Analysis of CNN Performance utilizing JPEG Compressed Images Created on an FPGA

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science
Computer Engineering

---

by
Timothy "Mac" McMahon Shaughnessy
May 2024

Accepted by:
Dr. Melissa Smith, Committee Chair
Dr. Jon Calhoun
Dr. Tao Wei

Abstract

JPEG (Joint Photographic Experts Group) was formed in 1986 to create a method to reduce image size primarily for ease of transfer on the Internet. Released to the public in 1992, JPEG compression is a form of lossless compression that has been a staple for compressing images [22, 1]. JPEG is the go-to image compressor because it provides high compression ratios while maintaining visual integrity for the human eye [10]. Growing image sizes have made JPEG compression increasingly relevant. It is vital to keep up with growing data sizes for improved image handling performance on an edge device like a Field-Programmable Gate Array (FPGA). The JPEG compression format has had extensive research, but many potential applications have not yet been realized. Convolutional Neural Networks (CNNs) have become increasingly researched. CNNs boast the ability to interpret data across a variety of fields. In many cases, the data being collected is generated on a different device than the CNN. High-performance computers (HPCs) allow CNNs to run faster than on an edge device where data is collected. This is desirable, especially for more compute-hungry models that benefit from the additional performance provided by an HPC.

This research aims to investigate improving the transfer of images to a more capable computer and the effect on accuracy on a few different Neural Networks due to a reduction in image quality. Compressing images to JPEG frees up bandwidth, allowing images to be transmitted faster since there is less data to transfer from memory [14]. One of the primary investigations is to determine if JPEG images can be correctly interpreted by CNNs by comparing JPEG quality (various levels of compression and information loss) to CNN accuracy. Exploring the impact JPEG has on the runtime of the network will be analyzed to determine if the idea is viable. If successful, providing a universal method to improve the performance of a range of neural networks and the post-hoc transfer of the images while minimizing the reduction in model accuracy. Secondly, this research investigates developing the JPEG algorithm on an FPGA to determine the required resources of such

an implementation and the performance benefits compared to the CPU variation. This investigation will demonstrate the ability to improve neural networks designed to run off data collected on an edge device.

Dedication

To Father, who taught me to walk

To Mother, who taught me to speak

To Sisters, who taught me to listen

For the Tigers!

Acknowledgments

I would like to acknowledge my professors for their advice, encouragement and support throughout my collegiate career. I would like to specifically thank Dr. Melissa C. Smith for her ongoing instruction throughout my research and all the guidance she has provided during my graduate career. The expertise on FPGAs and other hardware she provided to me has been extremely valuable. Next, Dr. Jon C. Calhoun, was a major inspiration for my research along with the foundation learned in his data compression class for the ideas I built upon. Finally, Dr. Tao Wei has been instrumental with his assistance to utilize the Xilinx environment for developing my design on the PYNQ. In his FPGA classes, I learned the fundamentals of HLS, a vital component to the implementation and design on the FPGA in my research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In modern computing, many have concluded that data generation is growing more rapidly than transmission and storage solutions can handle effectively. In this research, JPEG compression and its ability to reduce the demands on a system's resources by reducing the image data size will be explored. JPEG compression is a commonly used lossy compression method, accessible on a wide range of devices and platforms, allowing for comparisons across devices [16]. Lossy compression is a type of compression where some information is lost when the information is decompressed, not matching the original data. Applications using lossy compression pipelines must tolerate loss created during the compression phase to continue effective operations. For example, an edge device installed in a satellite captures images to be transmitted back to an HPC on Earth or processed on board by a CNN [12]. The satellite has limited bandwidth, making compression a viable solution for faster data transfers. Many applications do not require as high a level of preservation, making JPEG effective. This effectiveness at reducing data sizes and maintaining visual integrity for the application makes implementing JPEG compression into an image pipeline desirable as long as it meets performance goals to benefit the system. This pipeline aims to reduce data size to overcome compression time during data transfers while the application performs within an expected loss tolerance using the compressed images.

Currently, neural networks are implemented in a variety of methods. These neural networks are growing increasingly common due to their effectiveness at object detection, semantic segmentation, and a variety of other tasks. Some of these implementations require high processing demands, which can be a limitation when processed on edge devices. While others can accomplish similar

results using cutting-edge techniques like a Binary Convolutional Neural Network (CNN) to reduce size [4]. The other option is to collect data on an edge device and transfer the information to a more appropriate computing device, as described in the satellite scenario. Each of the previous approaches as opposed to using a CNN may better suit a specific application, but the second provides two areas of interest.

The second implementation has the potential to leverage JPEG compression to improve transfer times and the performance of a neural network pipeline. Since JPEG can thrive when images are transmitted over a limited bandwidth interface, the first objective is to compress the images, freeing up bandwidth and allowing images to be processed sooner since there would be less data to transfer from memory. In this situation, JPEG compression can provide a solution that allows for control over compression, thus the amount of data stored and transmitted. The second goal is to reduce model inference time, allowing a smaller device to take advantage of a neural network faster while maintaining high performance using JPEG. This area will provide additional room for compression overhead.

To achieve these goals, I set out to implement a CNN that could take advantage of JPEG compression in the hope of seeing a decrease in model execution time and the post hoc transfer of the images while minimizing the reduction in model accuracy. JPEG compression is valuable in this comparison because it is a commonly used compression method that is accessible on a wide range of devices. It can be deployed rapidly and is an ideal choice to test how various compression levels affect the performance and accuracy of a Neural Network.

1.1 Research Questions

The focus of this research was primarily about JPEG, FPGAs and CNNs. This work aims to answer the following questions regarding these topics:

- Can a Convolutional Neural Network (CNN) take advantage of JPEG compression?
- Does JPEG on an FPGA perform differently from the software implementation?
- How is post-hoc transfer of the images affected while minimizing the reduction in model accuracy?

Chapter 2

Background

The motivation for this research stemmed from the desire to increase performance using JPEG compression on an edge device before transmitting data to an HPC for a CNN to classify. Testing the effects of compressed images as the input for a Neural Network seemed to be a viable research path [13, 24, 18, 15]. The idea pursued was that JPEG compression could benefit a neural network when interfacing with an edge device through faster transfers. The rapid increase in the use of machine learning in everyday life has driven the need to use less powerful hardware to run a CNN, but new ways to squeeze performance on a device with limited capabilities need investigating. This research sets out to use an edge device, but a software implementation as a benchmark for expectations was necessary to determine the compression behavior. This idea compresses images on a local device and transfers them at a much greater speed. A compressed image can be transmitted to get a result from an HPC where the network is running. Since machine learning is becoming more common in local devices like IoT or other edge devices that gather information [3], faster data transfer in areas that can generate massive amounts will be needed. Current examples of machine learning in near real-time on an edge device include making predictions like how to classify a specific image, perform semantic segmentation, or perform speech recognition [9].

Due to its flexibility and reprogramability, the goal is to implement a JPEG compressor on an FPGA. This research targets systems that are in place with slow transfer speeds. If an FPGA is the edge device, the design can be implemented on part of the fabric and increase the system performance. The draw for this implementation is the ability to easily incorporate the design into an existing system. Additionally, the use of Vitis HLS for adapting the speed and size of the input

is a minimally invasive process [2].

Chapter 3

Related Work

3.1 Convolutional Neural Networks

Neural networks are becoming highly prevalent in today's society, performing tasks ranging from image identification to assisting in controlling self-driving vehicles [11]. Convolutional Neural Networks (CNN) are a specific type of Deep Neural Network (DNN) designed for visual data like images. In a basic CNN, the input layer is an image, and the output is a list of predicted probabilities that the image is of a particular class. The class with the top confidence is the model's prediction of what is in the image. The class with the highest output value is the model's prediction. CNNs can also perform tasks such as semantic segmentation, which performs similarly as previously described but can break out multiple objects, terrains, or features from a single image. The focus of Neural Networks in this research is on image classification CNNs, but many ideas are applicable across various CNN implementations with varying levels of success.

3.2 JPEG Compression

The JPEG Compression algorithm was introduced by the Joint Photographic Experts Group (JPEG) in 1992. After six years of development and many competing designs, the group determined an 8×8 DCT (discrete cosine transform) to compress the data was the best option they had come across [16]. This option was selected from 12 other options after six years of research into the best compression for visual integrity to the human eye [1]. Multiple improvements have been made in

the years since its introduction, helping JPEG remain one of the most widely used lossy image compression methods today. It is important to note that this research will not investigate JPEG-2000, a newer version of the JPEG standard that additionally supports lossless compression and uses a wavelet technique rather than DCT. JPEG-2000 has many advancements, but the simplicity of JPEG has kept it more prevalent on the internet and in most common use cases [19]. A basic breakdown of the methodology of the JPEG algorithm is as follows: First, an image is converted from the RGB color space to the YCbCr color space where Y is the luminance component (brightness) and Cb and Cr are the chroma channels for blue and red respectively [21, 23]. This results in three matrices of pixel values with dimensions equal to the original image dimensions. Next, blocks of four-by-four pixels in the Cb and Cr channels are averaged to provide an initial reduction by a factor of two without noticeable image degradation. Each channel is then split into 8 x 8 blocks and centered about 0 by subtracting 128 for a standard [0,256] value image. These newly centered values are then converted to coefficients in the frequency domain using the 2-D Discrete Cosine Transformation in equations (3.1) and (3.2) below where p(x,y) denotes pixel location in the input image.

$$D(i,j) = \frac{1}{4}C(i)C(j)\sum_{x=0}^{7}\sum_{x=0}^{7}p(x,y)\cos\left[\frac{(2x+1)i\pi}{16}\right]\cos\left[\frac{(2y+1)j\pi}{16}\right] \tag{3.1}$$

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}}, & \text{if } u = 0 \\ 1, & \text{otherwise} \end{cases} \tag{3.2}$$

The JPEG standard defines default quantization matrices, but different software may use custom or in situ matrices calculated based on image data for more accurate results. If using the preset matrices, a user-specified JPEG quality will scale the values for two quantization matrices, one for luminance and the other for chroma, down (higher quality image) or up (lower quality image). The selected quantization matrices are used to divide the coefficients with rounding, which results in a large number of them being reduced to zero and thus increasing compressibility. Run length encoding (RLE) is employed on the quantized DCT values, following a zig-zag pattern to group similar DCT coefficients together and reduce entropy. Huffman Encoding is used as a final method of data reduction on the output of the run length encoding. JPEG decompression executes the process in reverse, retrieving the quantization matrices and Huffman tables from the image

metadata.

3.3 Common Compression Metrics

A compression ratio describes how effectively data size is reduced. The compression ratio is the most common compression metric to determine a compressor's effectiveness. This metric provides a standard way to describe how much compression is done to the original file. The equation for calculating a compression ratio is straightforward dividing the uncompressed file size by the compressed file size. While simple to compute and understand, this metric doesn't provide details about the quality of the compressed data and what information is lost during compression.

Other metrics like Structural Similarity Index Measure (SSIM) and Peak Signal-to-Noise Ratio (PSNR) in tandem with compression ratio can provide better insight into how well an image is compressed and to what degree the data is degraded. SSIM is a method of measuring the similarity between two images. In this case, the uncompressed and compressed images. When computing the overall SSIM value between two images, multiple SSIM values are calculated on the "subwindows" of the two given images. The SSIM between the subwindows (x and y) of two images (a compressed and uncompressed version) is given by:

$$\text{SSIM}(x, y) = l(x, y)^{\alpha} \times c(x, y)^{\beta} \times s(x, y)^{\lambda} \tag{3.3}$$

In **Equation 3.3** above, l, c, and s are luminance, contrast, and structure, respectively, and are defined by the formulas in **Equations 3.4**, **3.5**, and **3.6**below:

$$l(x, y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1} \tag{3.4}$$

$$c(x, y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2} \tag{3.5}$$

$$s(x, y) = \frac{2\sigma_{xy} + c_3}{\sigma_x^2 + \sigma_y^2 + c_3} \tag{3.6}$$

These formulas use μ to indicate the pixel sample mean of x and y. σ represents the variance of x and y except when denoted as σxy where it indicates covariance. Lastly, c_1, c_2, and c_3 use L, the dynamic range of the pixel values, to be defined, and default values k_1=0.01 and k_2=0.03. These

definitions give the following values that aim to stabilize weak division:

$$c_1 = (k_1 L)^2 \tag{3.7}$$

$$c_2 = (k_2 L)^2 \tag{3.8}$$

$$c_3 = \frac{c_2}{2} \tag{3.9}$$

When using this metric to compare two images, it is the standard for describing when an image has high structural similarity with a result close to 1 or a low structural similarity closer to 0. SSIM will be a helpful measurement when determining why an image is identified correctly by a neural network.

Chapter 4

Research Design and Methods

This research is broken into two primary sections. The first is running a CNN using JPEG data. The other research area is the implementation of JPEG methods on an FPGA. The HPC used for testing is Clemson University's Palmetto Cluster. The FPGA used in this research is a Xilinx PYNQ-Z2. The PYNQ was selected as it offers a system-on-chip (SoC) design that uses an integrated processing system (PS) and a Programmable Logic (PL) unit, allowing users to create designs taking advantage of the entire chip on a single die. The PS utilizes a dual-core ARM® Cortex®-A9 processor to run the python environment [2, 17]. The only other device used is a Mac Studio with an M1 Max for decoding encoded data. The specific design setup will be detailed further in the following sections. **Figure 4.1** provides a design overview of how input data will flow from the FPGA to the CNN on another more powerful computer.

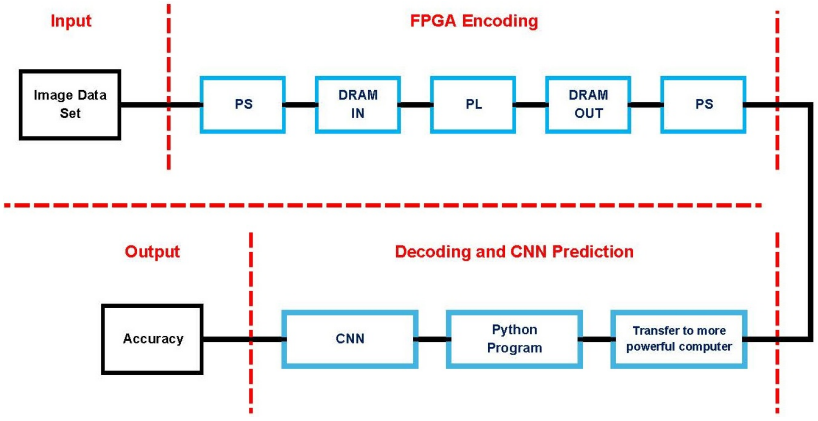

Figure 4.1: Design Overview

4.1 CNN using JPEG Data

For this research to be successful, a method to determine how to compress this data in a controlled fashion and determine what models and data were well suited for this research. The following portion of this research focuses on how compressing images affects the performance of CNNs.

4.1.1 Compression Phase

To effectively compress this data, prior to the model implementation phase, a Jupyter Notebook that leveraged the Python Imaging Library (PIL), referred to as Pillow, to compress a given JPEG image with either `.jpg` or `.jpeg` extension was implemented[7]. After the design successfully compresses images, the compression ratio and other metrics need to be analyzed to determine the effects. A dictionary and a pandas dataframe were utilized to collect post-analysis results for each image given a set of inputs. The Jupyter Notebook has the following parameters that can be selected as seen below in Figure 4.2.

Figure 4.2: JPEG Compressor Options

Here users can select one target metric: CR, SSIM, or PSNR, and then must select a desired range over which to iterate with a selected step size. Once the desired inputs have been selected

one must also set the input and output folders. The file structure of the input folder is crucial for the CNNs to run correctly. The original dataset uses a subfolder system to separate images based on classification, thus it is vitally important for the output folder to maintain this structure since the model relies on the structure to accurately correlate its predictions to the truth labels of the test dataset. The file structure contains a main dataset folder which contains 1,000 uniquely named sub-folders. Each of these sub-folders corresponds to a specific classification category and contains all images within that category. In order to maintain the integrity of this structure, the compressor is fed the input folder (the main dataset folder) and then iterates through each subfolder. The images within this folder are compressed to meet the target metric and subsequently saved to an output folder of the same name as their category folder within the user-defined output directory.

With the proper structure set up for the model, a method to quickly ensure the desired target metric was being met was required. A binary search tree that would take advantage of JPEG compression using a range of 0-100 allowed the search to take at most eight runs for the compressor to find the ratio closest to the desired ratio, always overestimating, was determined to be the most effective method to ensure a metric was as close as possible. This approach is a massive improvement over the naive implementation that required up to 100 iterations to achieve the target metric. The binary search tree was successful by varying the step size, evaluating if its result is above or below the target, and adjusting accordingly. This would narrow in on the correct target metric much more effectively providing a major speed-up. After the correct JPEG quality is found and the image saved, the program calculates the final metrics of an image by calculating the CR, SSIM, and PSNR and storing other data points like execution time and file size in a dataframe. The process of finding a metric and storing the results is done for each image in the input dataset. After the whole folder has been compressed, the data is ready to be passed to the model to run and test the accuracy among other parameters. A simplified version of the compression algorithm was later developed due to a variety of reasons. The simplified version maintained the file structure requirements, but instead of a binary search tree, compression overhead is reduced by compressing to a specified JPEG quality. This reduction speeds up the compression phase by up to eightfold, the importance of which will be discussed in the results below.

4.1.2 Modeling Phase

4.1.2.1 Programming Environment

For this implementation, Python was chosen as the programming language, implementing the TensorFlow library which is focused on training and implementation of deep neural networks. Within the TensorFlow framework is support for Keras which is an open-source deep learning API that provides access to many of the tools necessary for image import, image transformation in preparation for model input, and dataset selection. For simplification and standardization, all models were obtained from Keras Applications which is a set of deep learning models that can be implemented utilizing pre-trained weights.

4.1.2.2 Model Selection

Four models were selected from the available Keras Application models: Resnet50, VGG19, InceptionV3, and MobileNetV2. These models provide for a variety in model size, number of trainable parameters, and model layers. These model characteristics are displayed below in **Table 4.1**.

Model	***Size (MB)***	***Parameters***	***Layers***
ResNet50	98	25.6M	50
VGG19	98	143.7M	19
InceptionV3	92	23.9M	48
MobileNetV2	14	3.5M	53

Table 4.1: Model Characteristics

Model size has implications in federated learning where the storage capacity of client devices may be limited and model size will need to be weighed with model performance. VGG19 is one of the largest models available through Keras, MobileNetV2 is the smallest available model, and ResNet50 and InceptionV3 are in an average size range. In the context of deep learning models parameters are, in general, weights that are learned during training and applied layer by layer to the input image resulting in an output vector of predictions. In general, more parameters can result in increased model performance but with a tradeoff of increased risk for overfitting to the training data. As a result of overfitting the training data, the model's ability to generalize to unseen data is negatively

affected. By selecting a set of models at varying numbers of parameters or layers, a variation of complexity can be tested across the models.

4.1.2.3 Model Implementation

In this implementation, each model was loaded from Keras with pre-trained weights, developed from training on the uncompressed ImageNet database. Images were then selected from increasingly compressed versions of the ImageNet subset and passed through model-specific Keras preprocessing functions in preparation for model implementation. Preprocessing takes images from the input dataset and performs pre-defined manipulations (including resizing and color channel reordering). The output of the preprocessing is a tensor array containing the preprocessed images converted to floating point numbers, each in the format (image_height, image_width, color_channels). These test images were then run through each of the four models and post-analysis data was collected, including overall accuracy, the top five highest confidence predictions per image, and model runtime.

4.2 JPEG Algorithm on FPGA

The JPEG Algorithm consists of six primary stages as previously discussed. However, FPGAs do not operate well with data lengths that are not fixed, usually failing before a bitstream can be synthesized. This portion of the research led to an investigation into how to implement each part of JPEG except for Huffman encoding. After running RLE, the length of the output is dependent on the input data making it impossible to determine the output size before execution. The uncertainty in data length makes creating an FPGA implementation of a Huffman encoder as well as the ability to write to a file of unknown size with the appropriate headers is considered beyond the scope of this research. The most viable options for an implementation of the JPEG phases are on the PS side of the FPGA, by an HPC, or on the receiving end of the data if needed. Nevertheless, this section will focus on compressing image data to RLE data on the PL side of the PYNQ-Z2 and how the verification of these results will be conducted in this portion of this research.

The following design will be implemented as depicted in the data flow overview in **Figure 4.3**. The design uses Vitis HLS 2023.2 to handle creating an IP for Vivado [2]. The implementation is geared for an input size of 512 by 512. This input size was selected for two primary reasons.

First, nearly every image in ImageNet is smaller than 512 by 512 and can be scaled up to these dimensions if it is not the correct size, then reverted to the original shape after the compression is performed. Additionally, this size is pushing the upper edges of the board capabilities. However, the design accommodates any value divisible by 16 provided the board can support the size of the image in DRAM and potentially BRAM based on the RLE implementation. The design depicts the data flow of the implementation on the FPGA PL side.

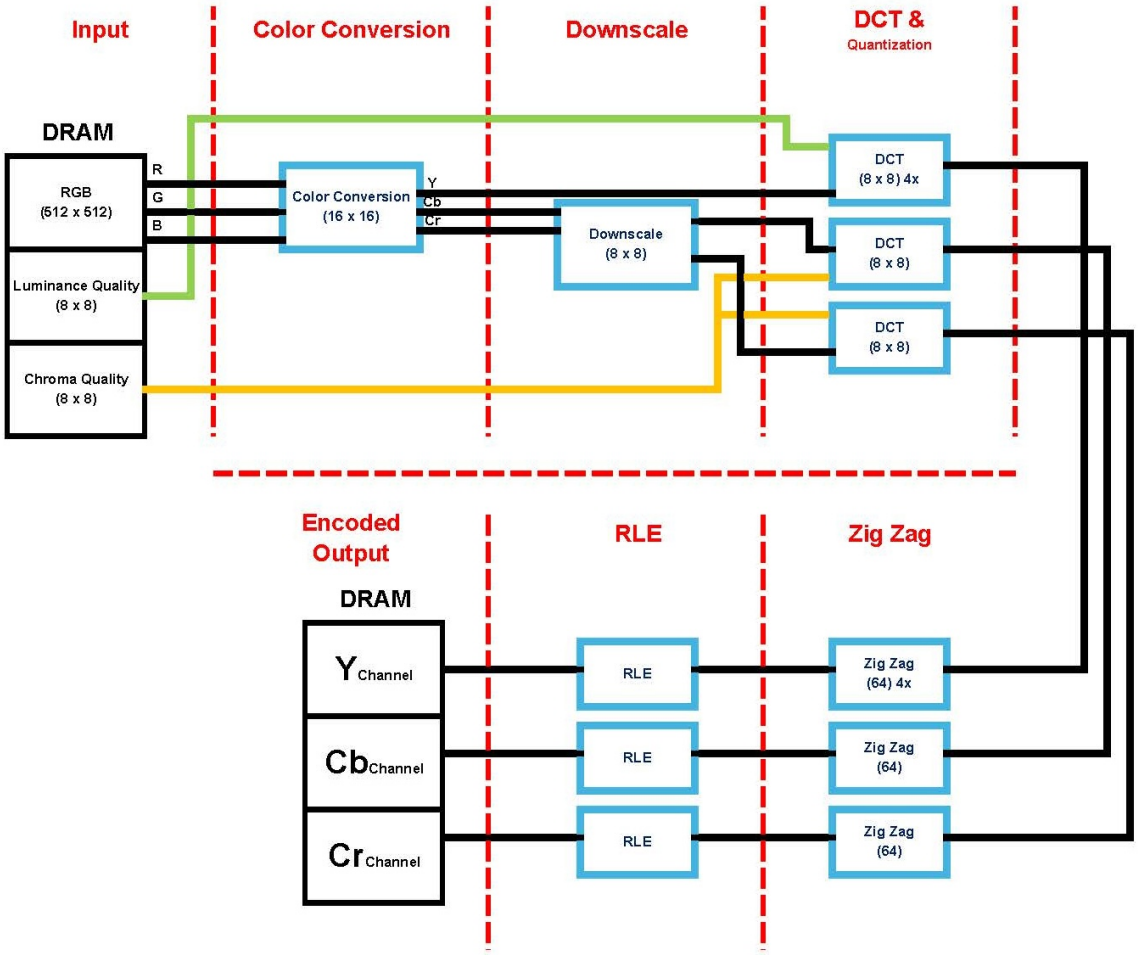

Figure 4.3: FPGA Image to RLE Dataflow

4.2.1 Color Conversion

Implementing an IP that was capable of converting an image from RGB to the YCbCr color space was the initial attempt at creating a Vitis HLS project of scale. Due to limited knowledge at

the time, two IPs were created one that uses AXI-stream, and the other stores the data in DRAM. In both designs and for all following implementations, the input data is a matrix of unsigned 32-bit elements with dimensions 512×512×3. Ideally, we could use 24-bits since we have three 8-bit channels, but due to the access pattern, we must use 32-bits with 8 do not care bits. Nevertheless, the objective of the IP is to pass RGB data (the input) through **Equations 4.1**, **4.2**, and **4.3**.

$$Y = 0.299 \times r + 0.587 \times g + 0.114 \times b \tag{4.1}$$

$$Cb = 0.564 \times (B - Y) \tag{4.2}$$

$$Cr = 0.713 \times (R - Y) \tag{4.3}$$

Once the data has been manipulated by the IP, the data is now represented in the YCbCr color space. The IP should additionally be able to handle passing the output to the appropriate receiver for the data.

4.2.1.1 AXI-Stream and DMA Approach

When starting this research, using Vitis HLS and technologies like DMA and AXI-Streams seemed like the appropriate implementation tools. They provide the ability to read the 32-bit input, operate on a range of bits as needed, perform the conversion, and write to a 32-bit output. However, the design only needs to write 24 bits of interest. To correctly communicate with this IP, a block diagram as seen in **Figure 4.4** is needed to pass the matrix of data from the PS to the PL side of the FPGA.

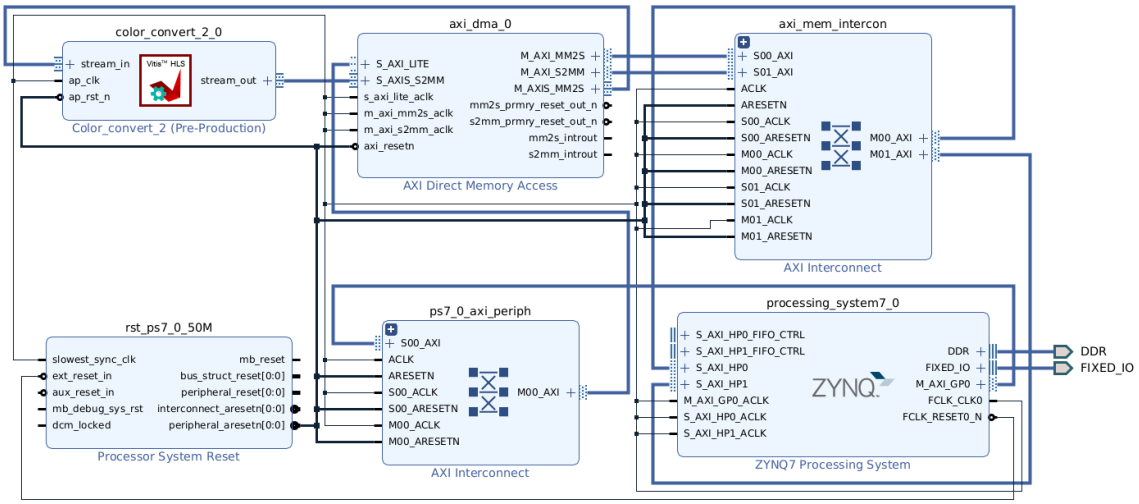

Figure 4.4: Color Conversion Block Diagram AXI-Stream

The data is passed from a Jupyter Notebook to using DMA to the color_convert block in the diagram. The benefit of using an AXI-Stream is the data is one in and one out, making it easy to keep track of how data is handled. Furthermore, this is ideal for timing and any parallelization can be done later knowing the serial potential is maximized. However, the AXI-Stream approach is only viable if functions for the final JPEG design are handled in independent HLS IPs.

4.2.1.2 DRAM Aproach

This was the second iteration of a color space conversion with newly learned best practices of the language and a better understanding of the requirements for later components such as chroma downsampling and matrix multiplication. Adopting a DRAM approach was vital for moving forward in a single HLS IP and keeping the manner in which data is read consistent throughout the project. These changes are helpful for understanding the design and data flow as well as allowing later functions to operate more easily.

In the DRAM approach, the expected input data in the design is changed on the HLS side. This approach expects the input type to be `struct RGB`. This structure contains four unsigned 8-bit numbers: three integers are associated with RGB, and the fourth is left blank. This was a massive improvement, and the implementation was no longer confined to writing and reading values from consecutive locations in the memory.

In this approach, an input image is read in blocks of 16×16 (reasoning for a block size of 16×16 discussed later in **Section 4.3**). The design then operates on each individual pixel in the block, passing RGB values through **Equations 4.4**, **4.5**, and **4.6**.

$$Y = 16 + (((r << 6) + (r << 1) + (g << 7) + g + (b << 4) + (b << 3) + b) >> 8) \tag{4.4}$$

$$Cb = 128 + (((r << 7) - (r << 4) - ((g << 6) + (g << 5) - (g << 1)) - ((b << 4) + (b << 1))) >> 8) \tag{4.5}$$

$$Cr = 128 + ((-((r << 5) + (r << 2) + (r << 1)) - ((g << 6) + (g << 3) + (g << 1)) + ((b << 7) - (b << 4))) >> 8) \tag{4.6}$$

These equations make use of shifts to accomplish the same multiplication as **Equations 4.1**, **4.2**, and **4.3**, respectively. The floating point implementation is highly readable, but a shift implementation creates an equivalent circuit to the floating point multiplication using only integers. This design uses fewer resources on the FPGA since it does not require additional Digital Signal Processing Units (DSPs) or other resources for complex multiplication.

After the IP generates the output of these equations, the YCbCr data can be output to a matrix of `struct YCbCr` with components `Y`, `Cb`, `Cr`, and a blank integer; similar to the RGB struct used to read the initial data. This can be used to verify results or to pass the results of `Y` to the DCT step and pass `Cb` and `Cr` to a function to downsample the results.

4.3 Chroma Downsampling

After completing the color conversion, the next step is to downsample the chroma channels, `Cb` and `Cr`, to a fourth of the original size. This type of downsampling is 4:2:0 as it reduces the size of the horizontal and vertical components to half by averaging groups of four pixels to a single value using the process seen in **Figure 4.5**. This required a data access pattern that was not easily usable for an AXI-Stream. The access pattern led to the creation of the DRAM implementation discussed in **Section 4.2.1.2**.

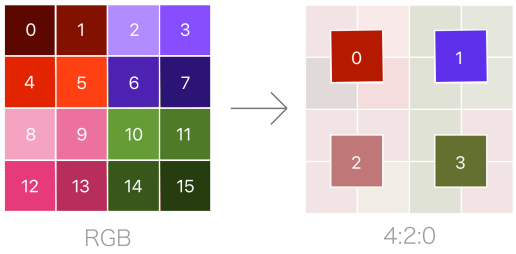

Figure 4.5: 4:2:0 Downsampling

In **Figure 4.5**, pixels at index 0, 1, 4, and 5 are averaged for the downsampled pixel at index 0. The ability to access indices 4 and 5 required an approach other than AXI-Stream. This led to an investigation of various approaches. The approaches included a line buffer and a DRAM implementation as potential options. Ultimately, the DRAM approach allowed access to pixels that were not in sequential order and the ability to access entire blocks out of order from a larger matrix. At this stage, keeping in mind that performing the DCT step requires an 8×8 matrix is necessary. For the chroma channels to have sufficient data for the DCT, a block of 16×16 pixels is needed to downsample since the result is an 8×8 block. Looping through the input data in steps of 16×16 blocks requires the design to iterate over a width of 32 and a height of 32 to completely scan the image (see **Figure 4** for more details). However, the design only downsamples the chroma channels leaving the luminance channel as a 16×16 block. This size difference forces the implementation to treat the luminance channel as four 8×8 blocks in subsequent JPEG steps.

4.4 Discrete Cosine Transform

The Discrete Cosine Transform requires two multiplications to take place [8]. The default 8×8 DCT matrix, as seen in **Equation 4.7**, is multiplied by an 8×8 block of data representing a single channel in a given color space. This result is then multiplied by the transpose of the default 8×8 DCT matrix.

$$DCT = \begin{bmatrix} 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 & 0.3536 \\ 0.4904 & 0.4157 & 0.2778 & 0.0975 & -0.0975 & -0.2778 & -0.4157 & -0.4904 \\ 0.4619 & 0.1913 & -0.1913 & -0.4619 & -0.4619 & -0.1913 & 0.1913 & 0.4619 \\ 0.4157 & -0.0975 & -0.4904 & -0.2778 & 0.2778 & 0.4904 & 0.0975 & -0.4157 \\ 0.3536 & -0.3536 & -0.3536 & 0.3536 & 0.3536 & -0.3536 & -0.3536 & 0.3536 \\ 0.2778 & -0.4904 & 0.0975 & 0.4157 & -0.4157 & -0.0975 & 0.4904 & -0.2778 \\ 0.1913 & -0.4619 & 0.4619 & -0.1913 & -0.1913 & 0.4619 & -0.4619 & 0.1913 \\ 0.0975 & -0.2778 & 0.4157 & -0.4904 & 0.4904 & -0.4157 & 0.2778 & -0.0975 \end{bmatrix} \tag{4.7}$$

This operation manipulates the data so that larger numbers will be in the top left of the matrix. In **Figure 4.6**, a visual representation of the DCT can be seen. The top left in this figure is a weight and will represent the average color of the 8×8 block. Other elements and details will also have a weight to add detail, with the most high-fidelity detail in the bottom right corner. This visual can help understand exactly what values in a matrix represent and which data is being discarded. Also related is quantization matrices, which will be discussed in **Section 4.5** along with how this visualization impacts their creation.

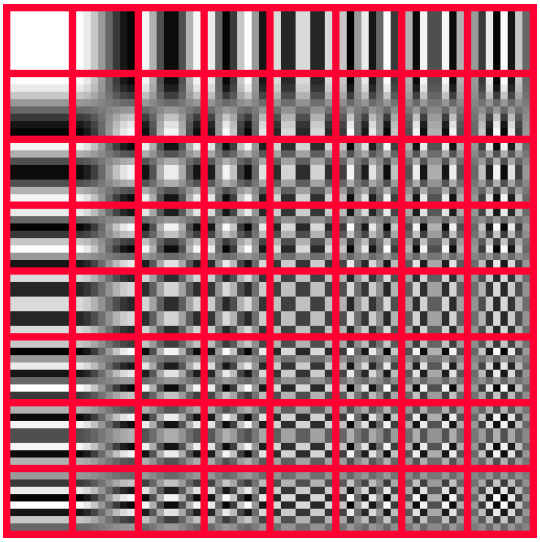

Figure 4.6: 8×8 Discrete Cosine Transform Visualization

In order to implement the design, the most straightforward approach is to break the DCT

step into two functions multiplication and multiplication_transpose. These two functions will accomplish the DCT as well as the quantization step which will be computed for each block of 8×8 at the end multiplication_transpose. Additionally, the FPGA implementation is not well suited for floating-point matrix multiplication so the standard DCT 8×8 matrix was multiplied by 2^{12} and converted to a signed integer. This scaled DCT integer matrix, seen in **Equation 4.8**, can be stored as a signed 16-bit 2-D array.

$$DCT * 2^{12} = \begin{bmatrix} 1448 & 1448 & 1448 & 1448 & 1448 & 1448 & 1448 & 1448 \\ 2009 & 1703 & 1138 & 400 & -400 & -1138 & -1703 & -2009 \\ 1892 & 784 & -784 & -1892 & -1892 & -784 & 784 & 1892 \\ 1703 & -400 & -2009 & -1138 & 1138 & 2009 & 400 & -1703 \\ 1448 & -1448 & -1448 & 1448 & 1448 & -1448 & -1448 & 1448 \\ 1138 & -2009 & 400 & 1703 & -1703 & -400 & 2009 & -1138 \\ 784 & -1892 & 1892 & -784 & -784 & 1892 & -1892 & 784 \\ 400 & -1138 & 1703 & -2009 & 2009 & -1703 & 1138 & -400 \end{bmatrix} \tag{4.8}$$

Due to the matrix being scaled, after each multiplication, the result is shifted down 12 providing the correct value. The ability to shift down by 12 was determined when selecting a factor to scale the DCT matrix. The scalar is the ideal method to implement the 2^{12} can be undone by shifting (each shift divides by 2) 12 times. A shift is less complex than any other form of division making scalers of the form 2^x the only reasonable selections. For more precision, the shift could be done at the end, but a larger register would be needed to store the accumulated sum.

4.5 Quantization

The quantization of the matrix is performed once the matrix data has been manipulated toward the top left via the DCT. This eliminates any smaller numbers and is where the majority of the loss during the compression occurs. This step generates the most loss since it makes many smaller values become zero which are then never recreated when decoding.

Worth discussing is the fact that luminance quantization matrices are asymmetric due to human eyes being able to distinguish vertical information better than horizontal[cite]. In **Equation**

4.9, L_{50} and C_{50} are the default quantization tables for a JPEG quality. In these tables, the values to the bottom left of the matrix are larger than those of the top right, accomplishing this bias towards vertical data using **Figure 4.6** as a guide to determine which location represents vertical vs horizontal data. Other tables can be generated using the **Figure 4.7**, but will retain the asymmetric nature on the luminance channel.

$$L_{50} = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \text{ and } C_{50} = \begin{bmatrix} 17 & 18 & 24 & 47 & 99 & 99 & 99 & 99 \\ 18 & 21 & 26 & 66 & 99 & 99 & 99 & 99 \\ 24 & 26 & 56 & 99 & 99 & 99 & 99 & 99 \\ 47 & 66 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \\ 99 & 99 & 99 & 99 & 99 & 99 & 99 & 99 \end{bmatrix} \tag{4.9}$$

```
if (quality < 50)
    S = 5000/quality;
else
    S = 200 - 2*quality;
end
luminance_QUANTIZATION_TABLE = floor((S*L_50 + 50) / 100);
chroma_QUANTIZATION_TABLE = floor((S*C_50 + 50) / 100);
```

Figure 4.7: Generate Quantization Matrix

The quantization matrix used must be stored or the quality saved so the data can be correctly reconstructed on another computer. If the incorrect matrix is used the result is an image that retains some structure, but visually parts are too light or dark (see Figure 8 in the appendix).

4.6 Zig Zag Matrix Transformation

Implementing the Zig Zag step from the JPEG Algorithm is fairly straightforward as the design is already operating on the appropriate block size after the DCT matrix has been calculated. The first step is to convert the 8×8 matrix to an array with 64 elements this was done by using `matrix_width`×`row`+`column` to give the new location in a 1-D array. This array is then iterated over from 0 to 63, but the index uses a predefined zig-zag array to appropriately index the array and read elements as in **Figure 4.8** [5].

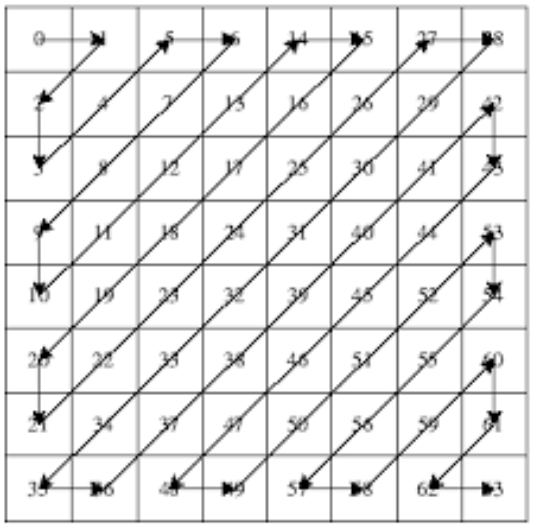

Figure 4.8: 2D 8×8 Matrix Zig Zag Transform to 1D Array [5]

This pattern is used for every 8×8 block to convert from a 2D matrix to a 1D array. This works well for the chroma channels, which only compute blocks of 8×8 at a time. However, for the luminance channel, the input is a 16×16 matrix, handling four 8×8 blocks simultaneously. The 16×16 matrix is not the expected 8×8 and must be handled accordingly. The implementation chosen follows the dataflow in **Figure 4.9**.

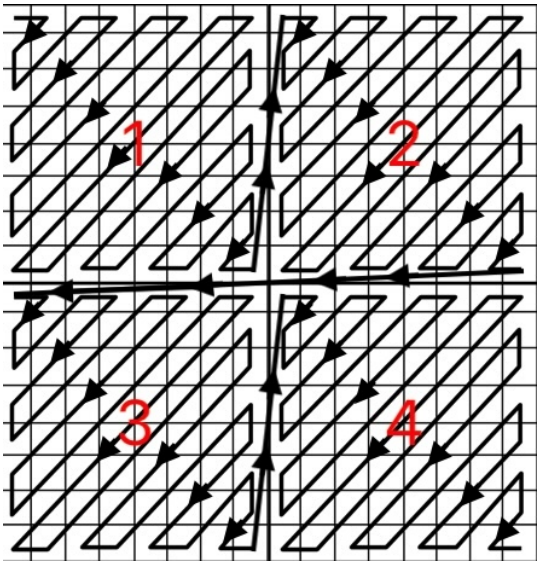

Figure 4.9: 2D 16×16 Matrix Zig Zag Transform to 1D Array

The design utilizes the standard zig-zag four times each zig-zag block is associated with a red number in **Figure 4.9**. These smaller 8×8 blocks are connected by connecting the last element in the block to the first element in the following block. The decision to connect the blocks in this fashion in the output means groups of 16×16 must be decoded and reconstructed appropriately. The luminance channel must be read differently than a chroma channel that can read a row of 8×8 blocks and then move down a row to read the next 8×8 block of data without issue. A row is referring to one of the 32 rows of 8×8 blocks in a 256×256 matrix (size of a chroma channel). If the same approach was taken on the luminance channel (512×512) when using the method in **Figure 4.9**, blocks 3 and 4 would be read as 65 and 66 if starting with an index of 1. This issue would affect the entire image when decoding the data, resulting in the image data being decoded incorrectly.

4.7 Run Length Encoding

Run Length Encoding is loosely defined, with implementations behaving differently depending on the required information to recreate the data. The basic idea is to compress data by noting how many times the data is repeated. The goal of RLE is to remove redundancy in the data. The most common technique creates one array for the values that are in the original array and removes all repeated values. A second array is used to store how many times the value in the first array at

the same index needs to be repeated. Lastly, if the implementation requires the ability to recreate subsets of the original data, a third array can be used that indicates where the value at that index should begin in the recreated array, this way we can recreate smaller portions and do not need to start at the beginning to accomplish the task. However, this RLE technique will stray from traditional implementations using only one array. For example, if the data has a repeated value instead of writing the value repeatedly, which uses an 8-bit elements for each repeated value, the RLE implementation uses two elements one for the value and how many times to print the value. An example array can be seem in **Equation 4.10**.

$$\begin{bmatrix} 8 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} [8 & 1] & [0 & 2] & [2 & 1] & [1 & 1] & [0 & 5] \end{bmatrix} \tag{4.10}$$

In the example, each pair of elements [x y] can reproduce a "run" of repeated numbers. In this technique, when a number is repeated more than twice, the data size is actually reduced. The first implementation of RLE uses a single array and alternates between the value needing to be printed and the number of times to print the value. This allows me to use one less array than the traditional approach. The data typically has over 50 zeros when the JPEG quality is 50, but the amount of zeros varies with quality and input. This initial implementation had a massive bottleneck of creating two data points where previously the data needed one. This scenario is not ideal for what the data from DCT is generating since only zeros are primarily repeated, and other numbers that repeat typically don't sequentially. This is expanding the data which is not ideal for a compressor. Instead, a new method of encoding was devised, such that only zeros have how many times they should be printed following the value. This new design can be seen in **Example 4.11**, which will always be the same or better compression compared to the initial design.

$$\begin{bmatrix} 8 & 0 & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \begin{bmatrix} 8 & [0 & 2] & 2 & 1 & [0 & 5] \end{bmatrix} \tag{4.11}$$

The benefit of this approach depends on how many non-zero, non-repeating values are in the data. The more data like this, the better this design is over the previous. For example, this new technique provided roughly 1.8x more compression at a JPEG quality of 50.

One consideration when implementing this design is how many arrays we need to store the information. We have three channels, so we will only need three total arrays as our example 4.11 is able to store the full data for a channel in a single array.

Furthermore, considering how large each array needs to be to hold the entire encoded data set in the worst case. Based on the initial RLE Implementation the worst case is no repeated data, this would expand the initial data to two times the initial size. This means if we have a JPEG quality 100 even with the new RLE implementation the data could be doubled. This is definitely a result that doesn't make for a good compressor so instead, using arrays the same size as the initial data is done this guarantees never expanding the data. However, the data may only be half there so instead of encoding (and expanding) the data, the initial data is returned and a flag is thrown to indicate to the decompressor that Run Length Decoding does not need to be performed. This guarantees never expanding the data and it will fit on the buffer.

Another consideration was how to implement this design for all of the channels while keeping in mind the board's resources. As in this phase of development, the utilization of the PYNQ was very high making designs difficult to synthesize. This led to two viable options moving forward.

4.7.1 Block RAM

In this version, the zig-zag output with 64 elements is stored in a large array of either 256*256 for the chroma channels or 512*512 for luminance. I selected this size since it is the original length of the data, meaning if the generated output is larger than this size, I should not perform RLE. These large arrays are written in groups of 64 for the chroma channels and groups of 256 for the luminance channel. These smaller arrays will fill the large arrays prior to the RLE being performed. Due to these write sizes, the BRAM approach for the luminance channel can create element values as high as 256 if zero is repeated this many times. The data is stored as signed 8-bit integers and 256 is out of the viable range forcing the data to be represented as signed 16-bit integers. This will double the size of the encoded data even if the data has fewer elements it will be half of the anticipated compression ratio. In order to combat this the design was altered so that after a chunk of 64 is read then the FPGA will write regardless of the next value in the data to ensure my count is not greater than 64. While I could have pushed the value to the upper limit of 8-bit the benefits of doing so did not outweigh the complexity of an implementation. Simply resetting after each 8×8 block is read is a much cleaner solution. It is worth noting that if the image data was 16-bit or higher the approach that does not reset after each block would be marginally better than an 8-bit

approach due to the ability to store the values like **Equation 4.12**.

$$\left[\begin{bmatrix}0 & 64\end{bmatrix} \quad \begin{bmatrix}0 & 64\end{bmatrix} \quad \begin{bmatrix}0 & 64\end{bmatrix} \quad \begin{bmatrix}0 & 64\end{bmatrix}\right] \Longrightarrow \left[\begin{bmatrix}0 & 256\end{bmatrix}\right] \tag{4.12}$$

4.7.2 Global RLE Index Variable Implementation

Instead of putting the output of zig-zag into a large memory, I begin to directly perform RLE after the zig-zag, requiring changes to how the data is written. In the BRAM version, a complete copy of the data in zig zag arrays is iterated through. The program needed to be altered to perform RLE after each zig-zag array was generated. This creates the problem of not knowing where a new portion of the RLE data needs to be written. By using a variable that is accessible to each block of 64 elements, the program can track the appropriate output index needed when writing. This design choice to use a "global" variable was made based on the best HLS approach given the limited board space. In this implementation, the data is still written sequentially since the data is pipelined and the "global" variable does not cause any issues being used in an incorrect order.

4.7.3 Implications of RLE Implementations

When implementing the design with a BRAM approach, the board can very easily use too much BRAM. The PYNQ-Z2 has 140 BRAM and HLS reports each chroma channel uses 60 leaving little room for other operations to use BRAM. This resource utilization is extremely high, thus the "global" variable implementation was created to decrease the demand for BRAM. This change did not come without side effects as my slack time increased with the change due to the variable becoming dependent in all RLE operations and needing to wait for the information to arrive compared to the BRAM which is not dependent on any external variable.

4.8 Vivado Implementation

After the HLS IP, named `IMG2RLE` was created, a block diagram in Vivado was needed to properly interface between the ZYNQ PS and the IP. The block diagram seen in **Figure 4.10** was implemented to accomplish this communication via AXI4 using the provided Vivado interconnect.

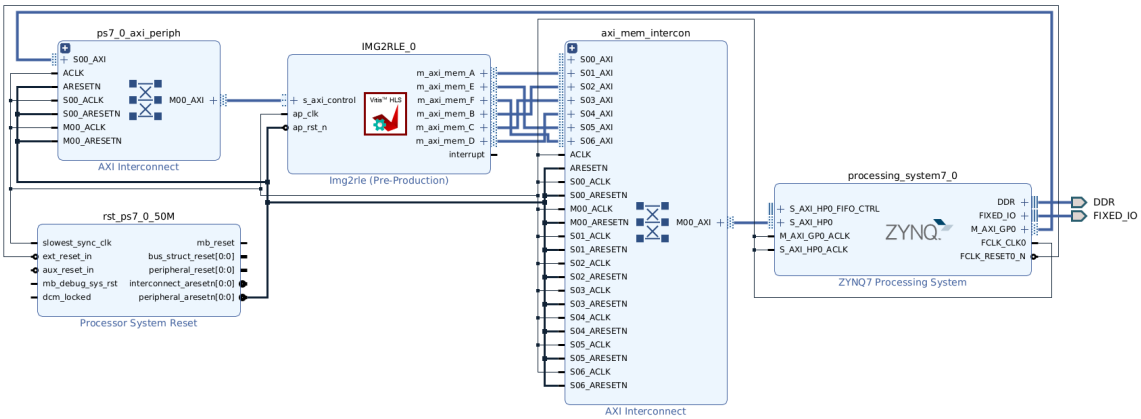

Figure 4.10: Image to RLE Block Diagram

From the block diagram, a bitstream was generated that allowed the input of a 512×512 8-bit image along with a luminance and chroma quantization table each of which is an 8×8 unsigned 8-bit matrix. The output is three 1D arrays of length 512^2 for the luminance and 256^2 for the chroma channels. These lengths ensure the data will not be expanded as if the program exceeds these lengths it is not an improvement over the downsampled data. Nevertheless, this output is variable length, and as such all elements after the last none zero can be discarded.

4.8.1 Resource Utilization

In order to understand how efficient this implementation is on the edge device, the resource utilization must be examined. For this design to be synthesized successfully, many resources on the board were used. This implementation has an extremely high resource utilization as seen in **Figure 4.11**, the light blue areas indicate a resource in use and the black portions mixed in the blue area (not including the two large black areas) are the unused areas on the PL fabric.

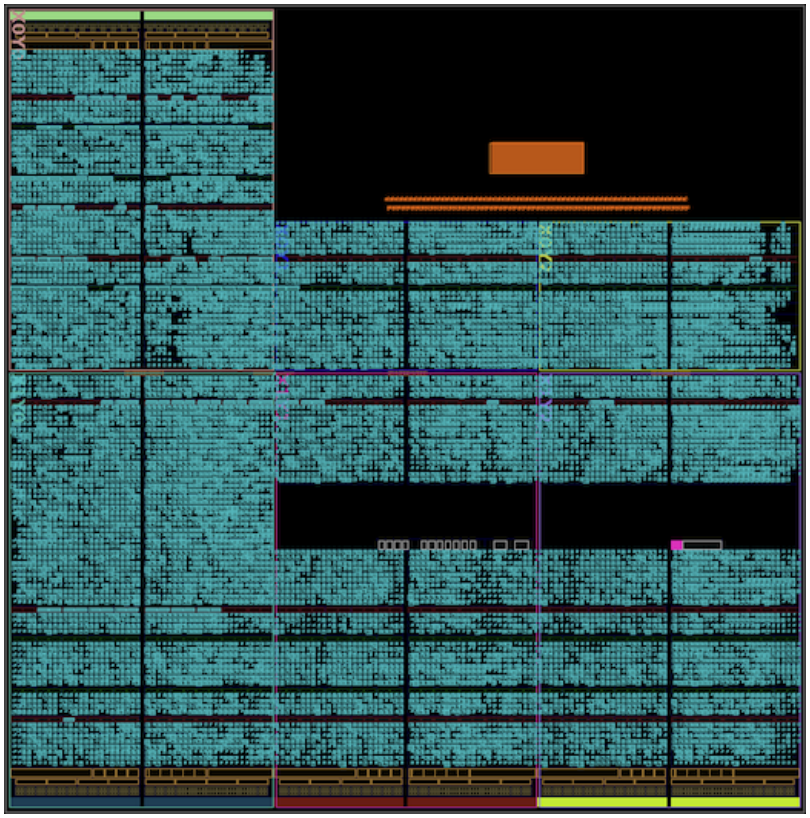

Figure 4.11: FPGA Resource Utilization

The figure is primarily blue since the majority of the fabric is used for the implementation. While this visual provides a quick overview of the usage, a breakdown with specific utilization of individual components is provided in **Table 4.2**.

Resource	***Utilization***	***Available***	***Utilization(%)***
LUT	37772	53200	71.00%
LUTRAM	1710	17400	9.83%
FF	48718	106400	45.79%
BRAM	32.5	140	23.21%
DSP	48	220	21.82%
BUFG	1	32	3.13%

Table 4.2: Resource Utilization

While the specific resources are not over-utilized, the Slice Logic Distribution requires 99% of the available slices. In addition to being resource-hungry, the design has the implication of utilizing 1.702 watts of power as seen in **Figure 4.12**. This utilization and power demands while functional, do not provide room for additional designs to run in tandem. If a larger design needed implementation on a single device a large more powerful board than the PYNQ-Z2 could be utilized.

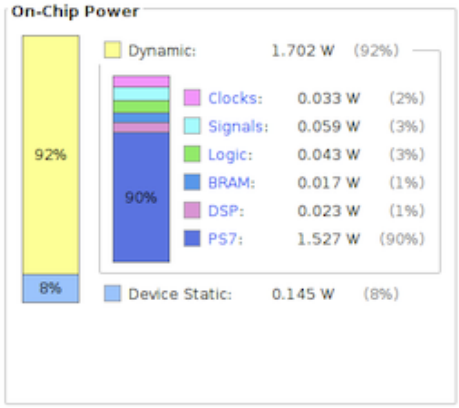

Figure 4.12: FPGA Power Usage

4.9 Python Implementation

Due to the PYNQ-Z2 board targeting Python productivity, the language was selected as the language to develop any additional functionality beyond the PL side of the PYNQ. This includes code to interface the PL side which will be discussed in section 4.9.1 on encoding. In section 4.9.2 on decoding the Python decoding implementation is discussed primarily focusing on the code running on the Mac Studio previously mentioned.

4.9.1 Encoding

In order to input an existing image to the FPGA implementation, on the PS side of the PYNQ, it must be formatted appropriately. There is a distinction between RGB images and gray so for the purposes of this implementation, all images are converted to RGB. This does not affect how well the gray image is compressed but provides zeros to the other channels ensuring normal functionality. Once the image is in RGB it must be scaled to 512×512 for the current implementation using the PIL library, discussed in section 4.1.1. After reshaping the image, the three 8-bit values

are combined into a single 32-bit unsigned integer. This is done for 512×512 pixels, then the data is put in DRAM for the FPGA to read. The PS must send the start signal to indicate the `IMG2RLE` IP should begin. Once the IP calculations are finished a result is written to another output variable in DRAM that the program will read and truncate any zeros after the last non-zero value. Finally, the three arrays for each channel are saved to a numpy file, `*.npy`, along with the quality compressed at and the image's original dimensions so it can be reconstructed during the decoding phase. It is worth noting that this does not create the same overhead as JPEG in its final form, but the expected order must be perfect or the data will not be able to reconstruct without manual intervention. Furthermore, if an ideal quality was found or a consistent image shape was used these data points could be removed with a minor increase in compression.

4.9.2 Decoding

In order to decode the numpy files, an additional Jupyter notebook was made to read the encoded data. The FPGA is connected over an ethernet connection to the Mac Studio so the program needed to be able to read data over an SMB connection. The file location is arbitrary, but rather than manually transfer and decode, the ability to scan a file structure for files is used. In order to accomplish this scan, all of the files in sub-folders for my image set, identical to the structure previously described, are scanned searching for files in the form `*.npy`. Each file is then decoded using the information in each array of the file. For example, the program will find the quality used to encode and then perform de-quantization by multiplying by the original quantization matrix used. These were stored on both computers so they did not need to be included only the value representing the table. Next, an inverse DCT is performed, and then 128 is added to the data to undo the normalization about 0 when encoding. The chroma channels are scaled up from 256×256 to 512×512 and then the channels are combined at each matrix element to make a pixel. These pixel values are in the YCbCr color space and must be converted back to RGB. This image can then be saved to the local computer completing the data transfer and decoding using this program. These decoded images can then be used as the input of a CNN.

Chapter 5

Results

5.1 CNN using JPEG Data

My results have been broken into multiple sections to highlight the performance of each technique used. For my test data, I created a randomized subset of the ImageNet Large Scale Visual Recognition Challenge database which contains over 1 million images divided into 1,000 categories. For consistency, all model inference runs were conducted on the Palmetto Cluster using two NVIDIA Tesla K40 GPUs.

5.1.1 Compression Metric Relationships

Figure 5.1 depicts the relationship between the JPEG quality and the achieved compression ratio across my test set. This graph shows a fairly positive linear relationship through JPEG 20, achieving a compression ratio of approximately 8. Additionally, a more exponential relationship that rapidly degrades the image when qualities are lower than 20 can be seen. Due to the exponential behavior, any fidelity for affecting compression ratio via JPEG quality is low when below 20. Small changes in JPEG quality have drastic effects on the compression ratio. The range from 100 to 20 provides the best controlled range for the compression.

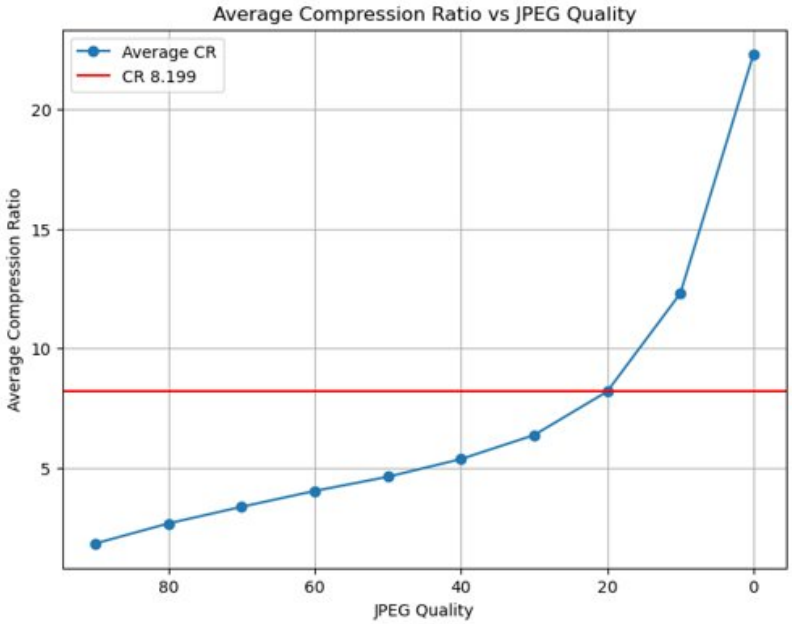

Figure 5.1: JPEG Quality vs. Compression Ratio

Figure 5.2 shows the relationship between JPEG quality and SSIM which follows a nearly identical, though inverse, trend to the one with compression ratio. From this graph, it can be seen that there is only a small change of approximately 15% in SSIM from JPEG 100 to JPEG 20 before a sharp decline begins thereafter. Combined with the results of the compression ratio testing above, this data demonstrates a relatively high compression ratios, upwards of eight times, can achieved while still maintaining a relatively high SSIM.

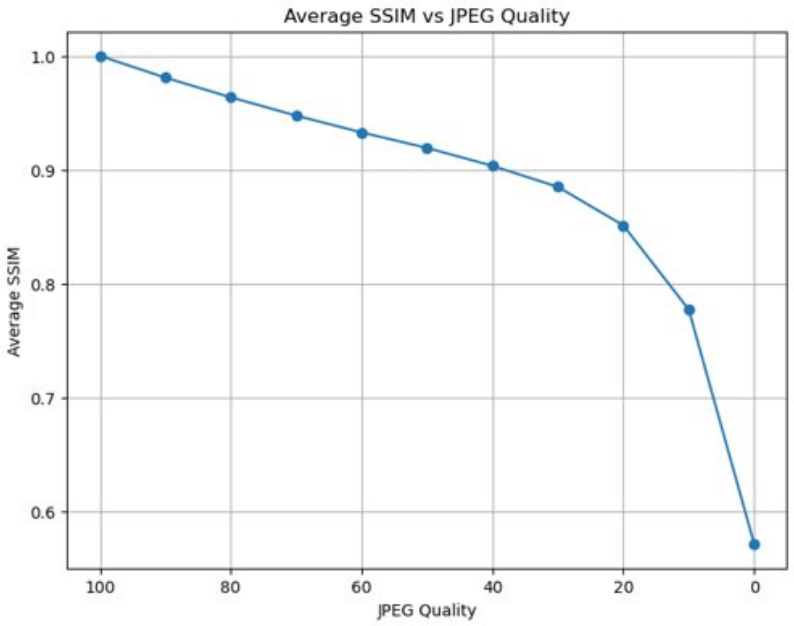

Figure 5.2: JPEG Quality vs. SSIM

5.1.2 Evaluation of Compression Ratio based images on Model Performance

In this section, an evaluation of the performance of each model on uncompressed images against images from various levels of compression ratio, ranging from two to 20 is performed. The results of these evaluations are displayed in **Figure 5.3**. All four models show only a small initial drop in accuracy at CR 2 followed by a steeper, near linear, reduction in accuracy through CR20. MobileNetV2 appears to be less resilient to this compression, experiencing a steeper accuracy reduction rate, rapidly falling below the performance of ResNet50 at CR5 and subsequently VGG19 at CR12. **Figure 5.4** shows the response of model confidence in its top prediction which generally follows the same trend as overall accuracy. Of note, even though ResNet50 maintains a high confidence, on par with the top performing model InceptionV3, its overall accuracy remains about 10% lower. This implies that even though it is confident in its predictions, these predictions tend to be less accurate overall.

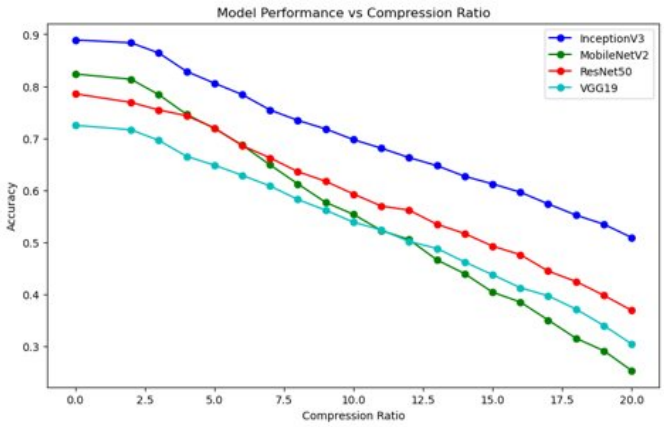

Figure 5.3: Model Performance vs Compression Ratio: Model Accuracy

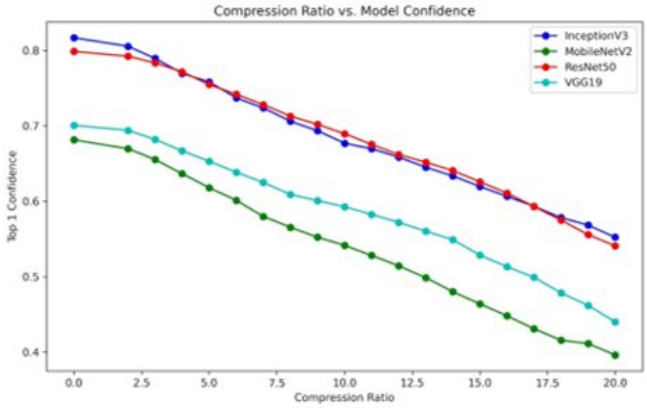

Figure 5.4: Model Performance vs Compression Ratio: Top 1 Confidence

5.1.3 Evaluation of JPEG Quality based images on Model Performance

Given that this compression algorithm utilizes the JPEG quality as a means by which to affect the compression ratio, I next chose to explore how compressing to discrete JPEG quality values (as opposed to compression ratios) would affect model accuracy. Based on the experimental results in **Section 5.1.1**, the accuracy was expected to follow a similar trend to SSIM, with a gentle reduction through JPEG 20 followed by a steep decline thereafter. **Table 5.1** shows the results of model accuracy on an uncompressed image down to JPEG quality 0 in steps of 10. The data has a sharp drop in accuracy after JPEG 20 as anticipated, with the results above a quality of 20 showing significantly higher performance. Even though the SSIM experiences a 15% reduction through JPEG 20, model accuracy only experiences a reduction of 6.7% over the same interval.

JPEG Quality	***Average SSIM***	***Average CR***	***Average Accuracy***	***Percent Reduction in Accuracy***
100	N/A	N/A	0.805	N/A
90	0.980	1.83	0.803	0.33
80	0.963	2.67	0.796	1.13
70	0.947	3.37	0.794	1.40
60	0.932	4.03	0.789	1.97
50	0.919	4.62	0.787	2.26
40	0.903	5.37	0.787	2.28
30	0.884	6.37	0.775	3.71
20	0.851	8.20	0.75	6.70
10	0.777	12.30	0.661	17.96
0	0.571	22.32	0.166	79.29

Table 5.1: Average Model Performance vs JPEG Quality

This relationship is plotted in **Figure 5.5** below. It can be seen that model accuracy is able to remain nearly unaffected through JPEG 20 before it rapidly declines.

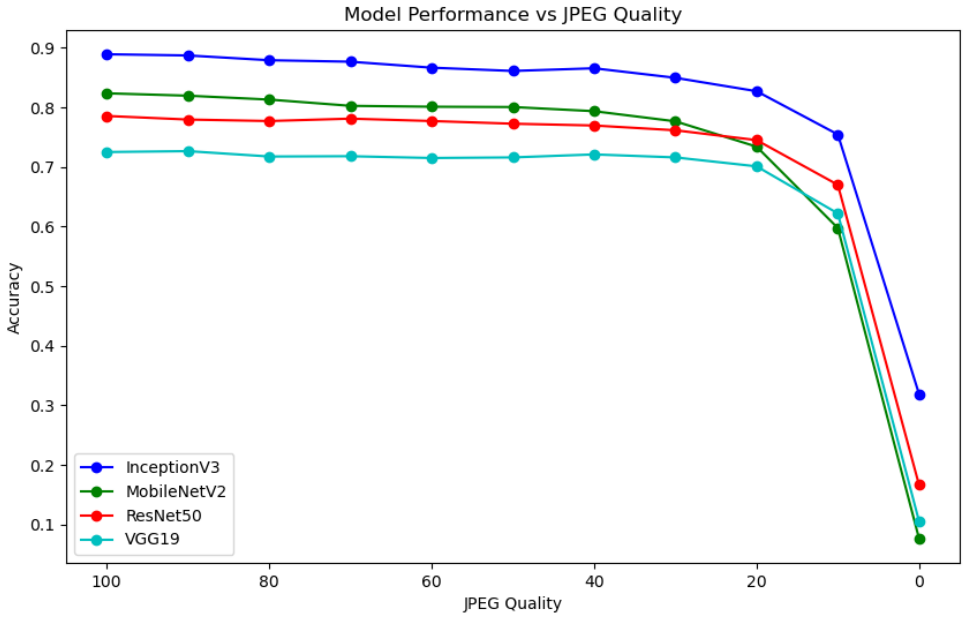

Figure 5.5: JPEG Quality vs Model Performance

When compared back to the results achieved in **Section 5.1.2** utilizing compression ratio as a target, a notable improvement in model accuracy is accomplished. In **Table 5.2**, a comparison of three different runs in which a comparable overall compression ratio is achieved by the two methods in the table. The results show JPEG quality achieves an overall higher model performance. JPEG quality 20 and 30 achieve not only a higher compression ratio but also an 11.1% higher average accuracy across all four models.

Target Metric	*Achieved CR*	*Achieved Accuracy*	*Accuracy Difference*
CR 4	4.04	0.760	0.044
JPEG 60	4.03	0.790	
CR 6	5.94	0.697	0.079
JPEG 30	6.38	0.776	
CR 8	7.77	0.641	0.111
JPEG 20	8.20	0.752	

Table 5.2: JPEG Quality vs. Compression Ratio Performance

5.1.4 Model Runtime and Compression Overhead

The time to compress an image is critical in my project since the implementation must maintain or improve runtime to be effective. This requirement left room for time savings in two areas, the model runtime and transfer times of the images. An implementation has a higher degree of control over the time savings due to transfer speed since a JPEG quality can be selected that should on average correlate to a compression ratio. Further, the specific device the images are compressed on and the rate at which they can be transferred can have a significant impact on the time saved. In the examination of my models' runtime in **Figure 5.6** for compression at various JPEG qualities, there is a reduction in each model's runtime in nearly every case except when JPEG quality is 0 where InceptionV3 and ResNet50 take longer to run the compressed data.

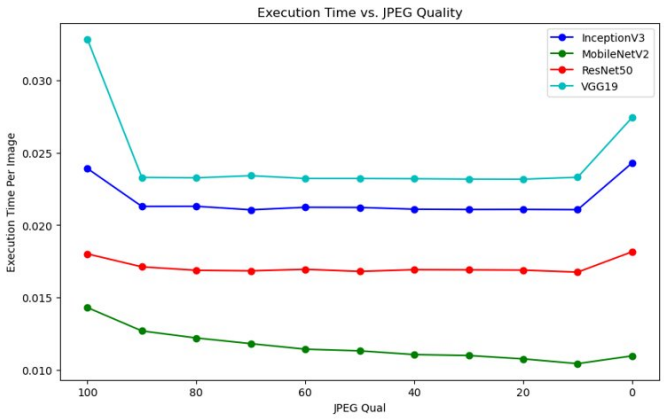

Figure 5.6: JPEG Quality vs Model Runtime

However, this time saving alone is not enough to overcome the compression time. As shown in **Table 5.3** below, the total overhead for a compressed execution of the model for a single image is on average 0.0064847 seconds, and the maximum execution overhead is 0.008275 seconds.

JPEG Quality	***Inception V3***	***MobileNet V2***	***ResNet50***	***VGG19***	***Compression Time*** (ms)	***Total Overhead***
90	2.614	1.609	0.891	9.525	11.9360	8.275
80	2.603	2.094	1.129	9.554	11.2709	7.425
70	2.847	2.486	1.167	9.406	10.8845	6.907
60	2.672	2.867	1.062	9.599	10.5836	6.533
50	2.684	2.982	1.206	9.602	10.3864	6.267
40	2.804	3.240	1.086	9.617	10.1745	5.987
30	2.826	3.303	1.098	9.642	9.9828	5.765
20	2.820	3.536	1.116	9.650	9.7182	5.437
10	2.837	3.863	1.255	9.517	9.4052	5.036
0	-0.380	3.328	-0.140	5.396	9.2637	7.215

Table 5.3: Runtime reduction by Model with Compression Time (ms)

Due to this increase in the execution time during this phase the design must compensate in other areas to make up the time. With the reduction in file size this deficit can be overcome

as the reduction in the transfer time of an image due to compression makes up for the overhead. If the images are needed after the model is run for post-hoc analysis of the image or stored for another reason, the benefit of using a compressed image increases with each needed transfer. While needing to store the image is a massive benefit, some edge devices are limited in their transmission capabilities. In **Figure 5.7** a variety of baud rates and bits per baud are compared to show the time savings per image when transferred at a given bit rate. For two-way communication, the maximum baud rate is 1,200, regardless of the bits per baud used to modulate the data the worst case saves more than 1.75 seconds per transfer potentially saving up to 14 seconds in a single transfer. Devices using lower baud rates are typically less complex systems and make use of Amplitude-Shift Keying (ASK) which is limited to 1 bit per baud. Phase-Shift Keying (PSK) can also be used to provide better performance allowing for more bits per baud while adding minimal complexity over ASK. As the bits per baud increase, the complexity in modulation is more complex, requiring Quadrature Amplitude Modulation (QAM) to achieve higher bits per baud. Wireless networks typically use QAM, which is much more complex than ASK or PSK, but QAM allows as high as 8 bits per baud.

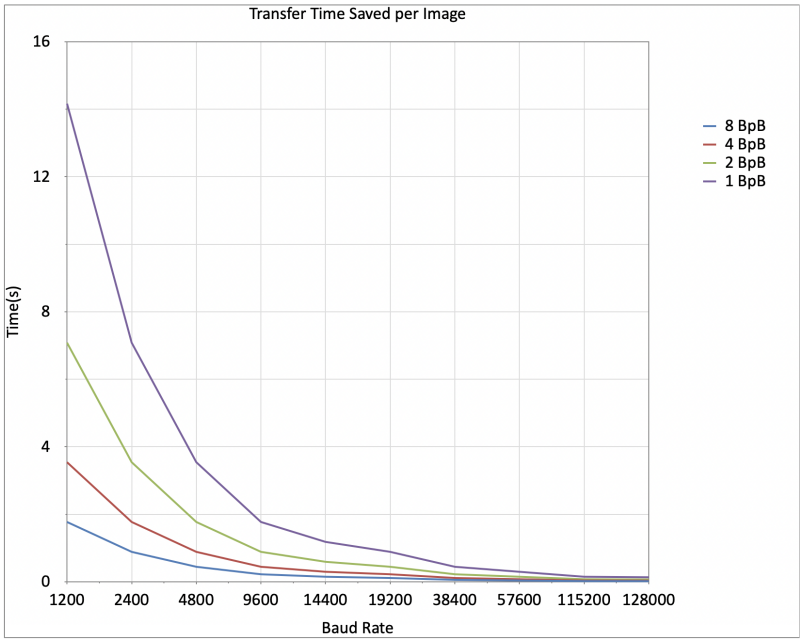

Figure 5.7: Transfer Time per Image across various Baud rates

To determine how much time would be saved ten thousand images were sampled. The average uncompressed image size was calculated to be on average 8,272.033 bits. While compressed

images required only 239.297 bits to store on average. For a baud rate of 115,200 with 8 bits per baud a corresponding bit rate of 921,600 bps is the operating speed. This is a high bps taking advantage of a high baud and maximizing bits per baud. A device like the Raspberry Pi or Pynq-Z2 operates at 921,600 bps over their serial communication [6]. This provides us 0.0087 seconds of saved time for every image transfer, which is more savings than my highest overhead.

5.2 FPGA Image to RLE implementation

As discussed the implementation is able to convert a 512×512 image these results will be focusing on the compressed RLE data and verifying these results will be the focus of this portion of my research. After implementation, using similar techniques to test the original JPEG implementation can be applied on these images. A visual representation of the compressions loss on an image at JPEG quality 50 can be seen in **Figure 5.8**. The figure indicates visually that the compressed image is still representative of the initial data, but also demonstrates that loss has occurred.

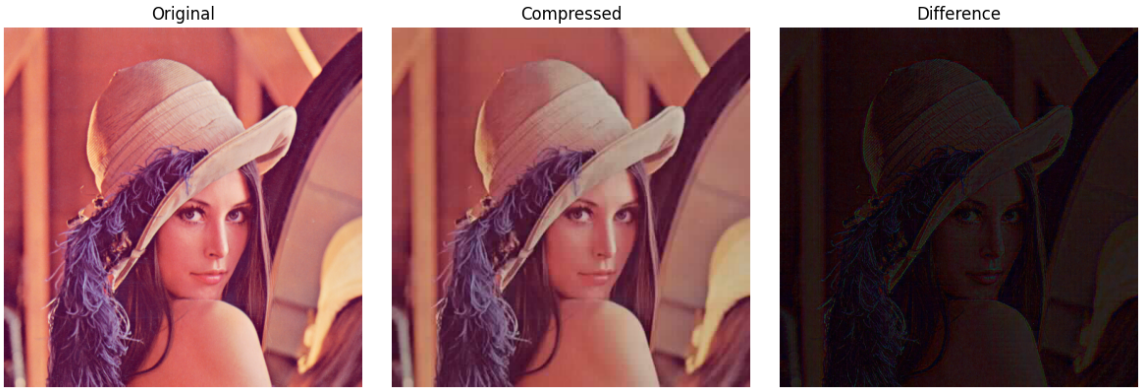

Figure 5.8: Difference between Original and Compressed Image

After concluding the `IMG2RLE` compression was functional, determining if this process of compression on the FPGA excessively degrades the images compared to the previous implementation and affects the results of the CNN will be observed.

5.2.1 Image Compression on FPGA Overview

These images are compressed in the same fashion as JPEG, but have more rounding error, for this reason the image will be observed using this knowledge. The SSIM structure can reveal

how compromised the structure of the image has become during compression. In **Figure 5.9**, the structural difference can be seen between the original and compressed image with white areas indicating more structural loss.

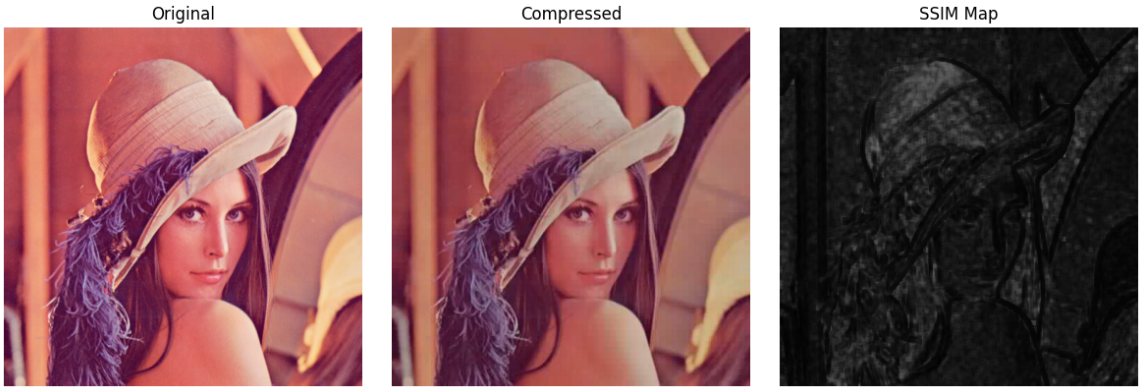

Figure 5.9: SSIM Map comparing Original and Compressed Image

The SSIM map has black "lines" that are similar to edge detection showing that the overall structure is maintained while information inside of these edges tends to be lost. This was also observed in JPEG images compressed using the original implementation when trying to determine which qualities affected accuracy the most. The image does not indicate that there is any observable difference in the implementations.

The operation was verified over a range of JPEG qualities with the range 25 to 75 being the primary interest. This range was selected for maximum compression while maintaining accuracy. In **Figure 5.10**, the trend of a higher CR as quality degrades is depicted. However, the large range of compression ratios for a single JPEG quality is worth noting, but individual images remain relative to the average so compression ratio always increases as quality increases for a single image.

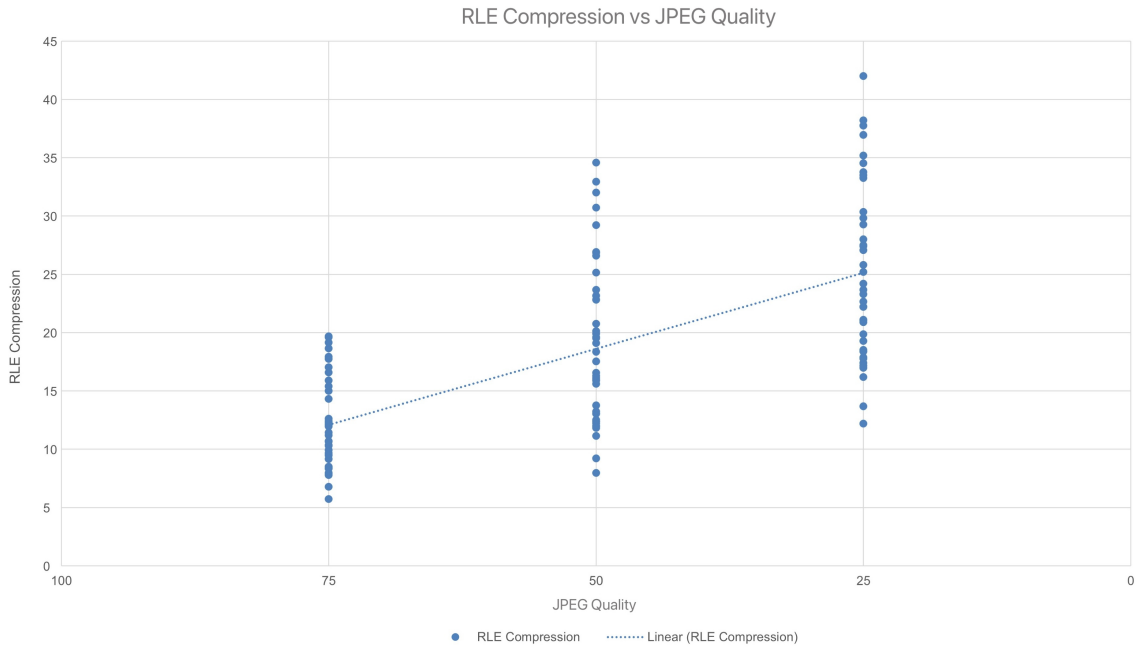

Figure 5.10: RLE Compression vs. JPEG Quality

The trend found in **Figure 5.10** provides us the ability to control how accuracy is affected. However, what values this trend occurs at needs to be verified. In **Table 5.4**, the compression metrics for the FPGA implementation are reported at JPEG qualities every 25, with 100 being the original image. Additionally, the drop in accuracy compared to the JPEG Accuracy at the same quality is reported. While the accuracy in the FPGA implementation is reported to be lower than the original implementation, this does not negate the viability of the approach. However, the accuracy required for a given application will need to account for this discrepancy. This difference between the two can be overcome by increasing the quality until sufficient accuracy is reached.

JPEG Quality	***Average Compression Time*** *(ms)*	***Average CR***	***Average SSIM***	***Average Accuracy***	***% Reduction in Accuracy***	***FPGA Accuracy vs JPEG Accuracy***
100	N/A	1	1	0.805	N/A	0
75	0.25171	11.99	0.82369	0.773	4.02	-0.022
50	0.25469	18.82	0.79645	0.750	2.92	-0.037
25	0.25515	25.02	0.76232	0.714	4.84	-0.049
0	0.25544	69.21	0.37771	0.0572	91.99	-0.406

Table 5.4: Average Model Performance vs JPEG Quality

5.2.2 CNN Accuracy

The CNN accuracy for the FPGA implementation compressed the ImageNet data using qualities in increments of 25. These compressed images were run through the CNN models used in the previous results for equal comparison. In **Figure 5.11**, the results of the CNN using Images compressed on an FPGA can be seen.

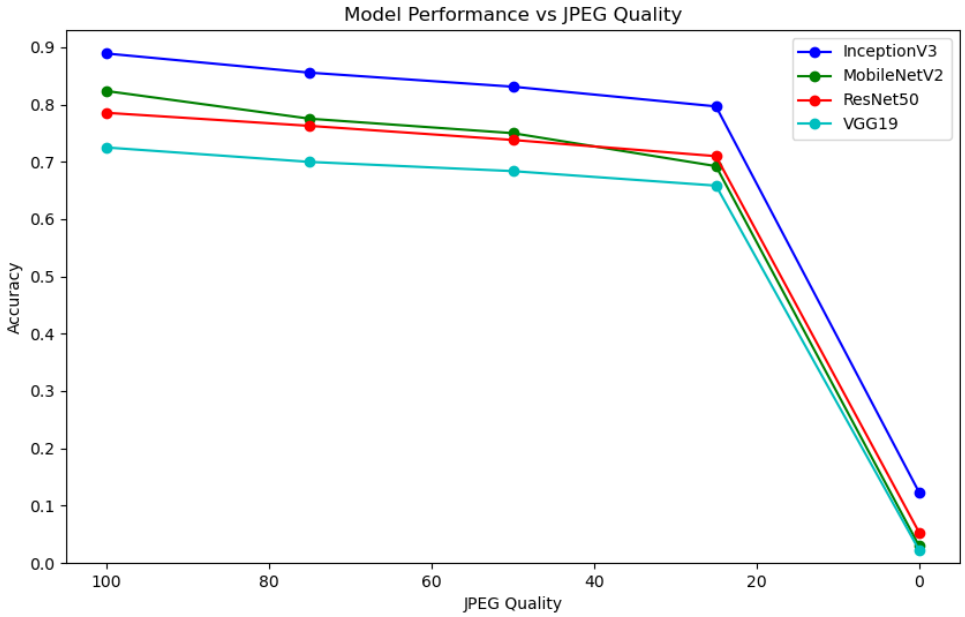

Figure 5.11: FPGA generated JPEG Image Quality vs Model Performance

These results closely resemble the accuracy reported in the python variation in **Figure 5.5** and maintain the same trend in accuracy decreasing as quality decreases. Furthermore, data before the drop off near 20 and in this case at 25 maintains a relatively high accuracy compared to uncompressed accuracy. The last result to verify is the reduction in model runtime, this analysis is in **Table 5.5**.

JPEG Quality	***Inception V3***	***MobileNet V2***	***ResNet50***	***VGG19***	***Compression Time*** (ms)	***Total Savings***
75	2.158	3.183	2.102	9.728	0.2517	4.106
50	1.962	2.854	1.863	9.497	0.2547	3.789
25	2.158	2.967	2.026	9.692	0.2551	3.956
0	1.053	4.464	1.598	10.450	0.2554	4.136

Table 5.5: Runtime reduction by Model with FPGA Compression Time (ms)

In this table, it is evident that the compression time is significantly faster than the JPEG

implementation from 5.3. Since the model runtimes are still reduced across all models and the compression time is less than the reduction, this implementation saves time and is able to send the data to another computer and get a CNN accuracy more quickly compared to the default implementation and the JPEG implementation. However, the total savings in the millisecond range does not account for the programs time to decode or transmit due to an inefficient python implementation used for development. For more holistic results a faster programming language to perform decoding should be used and timed to determine this cost. If it adds less than $12.064(ms)$ to the total time, the previous implications as when the overhead was $8.275(ms)$ would apply. The other option is to complete the JPEG compression by performing Huffman Encoding and writing to the file using standard techniques. For any longer times, **Figure 5.7** should be referenced to determine if the compression is worth while.

Chapter 6

Conclusions and Discussion

The growing need for new techniques to optimize CNN functionality across a wide range of applications remains an under-researched area that needs more attention. This work shows the viability of implementing the JPEG algorithm on an edge. The ability to maintain high accuracy while reducing compression time makes this solution to overcome limited transfer bandwidth from an edge device extremely viable.

The CNN models all demonstrated the ability to maintain accuracy as JPEG quality is decreased until roughly a quality of 20. This resilience to the compression JPEG compression should entice more research into CNNs using encoded data and lossy information. Further investigations into how to improve a CNNs ability to handle JPEG data should be explored. In the following section, a few recommended future work topics are discussed.

The ability to recreate the JPEG algorithm's core components on an FPGA with similar performance trends to the software implementation proves the device is a viable target for deployment. Collecting images on an FPGA for use on a CNN on an HPC or more suitable computer benefits from JPEG at the cost of a reduction in CNN accuracy. This is even more applicable for edge devices limited by transfer speed, JPEG compression is a viable option to alleviate systems resource demands.

6.1 Future Work and Recommendations for Future Research

6.1.1 Quantization

As discussed in **Section 4.5**, the default luminance matrix is asymmetric for human eyes. As the CNN is observing the image, not a human, an investigation into other default quantization matrices could yield marginal improvements in accuracy. This investigation would require running compression on the FPGA using custom quantization tables to determine the effect on the CNN. A similar approach could be studied solely using the initial Python software implementation for a proof of concept.

6.1.2 Run Length Encoding

Currently, any parallelization done here will fail as data will not be correctly ordered unless it is performed sequentially. If parallelization was needed, the likely solution would be using the third array discussed in **Section 4.7** to store position indices would be needed. Investigating a parallel implementation at this stage could be interesting and provide a speedup to the program.

6.1.3 Huffman Encoding and JPEG File Creation

More research investigating which device is the best to perform Huffman Encoding on and how beneficial the extra step is would provide helpful information about more potential time savings. As discussed the FPGA PL side will not handle this well so it is am limited to the PS and its performance. The Huffman encoding and subsequent steps may be executed faster on an HPC even with transfer overhead due to the vast performance difference in the devices. Due to the Huffman encoding not being performed the creation of a JPEG File would be needed using the appropriate markers to implement a complete JPEG algorithm.

6.1.4 Camera Pipeline

Implementing the ability for the FPGA to collect images from a camera and then compress the data for the CNN would be a beneficial addition to complete the collection to prediction pipeline and test the transfer times more thoroughly to determine if a speed-up is accomplished when capturing an image that needs to be sent to a CNN on another computer. In **Figure 6.1**, a picture at

Clemson Memorial Stadium was taken using a camera connected to the FPGA.

Figure 6.1: Mac Shaughnessy at Clemson Memorial Stadium compressed to JPEG 50

6.1.5 CNN Improvements

There are interesting options to improve the CNN. The two primary design alternates would be improving the existing models by better training the data and other general optimizations. The other could entail creating a CNN to make its predictions based on the encoded arrays [20].

For general improvements, it is worth noting that the image created is "blocky", meaning each 8×8 block does not transition well to the next as seen in **Figure 6.2**. This "blockiness" and its effects on CNN performance were not directly determined; however, more research on how this specifically affects the CNN could be interesting. Research into applying filters aimed at smoothing an image could increase the accuracy of the CNN.

Figure 6.2: 64 x 64 zoomed section of Image at JPEG Quality 25

Appendices

Appendix A Code Snippets

```
YCbCr Image[512][512]={};
ap_uint<8> Cr[8][8]={};
ap_uint<8> Cb[8][8]={};
YCbCr_ROWS:
for(int i = 0; i < 512/16; i++) {
    YCbCr_COLS:
    for(int j = 0; j < 512/16; j++) {
        color1:for(int l=0; l < BLOCK_SIZE; l++){
            color2:for(int k = 0; k < BLOCK_SIZE; k++) {
                ap_uint<16> r, g, b = MatRGB_DRAM
                ↪  [(BLOCK_SIZE*i)+l][(BLOCK_SIZE*j)+k];

                Image[l][k].Y = 16+(((r<<6)+(r<<1)+
                ↪  (g<<7)+g+
                ↪  (b<<4)+(b<<3)+b)>>8);
                Image[l][k].Cb = 128+(((r<<7)-(r<<4)-
                ↪  ((g<<6)+(g<<5)-(g<<1))-
                ↪  ((b<<4)+(b<<1)))>>8);
                Image[l][k].Cr =
                ↪  128+((-((r<<5)+(r<<2)+(r<<1))-
                ↪  ((g<<6)+(g<<3)+(g<<1))+
                ↪  ((b<<7)-(b<<4)))>>8);
            }
        }
        // New Channels can be used for DCT or Downsampling
    }
}
```

Figure 3: HLS Color Conversion 512×512 image

```
YCbCr Image[512][512]=RGB2YCbCr_image;
ap_uint<8> Cr[8][8]={};
ap_uint<8> Cb[8][8]={};
YCbCr_ROWS:
for(int i = 0; i < 512/16; i++) {
    YCbCr_COLS:
    for(int j = 0; j < 512/16; j++) {
        down_ROWS:
        for(int k = 0; k < 8; k++) {
            down_COLS:
            for(int l = 0; l < 8; l++) {
                Cr[k][l] = ((
                    Image[k*2][l*2].Cr +
                    Image[(k*2)+1][(l*2)+1].Cr +
                    Image[(k*2)+1][l*2].Cr +
                    Image[k*2][(l*2)+1].Cr
                )>>2);

                Cb[k][l] = ((
                    Image[k*2][l*2].Cb +
                    Image[(k*2)+1][(l*2)+1].Cb +
                    Image[(k*2)+1][l*2].Cb +
                    Image[k*2][(l*2)+1].Cb
                )>>2);
            }
        }
        // Downsampled Channels can be used for DCT
    }
}
```

Figure 4: HLS Downsampling 512×512 image

```
//If Y channel T=1 if Cb or Cr T=2 for 1/4 size
M1: for(int i=0;i<2/T; i++){
    M2: for(int j=0; j<2/T; j++){
        // 8x8 multiplication
        L1: for(int l=0; l<8; l++){
            L2: for(int k=0; k<8; k++) {
                real_ts32 temp[8]={};
                L3: for(int p = 0; p < 8; p++) {
                    temp[p]=
                     ↪  (dct[l][p]*MatIn[(i*8)+p][(j*8)+k]);
                }
                L4: for(int p = 0; p < 8; p++) {
                    MatMid[(i*8)+l][(j*8)+k]+=temp[p];
                }
                MatMid[(i*8)+l][(j*8)+k]>>12;
            }
        }
        // 8x8 Transpose multiplication
        L1: for(int l=0; l<8; l++){
            L2: for(int k=0; k<8; k++) {
                real_ts32 temp[8]={};
                L3: for(int p = 0; p < 8; p++) {
                    temp[p]=
                     ↪  (dct[l][p]*MatMid[(i*8)+p][(j*8)+k]);
                }
                L4: for(int p = 0; p < 8; p++) {
                    MatOut[(i*8)+l][(j*8)+k]+=temp[p];
                }
                MatOut[(i*8)+l][(j*8)+k]>>12;
            }
        }
        //Quantization
        L21: for(int l=0; l<8; l++){
            L22: for(int k = 0; k < 8; k++) {
                MatOut[(i*8)+l][(j*8)+k]=
                 ↪  (MatOut[(i*8)+l][(j*8)+k]/divide[l][k]);
            }
        }
    }
}
```

Figure 5: HLS DCT Matrix multiplication and Quantization

```
for(int i=0; i<8; i++){
    for(int j=0; j<8; j++){
        temp[i*8+j]=in[i*8+i][j*8+j];
    }
}

const int zz[64] = {
    0,1,8,16,9,2,3,10,
    17,24,32,25,18,11,4,5,
    12,19,26,33,40,48,41,34,
    27,20,13,6,7,14,21,28,
    35,42,49,56,57,50,43,36,
    29,22,15,23,30,37,44,51,
    58,59,52,45,38,31,39,46,
    53,60,61,54,47,55,62,63
};
for(int i =0; i < 64; i++){
    out[(64*(2*i+j))+i]=temp[zz[i]];
}
```

Figure 6: HLS Zig Zag

```
//If Y channel T=1, if Cb or Cr T=2 for 1/4 size
rle: for(int j=0; j<4/(T*T);j++){
    rle64: for(int i=0; i<64;i++){
        count=count+1;
        if(array[64*j+i]!=0){
            out[n++]=array[64*j+i];
            count=0;
        } else if(array[64*j+i]!=array[64*j+i+1] || i==63){
            out[n++]=array[64*j+i];
            out[n++]=count;
            count=0;
        }
    }
}
```

Figure 7: HLS RLE

Appendix B Error Images

Figure 8: Decoded image with incorrect quantization matrix

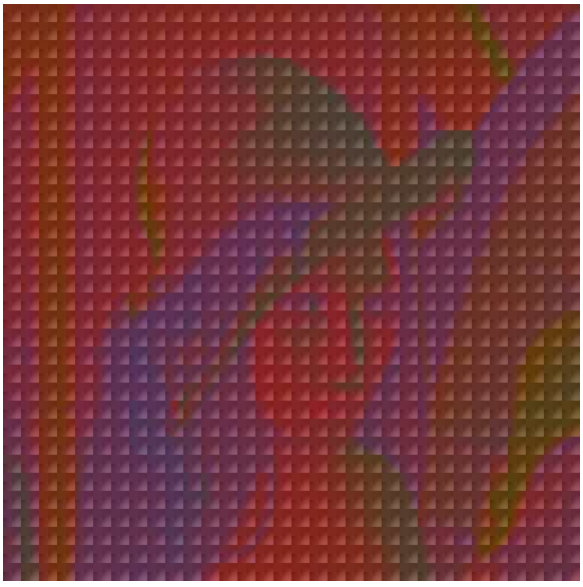

Figure 9: Repeated Chroma Channels

Figure 10: Incorrect Chroma Data Stored in RLE

Figure 11: Unknown Error in DCT phase

Figure 12: Incorrect decoding: Last 8x8 block in a 16x16 is Inverse DCT on 0

Figure 13: Incorrect data size

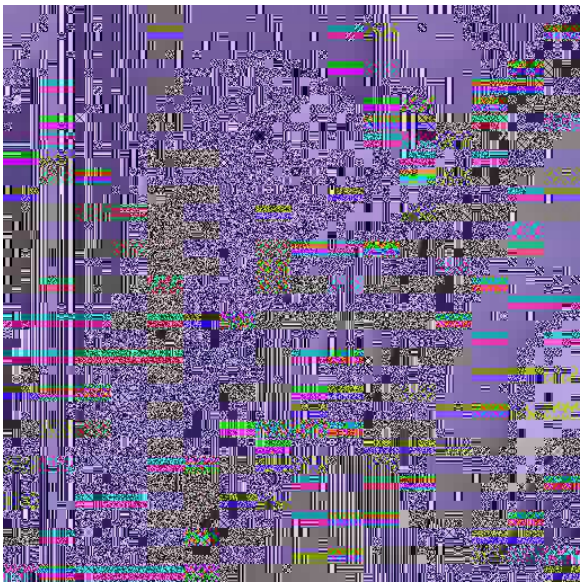

Figure 14: Decoded unsigned 8 bit data

Bibliography

[1] Muzhir AL-Ani and Fouad H. Awad. The jpeg image compression algorithm. *International Journal of Advances in Engineering & Technology*, 6(3):1055–1062, 07 2013. Copyright - Copyright I A E T Publishing Company Jul 2013; Document feature - Photographs; Diagrams; Equations; Graphs; Tables; Last updated - 2023-11-25.

[2] AMD. Vitis hls, 11 2023. Available at `https://docs.amd.com/r/en-US/Vitis-Tutorials-Getting-Started/Vitis-Tutorials-Getting-Started`.

[3] Ricardo Barreto, Jorge Lobo, and Paulo Menezes. Edge computing: A neural network implementation on an iot device. In *2019 5th Experiment International Conference (exp.at'19)*, pages 244–246, 2019.

[4] Kaustav Brahma, Viksit Kumar, Anthony E. Samir, Anantha P. Chandrakasan, and Yonina C. Eldar. Efficient binary cnn for medical image segmentation. In *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pages 817–821, 2021.

[5] R. Candra, Sarifuddin Madenda, S.A. Sudiro, and Muhammad Subali. The implementation of an efficient zigzag scan. *Journal of Telecommunication, Electronic and Computer Engineering*, 9:95–98, 04 2017.

[6] DTI. Available at `https://www.decisivetactics.com/support/view?article=using-serial-with-raspberry-pi#:~:text=The%20Raspberry%20Pi%20Zero%20and%20Raspberry%20Pi%204%20are%20the%20only%20exceptions.&text=Your%20terminal%20software%20must%20be.pdf`.

[7] MD Editorial. Jpg vs jpeg file formats: Is there a difference?, 2020. Available at `https://meridianthemes.net/jpg-vs-jpeg-file-formats/#:~:text=JPEG%20vs%20JPG%3A%20It%27s%20All%20the%20Same&text=The%20only%20difference%20between%20the,been%20JPEG%2C%20and%20vice%20versa.`

[8] Tao Wei et al. Real and complex matrix multiplication, 2024. Available at `https://uri-nextlab.github.io/ParallelProgammingLabs/HLS_Labs/Lab24.html`.

[9] GeeksforGeeks. Object detection vs object recognition vs image segmentation, 06 2022. Available at `https://www.geeksforgeeks.org/object-detection-vs-object-recognition-vs-image-segmentation/?ref=lbp`.

[10] Xiaoqiang He. Subjective evaluation of image color damage based on jpeg compression. In *2014 Fourth International Conference on Communication Systems and Network Technologies*, pages 838–842, 2014.

[11] Ayoush Johari and Preety D. Swami. Comparison of autonomy and study of deep learning tools for object detection in autonomous self driving vehicles. In *2nd International Conference on Data, Engineering and Applications (IDEA)*, pages 1–6, 2020.

[12] Temurbek Kuchkorov, Temur Ochilov, Elyor Gaybulloev, Nazokat Sobitova, and Ortik Ruzibaev. Agro-field boundary detection using mask r-cnn from satellite and aerial images. In *2021 International Conference on Information Science and Communications Technologies (ICISCT)*, pages 1–3, 2021.

[13] Enas Dhuhri Kusuma and Thomas Sri Widodo. Fpga implementation of pipelined 2d-dct and quantization architecture for jpeg image compression. In *2010 International Symposium on Information Technology*, volume 1, pages 1–6, 2010.

[14] Andrew Louie and Albert M. K. Cheng. Work-in-progress: Designing a server-side progressive jpeg encoder for real-time applications. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 379–382, 2020.

[15] Andreas Bay Madsen, Rasmus Faurskov, and Ali Sahafi. Jpeg-ls compression on fpga: A solution for wireless capsule endoscopy. In *2023 IEEE International Conference on Imaging Systems and Techniques (IST)*, pages 1–5, 2023.

[16] Fouad Hammadi Awad Muzhir Shaban AL-Ani. The jpeg image compression algorithm. In *International Journal of Advances in Engineering & Technology*, volume 6, pages 1055–1062, 2013.

[17] PYNQ. Pynq introduction, 2022. Available at `https://pynq.readthedocs.io/en/v3.0.0/getting_started/pynq_z2_setup.html`.

[18] G Sandhya, Aakarshitha Suresh, T Malini, and B Rajeshwari. Fpga implentation of jpeg image compression. In *2019 3rd International Conference on Recent Developments in Control, Automation & Power Engineering (RDCAPE)*, pages 143–148, 2019.

[19] D. Santa-Cruz and T. Ebrahimi. An analytical study of jpeg 2000 functionalities. In *Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101)*, volume 2, pages 49–52 vol.2, 2000.

[20] Ashutosh Singh, Bulla Rajesh, and Mohammed Javed. Deep learning based image segmentation directly in the jpeg compressed domain. In *2021 IEEE 8th Uttar Pradesh Section International Conference on Electrical, Electronics and Computer Engineering (UPCON)*, pages 1–6, 2021.

[21] Masayuki Sugawara, Seo-Young Choi, and David Wood. Ultra-high-definition television (rec. itu-r bt.2020): A generational leap in the evolution of television [standards in a nutshell]. *IEEE Signal Processing Magazine*, 31(3):170–174, 2014.

[22] G.K. Wallace. The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1):xviii–xxxiv, 1992.

[23] Guotian Yang, Minglei Han, Yingnan Wang, Zicheng Wang, and Geng Wei. Ycbcr color space based image segmentation algorithm with feedback in the underground cable tunnel. In *2019 Chinese Automation Congress (CAC)*, pages 692–696, 2019.

[24] Rui Zhan, Yi Ma, Shanshan Yang, Yufei Man, and Yu Yang. An advanced jpeg steganalysis method with balanced depth and width based on fractal residual network. In *2021 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 345–351, 2021.