# Convolutional Neural Network Hardware Accelerator

1st Youry Moise
*Department of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, MA, USA
yourym@mit.edu

2nd Jake Li
*Department of Electrical Engineering and Computer Science*
*Massachusetts Institute of Technology*
Cambridge, MA, USA
jakel238@mit.edu

***Abstract*—We plan to use our FPGA to design a hardware accelerator that is capable of running images from the MNIST dataset through a convolutional neural network (CNN) algorithm. Doing this in hardware has the potential to significantly speed up predictions, as computationally demanding operations such as convolutions and matrix multiplications do not need to be done in software. Using an FPGA specifically offers benefits in terms of flexibility, as we can change our architecture relatively easily to implement a variety of machine learning models. Images can come in through either a subsampled camera feed or a computer connected over SPI, and the output of the network is displayed on our seven segment counter and the HDMI monitor. We have defined our commitment to be sending an image to the FPGA from a laptop and running it through our pipeline. Our goal is to interpret camera frames as input images. Our reach goals include implementing backpropagation to train our model.**

## I. Architecture (Youry)

Images enter our network, described in figures 1 and 2, through one of two methods: they can be sent as frames from a camera, which will be downsampled and converted to grayscale, or from a computer over SPI through a Teensy 4.0. The network itself consists of one 5x5 convolutions followed by a 2x2 max pool layer. We take these outputs to a Global Average Pooling (GAP) layer, which calculates the average of the pixels in the image output by max pool. Our label selector makes a prediction based on the GAP output and displays it on a seven segment counter and an HDMI monitor. To meet our stretch goal, we would have to add a module to calculate the loss and feed its output to our convolution layers to update their weights using backpropagation.

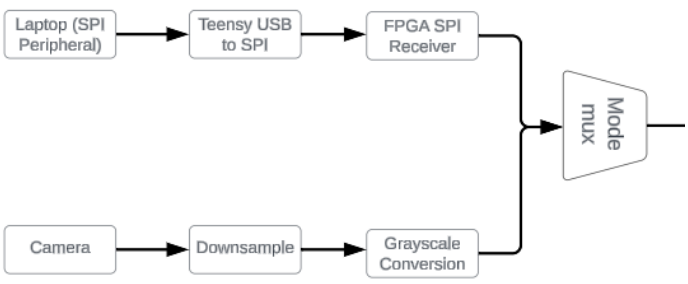


Fig. 1. High Level Architecture - Input

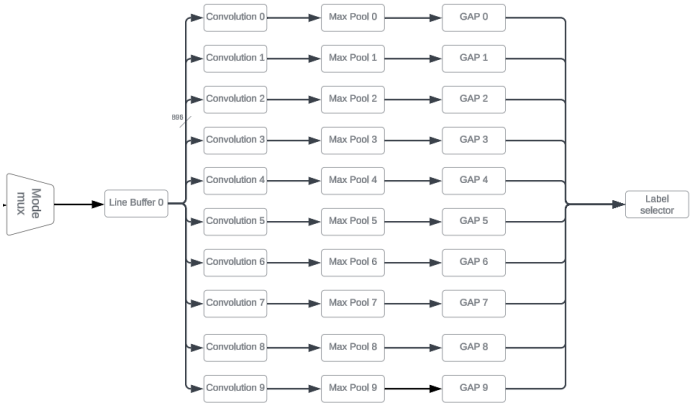


Fig. 2. High Level Architecture - CNN

## II. Checkoff List (Youry)

### A. Original

Commitment:

Our commitment originally involved sending data from a computer to the FPGA via a Teensy 4.0, running an image through layers of convolutions with predetermined weights, max pooling, and global average pooling, calculating running loss, displaying predicted labels on the seven segment counter, and being able to switch between ”training”, ”testing”, and ”user” modes.

Goal:

Our goal was to update the weights in the convolution through backpropagation, communicate bidirectionally between the FPGA and our computers, run camera frames through the network, display the predicted label on an HDMI monitor, and identify some minimum threshold for making a prediction on input images.

Stretch Goal:

As a stretch goal, we aimed to incorporate images from the CIFAR10 dataset, which consists of images of objects such as airplanes, birds, and trucks. We also wanted to implement a more complex architecture, use softmax and negative log likelihood loss for training, and increase the rate of data transfer to the FPGA, as that was the main bottleneck for our latency and throughput.

### B. Revised

We reconsidered our goals and architecture after receiving feedback from the course staff and running tests to determine how feasible our initial plans were.

Regarding architecture, we originally planned to have three convolution layers (two 5x5 and one 2x2), but Vivado often crashed during the placement stage. The original model was quite resource intensive, as it had 30 instances of convolution_mnist, thirty of max_pool, 10 of gap, 21 line_buffer5's, 2 filters (blur and sharpen), 1 label_selector, 1 downsample, 1 SPI transceiver, and 1 digit_sprite. This was all on top of the existing camera and HDMI video pipeline that we had already developed from previous labs. We tried to simplify our top level by removing everything involving the camera and HDMI pipelines, but even with just SPI and two layers, our builds tended to freeze. We also tended to run out of slices on the FPGA, requesting 2292 when we only had 2190 remaining. As a result, we decided to target an architecture with just one convolution and max pool layer. We discuss the resource utilization and congestion in more depth during the Evaluation section.

Commitment:

Our commitment remains largely the same, except there is no requirement for switching between training, testing, and user modes.

Goal:

For our goal, we decided only to target running camera frames through our network, and not pursuing backpropagation or identifying the minimum threshold at which the FPGA can make a prediction.

Stretch:

Because of the high resource utilization, we decided to set backpropagation and training as stretch goals, as adding these features would add more complexity that would require significant optimizations or reductions to the overall design.

## III. Goals (Jake)

We were initially targeting an accuracy of 70%, as a Pytorch model with what we thought was the same architecture achieved close to 90%. Later on, we realized that the model we were using had the output of each convolution in a layer connected to the input of every convolution in the next layer. Our plan, however, was to have each output connected to one input as described in the architecture section. Once we edited the Pytorch model to reflect this, the accuracy was closer to 50%.

To avoid having to work with floating point values, we decided to test how integer weights affected our accuracy. We took the final weights set by our Pytorch model, multiplied them by 100, and casted them to integers (the multiplication was necessary because many of the weights were small decimals). We ran our Pytorch model with these new weights and achieved an accuracy of 40% at best. As a result of this testing, we hoped to get an accuracy of at least 30%. After changing our architecture to only have one convolution and one max pool, our Pytorch model had an accuracy of about 25%-30%.

Regarding out latency goals, we planned to extend our SPI receiver to work with 20 MHz SPI. We need to send $28*28*8 = 6272$ bits to the FPGA to represent a full image. At a frequency of 20 MHz, it will take $\frac{6272}{20*10^6} = 0.0003$ seconds to send an image. It takes on the order of tens of cycles for one pixel to propagate through our pipeline, which adds a few millionths of a second of delay, so our final latency is still about 0.0003 seconds.

To meet our reach goal and successfully train on 60,000 images, we would need $0.0003*60000 = 18$ seconds of delay.

## IV. SPI (Jake)

A majority of the training time needed for our model comes from transferring the MNIST image data from a computer to our FPGA. For simplicity and robustness, we decided to use a Teensy 4.0 to serve as a peripheral that takes in USB data from the computer and outputs SPI data, with the controller being our FPGA.

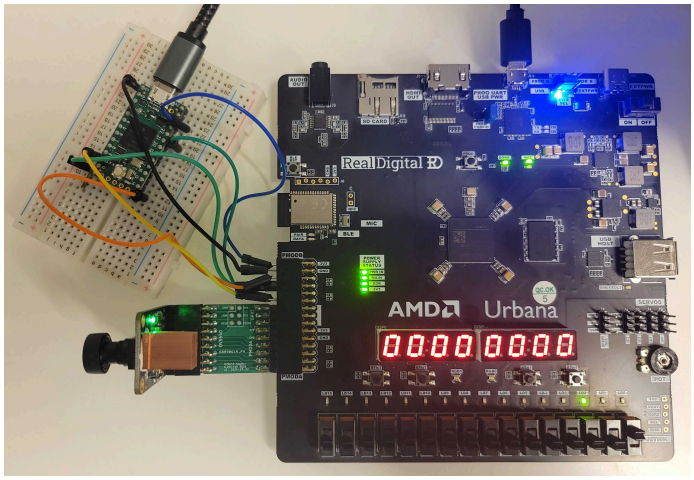


Fig. 3. SPI Setup with Camera

### A. Teensy 4.0 to FPGA

We started with lab 2's SPI controller and created a custom SPI Peripheral configuration on the Teensy as it is not natively supported and would allow us to better utilize the higher USB communication speeds. We created the SPI peripheral using the digitalFastRead() and digitalFastWrite() functions to quickly bit-bang our pixel values as we read the serial clock output from the FPGA. We set the trigger on the SPI controller to a PMODB input set by a digitalFastWrite() on the Teensy in order to automate this communication. The trigger is set right before the Teensy begins reading the FGPA DCLK.

Once the Teensy receives data from the laptop we set pin 21 on the Teensy to HIGH, then LOW after a 10 ns delay. We also shifted the pins in the top_level.xdc to allow for simultaneous connection of the camera and Teensy, with Teensy taking up the PMODB side and the camera remaining in PMODA.

### B. Laptop to Teensy 4.0

To more easily integrate MNIST image data, we created a python function and used pyserial to send 28x28 pixel image to the Teensy. On the Teensy side, this requires storing each of these values received from USB serial in a byte buffer of size 784. This implementation was used to reduce propagation delay between the laptop, Teensy, and FPGA. This way the Teensy only needs to refer to its own memory to write each pixel instead of waiting for the laptop to send another pixel.

To test the entire communication protocol, we edited the lab 2 top_level.sv to display each message it receives on the 7 segment display. This allowed us to verify that our FPGA was receiving the proper data and that our protocols were implemented correctly. The testing included the following functions:

**custom_pixel():** accepts integers 0 to 255 as a python input(). Sends a single pixel at a time.

**custom_image():** accepts an list of lists that represents a 28x28 mnist image. The python script has a few sample images to pass into this function.

**send_images():** sends multiple images and accepts a list of images.

Each of these pixels are turned into binary using bytearray() and sent across USB serial. This was displayed on the 7 segment display in hexadecimal.

### C. SPI Timing

With a period of 8 cycles of the 74.25 Mhz camera clock, we are able to transmit pixel data at 9.3 Mhz SPI. For 60,000 images, this would take 38 seconds to load all the images for training.

## V. Camera Pipeline (Youry)

### A. Downsample

For running camera frames through our accelerator, we will have to downsample the original outputs of the pixel reconstructor from 320x180 to 28x28. We initially attempted to do so by extending the method we used in class, simply taking every 45th pixel in every 25th row (previously using a camera outputing 1280x720), but as this is an extremely aggressive downsample, the output image was full aliases and looked almost nothing like the original.

Our approach to resolving this involved blurring and attempting to reduce the size in stages rather than using one large downsample. When we did this, however, our final image was usually full of green pixels and resembled colorful static by the time it got to 28x28, which we fixed by adding sharpen filters. We experimented with various combinations and amounts of blur and sharpen filters, but blurring, downsampling directly to 28x28, and sharpening seemed to work the best. When we used this strategy, we achieved these results for a 32x30 picture (these dimensions were chosen because they evenly divide 1280x720). We extended this to 28x28 and achieved the results in figure 5.

The most immediately noticeable issue is the fact that the black and white lines are disjoint in several places, making the picture look as if the lines are going from the bottom left to the top right instead of top left to the bottom right. There are also many artifacts on the right side of the screen, likely due to pipelining issues, as by the time the HDMI pipeline has received a valid pixel to display, the HDMI hcount and vcount values have already gone through several cycles, which also causes the image displayed to be far from the left edge of the screen.

Fig. 4. Original Image

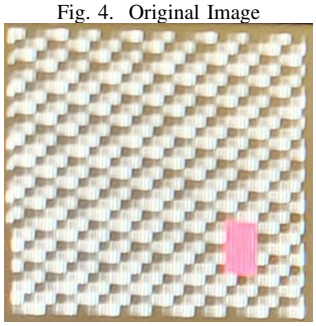

Fig. 5. Original downsample with extreme aliasing

Fig. 6. Downsample to 32x30

### B. Color to Grayscale

Finally, before putting the images through our pipeline, we had to convert them from 16 bit color to 8 bit grayscale. This is typically done using the formula 0.299*Red + 0.587*Green + 0.114*Blue. To reduce the complexity of this calculation, we approximated it to be 0.25 * Red + 0.5 * Green + 0.125 * Blue, or Red $>>$ 2 + Green $>>$ 1 + Blue $>>$ 3, which is a standard approximation for real-time systems. This approximation is unnoticeable to the human eye, but may cause issues with our calculations. If we find that they do during testing, another commonly used method involves approximating 0.299, 0.587, and 0.114 as 77, 150, and 29 divided by 256, which would just be a multiply and a right shift by 8 for each channel, which should not be too intense for our FPGA.

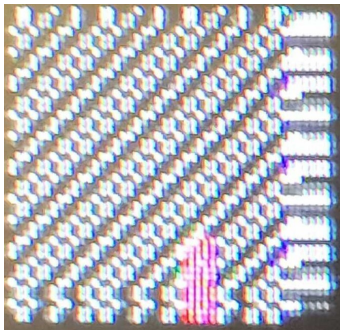

Fig. 7. Downsample to 28x28

## VI. Neural Network Pipeline (Jake)

### A. Convolutions

To create our convolution module, we started with the lab 7 version and adapted it to handle different sized data inputs as well as different kernel sizes to better suit our CNN architecture. The module has the new parameters including IMAGE_SIZE, DATA_SIZE, and K_SIZE as well as new inputs for weight coefficients for the kernel and the shift value. Currently we will input these values ourselves to test the accuracy of our model, but will implement the backpropagation portion to handle this in the future. For the MNIST dataset we set these parameters to 28, 8, and 5 or 2 based on the kernel size we choose. IMAGE_SIZE sets our hcount and vcount bounds, K_SIZE sets the size of our 2D array cache that takes in inputs from the line buffer, and DATA_SIZE sets the maximum value of our kernel weights, pixels, and helps compute the maximum value of our convolution output. We found that the maximum value we can expect from the convolution will be a product of the maximum value of the weights(255), pixel input(255), total number of kernel weights ($5^2$ or $2^2$) and found that adding the size of these inputs computes the size of this parameter. For a 5x5 kernel taking MNIST inputs for example, our max data size is $8 + 8 + 5 + 1 = 22$ bits. Compared to the lab 7 version of this module we also kept some of the clamping logic, lower bound of 0 and the upper bound is no longer 8 bit because we are not outputting RGB pixels. We will have to set a 16 bit upper bound, possibly as a separate output logic, if we end up implementing a visual representation of the convolution that displays onto the screen. This convolution module also utilizes a 3 stage pipeline split as: multiply weights, sum values, shift and clamp final output.

### B. Max Pool

Our max pool filter is a 2x2 kernel with a stride of 2 that starts at the top left pixel of our convolved image and returns only the pixel with the highest value, returning an output that is four times smaller than the input. To implement this, we reused the existing convolution architecture, using an array of pixels instead of BRAMs for simplicity. This has a "top row" and "bottom row", each of which is the width of the input image. At the beginning, when it receives "valid" signals, it adds as many pixels as it can to the top row. Once that happens, it starts filling in the bottom row. After every other pixel that it adds to the bottom row, it looks at the previous pixel, as well as the two corresponding pixels in the "top row", and sets the output pixel to be the maximum of those four values. Once the bottom row has been read, it switches to adding to the top row again. This process continues until it has received all the pixels. The Max Pool module also returns the new hcount and vcount of the pixel it has just processed based on the dimensions of the output, each of which should be half of the corresponding input dimension.

### C. Global Average Pooling and Label Selector

After going through our entire pipeline, the max pool layer will be returning pixels corresponding to a 12x12 image, or 144 pixels in total. We have 10 GAP layers, each connected to one max pool layer, responsible for calculating the average of each set of 144 pixels sent from the corresponding max pool module. Once 144 pixels have been received and the average has been calculated, GAP indicates that it has a valid output, which is fed into the label selector.

The label selector takes 10 values as inputs, the concatenated outputs of all GAP layers. It performs a pairwise comparison to find the index of the maximum value. The comparisons are done combinationally, and the output is returned sequentially, allowing this layer to complete in just one cycle. The label that returns with the highest value is displayed on the seven segment display and fed into the digit_sprite module.

### D. Digit Sprite

GAP's output is fed into our DigitSprite module, which is an extension of ImageSprite2. We have a sprite sheet with 10 32x32 pictures of digits 0-9, which are compressed through palettization. The images are black and white, so the palette consists only of 0xffffff and 0x8b8a8a, which were determined by using the img_to_mem program to find the two most common colors in the images. These colors are stored in a packed array instead of a BROM since we are only storing 48 bits of information. We still have an image BROM, which has a depth of 10240 (10 32x32 images) and a width of 1 (1 bit to encode the two palette addresses). DigitSprite has one new input, num_in, which it uses to determine which digit to display by setting the image BROM offset to be WIDTH*HEIGHT*num_in.

## VII. Evaluation (Youry)

### A. Status

We have reached our commitment goals, as we have successfully transmitted data from our computer to the FPGA and gotten images to travel through our pipeline end-to-end, with our final results being displayed on our seven segment counter. We have met some parts of our goal; we are able to run our model on camera frames, but have not reached the accuracy we were initially targeting. We did not get to start working on our stretch goals, as we do not have support for backpropgation, loss calculations, or the CIFAR10 dataset.

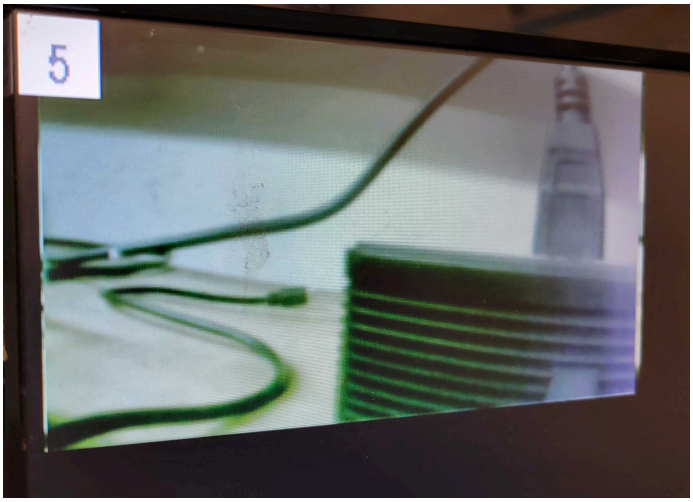


Fig. 8. Digit sprite on monitor

### B. Accuracy

| Label | Max Accuracy |
|---|---|
| 0 | 15% |
| 1 | 20% |
| 2 | 5% |
| 3 | 5% |
| 4 | 15% |
| 5 | 30% |
| 6 | 10% |
| 7 | 25% |
| 8 | 15% |
| 9 | 15% |

Our average accuracy was 15.5%, which varied depending on the digit in the picture, with 2 and 3 being our worst at 5% accuracy, and 5 being the best at 30%. During testing, we noticed that, although many digits were predicted incorrectly, they were often predicted to be digits that looked similar on paper. For example, while testing 9, half of the predictions were 4 or 6, many 4's were guessed as 9, 2's were seen as 7's, and 8's were seen as 5's. These digits are relatively similar, especially when accounting for the messy handwriting that appears in the MNIST dataset. Because of this, we believe that, if we were to add some more complexity to our model, such as by adding another layer, we could distinguish between similar digits and improve our overall accuracy.

When receiving pixels from the SPI pipeline, the accuracy of our model varies depending on the digit that is sent. It performs well with numbers such as 5, 7, and 8, achieving up to 25% accuracy, with our best run getting 40% accuracy when given 10 8's in a row. For other numbers, such as 0 and 3, our model performs poorly, getting only 10% of its predictions correct. When the correct label is 1, most predictions are incorrect, but many are 7, which is relatively close to 1. The same is true for 9, which is often predicted to be a 4.

The accuracy of our network when connected to the camera pipeline is quite low. Its predictions are quite inconsistent, and without any logic to limit the number of valid predictions, users can see the seven segment display or monitor update several times per second.

### C. Timing and Resource Utilization

A breakdown of the latency of each module in the network is as follows:

SPI - We are using 9.3 MHz SPI, so we send 1 bit every 108 nanoseconds. Each pixel is 8 bits, so this is 862 nanoseconds per pixel. The images are 28x28, so the time to send one image over SPI is 28*28*862 = 675,771 nanoseconds.

Blur and sharpen - Each of these depends on the filter module from lab 7, which has a propagation delay of three clock cycles, adding six clock cycles total. The other modules, including convolution_mnist, max_pool, GAP, and label_selector, take one clock cycle to complete. Each runs on the 74 MHz pixel clock, with 13.468 nanoseconds per clock cycle, for a total of 13.468*10 = 134.68 nanoseconds per pixel received from the SPI receiver or camera. This is the amount of time required for the final pixel to propagate to the label selector, so the latency for a full image is 675,771 + 134.68 = 675905.68 nanoseconds.

The throughput of our network depends primarily on GAP and SPI. Because GAP takes the average of 144 pixels, it only returns a valid signal once every 144 pixels. Over SPI, we receive one pixel every 862 nanoseconds, so the throughput is $\frac{1}{862*144} = 0.000008$, or 8000 predictions per second.

We were originally experiencing issues with our slack, as Vivado consistently reported a WNS around -9, almost reaching -10 at one point. This turned out to be an issue with clock domain crossing. We forgot about the cross from the 200 MHz camera clock to the 74 MHz pixel clock that we used in the labs. Because of that, we were running all our modules off the camera clock, demanding that everything, including the convolutions, produce outputs in just 5 nanoseconds, which would have been impossible to do. We reintroduced the clock domain switch, however, and we have reduced the WNS to -0.5 and the TNS to -2.1.

Our resource utilization is about 20%, which is quite high, as lab 7 only used 2.5% of the FPGA's resources. We believe this makes sense, however, as we have ten times as many convolutions, and although each convolution is simpler since it does not use a BRAM, the fact that we have ten max pools on top of that, as well as the GAP and label selector, brings us to 20% utilization.

Another, likely more pressing, issue is the congestion of the resources being used. The "West Dir" area of the chip has 86.7% congestion, which is almost double the amount of congestion we experienced in lab 7.

For our block RAM usage, we need 256 x 24 (camera settings), 10240 x 1 (digit sprite), and 4 x 320 x 16 (the line buffers for the blur), for a total of 36 kilobits of memory, which is about the size of 1 continuous block RAM. We do have the sharpen filter after the blur, which also uses BRAMs, but the sharpen works on pictures that are 28 x 28, so they only use 28 x 16 = 448 bits, which is converted to DSP.

### D. Additional Use Cases

With some tweaks, our project could apply to non convolutional neural network architectures. Without the convolutions, we would have resources for modules such as fully connected layers and various activation functions. We could also extend it to standard ML algorithms such as linear regression.

## VIII. Insights (Jake)

While working on our project, we realized we underestimated the complexity of implementing our architecture in hardware. Before switching to a one layer model, some of our builds would take over 10 minutes to complete, even without the use of external IP that had to be generated before our own modules. Starting over, we would attempt a much simpler architecture, such as a binary image classifier. The limited number of labels would have allowed for a much more complicated model with high accuracy for the binary image classes.

We would also primarily focus on the accelerator aspect of this project. From the beginning, we wanted to keep the complexity as low as possible, by not using floats or expensive logarithmic or exponential functions, for example, because we wanted to have resources left over for training and backpropagation. Had we prioritized just making an accelerator, we could have dedicated more resources and spent more time optimizing for that use case, rather than spend time trying to make a more general purpose project.

In order to do so, we would have also focused more on the SPI communication between the board and the computer. By fleshing out details such as start flags, signals indicating the number of pixels already sent, or even data packet structures, we would have been better able to keep our network aligned, making sure the predictions we were seeing corresponded to specific images and not some subset of pixels from several images. While prioritizing SPI, we probably also would have worried less about the camera pipeline. This pipeline likely requires much fewer resources than the camera and HDMI modules, which would have given us more freedom to add to our network.

When it came to the images, all of our tests were done using either SPI or the camera. The camera was not reliable because of the frequency at which the input pixels were changing, the distortion pattern, and the fact that it was recording a picture through a screen or on paper instead of having direct access to the pixels. Using SPI means we have external components, which adds several failure modes that we have to consider when debugging.

To prevent SPI and camera issues from bottlenecking other progress, we could have saved palette and image mem files for one mnist number and used the existing image sprite module to load an image directly onto the FPGA. This would have isolated the convolution pipeline from camera noise and issues with SPI communication, allowing us to test the heart of the project without worrying about the peripherals.

## IX. Author Contributions

Youry was responsible for the camera pipeline, max pool, digit sprite, gap, and label selector. Jake was responsible for the SPI pipeline, modified line buffer, and convolutions. Both authors worked on the writing for the technical reports.