Hardware-Accelerated Real-Time Video Share

Abdulaziz Aljuaid
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
aaljuaid@mit.edu

Dakota Goldberg
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
dakotag@mit.edu

***Abstract*—We propose a design that allows two FPGAs equipped with cameras to exchange high-quality video in real time. High-fidelity transmission has always been the target of in-hardware acceleration, one of the main use cases of FPGAs; we want to leverage the speed gains of utilizing an FPGA to process high amounts of data. The majority of our implementation lies in the JPEG-like compression scheme we use to encode raw camera frames. We transmit compressed data from one FPGA to a computer via UART or to another FPGA over a high-speed differential pair connection.**

I. Introduction

The fundamental constraint in real-time video transmission is *bandwidth*. A single frame of uncompressed HD video has at least 3 MB of data, which means that sending a second of 60 FPS video requires one to send *1440 Mb per second* if they are not compressing their data in any way. With the goal of making real-time video transmission between two FPGAs feasible, we designed and implemented a JPEG-inspired compression scheme and efficient packet construction pipeline to compress video frames in sequence, significantly reducing the number of bits needed to represent a second of video, then transmit the compressed frames to be decoded on another device.

II. Setup

For video capture, we use an OV5640 color image sensor and transform the raw camera signals into pixel space with a series of specialized modules designed to process and prepare image data from the camera sensor. The camera is clocked at 200 MHz, and we use a clock domain crossing FIFO buffer to generate a 74.25 MHz clock for our system logic.

III. Encoding

A. JPEG Compression

The **J**oint **P**hotographic **E**xperts **G**roup (JPEG) is a compression format which utilizes frequency-domain conversion to enable compression through partially discarding high frequency content, as well as other techniques. In total, the main steps in the compression procedure are initially chroma conversion and subsampling to discard unnoticeable chroma components and decrease amount of information fed into the rest of the pipeline. This is then followed by image division into 8x8 blocks for the purpose of applying the 2-dimensional Discrete Cosine Transform (DCT), then quantization to further narrow down higher frequency content, and finally entropy encoding to combine redundant zero-runs for further compression. As a compression technology, JPEG encoding can potentially achieve 1:10 to 1:50 image compression ratios with minimal impact on visual quality while being relatively simpler than other compression formats used for video.

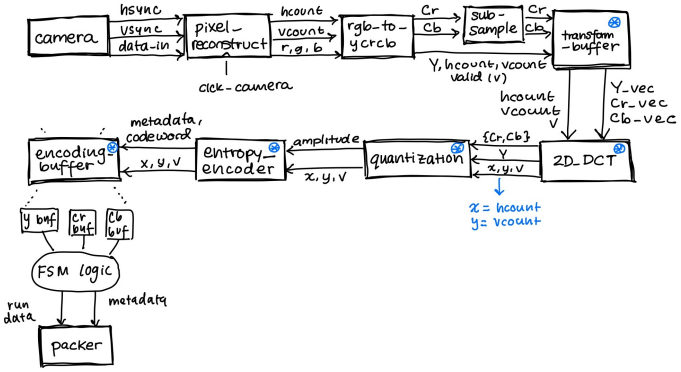


Fig. 1: Encoding pipeline block diagram. The blue ⊛ indicates that there exists one instance of this module for each channel.

B. Chroma Conversion and Subsampling (Abdulaziz)

Since most of the information is encoded in the luminance component of the image, this can be leveraged to down-sample the chroma content and lower the bandwidth requirement with minimal loss of quality. A subsampling ratio of 4:2:0 (half the vertical and horizontal resolutions for chroma) was implemented in the pipeline, reducing the total bits to be sent by half. However, one major trade-off was the separation of the three channels into separate pipelines, increasing the overall complexity by having multiple data-paths, and required a more robust packer module implementation that is able to combine the three streams into one serial connection.

C. Transform Buffer (Abdulaziz)

To enable processing of the image in 8x8 blocks in real-time, reordering of the camera input was needed. This is accomplished through the Transform Buffer module, which stores 16 lines of the input frame via one two-port BRAM, and outputs pixels in vertical fashion, incrementing the horizontal count every 8 cycles, thus sending 8-pixel vertical vectors one by one and sweeping an HRES*8 block in the process, as

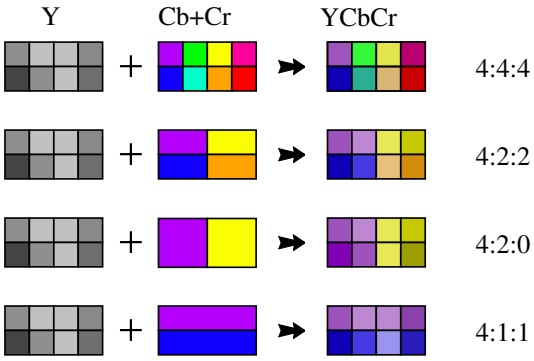


Fig. 2: Common subsampling ratios. [2]

shown in the figure. Moreover, since the reading and writing patterns are different for the transform buffer, pixel values are written to one 8-line section while read from the other 8-line section to avoid read-write problems. Lastly, the module keeps track of `valid_out`, `hcount_out`, and `vcount_out` as a function of input coordinates and validity.

Fig. 3: Read-write logic of the transform buffer

D. 1D Discrete Cosine Transform (Abdulaziz)

To construct the 2D DCT module, a pair of 8-point one-dimensional dct modules that can operate on signed fixed-point inputs were needed. Internally, the one-dimensional transform operates as a matrix multiplication between a precomputed 8x8 matrix of cosines, and an 8-value vector of pixels handed to the matrix multiplication module from a small 2x8 cache. To maintain consistent throughput, the module takes as input one pixel value per cycle and, in that cycle, computes the multiplication result of one row as a frequency component of a single value. Thus, for every eight pixels the module will calculate the eight frequency coefficients, all in the span of eight cycles.

E. 2D Discrete Cosine Transform (Abdulaziz)

The main compression method used in JPEG, DCT converts spatial data into the frequency domain, for the purpose of further compression by discarding high-frequency components in the 8x8 pixel block. In-hardware, the 2D DCT is constructed via a pair of 1D DCT modules operating differently on the data stream passing through the module. In particular, while one DCT module operates row-wise on the data, the other would need to operate column-wise. To accomplish this, a single 2-port BRAM was used to store two 8x8 blocks, such that, similarly to the transform buffer, the second operating DCT module can take data in transpose to the first one, without missing or stopping the data stream.

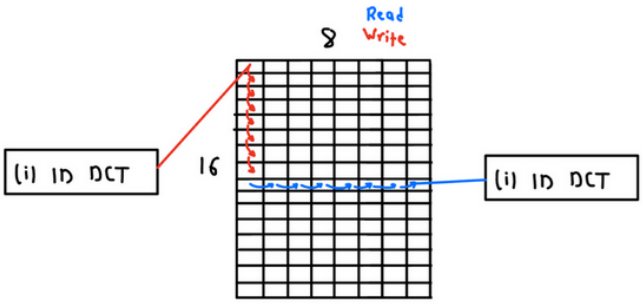


Fig. 4: 2D (i)DCT structure

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \qquad \text{for } k = 0, \ \ldots\ N-1\,.$$

Fig. 5: type-ii dct, in this case n=8. [3]

To accomplish this, the input pixels are transformed into signed fixed-point values, which are then fed to matrix multiplication that implements the Type-II DCT, computing one row-column multiplication value per cycle.

F. Quantization (Dakota)

To reduce the number of values needed to represent the 8x8 blocks in the frequency domain, we divide each value f_{ij} of the 8x8 block by q_{ij} from quantization matrix $\mathbf{Q}$ and round the quotient to the nearest integer. Quantization reduces the number of nonzero values (making the block easier to compress and transmit) while also optimizing for human perception by dividing higher frequencies by larger quantization coefficients.

We use one quantization matrix for the Y channel and another for the chroma channels [4]. These are both stored in BRAM and accessed using IP. The signed quantization division is carried out by a fully pipelined divider.

G. Entropy Encoding (Dakota)

Typical of JPEG compression pipelines, we use run-length encoding, compressed further with a few Huffman codewords, to encode the quantization coefficients. To optimize for runs with many contiguous zeros, we read the data in order from low frequencies to high frequencies, resulting in the zig-zag ordering in Figure 1.

We use two alternating dual read/write BRAMs to store elements from one block into memory as we read elements

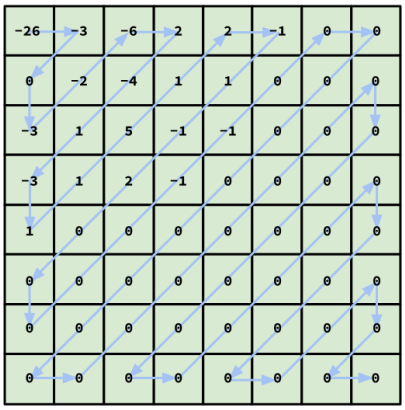


Fig. 6: The element ordering used for entropy encoding [1].

from the next one according to the zig-zag pattern. Then we carry out the run-length encoding algorithm, sending encoded runs as they are formed.

The first value in the sequence of quantization coefficients is encoded using Differential Pulse Code Modulation (DPCM), which stores the difference between the first value of the current block and the corresponding value of the previous block. The subsequent coefficients are encoded as runs of the following types:

1) Nonzero coefficients are encoded as `(run_length, value_width)(value)` where `run_length` is a number between 0 and 15 equal to the number of zeros occurring before this value. Because we `value_width` holds the bit length of `value`, we can save a bit by storing `value` in ones-complement.
2) If there are more than 15 zeroes in a row, they are encoded with fixed-width tuple `(15, 0)(0)`, where the middle zero in the tuple is represented with as many bits as we'd typically use for `value_width`.
3) The encoding ends with The encoding ends with Huffman codeword `(0, 0)` to indicate no more nonzero values in the block.

There are a few properties of this process that make its implementation non-trivial. First, although we output runs to the next module as we form them, we won't know which run contains the last nonzero coefficient until we reach the end of the block. To solve this problem, we keep track of the number of runs up to the most recent nonzero coefficient and send this number to the following modules on the last cycle of this block to ensure they only pack runs up to that point.

Further, because our packet format requires we know the entire length of the data we are sending and stores this in the packet *before* the data, we keep track of length as we do run-length encoding. On the last cycle of every block, we send all the metadata necessary to properly encode and pack the block, including the number of runs, total length of the runs (up to the last nonzero run), the encoded DC coefficient, as well as the x and y coordinates of this block, which are essentially `hcount` and `vcount` with the bottom three bits discarded.

This process reduces redundancy in the data, optimizing it for efficient storage and transmission. There is one entropy encoder module for each channel.

H. Encoding Buffer (Dakota)

As the entropy encoder is encoding a block, it writes the encoded runs and block metadata into a custom FIFO buffer for that channel, which uses both dual-port BRAM and registers to store these values for use by the packer module. Metadata is stored in register buffers, and run data is stored in BRAM.

This buffer is needed to accumulate the backlog of encoded blocks that builds up as the packer prepares packets for all three channels (Y, Cr, and Cb) to be sent over a single transmission line. Therefore, the encoding buffer is parametrized by `BUFFER_DEPTH`, which determines the number of BRAMs we use and the size of the register buffers. This is a function of the ratio of the rate that data comes into the encoding buffer from the entropy encoding module and the rate that the packer can pack and send data from each channel. Therefore, `BUFFER_DEPTH` is different for the luminance and chroma channels.

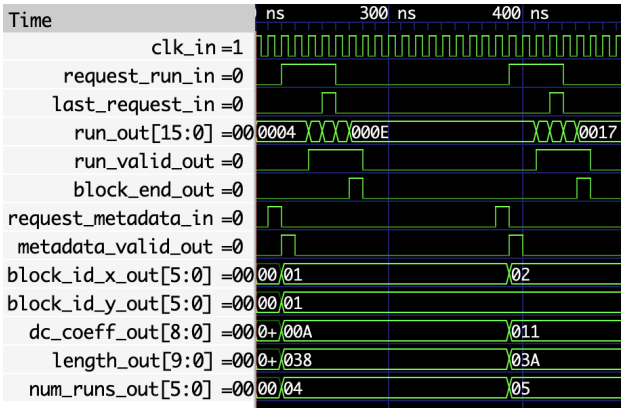


Fig. 7: Waveform illustrating the encoding buffer's communication interface with the packer. (For simplicity, does not show value-in signals from the entropy encoder.)

The packer interfaces with the encoding buffer through a request system. Each encoding buffer can accept two types of requests from the packer: `request_metadata` and `request_run`. The encoding buffer is a FIFO buffer by nature, so it handles this request by sending the data that has been in the encoding buffer longest.

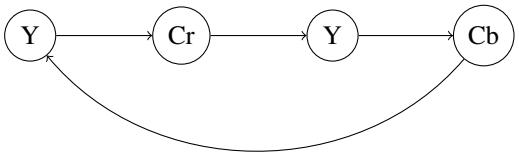
IV. Transmission

A. Channel Merging with a Round-Robin FSM (Dakota)

Up to this point, we process each color channel within its own series of modules. Because the packer outputs a single data stream, before we can give data to the packer to wrap into packets, we must combine these three channel streams together. To do this, we use a round-robin finite state machine in our top-level module that cycles through the channels and sends requests to fetch a complete block from one channel's

encoding buffer before moving to the next channel in the sequence.

Because we use 4:2:0 subsampling for the chroma channels, for every four Y blocks, we only have one Cr block and one Cb block. Or rather, for every 2 Y blocks, we have one chroma block. Therefore, our packing process is driven by the following state machine:



At each stage, we first send a request to the current channel's encoding buffer to get the metadata for the next block. Once we have the metadata, which includes the number of runs (`num_runs`) up to the final Huffman end token, we send one run request per cycle to that encoding buffer until we've made `num_runs` requests. One the final request, we send a single-cycle `block_end` signal and advance our state machine to the next state to repeat the process. With a multiplexer, we route the current channel's encoding buffer's output signals into the packer.

B. Packer (Dakota)

The packer takes in blocks, comprising of metadata followed by multiple cycles of run data and formats them correctly according to our packet format. The packer writes in-progress packets into a FIFO buffer workspace, and then ultimately outputs a single stream of bytes (or bits, depending on which transmission regime we are operating within). One byte is outputted every cycle, and the size of the FIFO workspace is a function of whether we output bits or bytes.

As we pack each bit of a packet, we also compute a parity bit for the variable-length data. The parity bit, which is followed by the stop bit, gets added to the end of the packet on the cycle that the packer receives the `block_end` signal.

C. Packet Format (Dakota)

0	CHANNEL	X	Y	LENGTH	DATA	PARITY	1

Fig. 8: Packet format for an encoded block.

Breaking this down:

1) We hold our transmission line high, so the start bit is represented with a 0.
2) `CHANNEL` is a 2-bit indicator identifying the channel of the packet's block.
3) `X` and `X` represent the location of the top-left corner of the block this packet carries. Their widths are a function of the image size, which is known by both parties prior to communication.
4) `LENGTH` is a $\log_2(64\cdot(4+(\log_2(\texttt{WIDTH}+1))+\texttt{WIDTH}))$ bit value containing the length of the encoded block data to follow. The bit length of `LENGTH` is derived from size of the maximum possible `DATA` bit-string.
5) `DATA` has length `LENGTH` and stores the encoded block data from our pipeline.
6) `PARITY` is a parity bit for redundancy and recovery. The `LENGTH` input provides a natural error-checking mechanism as well.
7) The stop bit is necessarily a 1.

D. Transmission Setup

For the version of our project where we decode using Python, we send our packets in bytes to a computer using UART to be decoded and reconstructed with a script.

Although we didn't end up having time to complete the FPGA-to-FPGA encoding/decoding transmission scheme, the design for this version of the project was to connect the packer's serial output channel to a Differential Signaling Output Buffer primitive (`OBUFDS`). The output of the `OBUFDS` instance is the final result of our data capture and encoding pipeline and would be sent to the paired device to be decoded and displayed.

V. Decoding (Abdulaziz)

We planned for two types of decoding: 1) sending our data to a computer to be decoded and displayed using a Python script and 2) sending our data over differential wire to another FPGA to be decoded on hardware and displayed via HDMI. We completed the first decoding scheme and half of the modules required for the second.

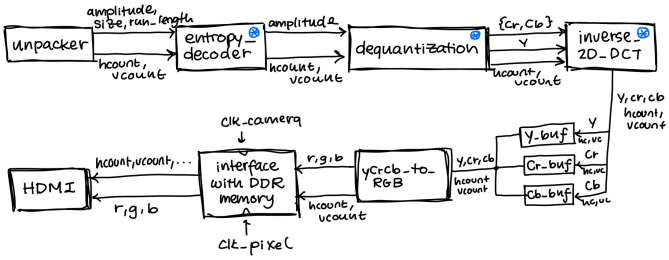
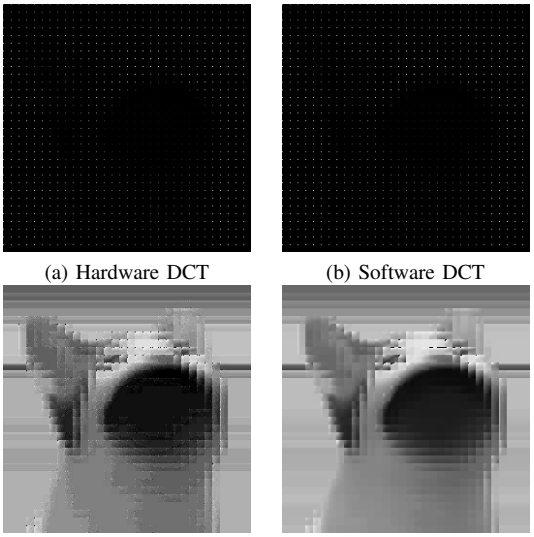


Fig. 9: Decoding pipeline block diagram. The blue ⊛ indicates that there exists one instance of this module for each channel.

The decoding stage is implemented very similarly to the encoding stage. In particular, The inverse Discrete Cosine Transform utilizes a very similar module, but with a different precomputed matrix (Type-III DCT is implemented in the iDCT module). However, some major differences include the dequantization module being implemented as a trivial multiplication step, and the lack of a Line Buffer, and in its place a FIFO is used to combine the three channels together before conversion into RGB colorspace. After this step, the reconstructed pixel, with its location, is handed into the familiar DDR-HDMI pipeline for display.

$$X_k = \tfrac{1}{2} x_0 + \sum_{n=1}^{N-1} x_n \cos\left[\tfrac{\pi}{N} \left(k + \tfrac{1}{2}\right) n \right] \qquad \text{for } k = 0, \ \ldots \ N-1 \, .$$

Fig. 10: Type-III DCT, or the iDCT, N=8. [3]



(a) Hardware DCT (b) Software DCT

(c) Hardware Reconstruct (d) Software Reconstruct

Fig. 11: A comparison of hardware and software implementations of DCT and Inverse DCT

VI. Evaluation

A. Color & frequency conversion

The Chroma Conversion to DCT part of `top_level` was implemented as one testbench, utilizing Cocotb to feed in pixel data from a few test images, data collected through the module's outputs were reconstructed again via Pillow, and some examples for luminance reconstruction are shown in FIGURE. Overall, the simulated hardware approach produces similar results to applying the DCT using scipy. However, some artifacts exist including some quantization due to using 4 bits to represent the channels in the current implementation (versus all 8 bits with the software approach). Additionally, both reconstructions suffer from noticeable quantization, as the software-level IDCT was operated directly on the 8-bit luminance values generated from Pillow images. Nevertheless, the simulation results are as expected, and the modules match the software approach.

VII. Retrospective

We learned a lot from working on this project — not only about hardware acceleration and how compression schemes work at an object-level but also how to approach a large design project, good heuristics for iterative development, and communication strategies for collaborating on an integrated pipeline. Some reflections on the project and specific lessons we learned include:

1) It's important to get a *very* basic version **working** before adding complexity. We initially had two development cycles planned: 1) develop our encoding pipeline on hardware and use UART and Python to decode 2) implement hardware decoding and transmit data between two FPGAs via a differential connection. Step 1) turned out to be a lot harder than we thought, but that was largely because we were trying to compress everything optimally the first time around. This more difficult task took longer to develop (delaying the amount of time we had working on hardware) and made things more difficult to debug. Further, especially on collaborative projects, integration often reveals necessary design adaptations, and the sooner you can address those, the better.
2) Beware the sunk cost fallacy, and don't fall into technical debt. When you have a project that depends heavily on an established protocol between modules, it's very tempting to not want to make a change that affects tested and working modules, even if that change will ultimately improve the entire system or is necessary for making another module work better. You may likely spend more time trying to make a module work under the constraint of not changing other modules than you would just updating the entire system. If you are really struggling to make a module work within the constraints of your overall design, maybe there's a problem with the design itself.

References

[1] J. Griffin, "The Ultimate Guide to JPEG Including JPEG Compression & Encoding," The Webmaster, Jan. 4, 2023. Available: https://www.thewebmaster.com/jpeg-definitive-guide/. [Accessed: Oct. 30, 2024].

[2] "Chroma Subsampling." Wikipedia, Wikimedia Foundation, 24 Sept. 2024. Available: https://en.wikipedia.org/wiki/Chroma subsampling

[3] "Discrete Cosine Transform." Wikipedia, Wikimedia Foundation, 14 Nov. 2024. Available: https://en.wikipedia.org/wiki/Discrete cosine transform

[4] libjpeg-turbo contributors, "jcparam.c - libjpeg-turbo." Available: https://github.com/libjpeg-turbo/ijg/blob/main/jcparam.c. [Accessed: Nov. 10, 2024].

[5] [redacted]