blocks $R_1...R_N$. These blocks are disjoint, and they cover the whole image.
- The second set of image blocks are known as domain blocks $D_1...D_K$. These blocks need not be disjoint and neither do they necessarily cover the entire image.



Stage 2: Transformations

For the second step, we need to transform the downsized domain blocks to find potential blocks that could resemble the corresponding range blocks. For our project, we employ the following transformations:

- Flip
- Rotate
- Contrast
- Brightness

$$f_l(x_{D_k}) = s \times rotate_\theta\left(flip_d\left(downsize(x_{D_k})\right)\right) + b$$

Here, s represents the contrast operation, and b represents the brightness operation.

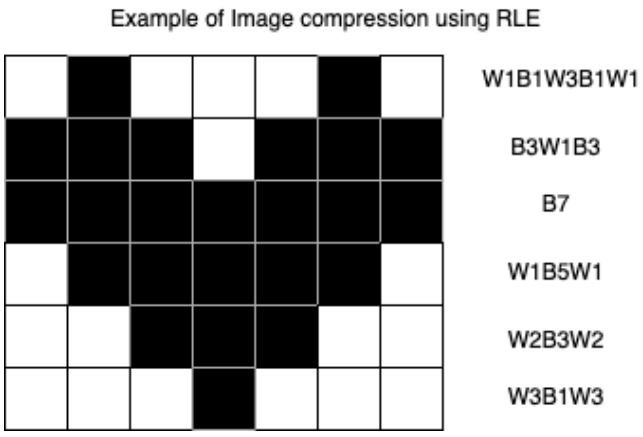
Stage 3: Similarity Calculation

In order to choose the range block that creates the closest approximation, it is necessary to use a similarity index to compare the two. We intend to experiment with the Frobenius norm as well as Structural Similarity Index (SSIM) for our project

B. Run Length Encoding

Another image compression algorithm is Run Length Encoding (RLE). While this algorithm is comparatively older and generally results in a worse compression ratio compared to fractal compression, this algorithm has a few advantages over it. First, RLE is lossless, making it better in situations where exact copies of data are necessary. Second, RLE can also be applied to datatypes other than images, such as text. It is even feasible to take a file compressed with fractal compression and further compress it with RLE, although the amount of added compression might be negligible.

The diagram below illustrates one example of how RLE could be implemented. This particular version works on a 2D black and white image (not to be confused with greyscale). It identifies similar pixels in every row, and it groups them together.



The diagram, while quite effective for demonstration, actually shows a rather poor implementation. For example, the first line, which was originally only 7 pixels long, got "compressed" into 10 characters. However, it was quite effective for line 3. This shows that the higher the number of grouped values in a row there are, the higher the compression ratio. This means that RLE is most effective on repeated data, and might actually be counterproductive for extremely non-repetitive data.

Additionally, the diagram shows a black and white image, so including the B and W characters into the encoding can actually be removed if we assume that the color changes after every number (we can use 0 if it needs to continue). But for images with more than two pixel-values (For example, greyscale or color images), the color values will be necessary.

II. Project Objective

Our project focused on improving levels of parallelization on Fractal and RLE Image compression algorithms using various concepts covered in purview of our course. This required a thorough understanding of the working of both existing algorithms and identifying stages where compression times can be optimized using GPU parallelization. Following is an introduction to how we approached the problem:

A. Parallelization in Fractal Compression

For Fractal Compression we investigated parallelization options in all three stages of compression. In the Compression phase, since we parallelized the segmentation task using CUDA based on the SIMT model we ensured that both sets of blocks were disjoint to avoid synchronization overheads. We aimed to experiment with grid and thread block dimensions to find the best performing configuration. This could not be done due to lack of time available for implementation and unexpected delays in implementing serial version of Fractal Compression. In the first case, each thread block was made responsible for downsizing a domain block to the dimensions of a range block by averaging neighborhood values.

In the transformation phase, we employed CUDA parallelism for transforming the downsized domain blocks. We also started work on parallelizing the image compression algorithm through OpenMP by replacing CUDA kernels we had used to process transformation phase and the compression phase in our project. While this part was not ready before our final presentation, some effort after enabled us to successfully achieve compression results in time for our final report.

B. Parallelization in RLE

We attempted parallelizing image compression using RLE Compression Algorithm by compressing different rows of an image simultaneously. There were a lot of hurdles in achieving this as we faced multiple challenges in handling end of a line and ensuring parallel execution of threads did not distort image by leaving out a few pixels from processing and that multiple lines did not merge with each other because they had the same color.

Once sequential parallelism was achieved, we extended the model to OpenMP parallelization algorithms to establish data points for comparing its relative performance to CUDA. At the end of this project, we detailed the compression time taken with different parallelization approaches that we covered as part of our course curriculum in section IV.

III. Code Implementation

A. Parallel Fractal Compression Implementation

For the implementation of Fractal Image Compression, we use the OpenCV library to abstract the image I/O and transformation functions. The key modules of the implemented code are discussed below

Compression Module:
This is the main function that reads the Bitmap (BMP) image and generates output in the Portable Pixmap (PPM) format. We chose to use the PPM format as a convenient intermediate format rather than a final choice of output file.

Generate Transformed Blocks:
This is the primary step of the compression which involves producing all the possible block combinations from the given image with arguments specifying the range and domain block sizes. The function first segments the image into square elements and reduces them by the compression factor, followed by computing the 8 affine transformations possible for that square (rotations of 0, 90, 180, and 270 degrees on the normal and flipped versions of the block). Along with the transformed blocks, the function returns a vector of specifications which is stored in the PPM file. Since the blocks are disjoint this provides an opportunity for parallelization. In the sequential version of the code, each resized block is iteratively processed to generate the transformed blocks. In the parallelized version, we use a mean reduce operation to compute the resized block and then generate the 8 transformed block in parallel. To avoid discrepancies at the block boundaries we use single-pixel padding.

Find Contrast & Brightness:
This function finds the best fitting contrast and brightness for a range block. It solves a simple least square problem for to find the values. OpenCV provides a direct abstraction for the sequential implementation which allowed us to keep the contrast as well as brightness as variable parameters. However, in the parallelized version we resort to a fixed value of contrast and just find the optimal brightness value. This was equivalent to computing the difference of the resized domain and range block and finding the absolute difference of pixel values. This function provided an excellent opportunity for using shared memory to speed up the calculation. However, the feature is not implemented in the current project and is a potential future improvement.

Find Similarity Score:
This module is used to compare the generated transformed block and current range block according to a given distance metric. We experiment with the Euclidean and Frobenius norm in our project since they provide the most straightforward implementation when parallelizing using CUDA.

Decompression Module:
The decompression module takes the PPM file as input and generates the corresponding image for display. Following the fractal image compression algorithm, we just start from a completely random image and apply the transformations for a fixed number of iterations. The decompression module itself does not have any data parallel tasks to be performed hence, the parallelization is implemented for the apply transformations function which is repeatedly called by the decompression module.

Apply Transformations:
This module is responsible for applying the affine transformations as well as the contract and brightness adjustments to the image block according to the following equation:

$$f_l(x_{D_k}) = s \times rotate_\theta\left(flip_d(x_{D_k})\right) + b$$

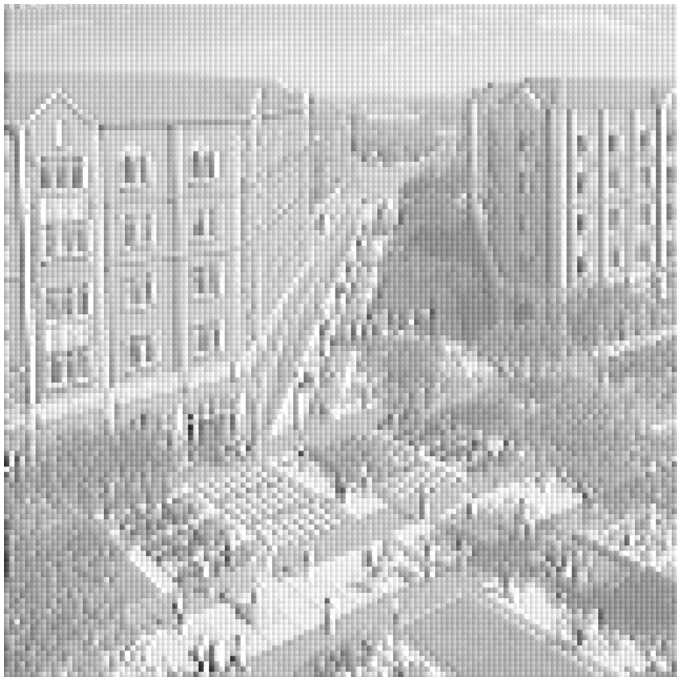
Since the image blocks are disjoint it allows us to modify it in-place. However, for the parallel implementation we must move the image block to device memory which leads slower performance compared to the sequential implementation. Hence, we omit the results from the parallelized apply transformations module since it skews the overall performance of the parallelized version. Once the functions fit we picked up images of different sizes to test the working of Fractal compression.

Below is the first image of dimensions 512*512 used.

Below you can see results of how the compressed version looks. An important thing to note is that we picked up the compression to happen in greyscale to be able

This image was tested for parallel processing using CUDA and OpenMP. Resultant compression ratio for parallel case was same as that of sequential because both were essentially achieving the same result with performance enhanced by multi cores or GPU.

Next, we saw if the algorithm can handle a 1024*1024 image. It was able to manage decent compression time with it as well.



We explored working with different destination block sizes to find the compression ratio:

For destination blocks of 4 x 4:
Space consumed in bmp image = 4 x 4 x 1 B = 16 B
After fractal compression, each destination block of 4 x 4 was represented using 6 parameters which collectively consumed a total of = 8 B
Compression Ratio = 2

For destination blocks of 8 x 8:
Space consumed in bmp image = 8 x 8 x 1 B = 64 B
After fractal compression, each destination block of 4 x 4 was represented using 6 parameters which collectively consumed a total of = 8 B
Compression Ratio = 8

However, as the destination block size increased the decompressed image turned increasingly susceptible to blocking artifacts.

B. Parallel RLE Implementation

To demonstrate the amount of data compression, we used BMP (bitmap) images as our input files. BMP files store uncompressed data, so ignoring overhead data (e.g. in the file header), they provide a good representation of the size of the uncompressed data. We then read the data into a 3D array. The first 2 dimensions represent the 2D nature of an image. The third dimension is relatively small in magnitude. It is an array of length 3, representing the 3 color channels: red, green, and blue.

To simplify matters, we restricted our input data to 256x256 and 512x512 images. This made it easy for us to divide the rows among the different CUDA threads and made it so that we did not need to worry about edge cases. This is not to say that RLE cannot be implemented on other image sizes; we merely wanted to focus on parallelization.

For testing purposes, we tested our code on a 512x512 image of the German flag. This produces a simple output, as each row of the image is a single color. There are three sections of the output, consisting of all black, then all red, then all gold. The image is shown below.

One issue we had to first address was how to deal with color images. Most digital images use three channels: red, green, and blue. This is commonly known as the RGB color model. We had to decide between treating each pixel as a single color, or compressing each channel separately. Because the latter would result in three separate run-lengths counts, we believed the former would result in better compression in the majority of cases.

Thus, we decided to combine each pixel into a single 6-digit hex value. This assigns 2 hex digits for each channel, which is able to represent each of the $16^2 = 256$ possible values on each channel.

As for the main algorithm, the sequential version simply processes each row of the image using a loop. For each row, the color of the first pixel is recorded, and a counter is started at 1. Then we loop through the rest of the pixels in the row. If the next pixel is the same color, we just increment the counter. If it's a different color, we record the new color and start a new counter. Once the row is completed, we have a list of colors and a count of how many times they appeared consecutively.

The parallel version expands on this by processing the rows in parallel. A simple way to think of it would be having one thread per row, and each of the threads processing their rows in parallel. In actuality, the threads may process more than 1 row, depending on the number of CUDA threads that were spawned. In any case, each thread only needs to focus on its own row(s), so there are no dependencies between the threads. This means that we were able to treat the problem as embarrassingly parallel.

One issue we had to address was CUDA's handling of multidimensional arrays. In particular, CUDA seems to be optimized for operating on linear 1D arrays, whereas our sequential implementation used a 3D array. We used a 3D array primarily because the library we used for reading BMP images read data into a 3D array. Thus, we had to flatten this data into a 1D array to allow CUDA to process it. This flattening was not included as part of the runtime, since we considered this to be logically part of the file IO.

However, this did mean we had to modify the kernel to calculate the index on the linear array. We initially ran into complications due to not calculating the indices correctly, but we eventually fixed this issue.

Initially, we wanted to further optimize this compression by combining consecutive lines of data. For example, in the heart diagram from section I, line 2 ends with B3, and line 3 starts with B7. This could be compressed further into B10. At the start of the project, we thought this would be simple, because we incorrectly assumed that we would just need to have each row check the row immediately before and the row immediately after, to see if they started/ended with the same color.

However, we realized this would not produce correct results in certain cases. To illustrate this, consider the case of a 10x10 image where all pixels are the same color say black. After each row is processed, we would have 10 rows, each saying 10B. The goal would be to combine them to achieve 100B. However, to achieve this, we would need, for example, row 1 to become 100B, and rows 2-10 to become an empty string. This would require (i) row 1 to receive communication from rows 2-10 about their color information, (ii) row 2 to know that it can combine with row 1, (iii) row 3 to know that row 2 is combining with row 1, and therefore it must send its information to row 1 instead of row 2, (iv) row 4 to know that rows 2 and 3 are combining with row 1, and therefore it must send its information to row 1 instead of row 2 or 3, (v) etc.

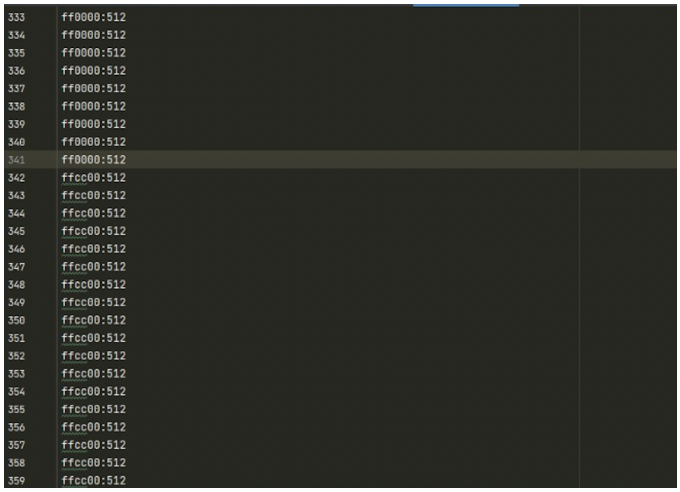
In essence, every row needs information from all of the rows before it, and because of the way they are interdependent, we were not able to find a way of combining them in this fashion that would be significantly faster than a sequential solution.

One other idea we had for this was to combine them in a tree-like manner, where each row would pair with another to create a "combined solution", and then those would (pseudo recursively) pair up until we reach a complete combined solution. However, this would have involved significant data communication between the different CUDA threads. Because the length of the compressed output is not known until runtime, each row is given two arrays (one for the colors and one for the counters), and the length of those arrays is initialized to the length of the row. This is to account for the worst-case scenario, where the row has no repeating colors. If we were to combine the row information in this tree-like fashion, then at each level, we would need to transfer so much data at each level that it would incur a significant time cost, which would negate the benefits of parallelism. Also, because of the nature of CUDA, using things like objects (e.g. C++ vectors) or pointers to memory in the heap would not be efficient solutions.

In the end, we decided not to combine consecutive rows together. While we could have used a hybrid solution, where we compress each row in parallel and then combine the rows in a serial loop, we wanted to focus on being able to analyze the benefits of parallelism in RLE. While our solution does not achieve maximum compression, it exhibits an extremely high amount of parallelism.

The primary output of the algorithm consists of 2 arrays. The first stores the color information of each line, and the second stores the run length (count) of each color. One might better think of this as an array of pairs of colors and run lengths. But because of the nature of CUDA, it was easier to implement using two separate arrays that are physically separate but logically paired.

However, this output had to be converted into a file. We opted to convert the output into a text (*.txt*) file. The primary reason for this was convenience. We wanted to be able to easily analyze our output to ensure that our implementations produced correct results. It should be noted that this is *not* efficient in terms of data compression for a number of reasons. First, each digit is stored individually, and this has effects on how the run-lengths are stored. For example, if the run length was 100, we would need 3 bytes to store this value (1 byte per digit), even though its binary representation can be stored in just 2 bytes. Secondly, ASCII wastes a lot of data because of all the control characters and punctuation it accommodates; our output only uses the digits 0-9, lowercase letters a-f, the colon (:), spaces, and the newline character (to put each row on a separate line). However, we were not heavily concerned with this, as our focus was on parallelizing the compression process (i.e. producing the 2 arrays of output), not on the details of the output file format (i.e. converting the arrays into a file format that is optimized for data compression).



A section of the text file, representing the RLE compressed German flag

As can be seen in the text output, between lines 341 and 342 (which is ⅔ of 512, the number of rows in the original image), the colors make a transition from #ff0000 to #ffcc00, which is the switch from red to gold. The hash symbol (#), which traditionally precedes a 6-digit hex encoding of an RGB value, is omitted to save data and maximize compression. A colon (:) is used to delimit the color code from the run length. If there are multiple colors on a row, their color-length pairs are output on the same row, separated by spaces. This is not pictured in the output for the German flag, but can be seen in another example in section IV. Each row of output represents a row of the image.

IV. Comparitive Results

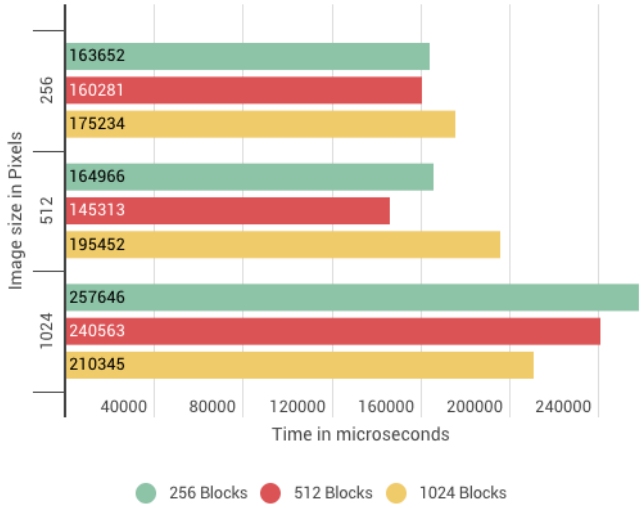
After parallelizing both RLE and Fractal Compression Algorithms for images, we attempted to compare their efficiencies and compression times in both sequential and parallel case. Before doing this, we established metrics point for compression times and efficiencies in both Fractal algorithms and RLE individually, attempting result analysis with different block sizes and parameters to compare sequential and parallel results in both algorithms.

Our expectation was that the Fractal Compression Algorithm should be able to handle any kind of image, and the RLE Compression algorithm should be inefficient while processing complex images with fewer pixel repetitions. On the other hand, we anticipated that RLE should overcome Fractal Compression Algorithm in terms of loss in data and compression times, as it does not involve any approximations to compress the image and has lesser calculations involved as compared to Fractal.

A. Fractal Compression Algorithm: Sequential vs Parallel

For Fractal Compression Algorithm, the first thing we did was to compare performances for sequential and parallel execution for image compression. We realized here that parallelizing different steps in the process helped us cut down the execution time for compressing a 256*256 image by ten times (All images compressed were of equal length and width for processing convenience. They were also of the dimensions 256, 512 and 1024 only). Execution time for sequential version of 256*256 image was 2719800 microseconds while execution time for the other was 175234 microseconds. (Chart comparison would be clearer in section C of Comparative results). We did execute the parallel results through OpenMP for a single stage and found that the time for execution was very close to that of the sequential version itself (1942550 microseconds). Hence, we did not work on extensively improving the model through OpenMP.

Next, we moved on to processing different image sizes using different block sizes for the Kernels. We found here that different block sizes worked better for different image sizes.



Here we noticed that execution timings were better when number of blocks used were same in number to the pixel size of the image. Although in 256*256 we see that parallel execution through 512 was more efficient, we could still conclude that when all threads were working on calculating range blocks for a single pixel row for transformation phase, the timings were better. We agreed this would be because of the impact of syncthreads() in between the parallel code for compression and range block calculation. Our code processed rows in the image by dividing the number of blocks available by the size of the image so that we could make sure the order of row processing remained consistent. This is also why we limited image processing to the three sizes seen in the comparison chart.

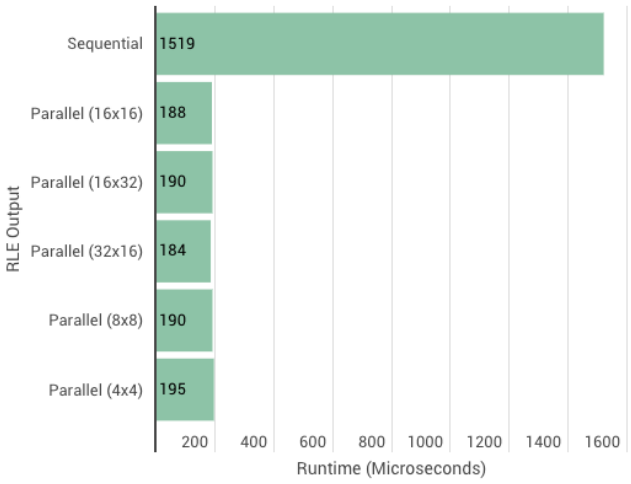
It was clearly seen that 1024 blocks seemed to demand more resources from the GPU and lead to greater memory usage, it still consistently provided most optimum results for different image sizes. Hence, we finalized the final code with a block size of 1024. 512 was close but the performance loss on higher resolution images was bigger in this case as compared to 1024, which would be a more realistic size of an image requiring compression in the real world.

Overall, there were quite a few learnings from experimenting with block sizes that weren't divisible with the image dimensions, optimizing image compression performance and playing around with the parallel CUDA code as we could see real implications of the learnings from our classroom in the project.

B. RLE: Sequential vs Parallel

In the case of the German flag image, we were able to compress the initial 769 KB file into a 5 KB text file, which is a mere 0.69% of the initial size. Notably, this performed slightly better than JPEG, which took 17 KB, although PNG was still more efficient at 4 KB.

On execution of RLE in both Sequential and parallel formats, we were able to compare execution times and output duration with different block sizes to check its impact on execution times. The execution times recorded here are for the German flag image.



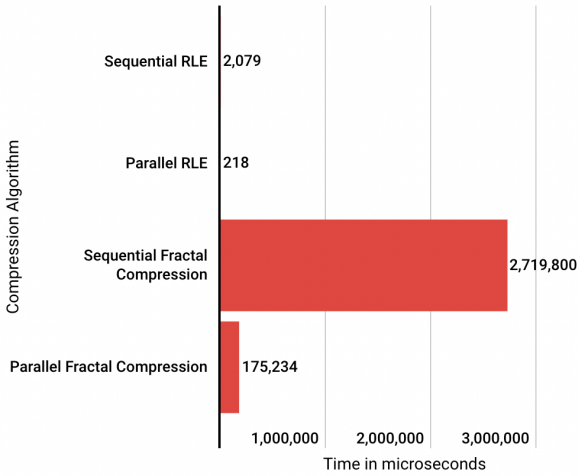
As seen in the graph, speedup from sequential to parallel was about 10 times going in sync with the output found for Fractal Image Compression as well. There were minor differences in the execution times found for varying block sizes while processing the image.

C. Fractal Compression Algorithm vs RLE

A final comparison between standard values for Sequential and Parallel implementations of Fractal and RLE was tabulated to find which one of the two performed better in terms of timing alone. Owing to a lot of calculations, as expected, we found that the average time for Fractal Compression was larger than that of RLE by an order of 1000.

However, as concluded from the previous sections on sequential vs parallel, we were able to bring down the execution time for both by an order of 10 by using CUDA and OpenMP to parallelize results.

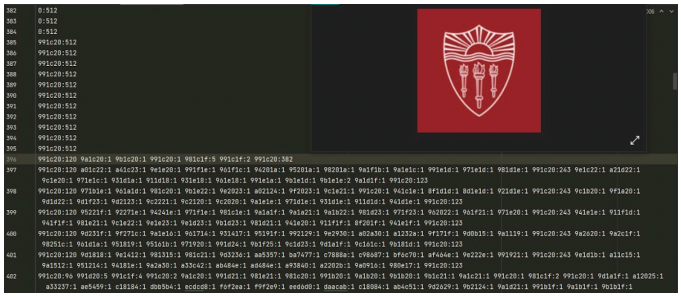
An important point to note is that the execution time for compression using RLE in this chart is different to that of the time captured for the previous section because it was run on the USC logo.



On comparison of the results for compressed image, we were also able to find that more complex images showed consistent compression rates with Fractal Compression algorithm than RLE as expected.

RLE was extremely effective with simpler images though, bringing down file size to 5% of its original form. However, compressed version of a more complex image gave a compression output of 60% through RLE.

Compressed form of the USC Logo image, that was used to make a comparative analysis of both compression algorithms, using Fractal compression algorithm can be seen above. In RLE, the Output file format used grouped pixels for storage which could not be seen in an image format. The resultant image was stored in a text file format containing long integer pixels in matrix form as shown in the next page.



In conclusion, we agreed on the point that both algorithms have their own benefits. For simpler images, RLE was both time efficient and effective with image compression. While Fractal is more time and resource consuming, its versatility and consistent performance makes it a very viable option for complex images. Its decompression power also enables image size modification with minimal losses in image quality.

V. Future Improvements

On completion of our project, we took a retrospective glance to understand the things we planned and what we can do to continue to make our code better. There were a few objectives we started work on but could not continue due to the lack of time and an underestimation of the complexity in handling image compression itself. Following are the observations on what we identified as possible improvements in the code for both algorithms.

In the final similarity calculation stage of Fractal Image Compression, we proposed to use an additional Peak signal-to-noise ratio (PSNR) criterion to see how it influenced the noise in the compressed image. Ideally each thread would calculate the similarity score for a given transformation and the corresponding range block. We proposed that we could assign each calculation to a thread block with each thread handling a smaller portion of the entire transformed block. This is still possible and even though not implemented, we thought we would take it up as a future enhancement to our project code.

As for RLE, the compression could further be improved by finding a way to combine consecutive lines that start/end with the same color, but we were not able to find a solution that can do this efficiently in parallel.

One further optimization might be allowing compression between colors that are similar. For example, the RLE output of the USC logo shows more than just 2 colors due to the “transition colors” that exist when the red changes to white (or vice versa). Allowing colors that are “similar enough” (e.g. having the RGB values fall within a certain margin of the previous pixel) to be combined would result in a better compression ratio, although this would turn the compression from lossless into lossy.

We also explored the possibility of improving both algorithms by introducing changes to their steps. Fractal Algorithm might have scope for improvement by identification of more processes that can be parallelized within the multiple steps introduced in the previous section, that can be taken as a future action item. RLE Algorithm could be enhanced by looking into the possibility of allowing compression if its nearby pixels are similar enough. This would turn it into a lossy compression but would result in a better compression ratio.

VI. References

[1] Run Length Encoding: https://iq.opengenus.org/run-length-encoding

[2] Imrpoved Fractal Image Compression Based On Robust Feature Descriptors: http://smartsenselab.dcc.ufmg.br/wp-content/uploads/2019/02/2011-Improved-Fractal-Image-Compression-Based-on-Robust-Feature-Descriptors.pdf

[3] Fractal Compression, Wikipedia page: https://en.wikipedia.org/wiki/Fractal_compression

[4] Fractal coding based on image local fractal dimension, Aura Conci, Felipe R Aquino: https://www.scielo.br/j/cam/a/6vf3NGkzvQmCnN6wkhZmZrv/?lang=en#

[5] Fractal Image Compression: https://pvigier.github.io/2018/05/14/fractal-image-compression.html

[6] Run Length Encoding: https://bbau.ac.in/dept/CS/TM/Image%20Compression%20and%20Run.pdf

[7] GeeksForGeeks and StackOverflow for a lot of debugging when we faced errors in coding

[8] Web resources for addressing CUDA compatibility issues to install libraries for proper compilation in local machines used