Hardware Implementation of Discrete Cosine Transform and Quantization for Image Compression

GitHub Repository:
https://github.com/bkaether/ee274_image_codec

EE 274
Data Compression, Theory and Applications

Brian Kaether, Marc Huerta
Department of Electrical Engineering
Stanford University
December 2023

Contents

1 Introduction

Image compression plays a pivotal role in addressing the ever-growing demand for efficient data storage, transmission, and processing. As digital content continues to proliferate across various platforms such as social media, the volume of images being shared and stored has skyrocketed. Image compression technology is crucial in reducing the size of these files without significant loss of visual quality, enabling quicker data transfer and minimizing the strain on storage infrastructure. This not only facilitates faster upload and download speeds but also optimizes bandwidth usage, making it indispensable for online communication, streaming services, and mobile applications. Additionally, since image compression is so common, it is extremely beneficial to have hardware that can accelerate this compression.

In particular, JPEG is one of the most widely used image formats in the world, so the compression of JPEG images is a very important process in today's world. For our project, we will implement the following critical components used in JPEG and other image compression/decompression algorithms using SystemVerilog:

- Transformation of the data stream into uncorrelated samples using the 2D discrete cosine transform (DCT)
- Quantization and rounding of the transform coefficients
- De-quantization of transform coefficients
- Transformation of the de-quantized coefficients to proper image data using the inverse discrete cosine transform (IDCT)

2 Software Implementation

In order to gain the best understanding of the algorithm before jumping into a hardware implementation, we decided to first implement a "gold model" in python. The python OpenCV library provides both a discrete cosine transform (DCT) and inverse discrete cosine transform (IDCT) function already, which we will use for the performance comparison, but we decided that implementing our own version from scratch would be a useful exercise for implementing the hardware version later on. It is important to note that for simplicity in hardware we decided on a fixed block size of 8 by 8.

2.1 Discrete Cosine Transform

For a two dimensional matrix, M, of size N by N, the discrete cosine transform output, T, also of size N by N, is defined as follows:

$$T(u,v) = \alpha(u)\alpha(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} M(x,y) \cos\left[\frac{u\pi}{2N}(2x+1)\right] \cos\left[\frac{v\pi}{2N}(2y+1)\right] \quad (1)$$

Where $\alpha(i)$ is defined as:

$$\alpha(i) = \begin{cases} \sqrt{\frac{1}{N}} & \text{if } i = 0 \\ \sqrt{\frac{2}{N}} & \text{if } i \neq 0 \end{cases} \tag{2}$$

Here is our corresponding python function for taking the DCT of a provided block:

```
def dct_2d(block):
    N = block.shape[0]
    dct_block = np.zeros_like(block, dtype=float)
    for u in range(N):
        for v in range(N):
            sum = 0
            for x in range(N):
                for y in range(N):
                    sum += block[x, y] * np.cos((2 * x + 1) * u * np.pi / (2 * N)) * np.cos((2 * y + 1) * v * np.pi / (2 * N))
            alpha_u = 1/np.sqrt(N) if u == 0 else np.sqrt(2)/np.sqrt(N)
            alpha_v = 1/np.sqrt(N) if v == 0 else np.sqrt(2)/np.sqrt(N)
            dct_block[u, v] = alpha_u * alpha_v * sum
    return dct_block
```

Similarly, the inverse discrete cosine transform for a two dimensional matrix is given by:

$$T(x,y) = \sum_{u=0}^{N-1} \sum_{v=0}^{N-1} \alpha(u)\alpha(v)M(u,v) \cos\left[\frac{u\pi}{2N}(2x+1)\right] \cos\left[\frac{v\pi}{2N}(2y+1)\right] \tag{3}$$

And here is our corresponding python function for taking the IDCT of a provided block:

```
    def idct_2d(dct_block):
        N = dct_block.shape[0]
        reconstructed_block = np.zeros_like(dct_block, dtype=float)
        for x in range(N):
            for y in range(N):
                sum = 0
                for u in range(N):
                    for v in range(N):
                        alpha_u = 1/np.sqrt(N) if u == 0 else np.sqrt(2)/np.sqrt(N)
                        alpha_v = 1/np.sqrt(N) if v == 0 else np.sqrt(2)/np.sqrt(N)
                        sum += alpha_u * alpha_v * dct_block[u, v] * np.cos((2 * x + 1) * u * np.pi / (2 * N)) * np.cos((2 * y + 1) * v * np.pi / (2 * N))
                reconstructed_block[x, y] = sum
        return reconstructed_block
```

2.2 Quantization

For image compression algorithms such as JPEG which utilize the DCT and quantization steps, there are established, predefined quantization matrices that are used for different color channels. For simplicity, we decided to center our project around grayscale images, so we used the quantization matrix that was associated with the achromatic or luminance channel, defined as follows:

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

The forward quantization step simply consists of element-wise division between the DCT output block and the quantization matrix, followed by rounding to the nearest integer. Similarly, the backwards "dequantization" step consists of just multiplication between the coefficient outputs from DCT and the same quantization matrix:

```
    def compress_block(block, quantization_matrix):
        dct_block = dct_2d(block)
        return np.round(dct_block / quantization_matrix)

    def decompress_block(block, quantization_matrix):
        dequantized = block * quantization_matrix
        return idct_2d(dequantized)
```

The following python code completes our software implementation. We read in a grayscale image using the OpenCV python library, then process the image one block at a time, saving both the compressed coefficients and the reconstructed image data. We then use the CV2 library to write the output image (numpy array) to a JPEG file for viewing.

```
    def process_image(image, quantization_matrix):
        h, w = image.shape
        compressed = np.zeros_like(image, dtype=np.float32)
        decompressed = np.zeros_like(image, dtype=np.uint8)
        for i in range(0, h, 8):
```

```
            for j in range(0, w, 8):
                block = image[i:i+8, j:j+8]
                compressed_block = compress_block(block,
    quantization_matrix)
                decompressed_block = decompress_block(
    compressed_block, quantization_matrix)
                compressed[i:i+8, j:j+8] = compressed_block
                decompressed[i:i+8, j:j+8] = np.clip(
    decompressed_block, 0, 255)
        return compressed, decompressed
```

2.3 Results

Here is the raw grayscale image that we chose to use for our project, followed by the software image reconstruction. As a reminder, this image reconstruction is the output image after the raw image is put through DCT and quantization, followed by "dequantization" and IDCT. In full image compression, entropy encoding would take place after quantization. The mean squared error (MSE) between these two images is 43.14, but it is basically impossible to tell the difference. This MSE value will be a useful metric for determining just how much is lost due to the hardware algorithm adaptations we made, which we discuss further in the following sections.

Figure 1: Raw grayscale image

Figure 2: Software image reconstruction

3 Hardware Implementation

3.1 Algorithm Adaptations

Our goal for the hardware implementation was to create synthesizable SystemVerilog modules that would be able to perform all the same steps as our python model, and produce a reconstructed image with imperceptible differences from the raw image. Given these self imposed constraints, there were a few main problems which we had to work around.

First, the division operator is not synthesizable in SystemVerilog. Given that our quantization step involves element-wise division between the DCT block output and our quantization matrix, we needed to find an alternate method of quantization. Knowing that bit shifting is not only synthesizable, but efficient as well, we decided to round each element in the quantization matrix to the nearest power of two. This way, we could simply create an 8x8 register file that would act as a lookup table on chip, where each element is the log base 2 of the corresponding rounded quantization matrix value, then perform element-wise bit shifting. These lookup table values could be pre-computed ahead of time, because the quantization matrix is always the same. We verified that this would not create an overly distorted image reconstruction ahead of time by using the rounded quantization matrix for both compression and decompression in python. Then we created a utility python function for converting a numpy

```
[[ 16   8   8  16  32  32  64  64]
 [ 16  16  16  16  32  64  64  64]
 [ 16  16  16  32  32  64  64  64]
 [ 16  16  16  32  64  64  64  64]
 [ 16  16  32  64  64 128 128  64]
 [ 32  32  64  64  64 128 128 128]
 [ 64  64  64  64 128 128 128 128]
 [ 64 128 128 128 128 128 128 128]]
```

Figure 3: Quantization matrix rounded to nearest powers of two.

```
[[4 3 3 4 5 5 6 6]
 [4 4 4 4 5 6 6 6]
 [4 4 4 5 5 6 6 6]
 [4 4 4 5 6 6 6 6]
 [4 4 5 6 6 7 7 6]
 [5 5 6 6 6 7 7 7]
 [6 6 6 6 7 7 7 7]
 [6 7 7 7 7 7 7 7]]
```

Figure 4: Shift values that are loaded into memory for the quantization module.

array to a .mem file format, so that the correct values could be easily loaded into memory using the $readmem system functions in verilog. These readmem functions are the only non-synthesizable code in our implementation. For reference, here is what the rounded quantization matrix looks like along with the actual shift values that are loaded into memory.

Second, the cosine operator is also not synthesizable in SystemVerilog, but we were able to solve this problem in a similar way to the quantization problem. If we take a close look at the cosine terms in our DCT and IDCT equations, we see that the variables u, v, x, and y all go from 0 to N-1, where N is our fixed block size of 8. This means that the cosine terms can only ever be a total of 64 different values, since they only ever depend on either u and x, or v and y.

$$\cos\left[\frac{u\pi}{2N}(2x+1)\right] \tag{4}$$

$$\cos\left[\frac{v\pi}{2N}(2y+1)\right] \tag{5}$$

Again, we compute the 64 values that the cosine terms can be ahead of time and load them into an 8x8 memory on chip to be used as a lookup table using the verilog readmem function.

The last problem is that our python version worked with floating point numbers, but floating point is a real pain to work with in hardware. Although it would likely be another source of distortion, we decided to use fixed point for all our hardware computations, then interpret the final results accordingly based on the expected number of fractional bits. We wrote some utility functions in python to convert signed floating point numbers into fixed point binary representations for the cosine value lookup table and other floating point constants such as α. All these functions can be seen in our GitHub repository.

3.2 Design Decisions

Our python implementation of both DCT and IDCT contains four nested loops, and even though we are only running simulations and don't have hardware to

synthesize on, we wanted to avoid an unrealistic amount of hardware unrolling, so we decided to unroll only the two innermost loops. With this implementation, the DCT output for a single pixel value is calculated in a single cycle, and the DCT output for each pixel in the block is done sequentially, taking a total of 64 cycles to complete one block. We implemented a double counter module in SystemVerilog to output the correct outer loop indices, and used flops with enables for each pixel in the block to save the current combinational result to the correct location. As a result of this sequential, pixel by pixel design, The DCT and IDCT modules are both simple FSMs, and the quantization and dequantization modules are purely combinational logic. Although this would likely be something to change given more time, our top level module simply instantiates as many blocks as it takes to cover the entire input image in parallel, so the entire compression and decompression of the image can be completed in 64 cycles (admittedly at the cost of large area most likely). The verilog modules can get quite large and are more difficult to understand intuitively so I have not included the code in the report. All the RTL, testbenches, python scripts, and output images can be found in the GitHub repository!

Another design decision we made that could be changed is the number of fractional bits preserved throughout all the fixed point multiplications found in the algorithm. When you multiply two fixed point numbers together, in order to not lose information and prevent overflow, the result must be stored using an amount of integer bits equal to the sum of integer bits in the multiplicands, and an amount of fractional bits equal to the sum of fractional bits in the multiplicands. For example, if we multiply a fixed point number of format Q6.4 (6 int bits, 4 frac bits) with another number of format Q8.2, the result should be stored in format Q14.6. Once you have the result you can then shift or take a subsection of the bits to achieve your desired precision.

There is a lot of multiplication in this algorithm, and in the current implementation, we didn't shift or slice the result before doing the next multiplication, so the bit widths of signals ended up quite high. In some cases we maintain 32 bits of fractional precision. This isn't a problem at all in terms of simulation results, in fact it helps minimize the distortion between the software and hardware versions, but high bit width multiplication is quite costly in terms of energy and area. If we were to synthesize this design and run it on real hardware, we would likely want to make the bit widths as small as possible at every step.

3.3 End to End Simulation

We tested our design gradually, beginning with concrete testbenches for the more simple modules such as the double counter and quantization modules, and then moving on to the more complex DCT and IDCT modules. We tested and ensured that the results of compression and decompression of the first block of the image in hardware matched the results of the python implementation before moving onto testing the entire image. In this section I will describe the end to end process for getting the reconstructed image out of our hardware design in order to sufficiently cover the entirety of our project efforts and allow the results

to be reproduced if desired.

First, we run the mem_gen.py python script to generate all the necessary memory files in the right format, so that we can load the correct data into the design. This script will generate memory files for the input image data, the fixed point cosine values, and the rounded quantization matrix shift values.

Second, we run the top level compressor testbench, compressor_top_tb.sv in Vivado. This performs the DCT and quantization steps on the input image and writes the quantized coefficients to its own output memory file.

Third, we run the top level decompressor testbench, decompressor_top_tb.sv in Vivado. This testbench reads in the values that were output from the previous step, and performs dequantization and IDCT steps, and writes the reconstructed grayscale image data to its own memory file.

Fourth, we run the process_rtl_output.py python script. This script converts the raw output from the memory file to an appropriately shaped numpy array, and then uses the CV2 library to create the JPEG image based on the numpy array.

3.4 Results and Performance Comparison

Here is the raw grayscale image, followed by the RTL reconstruction (next page). For this reconstruction, both compression to coefficients from the raw image and reconstruction from these coefficients are done in hardware, using fixed point arithmetic. Again, it is basically impossible to tell the difference, however we do see that the MSE basically doubles when compared with the software reconstruction, increasing to 81.51 from 43.14.

In terms of speed, the compression of the image using python's CV2 library takes 48.80 ms on average, while the decompression using the CV2 library takes 46.69 ms on average. Our hardware implementation, for both compression and decompression of the entire image, takes just 64 cycles. Using this information, we can solve for the minimum clock frequency our design must run at to outperform the python implementation:

$$\begin{aligned} \frac{1}{F} * 64 &\leq 46.69\ ms \\ \frac{1}{F} &\leq 0.7295\ ms \\ F &\geq 1.37\ kHz \end{aligned} \tag{6}$$

Although this looks like an extremely easy target to hit, there are a couple things that are worth mentioning. As it stands, with a single combinational block handling all four multiplications and the summation of 64 elements, this would surely result in a very slow critical path, severely limiting our clock frequency. However this problem can be solved easily with a few pipeline stages, which would be the next thing we would implement if we were synthesizing the design.

Figure 5: Raw grayscale image

Figure 6: RTL image reconstruction

Additionally, we are working with processing 640x480 images, meaning we have a total of 4800 compressor block modules instantiated in parallel in order to complete the design in 64 cycles. This likely results in a massive chip area. However, in a more realistic design, even if we were only processing one image block row in parallel at a time (80 instantiations) and doing the rest sequentially, the frequency target should still not be too hard to surpass, especially with pipelining. The new frequency target with this design without pipelining can be calculated as follows (64 cycles for a block row to complete, 60 block rows in total):

$$\begin{aligned} \frac{1}{F} * 64 * 60 &\leq 46.69\ ms \\ \frac{1}{F} &\leq 0.0122\ ms \\ F &\geq 82.24\ kHz \end{aligned} \tag{7}$$

In conclusion, we achieved our goal of doing full image compression and reconstruction using synthesizable hardware, achieving a reconstructed image with imperceptible differences from the raw image. Although concrete performance numbers are a bit difficult to estimate without having a hardware target to synthesize for, it looks like we should easily be able to surpass software performance, especially with some slight design modifications.