Implementation of a Hardware Optimized ANN on a Low-Cost CPU-FPGA Hybrid Platform

Mowly Krishnamoorthy
Toronto Metropolitan University
Toronto, Ontario

***Abstract*—Deep learning has been embedded in many areas of technology. However, unlike in CPUs and GPUs it has not matured for FPGAs due to barrier of hardware expertise required. FPGAs, with careful planning, are more power efficient yet capable of matching in performance compared to CPUs and server/embedded GPUs. With Hybrid CPU-FPGA architectures this barrier is lowered where hardware-software codesign framework can be used to parallelize and accelerate certain processes of deep learning models in FPGA while making the CPU as the master. This framework is used to implement an ANN in a low-cost Hybrid device called the Zedboard.**

I. Background & Literature Survey

RECONFIGURABLE computing gives Field Programmable Gate Arrays (FPGAs) a distinctive edge over CPUs, GPUs and ASICs (Application Specific Integrated Circuits). Generally FPGAs provide higher energy efficiency than both GPUs and CPUs and higher performance than CPUs. ASICs have better performance/throughput due to their full customization but at the expense of high capital investments, complex design cycles and lack of reconfigurability. Deep Learning and its constituent methods such as Convolutional, Artificial Neural Networks (CNN/ANN) are constantly evolving conceptually and architecturally that applications need to be able re-deployed regularly. GPU and CPU use Software models to reprogram. However, as channel length of CMOS technology continues to decrease both in size, power consumption and price, there is demand for low-cost and low-power devices for processing raw data at the point of collection, close to the sensors and actuators like in autonomous driving, robotics, internet of things (IoTs) and space missions to name a few. Software programs are power hungry and not energy efficient especially when running on batteries or solar. FPGAs offer an attractive alternative by virtue of customizable data types and hardware reconfigurability at the "Edge" of computing for and is especially well suited for the quick exploration of vast design-space of CNN configuration/parameters

However, FPGAs have large design time and require considerable hardware expertise. Training in FPGAs is generally challenging due to the compute-intensive backpropagation algorithm and memory requirements for parameters. Therefore usually the training is done on a CPU/GPU and often the FPGA is deployed as accelerators and inference engines at the Edge. The literature review will discuss the difficulties, tradeoffs and working implementations of deep learning models (generally CNN) in FPGA. This report will deploy a parametrized Deep Neural Network (DNN), an ANN with many layers, Model on a Zynq-7000 System-on-Chip FPGA interfaced to two ARM CPU cores, commonly known as Zedboard.

A. General Considerations and Challenges of FPGA Design for Deep Learning

The FPGA architecture provides four types of resources[1]:

- Digital Signal Processors (DSPs) - This is an Arithmetic Logic Unit (ALU) that can perform logic/add/subtract/multiply operations. DSP48 means it can perform a maximum of 48-bit addition. DSP slices are an array of DSPs chained to do multiply and accumulate (MAC) operation.
- Look Up Tables (LUTs) - These are x-input, x being a multiple of 2 ranging from around 2 to 8, multiplexers used in implementing combinatorial logic operations.
- Flip-Flops - Logic gates used in sequential operations. Usually they are D positive-edge triggered flip flops.
- BRAMs - Block RAMs are on-chip units for storing of data. They are much smaller than system memory but operate faster. For fast convolution or dot products it is prudent to have intermediate data stored on-chip.

The Zedboard is a CPU-FPGA shared memory architecture where both processes can access shared memory for performing read/writes. This way, they both can simultaneously operate on different parts of shared data using a synchronization scheme.

1) CNN model: CNN model generally has many layers composed of convolution (CONV), pooling and nonlinear (activation) and fully connected (FC), also referred as ANN, layers. FC performs the classification. CONV and FC layers are compute and bandwidth (BW) intensive. In AlexNet CONV and FC layers account for $\sim$93% and $\sim$7% of total computations, respectively. Pooling and nonlinear layers do not require large amount of resources and have no data reuse. ANN layers account for only 0.11% of execution time of AlexNet where it is reduced further in GoogleNet. Using the massive parallelism of FPGA it is the goal of many to accelerate CONV operations without sacrificing BW using very little power.

2) Binarized CNN (BNN): Since CNNs are deployed on error tolerant applications it is possible, instead of using 32-bit Floating point (FP) or 8-bit/16-bit Fixed point (FxP) values, to use 1-bit values binary values which reduce memory capacity

and off-chip BW requirements. AlexNet requires about 50MB memory for storing parameters while the binarized AlexNet requires only 7.4MB which can stored on-chip.

BNNs also allow converting MAC operations of CONV layer into XNOR and bit-count operations performed in LUTs rather than DSPs[2]. This leads to significant reduction in area and energy consumption. FPGAs provide native support for bitwise operations. Since LUTs are used BNNs allow FPGAs to become competitive with GPUs. However, the major drawback is loss of accuracy. But this is where the need for careful design and optimization required, given the unique properties of both CNN models and FPGA architecture, in order to determine the tradeoffs that limit the loss in accuracy[3].

3) CNN Simplification: CNNs contain significant amount of redundant information, which is advantageous when used for error tolerant applications but not very practical for hardware implementations. Thus, many techniques are incorporated to to simplify CNN models that yield lesser hardware overhead. Some of the most common ones are:

- ***Pruning*** Removing the least important connections between input feature channels and FMs dynamically and statically. Dynamic pruning identifies mutually exclusive abstract features and assumes only those few features are required for classification and removes the rest. Pruning also contributes to sparsification of matrices which can have hardware implications that need to be accounted for.
- ***Quantization*** FMs, kernels, input images are quantized, dynamically, to represent them with less bit widths in on-chip memory while maintaining classification accuracy.
- ***Encoding*** This technique is used to encode a set of CNN parameters that are in FP format by clustering, replacing the parameters by their centroids and encoding the centroids with a 1-bit index of a dictionary. This reduces multiplications in a traditional dot product.
- ***Data Compression*** CNN parameters are computed in training phase and remain constant after. With dictionary or entropy encodings, those parameters can be compressed and decompressed during runtime in order to mitigate the BW bottleneck of CNNs in FPGA.

4) Utilization Schemes: Naturally FPGAs offer customizable resource due to their native support of custom data types but that alone is not a good indicator of overall performance because different dimensions of MAC arrays can provide vastly distinct throughputs for different layers and NN topologies. This often improves resource efficiency but poor throughput due to mismatch with computation-patterns of different layers and overhead of reconfiguration. In addition, since BW utilization of FPGA crucially depends on burst length of memory access, memory access patterns of different NN topologies need to be carefully analyzed and optimized.

5) Hardware-aware Algorithmic Optimizations: This will be quick overview of algorithmic strategies used in Hardware architectures for Deep Learning.

- ***Loop reordering, unrolling and pipelining*** Reordering seeks to increase cache usage efficiency by avoiding redundant memory access between different loop iterations. Unrolling and pipelining use parallelism between loop iterations by using FPGA resources. If loop has independent loop iterations multiple iterations can simultaneously be unrolled (or unfolded).
  Pipelining decreases latency by allowing an iteration to proceed even before the previous one is finished as long as the current iteration is provided the right data in a timely manner[4].
- ***Tiling*** Tiling is needed when working with large datasets where data item fetched in cache may get evicted even before being used. Therefore to alleviate this issue, data is fetched from memory in chunks (tiles) that can fit in the cache. The kernels are usually small and not tiled but they can be if required.
  The tradeoff with this method in hardware architecture it scrambles the memory access pattern because it is not contiguously read from memory if it were not used. Therefore burst access cannot be used with tiling. The tradeoff is by improving cache utilization memory access efficiency or memory BW is reduced.
- ***Batching*** Multiple input frames/images are simultaneously processed in order to increase throughput, where matrix-vector multiplications (MVM) are converted to matrix-matrix multiplications (MM). This is especially useful for improving reuse of weights in ANN layers. However since it increases latency it is not suitable for latency-critical applications.
- ***Fixed Point (FxP) numbers*** Fixed point incurs smaller area and latency overhead than FP format. Many works are justified in using FxP because CNNs are error-tolerant and using bounded nonlinear activation functions like tanh and Sigmoid also limits the range of parameters and does not harm accuracy.
- ***HW-overhead reduction schemes*** There is a high cost in implementing exact exponent functions so some works use piece-wise linear approximation of activation functions. To reduce logic for multiplications, dictionary encodings, Winograd algorithm, AND gates with half-tree adders instead of DSPs are employed. When it comes to padding in BNNs 0 is avoided if $\{$-1,+1$\}$ used for weights and activations since incorporating 0 would mean 2-bits of storage rather than 1-bit as was discussed earlier. Thus -1 or +1 is used for padding.
- ***Frequency-Fomain (FDC) and Winograd CONV*** In traditional CONV (spatial) every element in output feature map (FM) is separately computed. With Winograd CONV a tile of output FM is concurrently generated based on structural similarity between the elements in the same tile of the input FM or image. This reduces the number of multiplications by reusing the intermediate outputs and is very good for small kernel sizes and stride values. Since it uses tiling memory BW is reduced even though it reduces computations. For large kernel sizes this algorithm incurs higher overhead due to addition and constant multiplications[5]. The FDC is a 2D CONV that can be computed using 2D FFT where it is deconstructed as 1D FFT of every row followed by 1D FFT of every column. Despite being less complex having small kernels

leads to a large number of operations with FDC, however. This can be further reduced with an overlap-and-add (OaA scheme). Since FDC is in the complex domain, the memory requirements increase compared to Winograd Algorithm.

Moss et al[6] accelerate a BNN using a CPU-FPGA architecture running on an Intel Xeon+FPGA system, using a systolic array design where every processing engine (PE) performs multiplication using XNOR and bit-count operations. Inputs to every PE are interleaved to reduce BW requirements. [7]. Concluding this brief overview is a focus on CPU-FPGA collaborative architectures because the project is done using this scheme.

II. Problem Statement

FPGAs, delivering better throughput/watt compared to CPUs and GPUs[8], are built conceptually from the dataflow model rather than the Von Neumann Architecture that dominates PCs, laptops and mobile devices. Deep Learning can be modelled as a dataflow network which uses a data-driven execution scheme, such that the availability of inputs trigger the execution of the network [9]. This inherent similarity between Deep Learning and FPGAs appears to be a natural fit but how to make it efficient in power consumption, non-contiguous memory/data access, bandwidth usage while maintaining comparable performance is a tricky proposition. I propose to initially demonstrate a proof-of-concept using an ANN that is hardware optimized and parameterized for classification on a hybrid FPGA platform rather than the DNN presented in [10].

III. System Model

This exercise will build on the ZyNet (portmanteau of Zynq SoC + Neural Network) architecture presented in [10]. In that paper, a 4 layer NN is implemented along with a final classification layer called "Hardmax" - as opposed to a softmax implementation which is challenging in hardware. These tools are hosted in Github[1]. The system architecture is shown in Figure 1.

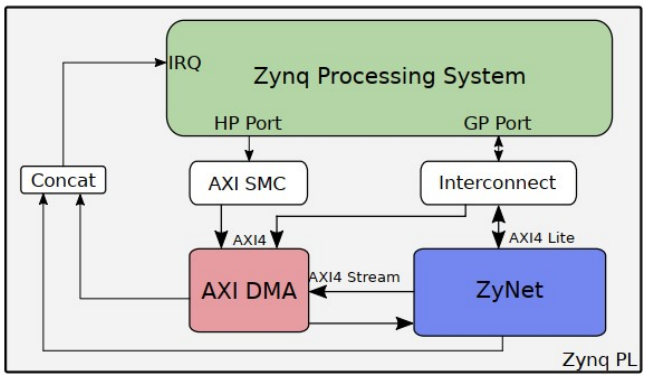

Fig. 1: *System architecture. AXI - Advanced eXtensible Interconnect. HP - High Performance, IRQ - Interrupt Controller, PL - Programmable Logic, SMC - Smart Interconnect.*

[1]https://github.com/vipinkmenon/neuralNetwork/tree/master

The neuron cell architecture is carefully designed without using any Xilinx IP in order to make it scalable and flexible on any FPGA platforms. There are python scripts that automate the creation of any desired neural network as long as hardware resources are able to support it.

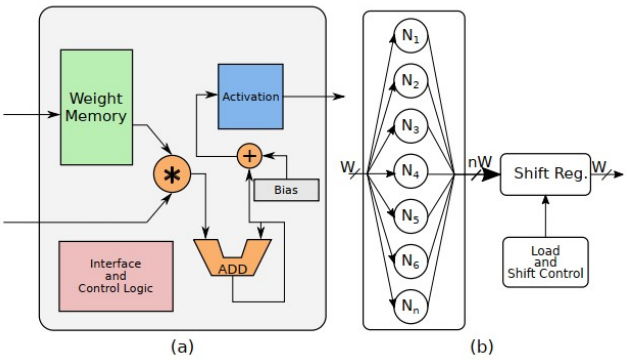

Fig. 2: *a) Single Neuron Architecture. b) How neurons form a layer in neural network.*

Initially, the latest release of Vivado-2023.2, was installed. However, since the ZyNet program was compiled in Vivado 2017.4 with Xilinx SDK part of it for hardware-software codesign, the archived version was installed to avoid any back compatibility issues. All the simulation, block design, synthesis, implementation, bitstream generation and C driver design was done in Vivado 2017.4.

A. Design Considerations in Hardware

In order to automate the generation of a native NN in hardware, it needs to be made scalable. It is possible to make one big circuit for the NN but it will also be one big combinational circuit, considerably limiting the performance and max frequency, FPGA can operate at. Therefore, the circuits need to be pipelined, sequentially, using state machines, interface and control logic, to realize scalability, flexibility and speed. The following points will list the design decisions made for this architecture.

- ***Why use pretrained parameters?*** Algorithms in general, especially backpropagation, are not very hardware friendly. Dot product multiplication, however, is very hardware friendly and it is efficient to use FPGAs in deployment/production environments rather than in training and testing environments. Besides training usually takes place once to determine the weights and biases. Therefore, it is impractical to dedicate intensive resources to train the parameters for FPGAs intended for low-cost, low-power computing. Another important factor is training for ANN is usually not time critical. Thus using pretrained parameter values is best suited for low-cost and low-power, edge computing devices. The parameters are obtained from a Tensorflow model designed in Google Colab.
- ***To ReLU or not to ReLU?*** Rectilinear activation functions are widely used in Tensorflow and its been proven they are sufficient in ANN for classification. However

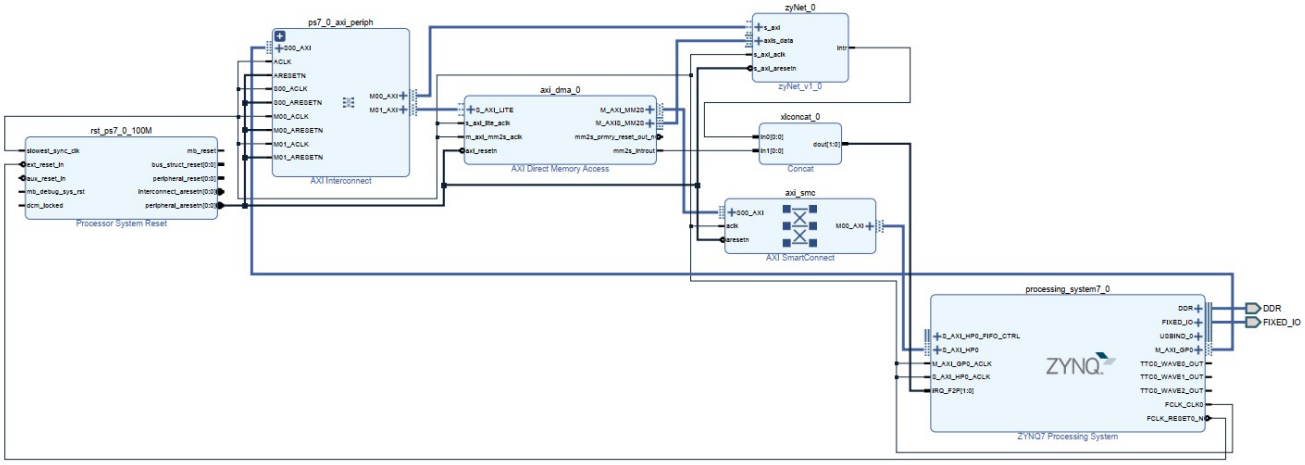

Fig. 3: *The generic block diagram of a ZyNet implementation displayed by Vivado. ZyNet has an interrupt signal which is concatenated with the interrupt from the AXI DMA controller and sent to the Zynq processor system (PS). The ZyNet block houses user defined NNs specified in step 2 of Evaluation procedure.*

when it comes to hardware design ReLU presents a major issue: Unboundedness. When y = x if x > 0, x can be large where its digital representation becomes problematic if the width of the data bus is immutable. Therefore, since sigmoid activation is bounded between 0 and 1, the signed representation will only need 1 bit to represent positive and negative fractions. This leaves a majority of data width bus to handle the fractional bits in a FxP representation which consequently doesn't adversely affect accuracy unlike ReLU. Thus only Sigmoid activation will be used.

- ***Sigmoid needs memory. How to optimize that?*** The major drawback of using Sigmoid activation is the need for memory to store the precalculated values of the function. Realizing exponential terms of Sigmoid in hardware is tricky and to avoid that its best to use 2D memory array. In FPGAs, memories can be synthesized using BRAM and distributed RAM, using FPGA Look Up Tables (LuTs) and Flip Flops. Distributed RAM is faster, but since LuTs and Flip Flops are used as memory elements, the total resource utilization used for actual calculations is reduced which is not preferable especially if varying sizes of neurons are needed. Therefore, specific coding styles in Verilog/VHDL will make AMD Vivado synthesize BRAMs over distributed RAM. These programming constructs are employed to make sub-circuits more resource efficient.

B. Evaluation Procedure

The procedure to evaluate different NN models is a bit circuitous. There are a few stages where manual changes are required for the proper evaluation.

First, as mentioned earlier, the Tensorflow program in Google Colab is run to generate the weights and biases of the desired NN. In this case it is usually a 2 layer network. The weight and biases are stored in a text file and exported to the directory where the MNIST data and other python ZyNet modules reside.

Next, modify the dataWidth parameter and ZyNet ANN layers accordingly in *mnistZyNet.py* file. In addition, add the weight and biases file saved from the first step. this python module creates the Vivado project for the ANN created and optionally block diagram and IP-XACT file using a TCL script.

Once the project is created, descend into the AXI DMA (Direct Memory Access) block of block diagram 3 and change stream width to 8 bits since MNIST uses *.bmp files which are in 8 bit raw format. And also check "Allow Unaligned Transfers" box. Once that is done save it and now load up the test data samples for Simulation by running *genTestData.py* program. Ensure the directories are matched and dataWidth parameter is the same as in previous step. The simulation runs the testbench called *top_sim.v* where 100 samples are chosen, not randomly, the accuracy is calculated and reported. If satisfied with the accuracy then we can move on to synthesis and implementation.

After synthesis and implementation have been successfully completed, before running bitstream generation, HDL wrapper for the block diagram needs to be created. Once that is created it should be 'set as top'. Only then can the bitstream generation be completed.

Once bitstream generation is done and exported to Xilinx SDK, create an empty application project in C. Now run the *genTestData.py* again but **after** modifying it to output from the function *genTestData* in that python file and commenting the line with the function used in step 3 above. This will create a header file called *dataValues.h* that will contain the pixel values for one user specified test data sample. This header file is imported into the SDK along with source file *nnTest.c* for programming the Zedboard with the generated Neural Network. The output of the Zedboard is displayed via an application called TeraTerm connected to the PC through a serial link. Unfortunately only a single image file is loaded

Resource	Utilization	Available	Utilization%
LUT	59	53200	0.11
FF	66	106400	0.062
DSP	2	220	0.91
IO	36	200	18.0
BUFG	1	32	3.125

TABLE I: *Resource Utilization of a single Neuron coded in Verilog for a ReLU activation function.*

Resource	Utilization	Available	Utilization%
LUT	48	53200	0.09
FF	52	106400	0.049
BRAM	0.5	140	0.357
DSP	2	220	0.91
IO	36	200	18.0
BUFG	1	32	3.125

TABLE II: *Resource Utilization of a single Neuron coded in Verilog for a Sigmoid activation function. The Sigmoid size here is 10 but the sizes used all of the NNs here use 5. This should constitute to less resource usage.*

and compared to the result at this point. Further modifications couldn't be cemented prior to this report.

IV. Results & Discussion

Originally the 4 layer DNN was evaluated in the Zedboard and it conformed to results reported by Vipin. Those results will not be discussed here.

A. Single Neuron

The analysis begins fundamentally with the neuron cell as constructed in *neuron.v*. One cell is evaluated on its resource usage based on activation functions used. In Table I ReLU is seen as more resource intensive as its LUT and FF count is higher. There is no need for memory and hence no BRAMs. The last 3 rows are the same for both activating functions.

Using the Sigmoid activation function actually decreases the resource utilization compared to ReLU for LUTs and FFs as shown in Table II. This reduction is due to the fact that memory elements needed are made to use BRAM, as discussed earlier, in order to speed up the per cell execution. This surprisingly also makes the count for LUTs and FFs smaller in comparison and consequently the resource utilization is better than for ReLU.

B. 2-Layer ANNs

There were many different NNs analyzed but only four are presented in Table IV. Even though ANNs might have better accuracy in simulations, it does not mean that they can be synthesized and implemented on the FPGA. Some of them are more resource intensive due to higher number of Neurons in first layer and data width used.

It is interesting to see that the last row ANN with 256 Neurons in first layer was implemented with 8 bits. However its simulation accuracy is a bit lower than for 128 Neurons, probably due to greater loss of accuracy in accounting for weights for all 256 neurons with 8 bits as compared to 128 Neurons.

The hardware validation of ANN2L256_8b (short-form notation indicating type of NN, total number of layers, number of neurons in first layer and the data width bits use d) for last row in Table IV) returned an error in the first try where number 5 was detected when the expected number is 2. No further numbers were tested. For ANN2L128_8b, the results were accurate for the MNIST dataset. The total resources for ANN2L128_8b are shown in III. No more than 50% of resources are utilized to implement this 2-layer NN. 10 datapoints were hardware validated and results shown in Table V. As can be seen the detected digits all matched the result. Of course this is only for 10 iterations and more needs to be done to examine its robustness.

Resource	Utilization	Available	Utilization%
LUT	17463	53200	32.83
FF	8267	106400	0.062
BRAM	69	140	49.29
IO	97	200	48.50
BUFG	1	32	3.13

TABLE III: *Total Resource Utilization of a 128-first layer coded with Sigmoid Activation Function.*

no. of Neurons in First Layer	Data width in bits	Simulation Accuracy %	Implemented?
256	16	98	No. Needs 440 DSPs
128	16	95	No. Needs 276 DSPs
128	8	95	Yes.
256	8	94	Yes.

TABLE IV: *Results of 2-Layer ANNs. Simulation was run over 100 samples.*

A physical picture of the Zedboard along with the results displayed on TeraTerm is shown in Figure 4.

The results reported in Table V are the same displayed in Figure 4 if it can be zoomed in.

V. Conclusion

A proof-of-concept for a 2-layer ANN has been successfully demonstrated on the Zedboard via a hardware-software codesign (CPU-FPGA) framework. This work builds heavily on the research done by [10]. However, this is just the beginning: Timing, performance, power consumption and temperature analysis need to be investigated in the future to rigorously compare the efficacy of ANNs to DNNs, not only on MNIST but on other datasets as well. CNNs models can be further incorporated for better processing capabilities in Edge computing applications, especially ones that employ a camera.

This demonstration almost resembles full-stack development from Python to C to Verilog. This process was already

testnumData	Result	Detected Digit
5	1	1
9	1	1
20	9	9
100	6	6
13	0	0
51	3	3
17	7	7
19	4	4
15	5	5
1	2	2

TABLE V: *Hardware validation results for ANN2L128_8b. The first column is the index number of test dataset ranging from 0 to 9999 for a total of 10,000 samples.*

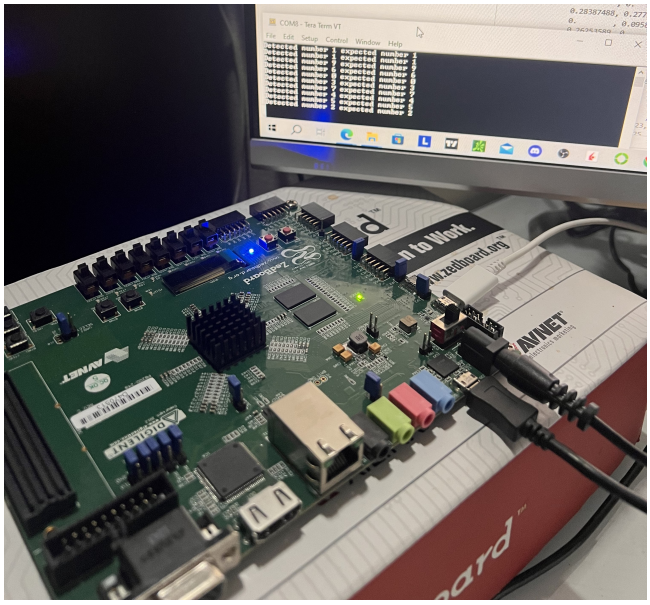

Fig. 4: *A picture of the setup of the Zedboard and its operation regarding ANN2L128_8b.*

automated at certain levels but needs better integration for extensive hardware validation. It would be interesting to attach a monitor or another display device, perhaps the OLED in Zedboard, to display the detected digits along with the result for further visual confirmation.

REFERENCES

[1] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural computing and applications*, vol. 32, no. 4, pp. 1109–1139, 2020.

[2] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[3] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Scaling binarized neural networks on reconfigurable logic," in *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, 2017, pp. 25–30.

[4] Y. Li, Z. Liu, K. Xu, H. Yu, and F. Ren, "A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 14, no. 2, pp. 1–16, 2018.

[5] H. Yonekawa and H. Nakahara, "On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an fpga," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 98–105.

[6] D. J. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "High performance binary neural networks on the xeon+ fpga™ platform," in *2017 27Th International conference on field programmable logic and applications (FPL)*. IEEE, 2017, pp. 1–4.

[7] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 535–547, 2017.

[8] Y. Qiao, J. Shen, T. Xiao, Q. Yang, M. Wen, and C. Zhang, "Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 20, p. e3850, 2017.

[9] K. Abdelouahab, M. Pelcat, J. Serot, C. Bourrasset, and F. Berry, "Tactics to directly map cnn graphs on embedded fpgas," *IEEE Embedded Systems Letters*, vol. 9, no. 4, pp. 113–116, 2017.

[10] K. Vipin, "Zynet: automating deep neural network implementation on low-cost reconfigurable edge computing platforms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 323–326.