ECE420 Final Report
RAW to JPG Data Conversion

Yoonseo Choi, Elias O'Malley, Minho Lee
{yschoi4, eliasco2, minhol2} @ illinois.edu

I. Introduction

Although we live in a data-driven world, it's ironic that we don't always have enough storage to keep every single portion of it in our systems. Hence, engineers tend to utilize a widely used method called data compression to minimize the amount of data stored in memory and processing pipelines.

Today, we take this concept and translate it into the realm of image data: raw to jpg conversion. To briefly describe, raw image files are data that is minimally processed from a camera source to memory. Simply said, it's an image data type that is most accurately stored in memory as its physical representation. Although accurate, this form of data consumes massive amounts of a device's RAM and can cause detrimental latency in systems that communicate image data between submodules.

Our project aims to implement the fundamental ideas of the JPEG file type, including variable compression rate to allow for a trade-off of quality and size, and the ability to save binary files containing the image's converted information.

II. Overview of the algorithm

Given a rough sketch of the timeline of a raw image, the part our group will focus on is the RAW to JPG conversion block. To achieve this, we will implement each of the individual portions of the conversion.

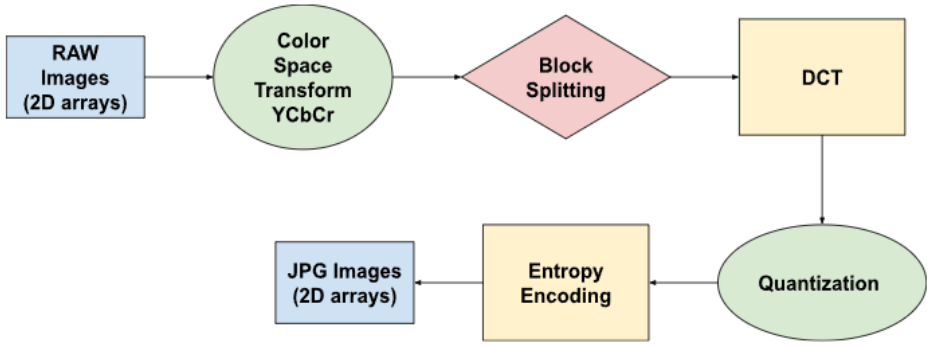

Fig. 1. RAW to JPG Data Conversion Flowchart

A. sRGB to YCbCr Conversion

Most modern devices today - whether that be smartphone, tablet, or TV - process images in the sRGB (Standard RGB) color space. The advantage of converting to this new color space is to allow for users to more conveniently modify the luminance and chroma of images/videos independently. Because luminance is more sensitive to the human eye, modifying its value even minute amounts can greatly impact the overall image quality. By transforming the picture to the YCbBr color space, the less important chroma channels can be compressed more while maintaining image quality.

To achieve this transformation, a simple matrix multiplication operation is performed on the RGB-valued pixels. The following is from an article published by Microsoft [6]:

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

Fig. 2. sRGB to YCbCr Conversion via Matrix Multiplication

The Y level is then shifted down by 128 to fit in to translate the values into the range -128 to 128

B. Block Splitting

JPEG splits the image into 8x8 tiles and then compresses each tile individually. This allows the algorithm to compress the image locally, where quality is less perceptible, while keeping the overall picture almost constant.

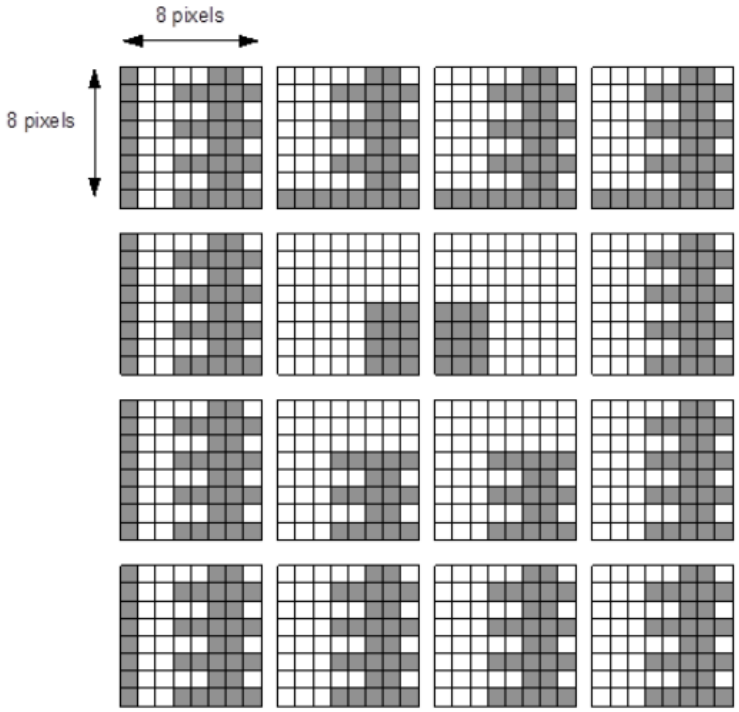

Fig. 3. 8x8 tile in a raw image

In "Image Transformation and Compression using Fourier Transformation", researchers tried running the compression algorithm on different block sizes (4x4 through 32x32). The performance of each algorithm was measured using the Mean Squared Error (MSE) from the original image as well as the Peak Signal Noise Ratio (PSNR). It was found that using an 8x8 block on a large image (1024x1024) resulted in the best compression ratio as well as good image quality (lowish PSNR). It was also found that large blocks on large images actually results in a larger file size rather than a compressed file. This demonstrates the effectivness of using 8x8 blocks.

C. Direct Cosine Transform (DCT)

Instead of compressing the image directly, JPEG compresses the frequencies of the image data. This is because the majority of the information stored in an image is part of the image signal's low frequency portion, so the higher frequency data is preferable to lose. A DCT is used since the image's color data is all real numbers. (See reference 1)

Direct Cosine Transform is given by the following equation:

$$F(u,v) = \frac{1}{4}C(u)C(v)[\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}] \quad (1)$$

where C(z) = $\frac{1}{\sqrt{2}}$ if z = 0
or = 1 if z $\neq$ 0

D. Quantization

In order to compress the frequency data, we will divide the DCT output by a quantization matrix. This allows us to get rid of smaller amplitude frequencies by dividing them down to zero. We will be using the standard JPEG quantization matrix. This matrix can be changed using a quality factor to allow for less or more compression.

As stated above, the luminance and the two chroma channels can be compressed at different qualities, so two quantization matrices are used:

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

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Fig. 4. The JPEG standard quantization matrices

E. Entropy Encoding

Since quantization reduces many small amplitude frequencies to zero, our data contains many zero values. In order to compress the amount of data needed to store the image, we format the data using Run-Length Encoding. This type of encoding algorithm allows data to be stored minimally in memory. Additionally, it is lossless, which means it preserves all the values it takes in as inputs. The first step in this encoding format is to transform the data into a 1D array. This is done through the "zigzag" pattern.

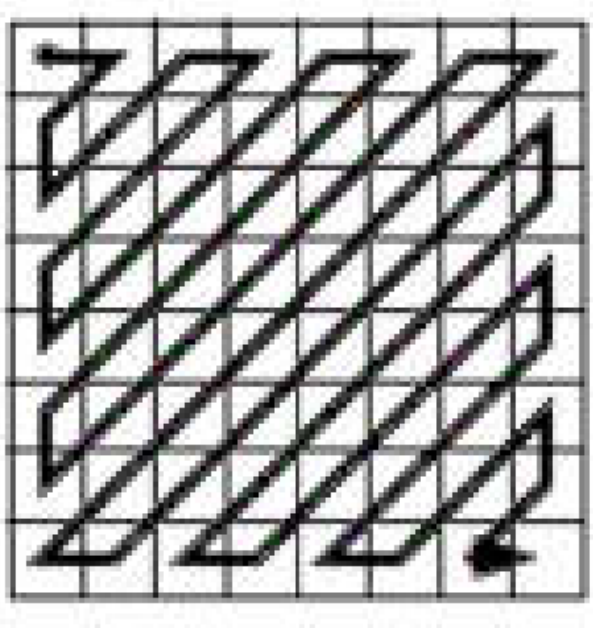

Fig. 5. The 2D to 1D zigzag transformation

Next, we represent each non-zero value in the array as the number of zeros before it, the number of bits to represent it (this is done for the decoding process where the data will come in as a byte stream), and finally the bit representation of the number.

Then, the run-length encoded data is written sequentially into a binary file.

III. App Structure

The app first asks the user to select a quality factor via a slider. If the compress button is selected, a small picture of a kitten is compressed. This is done so that testing of the full app can be done is a timely fashion since larger pictures cause much larger runtimes.

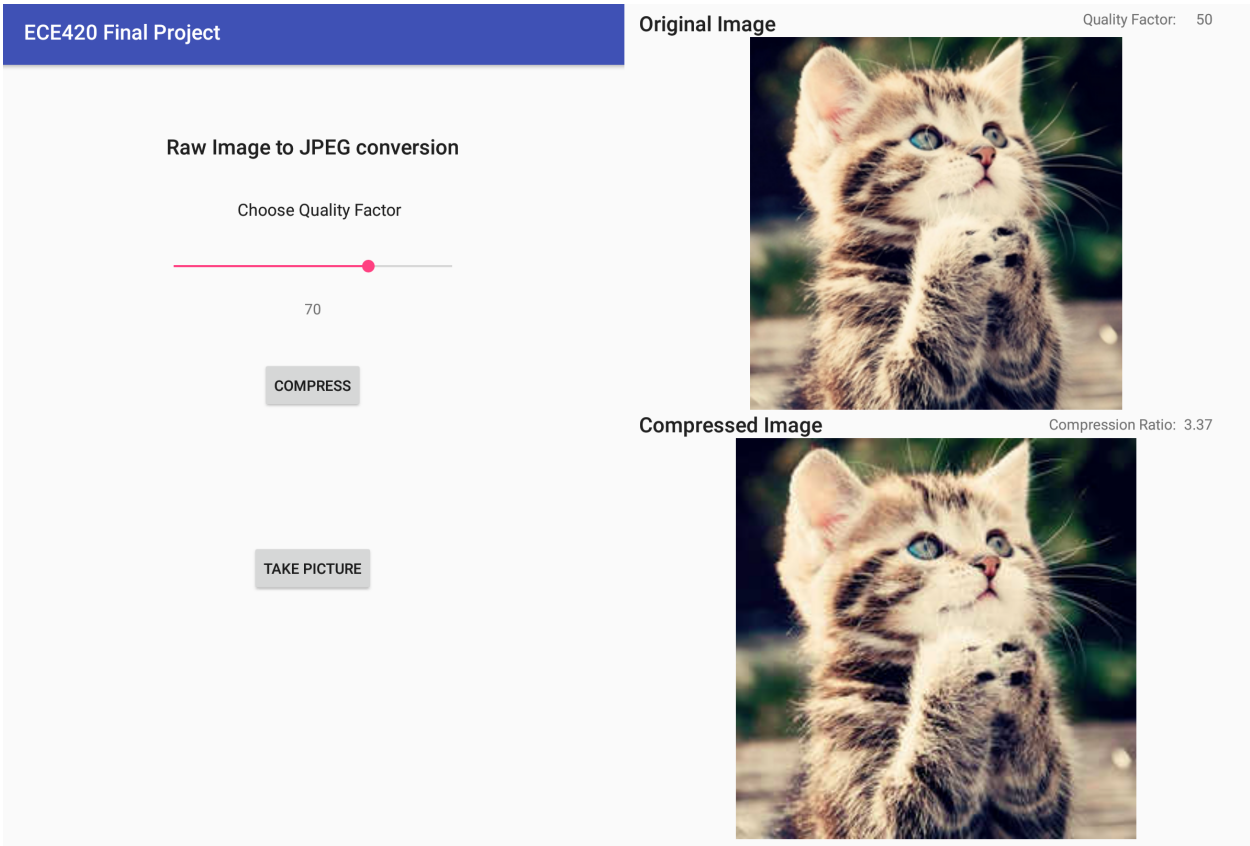

Fig. 6. The main page of the app (left) and result of compressing the kitten photo (right)

The result page includes the chosen quality factor, and the compression ratio, which is calculated by dividing the original size of the picture by the output file's size.

If the take button is selected, a page with a live preview of the camera is shown. Once the user selects the "Take Picture" button, a photo is saved and the compression algorithm is run, with the same results page as the kitten.

Fig. 7. The camera page (left) and result of compressing the taken photo (right)

IV. Results

Once the picture is compressed and saved to a binary file, in order to verify that the image was properly maintained, reversal algorithms were made so that the image can be read back from the binary file. This allowed verification of our programs as well as the ability to demo the results of compressing an image at different quality factors. As can be seen from the below image, at low quality factors, the block splitting becomes more obvious as the micro-level detail is removed almost completely.

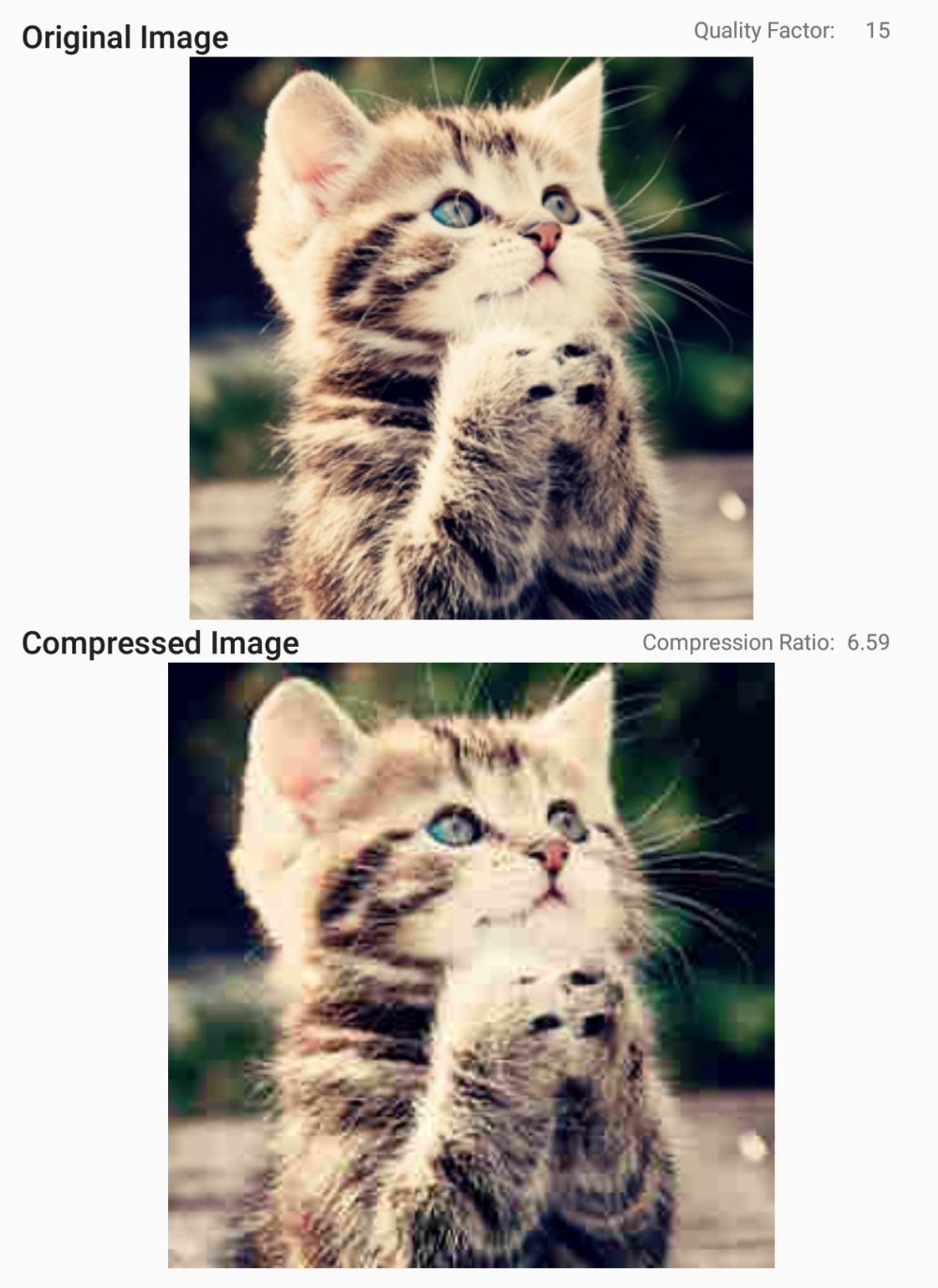

Fig. 8. The result of compressing the kitten at a low quality factor

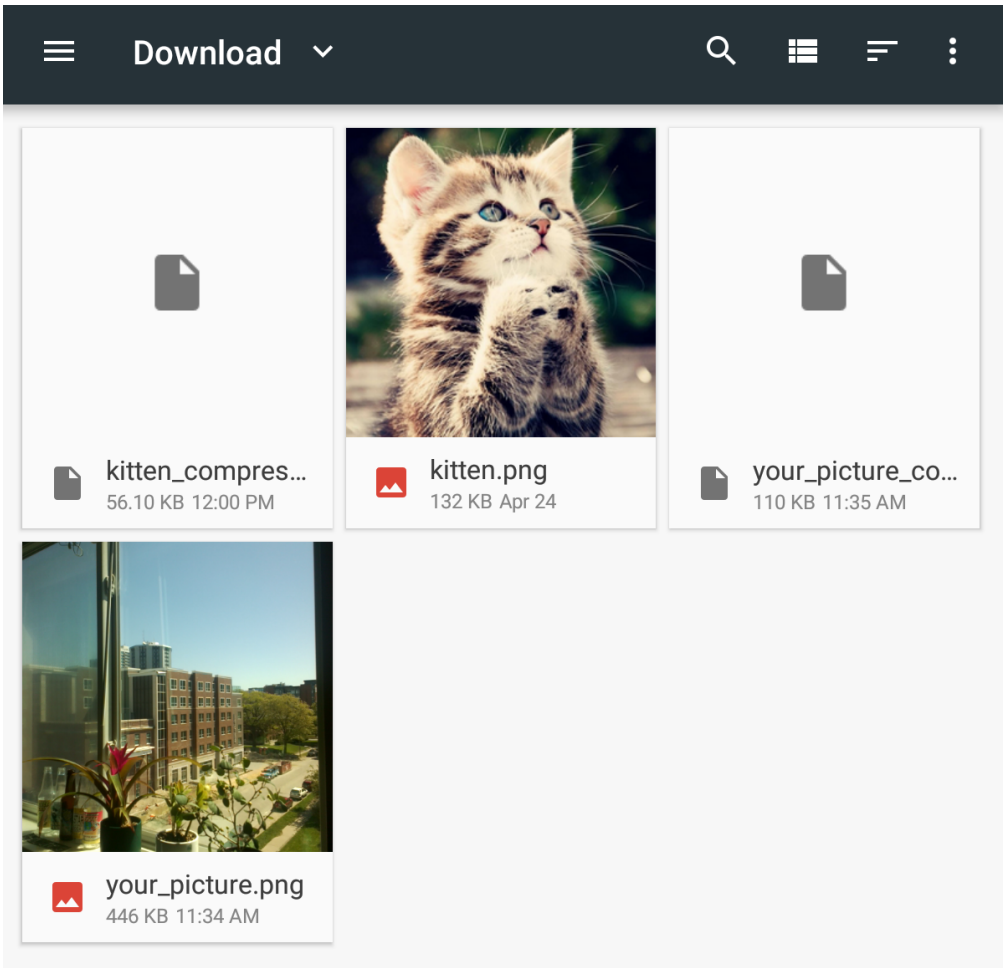

Fig. 9. The download folder containing the original images and compressed binary files

V. Suggestions for Extension/Modification

One major way to improve compression is by implementing huffman encoding. Huffman encoding is a way of encoding bytes through their frequency to decrease total size. By encoding more frequent bytes with smaller values, the total size of the file could be reduced. This was not implemented due to its complexity and it not being fundamental to the project's idea, but a future implementation could utilize huffman encoding to improve compression ratio.

Originally, compressing a live video feed was intended, but the algorithm proved to be too slow/complex for this to be possible. If the algorithm was dramatically improved, perhaps with implementing it in a faster environment than an android app on a tablet, then this might be possible.

REFERENCES

[1] "Image transformation and compression using fourier transformation," April 2015. [Online]. Available: https://inpressco.com/wp-content/uploads/2015/04/Paper1061178-1182.pdf

[2] "Jpeg - idea and practice," March 2013. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/9/9e/JPEG_-_Idea_and_Practice.pdf

[3] "Jpeg," Mar 2023. [Online]. Available: https://en.wikipedia.org/wiki/JPEG

[4] "Discrete cosine transform," Jan 2023. [Online]. Available: https://en.wikipedia.org/wiki/Discrete_cosine_transform

[5] "lossy data compression: jpeg." [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossy/jpeg/dct.htm

[6] "Discrete cosine transform (algorithm and program)," Aug 2022. [Online]. Available: https://www.geeksforgeeks.org/discrete-cosine-transform-algorithm-program/.

[7] "Discrete cosine transform - matlab & simulink." [Online]. Available: https://www.mathworks.com/help/images/discrete-cosine-transform.html

[8] "[ms-rdprfx]: Color conversion (rgb to ycbcr)." [Online]. Available: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rdprfx/b550d1b5-f7d9-4a0c-9141-b3dca9d7f525

[1] [2] [3] [4] [5] [6] [7] [8]