Medium Distance Livestream with Shape Detection Final Report

1st Robin Mia Tian
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, MA, USA
miatian@mit.edu

2nd Brady Sullivan
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
bsully@mit.edu

***Abstract*—We present a design that livestreams a video feed over distance and performs shape detections on the video feed. We utilize a hardware networking offload engine to both send and receive live video feed from one FPGA to another over ethernet. The receiving FPGA implements image processing in hardware to identify shapes in the imagery and display detections. The computer vision component takes a bitstream of the video of a whiteboard with rectangle(s) drawn on and uses a pre-trained neural network to output the video with detections of said rectangle(s) overlayed on top.**

Our goal is to reliably deliver a streamed and computer vision processed video feed in as close to real-time as possible.

***Index Terms*—FPGA, Ethernet, Neural Network, Convolution**

I. Background

Implementing computer vision via hardware allows us to take advantage of the parallelism inherent in neural networks. Hardware implementation is also more storage efficient, so it is useful for applications with minimal storage such as satellites. This project explores implementing a very simple neural network in hardware.

II. Networking Offload Engine (Brady)

A. Link Layer - Physical Layer

The Nexys 4 Ethernet PHY has been implemented according to IEEE802.3 ”Fast Ethernet” [1]. While the IEEE802.3 standard is rated for full-duplex operation, our project isolates the encapsulation and de-encapsulation engines on separate FPGAs. It is thus unnecessary to implement full-duplex capability on either FPGA. Per spec, we implement a CRC-BZIP32 checksum following the transmission of data data to enforce reliable data transmission.

B. Local Area Network (LAN) - Media Access Control

We have implemented MAC to enforce data access over LAN. Both FPGA’s have been assigned MAC addresses for the transmitting FPGA to send data to the receiving FPGA.

III. Image Transmission (Brady)

A. Disassembly

We transport ethernet packets over LAN. This is enabled through our image disassembly module. On the sending-side FPGA, as video frames are captured by the camera and stored

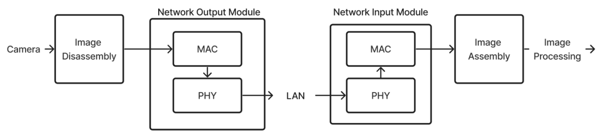

Fig. 1. LAN Offload Engine Architecture - The design uses AXI protocol between Image Disassembly and Network Output (modules and submodules) as well as between Network Input and Image Assembly (modules and submodules).

into onboard storage, this module loops over the storage and then sends pixel location and color data to the network engine for encapsulation. We are currently sending data at 320x240 image resolution with pixel colors in RGB565 color space. The module sends one of the 320 horizontal lines per packet meaning each packet includes 240 pixels of color data and a vertical line indicator. Each pixel is 16 bits wide with 240 pixels per line, including 16 bits per packet for the line information. Each packet has the following structure:

`v[m]`	`h[0]`	`h[1]`	`...`	`h[239]`

- `v[m]`: The vertical position of the pixel where m increments by one between packets. This value goes from 0 and 320. This value is zero-extended to 16 bits for two complete bytes.
- `h[n]`: The horizontal position of the pixel where n increments from 0 to 240 within each packet. Pixel values are in RGB565 colorspace format for a total of 16 bits per position.

B. Assembly

On the receiving-side FPGA, as image data packets are received by the network engine, a separate image assembly module reads off the line data. At first, the data is held in temporary register storage. This is due to the ethernet checksum requirement in order to validate the packet data. If invalid, the packet data is dropped and the module moves on to receive the next off the wire. If validated, the module writes 16-bit pixel data every two cycles until all 240 pixels from the line (packet) are in memory.

The design choice to include this onboard memory rather than pipeline packets to image processing module serves to ensure reliable data delivery. If a packet were to be corrupted during transmission or the checksum otherwise invalidated a packet, an entire line of data is lost. However, the image processing module reads directly from the onboard memory in it's own clock domain. Assuming a continuous stream of data, the same line from the previous frame entered the memory 0.013 seconds before which is likely hardly distinguishable to the human eye.

IV. Image Processing (Mia)

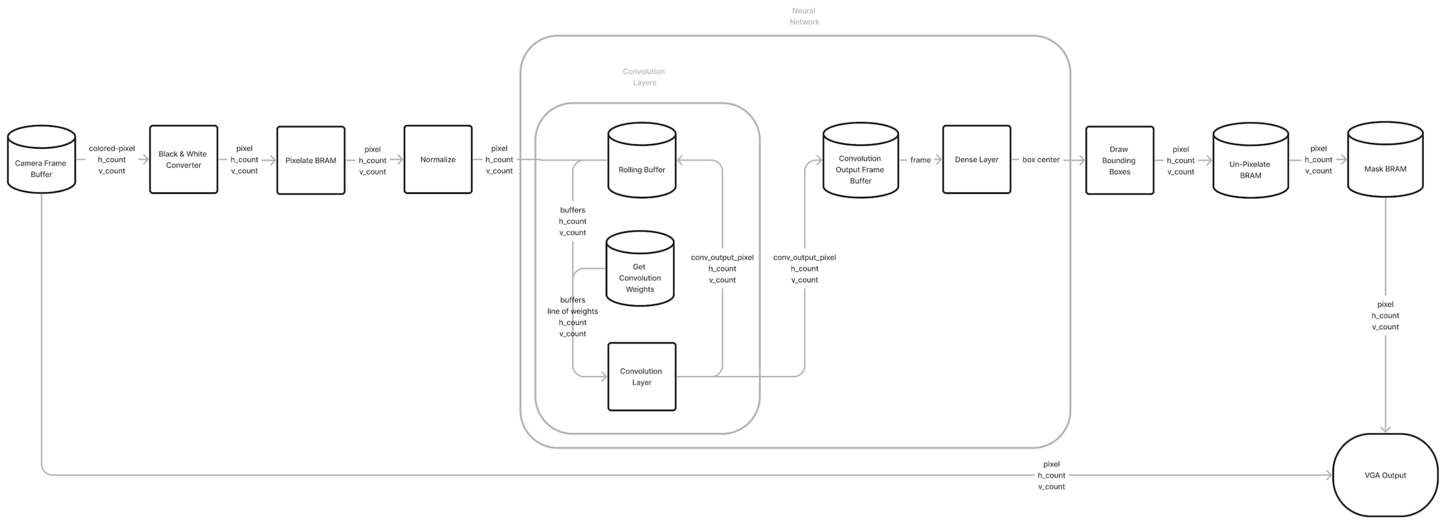

Fig. 2. Image Processing Architecture

The image processing component takes a bitstream of the video of a whiteboard with rectangle(s) drawn on and output the video with detections of said rectangle(s).

At a high level, the image processing architecture is broken into 3 working parts– Pre-Neural Network Image Processing, Neural Network, and Display and VGA.

In Pre-Neural Network Image Processing, we convert all pixels to black and weight, pixelate the 240x320 frame to a 24x32 frame, and then normalize the pixels to prepare them for the neural network.

In the Neural Network, we have two convolution layers as well as a dense layer to output the predicted center of the rectangle. The convolution layers use a filter to scan over the frame, identifying key features of the rectangle, and the dense layer will output the hcount and vcount values of the predicted center of the shape.

The output of the neural network is used to draw bounding boxes, create a mask to overlay over the existing video stream, and display on an external monitor.

A. *Pre-Trained Neural Network*

First, the Neural Network was implemented via Python to obtain the weights that we used in hardware. I used matplotlib to generate 500,000 images of black rectangles on white backgrounds. Then I trained a model to detect these rectangles.

We were conscious of the difficulties of implementing this network via hardware, so we made choices to simplify the model as much as possible without sacrificing too much of the accuracy. We chose to use black and white images as opposed to colored because in order to multiply a pixel by a weight, we only have to conduct one multiplication operation as opposed to three– conserves DSP slices. We began with at least 32 convolution layers (yielded very accurate results), but we reduced to 2 convolution layers to decrease the amount of weights to store and multiply on the FPGA.

Originally, we had planned to train the network on a 240x320 images, but we were limited by the memory of my MIT loaner laptop. We decided on training the model on 24x32 images. The convolution layers take up significantly less memory to train on the smaller images.

The model is as follows:

- one 9x9 convolution layer (9*9 + 1 bias = 82 weights)
- one 7x7 convolution layer (7*7 + 1 bias = 50 weights)
- dense layer (32*24*2 outputs + 2 = 1634 weights)

Training this model with 25 epochs has yielded successful results. My performance metric is Mean Euclidean Distance Squared. Using this metric on a 24x32 image has yielded a Mean Euclidean Distance Squared of 1.53.

B. *Pre-Neural Network Image Processing*

Pre-Neural Network Image Processing contains the following modules:

- Black and White Converter (latency: 1 cycle per pixel, throughput: 1 cycle per pixel). The input video is a whiteboard with black rectangles on it. We convert all white, off-white, and light gray pixels to white and all other pixels to black (represented by one bit).
- Pixelater (latency: 100 cycles per pixel, throughput: 100 cycles per pixel). Per the above, the neural network runs on a 24x32 grid. We convert the 240x320 frame to a 24x32 frame by outputting the center pixel of every 10x10 on the grid. Consideration was made to use rolling buffers to find averages of each 10x10, but we deemed that choosing the center pixel is accurate enough for the neural network and saves BRAM storage.
- Normalizer (latency: 1 cycle per pixel, throughput: 1 cycle per pixel). Inputs to the neural network must have mean 0 and standard deviation 1. We decided to use fixed point math– the output of this module are the pixels normalized and then shifted 17 bits left.

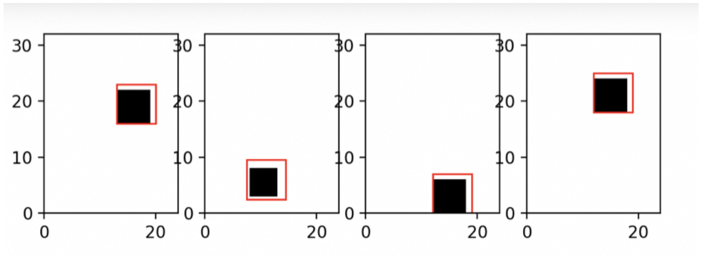

Fig. 3. Samples used to train the model

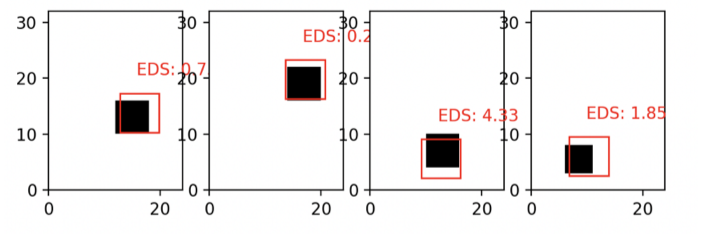

Fig. 4. Results of using the model on test images

C. *Convolution Layers*

There are two convolution layers. The first outputs 24x32 frame of the input convolved with a 9x9 kernel. The second outputs a 24x32 frame of the output of the first convolution layer convolved with a 7x7 kernel. Each convolution contains the following modules:

- Rolling Buffer (latency: 9 cycles per line, throughput: 1 cycle per pixel). We implemented 10 rolling buffers, each storing a row of pixels. As we are writing to the 10th buffer (10th row), we are outputting a 9x9 of the buffer from the previous 9 buffers (1-9 rows).
- Get Convolution Weights (latency: 1 cycle per line, throughput: 1 cycle per line). We processed the convolution weights from the pre-trained network into a .mem file where each line stores one row of weights (9 weights). Python processing was done to convert the pre-trained network weights to binary, sign them using two's

complements, and pack them together into a line. The get-conv9-weights module outputs one row of weights.

- Convolution (latency: 27 cycles per frame, throughput: 27 cycles per frame). Inputs to the neural network must have mean 0 and standard deviation 1. We decided to use fixed point math– the output of this module are the pixels normalized and then shifted 17 bits left.

D. Convolution Output Frame Buffer and Dense Layer

The Convolution Output Frame Buffer stores the output from the 7x7 convolution above into a frame buffer. This framebuffer includes signposting to indicate when an entire frame is in the buffer.

The Dense Layer (latency: 768 cycles per frame, throughput: 768 cycles per frame) takes in pixels from the frame buffer and multiplies it by weights stored in a .mem file (same implementation as weight storage for the convolution weights). There are two sets of weights for the dense layer– one for the predicted hcount of the rectangle center, one for the predicted vcount of the rectangle center.

E. Post Neural Network Processing

Post Neural Network Processing contains the following modules:

- Draw Boxes (latency: 24 cycles per pixel, throughput: 1 cycle per pixel). We take the predicted center of the rectangle and output the hcount and vcount of 24 pixels that serve as the lines of the rectangle. We also output the color (red) of those pixels in rgb565.
- Unpixelate (latency: 1 cycles per pixel, throughput: 1 cycle per pixel). We take the hcount and vcount of the red pixels in a 24x32 frame and output the corresponding hcount and vcount in a 240x320 frame.
- Display through VGA.

V. Evaluation

A. Networking Offload Engine & Image Transmission

The image disassembly and networking output modules are pipelined to provided as high of a data rate as possible. Each packet contains an image line which contain 240 16-bit color pixels and one 16-bit vertical location resulting in 482 bytes of data per packet. With the preamble (7 bytes), sfd(1 byte), destination and source MAC addresses (12 bytes), ethertype (2 bytes), frame check sequence (4 bytes), and interpacket gap (4.5 bytes) included, that results in a total of 508.5 bytes per packet. This means each packet has a $\frac{482}{504}$ data ratio or is 95.6% full of data.

With the ethernet clock operating at 50 Mhz or 20 ns per cycle, this means one packet is transmitted about every 0.00004 s. Multiplying by 320 packets (lines) to transmit the entire frame, this gives us 0.013 seconds per frame or the inverse of 76.819 frames per second.

There is room for improvement in both data rate and data density. Fast ethernet frames have a maximum transmission unit of 1500 bytes. A potentially quick improvement may be to transmit two or three lines of data per packet which would result in data ratios of about $\frac{962}{984}$ (97.7%) and $\frac{1442}{1464}$ (98.5%) useable pixel data to total transmitted data. Because our packets are twice or three times as dense, the number of packets to transmit is decreased two or threefold and thus frames per seconds increases to 153.6 and 230.5 respectively.

As far as delay from pixel capture to display, since the data is transmitted linearly, we are concerned with the average time for a pixel to be taken from the onboard storage of the transmitting FPGA, to when it is stored in the onboard memory of the receiving FPGA. Data is continuously streamed on the transmitting FPGA with approximately 92 registers or 1.84 microseconds between the onboard storage and when data goes off the wire. This delay is used to add the ethernet header, to flip the bit order, and to send the preamble.

On the receiving side, a pixel experiences approximately 4 registers or 80 nanoseconds of delay before reaching a large temporary register storage. This storage is used as the packet must fully transmit in order for the FCS to be calculated. At this point, if valid, the data streams off linearly into the on board memory at 2 cycles per pixel. At this bottleneck, the average pixel must wait for on average half of the other pixels to be loaded into the memory which results in a delay of 2.48 microseconds from receiving to storage.

Another unknown delay is the bottleneck that may result is when we transmit data over the wire. Because the current design utilizes point-to-point transmission with a single ethernet cable, we can assume this delay is negligible.

Overall, the delay from pixel capture to pixel storage between the two FPGAs has a floor of 4.32 microseconds. Reducing this delay would be tough as upstream modules are already triggered to seamlessly transmit data. Parallelizing data transmission, perhaps to 4 bit ethernet could cut the delay in half when the signal is on our hardware as a 4-bit data bus could replace the 2-bit bus used throughout the design.

B. Image Processing

All of the image processing modules run on the 65MHz clock. The above design runs at a throughput of 768 cycles per pixel = 58,982,400 cycles per frame = .9 seconds per frame. The background video still has very low throughput, but the overlay updates very slow. This does not meet our goal because our detections would update noticably slow to the human eye. We can improve this by decreasing the throughput of the dense layer– conducting more multiplication operations per clock cycle.

The image processing design runs at a latency of 932 cycles per pixel = 77,732 cycles per frame = .001 seconds per frame. This does meet our goal because the detections are performed at very close to realtime– the human eye would not notice the lag.

The image processing design uses 1,379 Kbits of BRAM (out of the alotted 4,860 Kbits). The BRAM storage was taken into account when creating the pre-trained network. Pixelating the frame also saved a significant amount of memory. We were able to use more bits to denote each pixel– we chose to shift the weights and inputs by 17 bits when doing the fixed point multiplication so as to not lose precision.

The image processing design could be adapted to be used in any convolutional neural network.

We did not reach our goals. We were able to rigorously test the pre-neural network processing as well as all modules in the convolution layer. The dense layer and the post-neural network processing are at varying levels of tested due to time limitations and poor planning.

VI. Retrospective

- Ethernet is incredibly difficult to implement and has very particular timing specifications.
- Floating point numbers are unwieldly in hardware.
- We should consider sticking to the schedule.

VII. Team Repository

https://github.com/bmsully/peoplewatching

References

[1] IEEE 802.3u Fast Ethernet Specification. IEEE802.3u, June 1995.