FPGA-Based Color Image Compression Using 2D DCT

Author: Venkatesh Arumugam

Course: EECE 5698 – FPGA's in the Cloud

Phase: Second Project Update

Abstract / Overview

This project aims to design and implement a color image compression accelerator using High-Level Synthesis (HLS) targeting FPGA. The design will employ the **2D Discrete Cosine Transform (DCT)** to compress color images by converting spatial domain pixel data into the frequency domain, significantly reducing data redundancy. The project will begin with a functional HLS implementation and progress toward optimization in terms of performance, resource utilization, and design complexity. This project is both computationally complex and relevant to real-world applications in image processing and FPGA acceleration.

Work Completed in Update 1

A full software reference pipeline was completed using **C++ and STB image libraries**, including:

- Loading PNG images
- Splitting into R/G/B channels
- Zero-padding to multiples of 8
- Implementing a textbook 2D DCT (float-based)
- Generating CPU DCT outputs (for correctness reference)

This provides a baseline for verifying FPGA correctness.

Pipeline implemented for Update 2:

1. Load a PNG image using STB library (CPU)
2. Split into R/G/B channels
3. Transfer each channel to FPGA global memory
4. FPGA kernel performs 8×8 forward DCT on each channel
5. Transfer results back to host
6. Recombine channels and save as an output PNG

This update completes the full hardware-accelerated forward DCT pipeline, validated using hardware emulation (hw_emu).

How To Run

1) Get a Build Machine and clone the following Repository in terminal.
https://github.com/Venkatesh-Arumugam/fpga.git

2) To configure the environment to run Vitis commands, run the following shell commands.
source /tools/Xilinx/Vitis/2023.1/settings64.sh
source /opt/xilinx/xrt/setup.sh

3)Then Run the following
cd fpga
make hw_emu
export XCL_EMULATION_MODE=hw_emu
once Build is Finished
./build/host.exe build/dct_hw_emu.xclbin data/input.png data/output.png

Output Obtained during hw_emu

```
Venkat14@fpga-tools:~/fpga$ ./build/host.exe build/dct.xclbin ./data/test_64x64.png output.png
Loaded 64x64 (3 channels)
Padded to 64x64
Opening device 0...
CRITICAL WARNING: [HW-EMU 08-0] Unable to find emconfig.json. Using default device "xilinx:pcie-hw-em:7v3:1.0"
Loading xclbin: build/dct.xclbin
INFO: [HW-EMU 05] Path of the simulation directory : /users/Venkat14/fpga/build/.run/9312/hw_emu/device0/binary_0/b

 server socket name is  /tmp/Venkat14/device0_0_9312
INFO: [HW-EMU 01] Hardware emulation runs simulation underneath. Using a large data set will result in long simulat
 dataset is used for faster execution. The flow uses approximate models for Global memories and interconnect and he
roximate.
configuring penguin scheduler mode
scheduler config ert(0), dataflow(1), slots(16), cudma(1), cuisr(0), cdma(0), cus(1)
Opening kernel 'dct'...
Opening kernel 'dct'...
Running kernel dct(64x64)...
Kernel execution complete.
Wrote output image: output.png
INFO: [HW-EMU 06-0] Waiting for the simulator process to exit
INFO: [HW-EMU 06-1] All the simulator processes exited successfully
INFO: [HW-EMU 07-0] Please refer the path "/users/Venkat14/fpga/build/.run/9312/hw_emu/device0/binary_0/behav_wave
```

Input Image:

Output image obtained

The output produced by the FPGA hardware-emulation run appears as a uniform gray, block-structured pattern because the accelerator currently implements only the forward 8×8 Discrete Cosine Transform (DCT) for each RGB channel. DCT coefficients are frequency-domain values, not pixel intensities, so they do not form a recognizable image when written directly to a PNG file. Since hardware emulation models the FPGA at an RTL-simulation level—including AXI transactions, DDR behavior, and kernel scheduling—it runs extremely slowly, often 100× slower than real hardware. For this reason, a small 64×64 synthetic test image was used instead of a full-resolution image from Project Update 1; using a real image would take several hours or fail to complete within the available runtime. The purpose of Update 2 is to demonstrate that computation is successfully offloaded to an accelerator (HLS kernel + Vitis hw_emu), not to perform full JPEG reconstruction. Thus, the simplified test input and the resulting DCT-coefficient visualization are expected and fully sufficient to prove correctness of the FPGA flow at this stage.

FPGA Implementation

Kernel: dct

The HLS kernel implements:

- 8×8 separable 2D DCT
- Fully unrolled inner loops for high parallelism
- Complete array partitioning for high throughput
- Processing of full RGB separately within DATAFLOW
- M_AXI interfaces with burst reads/writes
- AXIS-lite control interface

Key optimizations used:

- #pragma HLS UNROLL for all 8×8 loops
- #pragma HLS PIPELINE II=1 for block traversal
- #pragma HLS ARRAY_PARTITION complete
- #pragma HLS DATAFLOW to process three channels concurrently

Host-Side Implementation

The host program performs:

- Reading input PNG using STB
- Splitting into R/G/B vectors
- Allocating xrt::bo buffers
- Writing data into device memory
- Launching the kernel
- Reading DCT-transformed outputs
- Recombining into output.png using STB

Experimental Platform

I used a Build machine for this project update as Alveo U280 was not available for usage. Hardware Emulation was done using platform set to xilinx_u280_gen3x16_xdma_1_202211_1.

Next Steps

For the final project update, I will complete the remaining stages of the DCT acceleration pipeline by moving from hardware emulation to real execution on a U280 FPGA node (once available). I will run the hardware-target DCT kernel on the actual device and measure true performance using XRT kernel-event profiling and host-side timing. To validate correctness, I will use the golden outputs generated by my CPU encoder—specifically the padded channel files and the reference RLE bitstreams—as ground-truth. I will extend the CPU implementation to include an IDCT stage that exactly mirrors the FPGA's DCT basis, allowing me to reconstruct full images from the FPGA-generated coefficients and compute quantitative correctness metrics such as MSE and PSNR. Beyond functional testing, I will expand the FPGA kernel toward the full JPEG-style pipeline (DCT → Quant → Zigzag → RLE) and compare its outputs bit-for-bit against the CPU golden binaries. I also plan to evaluate performance using multiple real images of different sizes and compare CPU vs FPGA throughput. Finally, I will conduct systematic design-space exploration (DSE) by varying HLS optimizations such as loop unrolling, pipelining, dataflow restructuring, array partitioning, and fixed-point precision to study their effect on latency, initiation interval, and resource usage, and identify a Pareto-optimal hardware configuration. The final submission will include correctness analysis, reconstruction results, performance charts, and a summary of architectural trade-offs.

AI Usage

Portions of the code structure and documentation were refined with the assistance of AI to improve clarity and implementation accuracy.

Explanation for Previous Project Update Questions

1) What is the difference between out and out_padded?

Since we do DCT on 8*8 pixel block if the provided input image dimension is not exactly in multiples of 8 we add extra zeros at the end to round off to nearest multiple of 8 in the encoder. The image contains black bars at the output as we pad 0 to the end. This is the outpadded file. For the decoder if we provide the dimension of what our input was it removes the padding and provides the image with exact dimension of input which looks neat.

Outpadded -> output when the deocoder is run without knowledge of dimension of the input data that was compressed.

2) Explain the metrics used in your project.

I am using PSNR and MSE as comparison metric. PSNR, or Peak Signal-to-Noise Ratio, quantifies overall visual quality in decibels. It is inversely related to MSE —when MSE decreases, PSNR increases. Typically, PSNR above 40 dB indicates excellent quality, while 30–40 dB is acceptable for most visual applications.

MSE, or Mean Square Error, measures the average squared difference between the original and reconstructed pixel values. Lower MSE means the reconstructed image is closer to the original. In this project, the Mean Squared Error (MSE) was computed separately for each RGB channel to identify which color component contributes most to overall distortion. The Peak Signal-to-Noise Ratio (PSNR) was then computed per channel, and the final PSNR reported is the average across all three channels (R, G, and B).

3) Please provide the sources of the code.

The source code for this project (encoder, decoder, and PSNR calculator) was developed based on the JPEG image compression algorithm described in academic and open technical references. Ref: DCT Implementation

The implementation uses publicly available single-header C libraries — stb_image.h and stb_image_write.h by Sean Barrett — for reading and writing PNG images. Ref: https://github.com/nothings/stb

References

https://dl.acm.org/doi/full/10.1145/3530775#sec-2-3

stb_image and stb_image_write libraries, Sean Barrett, MIT License

https://www.geeksforgeeks.org/dsa/discrete-cosine-transform-algorithm-program/