Parallelization of the JPEG Compression Algorithm

Kyle Barrows
Undergrad Student, Dept. Computer Science
University of Central Florida
Orlando, Fl, USA
barrowsk@knights.ucf.edu

Caleb Chase
Undergrad Student, Dept. Computer Science
University of Central Florida
Orlando, Fl, USA
calebchase2001@knights.ucf.edu

Mari Peele
Undergrad Student, Dept. Computer Science
University of Central Florida
Orlando, Fl, USA
mpmorrell@knights.ucf.edu

***Abstract*— JPEG image compression is widely used to reduce storage size of photographs for internet applications, digital cameras, medical imaging, surveillance systems, and wide variety of other uses. This lossy compression allows the user to adjust their output image to the ratio of image quality and file size appropriate to their use case. The compression method uses discrete cosine transformation (DCT) combined with a statistical encoder. In this project we investigated the possible benefits of parallelization on each the DCT and encoder algorithms. The experimental results were analyzed in two ways. The first by comparing runtime and correctness of parallelized vs non-parallelized implementation of the algorithm we created. The second method was to compare the runtime of our parallelized algorithm to well-known JPEG compression tools. The analysis involved statistical methods and visualization of data collected during the analysis describing execution time and image quality. The experimental results show a decrease in average execution time when using parallelization to compress images to JPEG.**

Index Terms—JPEG, concurrent, c++, lossy, DCT

I. Introduction

The JPEG format was first released as a standard in 1992 as a joint project from the Joint Photographic Experts Group committee, a joint working group of the International Standardization Organization (ISO), the International Telecommunication Union (ITU), and the International Electrotechnical Commission (IEC). Since that time, the standard has been updated eight times to include features like compliance testing, extensions, and a file interchange format.

JPEG image compression takes advantage of the human eye's increased sensitivity to luminance over chrominance. In other words, human vision is more sensitive to light than to color tone. This difference allows a method of compression that discards high-frequency information through discrete cosine transformation (DCT) and creates an image that appears almost identical but does not contain much of the original information. This is a type of lossy compression that greatly reduces the image storage size.

A compression ratio of 10:1 has little effect on the image quality while still significantly reducing image size. The user can choose the level compression or image quality that fits their specific needs. This methodology works best with images like photographs that lack sharp transitions and variations in tone. Images like sketches or photographs with sharp textures are more likely to create artifacts that distort the output image.

Additional caution should be taken with repeated edits, resizing, or cropping of the original JPEG output. Each alternation of the file decompresses and recompresses the file and loses additional information through lossy compression. Some tools have been created to provide lossless JPEG editing, but they only work for specific alternations and images of a minimum quality.

In comparison, PNG is an alternative image format that uses lossless compression to decrease image size while maintaining all original image data. PNG uses a palette of colors stores within the image data and lossless compression to create images that are capable of transparency and editing without losing quality or information. They are capable of handling images with sharp transitions, such as crisp logos. This capability comes at the cost of increased file storage size compared to JPEG images.

II. Methodology

A. Algorithms

For our experiment we implemented two different versions of the JPEG compression algorithm. The first algorithm we implemented was a standard JPEG compression algorithm that does not utilize any parallelization. The purpose of this implementation is to provide a baseline in terms of performance.

The second JPEG compression algorithm utilized parallelization to streamline the compression. During the compression process, the algorithm runs in 8x8 blocks on the

image. Parallelization will be used to run these 8x8 block operations on separate threads using dynamic load balancing.

B. Testing and Analysis

To test these algorithms, the following process was applied.

1. Image input to the algorithms was prepared. Three images at a pixel size of 2560 x 1440 were selected in PNG format. Two of the images are landscape images. The third image was a white image. These images were then down sampled to a pixel size of 1280 x 720, and 640 x 480.
2. These images were fed into the non-parallel and parallelized JPEG compression algorithms. Specifically, we tested using 1, 4, and 8 threads for every image and size. The average time to perform the algorithm for each image was recorded. To analyze our results, we have provided analysis that aims to answer the follow questions:
    a. What is the relationship between image size and runtime?
    b. Is there significant variation between image contents for a given size and runtime?
    c. What were the performance effects on the parallelized algorithm compared to the non-parallelized algorithm?
4. The results of the analysis have been provided in tables and data visualizations. In addition, written analysis has been included to accompany the visualizations.

III. BACKGROUND

JPEG compression can be broken down into five main components: YCbCr Color Space, Chroma Subsampling, Discrete Cosine Transformation (DCT), Quantization, and Encoding. In the following sections we will give a brief overview of each of these components.

A. YCbCr Color Space

YCbCr is a type of color space often used in digital media and photography. Like the RGB color space (red, green, blue), each letter in YCbCr represents its color format. Y represents the luminance component. Cb and Cr are the chroma components that can generally be described as the blue chrominance and red chrominance values. To utilize YCbCr, an image's RGB pixel values are converted. From Microsoft's specifications [8], a method is provided to convert RBG to YCbCr using matrix multiplication:

$$[Y\ Cb\ Cr] = [RGB] \begin{bmatrix} 0.299 & -0.168935 & 0.499813 \\ 0.587 & -0.331665 & -0.418531 \\ 0.114 & 0.50059 & -0.081282 \end{bmatrix}$$

Fig. 1. RGB to YCbCr Matrix [8]

The reason that YCbCr is used over other color spaces is because YCbCr corresponds with the sensitivity of the human eye to light. Humans tend to be more sensitive to change in luminance than chroma in images [7]. Therefore, YCbCr allows for compression of Cb and Cr components without having a minimal impact on the visible quality of the image.

B. Chroma Subsampling

In the JPEG compression algorithm, YCbCr properties are taken advantage of in a process called chroma subsampling. For a 2D array of pixels, the subsampling of the image is controlled by a parameter J:a:b. "J" represents a filter of J x 2 pixels. "a" represents the number of chrominance samples in the first row and "b" represents the number of chrominance samples in the second row. For example, 4:4:4 would provide no subsampling whereas 4:2:2 provides subsampling by a factor of 2.

C. Discrete Cosine Transformation

The Discrete Cosine Transform (DCT) is the main algorithm behind JPEG compression. DCT is a block compression algorithm meaning it compresses the data into blocks (typically 8x8 pixels for standard DCT) and attempts to represent each block as a sum of cosine functions with varying frequencies.

The input into the DCT algorithm is an array of integers, where represents the intensity of the pixel. Each pixel has a value ranging from 0-255. Using a set of precomputed basis functions, the DCT method will take each image block and output an array of integers containing DCT coefficients ranging from -1024 to 1023 [4]. The DCT coefficient represents the weight, or how much each one of the basic functions influences the final image.

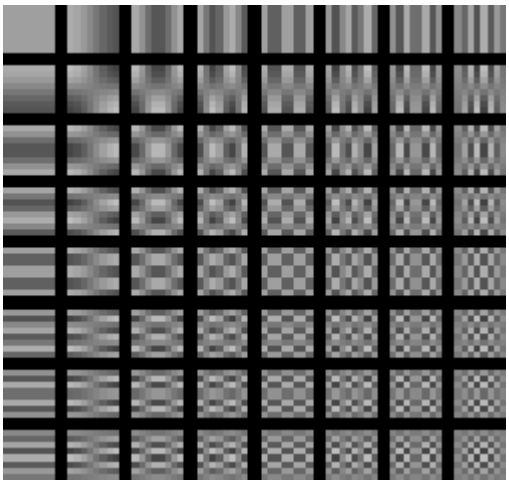

Fig. 2. DCT: basis images [9]

This image shows the 64 basis functions that the original image can be summed to recreate the original image. Each cell in the grid represents a cosine wave with a different frequency. As it moves right, the horizontal frequency increases and as it moves down vertical frequency increases.

As higher frequency ranges are not as perceivable to the human eye, the DCT coefficients for high frequency cells are often smaller.

Each 8x8 block of an image is given a matching 8x8 matrix, each cell represents a DCT coefficient for how much the matching basis image influences the final image. A higher coefficient means it has more influence on the final appearance. The top left cell is called the direct coefficient, or DC coefficients and represents an image with no frequency variance (i.e. no color differences). The remaining cells are alternating coefficients, or AC coefficients, and represent cells with different degrees of frequency variance. Generally, the lowest frequency cells have the highest coefficients and are thus the most important which lends itself to the next step.

D. Quantization

Once the DCTII values are calculated, JPEG compressions uses coefficient quantization to reduce the number of values stored for the image. It does this by reducing some of the values to zero so they can be compressed using runtime-length encoding. By using a precomputed quantization matrix that varies based on the compressor and the quality of the compression selected, the quantized value for each position in the DCT matrix is obtained with [5]:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Fig. 3. Quantization matrix example [10]

The larger the value in the quantization matrix, the more information will be discarded for each frequency range. This results in many values for less important, higher-frequency areas getting set to zero.

E. Encoding

To store the values in the table, each value gets collected in a zig-zag manner collecting data from lowest frequency to highest-frequency.

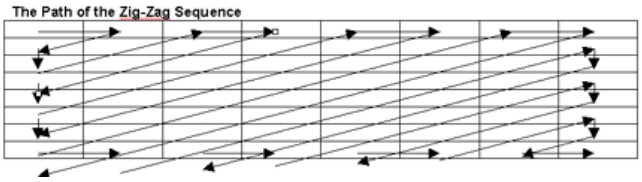

Fig. 4. Order cells are collected [6]

This means the most quantized parts of the image are grouped together. This can result in long strings of zeros since many values are usually set to zero during quantization. Using run-length encoding (RLE), a lossless compression algorithm, sequences of repeated data are represented by a single integer representing how many times it appears in the sequence [6].

IV. Parallelization Approach

Our approach for parallelizing the JPEG Compression algorithm involved coordinating threads to operate on sections of the entire pixel array. Each thread operates on its assigned 8 x 8 section and handles all steps of compression. First it will convert the RGB value of each pixel to the equivalent YCbCr values, then it computes the DCT coefficients, quantizes them, and encodes them.

To assign each thread we use the following method:

```
1   Procedure: block assignment
2       int row, col
3       row = row_index.load
4       col = col_index.load
5       while !done.load()
6           Col =col_index.fetch_add 8
7           Row = row_index.load
8           If col >= width
9               If row >= height
10                  break
11              else
12                  If row == row_index.load and col < width + 7
13                      row_index += 8
14                      col_index.store(0)
15          else
16              compressImage(image)
17      end while
18      done.store(true)
19  end procedure
```

This method takes in references to atomic integers, row and col. These atomics keep track of what row and column of the image array each thread should start at. Lines 2-4 initialize the variables row and col. In line 5 it iterates over the image array until every 8 x 8 block of the image is complete. Line 6 updates the col variable and increments the atomic col_index by 8 because that is the width of a block. Line 7 updates the

row variable with the content of row_index. Lines 8 and 9 check if col and row are within the width of height of the image respectively, if they are the code will run the compression algorithm on the image block. If col and row are both out of bounds, then each block has successfully been compressed and it can break out of the loop. Line 12 checks if col is out of bounds and shifts the row_index down by 8 as that is the height of a block and sets col_index back to 0.

The previous method handled assigning the blocks to each thread. To handle compressing each block of the image we use the following method as seen above:

```
1    Procedure CompressImage
2        If row >= height or col >= width
3            return
4
5        CreateYCbCrArray(image)
6        Vector<PixelYCbCr>  dctCoefficients(64)
7        DCT(image, dctCoefficients)
8        Quantize(dctCoefficients)
9        ZigZagEndcoing(dctCoedfficients)
10       For each component of YCbCr
11           Vector<pair<int, int>> rle = runLengthEncoding(dctCoefficients)
12           HuffmanNode tree = buildTree(rle)
13           Map<int, string> table = buildTable(tree)
14           String encoded = encode(tree, table)
15   End Procedure
```

The procedure compress image operates on each 8x8 block of the image and computes the Huffman table and tree which it can use to encode the array. Lines 1 and 2 are just a sanity check. Lines 5 through 9 run the primary steps mentioned above to handle compressing the image. Finally, lines 10 through 14 generate the encoded string for each component of a YCbCr image.

V. Results

Table 1 describes the raw data output from the algorithm performance testing. Here the size of an image, number of threads used, and average execution time for 10 trials is displayed. The execution time was tracked starting after the image was loaded into a pixel array in memory. Execution time was stopped being tracked after all components for the JPEG compression algorithm were calculated.

Table 1

Image	Image Size	Threads	Average Execution Time
(-)	(px * px)	(n)	(ms)
forestL.png	2560 x 1440	1	27281.8
forestL.png	2560 x 1440	4	6970.57
forestL.png	2560 x 1440	8	3918.74
forestM.png	1280 x 720	1	6816.71
forestM.png	1280 x 720	4	1738.44
forestM.png	1280 x 720	8	974.858
forestS.png	640 x 480	1	2266.81
forestS.png	640 x 480	4	578.33
forestS.png	640 x 480	8	329.261
flowersL.png	2560 x 1440	1	27339.7
flowersL.png	2560 x 1440	4	6955.7
flowersL.png	2560 x 1440	8	3921.03
flowersM.png	1280 x 720	1	6818.79
flowersM.png	1280 x 720	4	1742.28
flowersM.png	1280 x 720	8	978.168
flowersS.png	640 x 480	1	2265.05
flowersS.png	640 x 480	4	580.987
flowersS.png	640 x 480	8	326.418
whiteL.png	2560 x 1440	1	27337.9
whiteL.png	2560 x 1440	4	6927.57
whiteL.png	2560 x 1440	8	3917.88
whiteM.png	1280 x 720	1	6813.8
whiteM.png	1280 x 720	4	1735.67
whiteM.png	1280 x 720	8	974.075
whiteS.png	640 x 480	1	2273.22
whiteS.png	640 x 480	4	585.712
whiteS.png	640 x 480	8	325.86

Table 1. Average Execution Times for each image and thread count.

Table 2 describes the speed up in execution time caused by introducing more threads. For example, the speed up for 2560 x 1440 image going from 1 to 4 threads was 3.93 on average. Similarly, the speed up for 2560 x1440 image going from 4 to 8 threads was 1.77. Each entry that uses only 1 thread is marked with “baseline” for speed up to indicate that a speed up value is not applicable. For this table, it can be concluded that introducing additional threads had a positive impact on execution time. For example, going from 1 to 4 threads for each image size had the execution time sped up by a factor of 3.9 to 3.93. In addition, going from 4 to 8 threads sped up execution time by a factor of 1.77 to 1.78. The speed up from 1 to 4 threads was greater than 4 to 8. A possible explanation for this is diminishing returns described in Amdahl’s law. In addition, the implementation did not run into an issue where the number of threads introduced overhead to cause slowdowns. For a very small image size, the possibility of introducing slowdown with a high number of threads might occur. Further studies are needed to investigate this issue.

Table 2

Image Size	Number of Threads Used	Average Execution Time	Speed Up From Previous
(px * px)	(n)	(ms)	(-)
2560 x 1440	1	27319.80	Baseline
2560 x 1440	4	6951.28	3.93
2560 x 1440	8	3919.22	1.77
1280 x 720	1	6816.43	Baseline
1280 x 720	4	1738.80	3.92
1280 x 720	8	975.70	1.78
640 x 480	1	2268.36	Baseline
640 x 480	4	581.68	3.90
640 x 480	8	327.18	1.78

Table 2. Average Execution Times by image size with Speed Up.

Table 3 demonstrates the percent increase from the smallest average execution time for an image at a particular size to the largest average execution time for an image at the same image size. This table represents the effect of image data for a particular size on execution time. From this data, the greatest percent increase was only .36 percent. Because there was such little variation on runtime compared to the pixel data for a particular image size, it was concluded that pixel values have a negligible effect on runtime. However, it is important to note that this evidence has no claim over the effects on image quality.

Table 3

Image Size	Fastest Runtime Avg 1-Thread	Longest Runtime Avg 1-Thread	Percent Increase
(px * px)	(ms)	(ms)	(%)
2560 x 1440	27281.80	27339.70	0.21
1280 x 720	6813.80	6818.79	0.07
640 x 480	2265.05	2273.22	0.36

Table 3. Percent Increase in per image size with parallelization.

VI. Conclusion

From the initial findings of this experiment, we concluded that introducing parallelization to the JPEG compression algorithm would introduce a positive impact on the algorithm's performance. For each image size and content, there was a significant increase in performance measured by average execution time. This evidence provides support for further investigation regarding further optimizing our algorithm to see if even more increases in performance are possible. One other possible area to investigate is the effect on small image sizes. In particular, seeing if having larger number of threads for a small image size would still have increased execution time. Based on the initial success of our results, it would be interesting to see how our approach could be applied to other algorithms. For example, other image algorithms that work on block processing of data could also benefit from a similar approach described in this paper.

References

1. "JPEG vs. PNG: Which one should you use? | adobe." [Online]. Available: https://www.adobe.com/creativecloud/file-types/image/comparison/jpeg-vs-png.html. [Accessed: 11-Mar-2023].
2. K. Iqbal, "JPEG - image file format," *JPEG - Image File Format*, 08-Aug-2020. [Online]. Available: https://docs.fileformat.com/image/jpeg/. [Accessed: 10-Mar-2023].
3. "Overview of JPEG 1," *JPEG*. [Online]. Available: https://jpeg.org/jpeg/index.html. [Accessed: 10-Mar-2023].
4. The discrete cosine transform (DCT). https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node231.html.
5. *Lossy data compression: JPEG*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/coeff.htm.
6. *Lossy data compression: JPEG*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/lossless.htm.
7. S. Winkler, M. Kunt, and C. J. van den Branden Lambrecht, "Vision and video: Models and Applications," *Vision Models and Applications to Image and Video Processing*, pp. 201–229, 2001.
8. "[MS-RDPRFX]: Color conversion (RGB to ycbcr)," *[MS-RDPRFX]: Color Conversion (RGB to YCbCr) | Microsoft Learn*. [Online]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rdprfx/b550d1b5-f7d9-4a0c-9141-b3dca9d7f525. [Accessed: 10-Mar-2023].
9. IMAGE PROCESSING TOOLBOX. *Image Processing Toolbox Documentation*. https://www.mathworks.com/help/images/.
10. JPEG STANDARD QUANTIZATION TABLE | DOWNLOAD SCIENTIFIC DIAGRAM. https://www.researchgate.net/figure/JPEG-standard-quantization-table_fig1_331969197.