

DESIGN DOCUMENT FOR FPGA NEURAL NETWORK

ACCELERATION

Stephen Ogunmwonyi (iso2107)

Aymen Ahmed Norain (aan2161)

Bradley Chen Jocelyn (bcj2124)

Connor James Espenshade (cje2136)

Table of Contents:

Introduction	2
Algorithm	3
Algorithm: CNN development and training	3
System Level Block Diagram	4
System Level Dataflow	4
Computation Module	6
Computation Block	8
Memory Resource Allocation	9
Hardware/Software Interfaces	13
Bibliography	14

Introduction

This project is focusing on building specialized architecture for neural network acceleration and will utilize the MNIST (Modified National Institute of Standards and Technology)¹ dataset as a training and testing dataset. The base aim of this project is to accelerate inference of a newly trained neural network by implementing a convolutional neural network (CNN) architecture on a field programmable gate arrays (FPGAs). FPGAs offer us the ability to tailor specific aspects of a CNN in hardware and they offer extremely low latency when processing real time applications. We also hope to be able to take advantage of pipelining to increase throughput in our designs and take advantage of parallel processing capabilities. MNIST is a publicly available dataset that we will be using to train our neural network. It consists of a series of handwritten digits and is a common benchmarking test for classification algorithms in machine learning. The dataset contains 70,000 greyscale images of digits 0 through 9 with each image being 28 x 28 pixels in size. 60,000 of these images will be used in training and 10,000 will be used for testing. The low pixel resolution and monochromatic nature of the images allows even basic models to achieve accurate results (90+ % accuracy).

A secondary aim of the project is the adjustment of certain hardware components i.e. changing the pooling architecture from max pooling to average pooling, and measuring if there are any potential performance drop offs. This would enable us to make informed comparisons about what aspects of neural networks provide the most tangible acceleration when implemented in hardware.

¹ (“MNIST database”)

Algorithm

General Practice:

1. For each layer in either model:
 - a. Read 784 bits of image, weights for layer 1, and biases for layer 1, load into HW
 - b. Process layer 1
 - c. Send outputs from layer 1 to SW
 - d. Layer = next_layer

Algorithm: CNN development and training

We carefully evaluated multiple algorithms to implement on the device. We considered implementing large general CNNs for general image classification: Resnet-18, SqueezeNet, and MobileNet. These CNNs do not require any additional training, but are large and have their own quirks including residual connections and complex interconnected webs of convolutions. As we are developing models to detect MNIST data samples, this is a much simpler problem space and does not require implementing a full CNN off-the-shelf. Instead, we intend to construct our own neural networks from Keras and TensorFlow tutorials, one leveraging convolution and one only leveraging dense layers (Chollet, “Training”). We intend to build the basis for a system that can take simple Keras models trained off device and perform inference live, allocating various hardware accelerating components to speed up different Keras layers. Therefore, convolution, pooling, padding, flattening, and dense models (matrix multiplication) should be built in hardware as much as possible.

The first model (from the TensorFlow documentation on “Training a Neural Network in MNIST with Keras) is simple, where it does not even require convolution, but only two dense

layers and one flatten layer. This model's weights and biases will be transferred from a laptop computer performing the training to the FPGA's arrays.

Python

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10)
])
```

The second model from the second tutorial is more complex, though not significantly. Building this model into our platform will require Conv2D and flatten layers, though we can ignore dropout layers that the model skips during inference. Here, we will likely need to reduce the number of filters from 32 and 64 as this configuration consumes 136.04KB with 34,826 parameters ($34,826 * 32\text{-bit float} = 34,826 * 4\text{B} = 139,304\text{B} \approx 136\text{KB}$).

Python

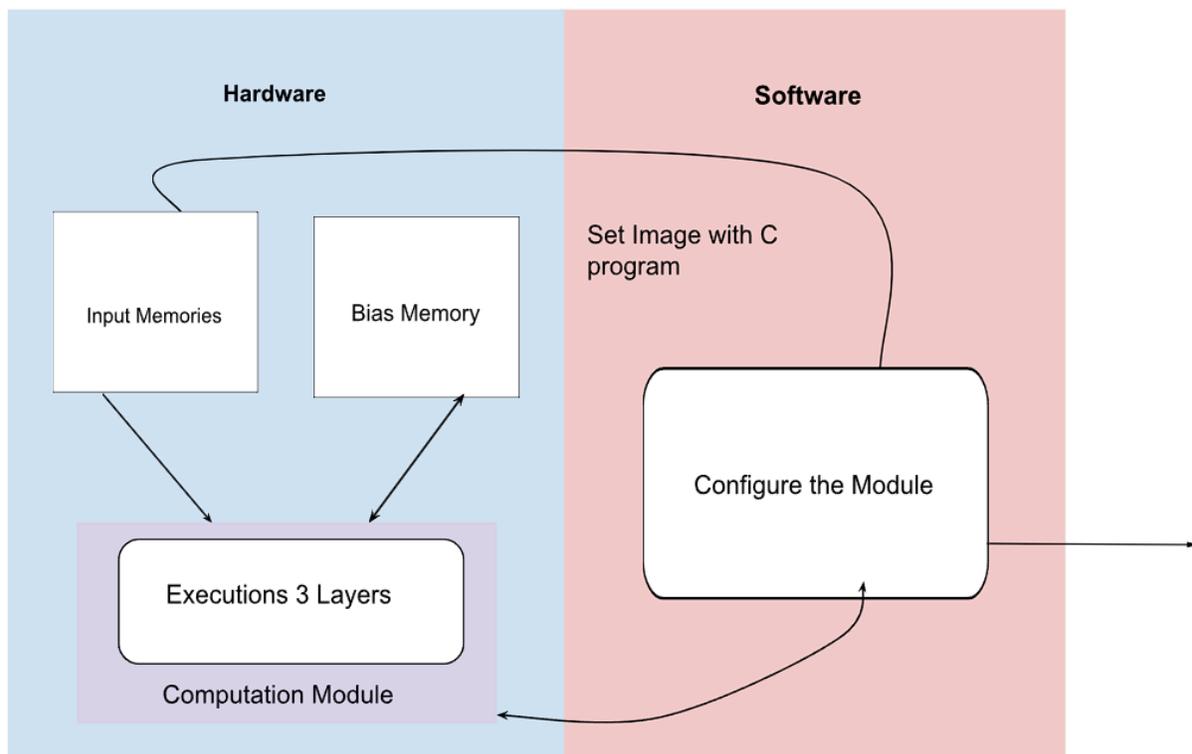
```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3),
            activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3),
            activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)
```

We will also quantize the model weights to int8 to fit on device, which alone would reduce the model size to 34.8KB. The data itself can further be quantized down from 8 bits of input with grayscale images, down to 1 bits per pixel: 0 if the original value is dark < 127 , and 1 otherwise. There are 28×28 pixels in each image, or 784 pixels total, resulting in images that are 784 bits, or 98B, each. Combined with optimizations of lower filter numbers, this model should be deployable on De1-SoC to accurately accelerate detection of numbers in the MNIST dataset.

System Level Block Diagram

System Level Dataflow

This is a high level block diagram of the system we hope to implement. As you can see in the figure below we will have a HW/SW interface on the FPGA that will enable our hardware to communicate with our control C program on the local terminal:



As you can see above the overarching system level architecture is relatively simple to implement. The hardware side of the system consists of 3 major components. There will be N image RAMs for us to store the dataset images. As the MNIST dataset consists of 28 x 28 pixel images, the RAMs will be just under half a kilobyte in size. There will also potentially be a bias memory implemented. A bias can be defined as a constant which is added to the product of features and weights. It is used to offset the result and it helps the models to shift the activation

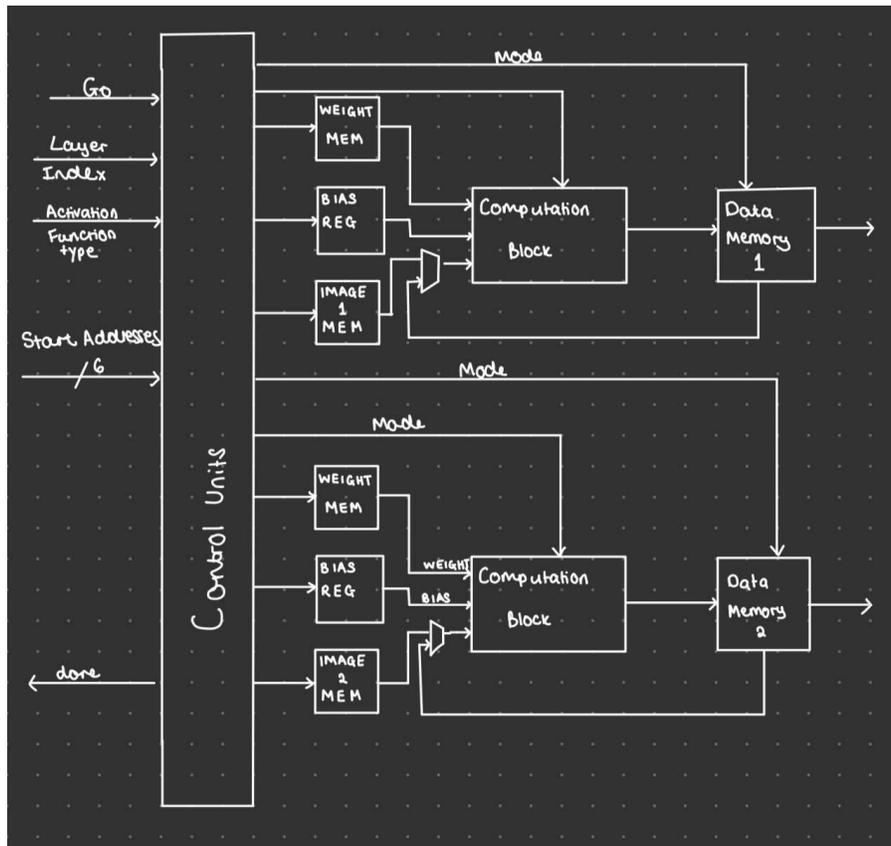
function towards the positive or negative side.² The main component of the hardware design would be the computation module. Our computation module would have the ability to perform a convolution layer, dense layer, max pooling layer, average pooling layer, spare layer and a series of activation functions. Initially, the focus would be on getting one set of these layers working but the final aim would be to have all of these working and interchangeable depending on the c file to configure the module.

The software side will have a C program that will configure the hardware side of the design depending on the neural network being implemented.

² (“Importance of Neural Network Bias and How to Add It”)

Computation Module

The computational module is the main block on the FPGA side in our NN accelerator design. In order to enable high speed inference on the FPGA, we would design a module that is controlled by the on board software in real time. Initially, we will implement a computational module that performs all necessary computations for a CNN. However we hope to expand this design to include a multitude of different types of layer computations such as dense layer, sparse layer etc. This will enable us to take a more generic neural network design and accelerate it using the hardware. We also hope to take advantage of parallelism during this process so we will have multiple pipelines running simultaneously within this module. The microarchitecture for the computation module can be seen below:



We want to be able to run inference on multiple images at a time, hence why, as you can see above we hope to implement 2 pipelines. There will be a control unit (CU) that will dictate the state of the pipelines above. This will be the part that interfaces with the C program and as soon as the registers and memory blocks are loaded with the correct information will begin to drive the pipeline.

Initially, CU supplies the address from which the Image Memory (IMEM), Weight Memory (WMEM) and the Bias Registers (BREG) should be read from. The computation blocks take these inputs and perform a multiply and accumulate on the values. After 9 cycles (9 due to a 3 x 3 convolutional window), the computation module performs the required activation function on the value and writes it to the Data Memory (DMEM). This continues until we have performed the necessary convolutions on the entire image. This would be considered the first layer in a typical neural network. Instead of using the IMEM now we would utilize a mux and select the DMEM as the new input data source. Now we would perform the new calculations with the DMEM input data, new bias value and new weights from the WMEM. This would represent the second layer in our neural network. One key design feature of the computation block would be that it is both capable of performing convolution layer computations as well as pooling layer computations. This will be occurring within the same block.

Computation Block

The computation block is where we actually perform convolutions, pooling layer calculations and implement an activation function. The microarchitecture of the computation block can be described as a rotating series of line buffers that feed into a convolution module. Effectively, each line buffer stores the value of each row. As we are using a 3 x 3 kernel, we need 3 MEM arrays for the current calculations and 1 for the next row being loaded. On each clock cycle, we can push the pixel values of the first 3 rows into our convolution kernel. At the same time we will feed the values of the next row into the 4th MEM array. When the 4th MEM array is full and the 3 MEM arrays for the current calculation have been read all the way, the following happens:

- The 4th MEM array becomes the bottom row in the convolution kernel.
- The oldest MEM array so MEM array 1, now becomes the new loading MEM array as the information in this MEM array is no longer needed.

MEM array 1	CALCULATIONS
MEM array 2	
MEM array 3	
MEM array 4	LOADING ROW 4
NEXT STATE	
MEM array 2	CALCULATIONS
MEM array 3	
MEM array 4	
MEM array 1	LOADING ROW 5

Memory Resource Allocation

Memory Resource Requirements and Module Roles

The CNN accelerator design uses six total memory modules—two each for storing image data, convolutional weights (with bias parameters), and intermediate calculation results (data). Duplicating these modules allows us to enable dual channel processing // parallel computations throughout the pipeline.

Image Memory Modules (×2):

There is one 28×28 pixel MNIST image in every module. The MNIST images are typically in 8-bit grayscale, but in our design, we use them as binary values of either 0 or 1 but use 8 bits for each pixel to maintain normal resolution as defined by the data set. Such memories store the main input data for convolution operations as well as to perform secondary operations in processing.

- Per Module:
 - MNIST image size = 28×28 pixels = 784 pixels
 - Pixel bit length = 8 bits normally, 1 bit quantized in our design
 - Memory per module = 784 pixels \times 1 bits = 784 bits (which equals 98 bytes)
- Total for 2 Modules:
 - 98 bytes/module \times 2 = 196 bytes (or 1,568 bits)

Weight Memory Modules (×2):

These modules store weights needed for a given layer, along with associated biases. For different layers these modules must be different sizes, but they store a collection of int8 weights that correspond to the product of the input and output channels for Dense layers, or their kernel shape, input channels, and filters/output shape for convolutional layers. Each output channel also has a given bias. All weights and biases are stored in int8 data types.

For simple model (“Training”):

Layer	in_channels	out_channels	Weights	Weight Size	Bias Count (1/filter)
Dense #1	784	128	100,352	100.352KB	128B
Dense #2	128	10	1,280	1.280KB	10B
Total				101.632KB	138B

For a total of 101.77KB.

For convolution model (Chollet):

Layer	Kernel Shape	in_channels	# Filters / out_channels	Weights	Weight Size	Bias Count (1/filter)
Conv2D #1	(3x3)	1	32	288	0.288KB	32B
Conv2D #2	(3x3)	32	64	18432	18.432KB	64B
Dense	–	1600	10	16000	16KB	10B
Total					34.72KB	106B

To calculate number of weights = kernel shape * in_channels * filters

- Per Module:
 - Memory per module, model 1 = 101.77KB
 - Memory per module, model 2 = 34.826KB
- Total for 2 Modules:

- (Module 1): $101.77\text{KB/module} \times 2 = 203.54 \text{ KB}$
- (Module 2): $35 \text{ KB/module} \times 2 = 70 \text{ KB}$

Data Memory Modules ($\times 2$):

Intermediate results and final outputs of CNN computations are stored in data memory modules. They offer direct access to computed values to be passed to subsequent stages of processing or to be stored for subsequent post-processing. With their dual banks of data memory, the design supports concurrent streams of processing, increasing overall throughput and efficiency of the accelerator. For memory constraints, it is important to consider these too on a per-layer basis, mainly focused on the product of the output sizes with the int8 data type for the weights. As such, for model 1:

- Flatten: 98B (784 bits, just rearranging input, so no need to multiply by sizeof(int8))
- Dense 1: 128B (1024 bits = 128 output channels * int8)
- Dense 2: 10B (80 bits bits = 10 output channels * int8)

For model 2:

- Conv2D #1(3x3x32)+MaxPooling2D (2x2): 5408B ($13 * 13 * 32 * \text{int8}$)
- Conv2D #2(3x3x64)+MaxPooling2D (2x2): 1600B ($5 * 5 * 64 * \text{int8}$)
- Flatten: 1600B (rearranging last layer)
- Dense: 10B (80 bits bits = 10 output channels * int8)

For memory constraints per module, we will define best case performance to be processing one image per system, and taking the maximum value to find the memory requirements for the worst layer. For worst case performance with an image in each layer of the system pipelined through (more below in hardware/software interface), we will sum the memory constraints for all layers as we assume all are in use by the system.

Per Module:

Model	1	2
Best Case (Maximum)	128B	5408B
Worst Case (Sum)	236B	8618B

Total for 2 Modules:

Model	1	2
Best Case (Maximum)	256B	10,816B
Worst Case (Sum)	472B	17,236B

Total Memory Considerations/Requirements:

- Image Memory = 196 bytes
- Weight + Bias Memory = 203.54KB Model 1, 70KB Model 2
- Data Memory (assuming pipelined worst case) = 0.472KB Model 1, 17.236KB Model 2
- Total Memory Req = 204.012KB or 87.236KB

Hence, we can see that the total system memory requirements for the computationally challenging convolutional network are acceptable at **87.24KB**, leaving a significant amount of memory for other processing tasks within the system design. For the simple to compute dense layer ‘simple’ model, we can batch weights in sequentially for a dense layer much more easily than a convolutional layer, allowing us to not store all weights in memory at the same time.

All numbers for Model 1 come from “Training” source, and all numbers for Model 2 come from François Chollet’s Keras tutorial.

Hardware/Software Interfaces

Software Connects the Layers Accelerated by Hardware

Each machine learning layer's various computations are to be carried out by separate Verilog hardware modules, each with a load, compute, store workflow.

The HW/SW interface is based on the Intel Avalon Bus Protocol, where we have ADDRESS, CHIPSELECT, WRITE, WRITEDATA, READ, and READDATA signals. An 8-bit register will be used to write data (layer input (image data/last layer's output), convolution weight values, bias) from the software to the hardware, and another 8-bit register will be used to read data from the hardware into the software (result of accelerator layers).

HW and SW communication will occur between every layer; the output of the HW accelerator will be read into the SW through the register, and then SW will write this output to the next accelerator along with the corresponding weights and bias. This process repeats until the image data has been processed through all the layers.

Pipelining is implemented through the software by the software passing in the next set of WRITEDATA to the corresponding accelerator once it finishes its current task. Thus, the system can process the output of layer 1 from image 1 on layer 2 while a new image (image 2) is loaded into layer 1, and so forth.

As we are communicating through an 8 bit register, we need to establish a communication ordering/protocol so the HW/SW interface knows what information is being sent at a given time, as well as when a set of information has been successfully sent in completion. The bitstream will look like this:

WRITEDATA:

1. The first 2 bits will correspond to the layer address so the control unit can locate the layer we want to interact with
2. Then the result of the previous layer will be sent to the current one (for first layer, just read from data memory block) (bit count is shown in the data memory table)
3. Then the weights will be sent, followed by the bias values (bit count is shown in the weight memory table)
4. Finally, a 1 bit “Go” flag will be sent to signal the accelerator to begin its computations

READDATA:

1. Once the accelerator of a given layer is finished computing, it will send a 1 bit “Done” signal
2. This indicates to the interface to have the SW begin reading in the data from the data memory modules (specific bit count is shown in the data memory table)

Testing the Accelerator

There are two main levels of verification for this accelerator:

1. **Functionality:** comparing the accuracy of the accelerated system when performed on a baseline laptop system.
2. **Performance:** comparing the speedup of the system compared to performing all operations in software on the same system

The functionality benchmark is essential. By finding which value of the final Dense layer is the maximum value, we can determine which of 10 digits the model predicts a given image to represent. This can be checked against the label of that image: if the prediction matches the label, then the model is correct. Otherwise, the model is incorrect. Accuracy is defined as the number of correct tests divided by the total number of tests. For this system, the accuracy should not just be strong, but should match a laptop system's accuracy for the same model: this will ensure the accelerator system is performing the proper tasks.

The performance benchmark should be measured at minimum on the De1-SoC in a vacuum, evaluating how quickly the model runs on this system. Ideally, this verification would include a direct comparison to a model carrying out all computations on the De1-SoC in C software, and compare the speedup.

Bibliography

“Importance of Neural Network Bias and How to Add It.” *Turing*,

<https://www.turing.com/kb/necessity-of-bias-in-neural-networks#what-is-bias-in-a-neural-network?> Accessed 15 April 2025.

Chollet, François. Keras Documentation: Simple MNIST Convnet. 19 June 2015,

https://keras.io/examples/vision/mnist_convnet/.

Kizheppatt, Vipin. *Neural Networks on FPGA: Part 2: Designing a Neuron*. 2021. *Youtube*,

https://www.youtube.com/watch?v=a2wOjxRf_xg&t=488s. Accessed 15 April 2025.

“MNIST database.” *Wikipedia*, https://en.wikipedia.org/wiki/MNIST_database. Accessed 15 April 2025.

Patel, Prathmesh, et al. *Acceleration of Digit Classification Using Custom CNN on a SoC FPGA*. 2024.

“Training a Neural Network on MNIST with Keras | TensorFlow Datasets.” *TensorFlow*, 14 December 2024, https://www.tensorflow.org/datasets/keras_example.