ZyNet: Automating Deep Neural Network Implementation on Low-cost Reconfigurable Edge Computing Platforms

Kizheppatt Vipin
Department of Electrical and Computer Engineering
Nazarbayev University
Nur-Sultan, Kazakhstan
vipin.kizheppatt@nu.edu.kz

***Abstract*—Prevalence of internet of things (IoT) enabled applications provide a new opportunity to low-cost FPGA devices to act as edge computing neural network nodes. Although FPGA vendors provide neural network development environments, they often target high-end devices. At the same time these development platforms are not as user friendly as their software counterparts. In this work we introduce ZyNet, a Python package, which enables faster implementation of deep neural networks (DNNs) targeting low-cost hybrid FPGA platforms such as the Xilinx Zynq. Based on hardware-software co-design approach, this platform supports pre-trained or on-board trained networks with development environment very similar to the popular TensorFlow. Implementation results show that the DNNs generated by the platform achieve accuracy very close to software implementations at the same time gives throughput by an order of magnitude compared to other edge computing devices at lower energy footprint. The platform is integrated with Xilinx development tools and is distributed as open source.**

***Index Terms*—neural networks, hardware-software co-design**

I. Introduction

Advent of IoT-enabled applications have stimulated the interest for neural networks on edge computing devices. These are low-cost devices with limited computing and memory capabilities and require a low energy footprint. It has been shown many a times that FPGAs can outperform software implementation of deep neural network (DNN) as their concurrent computation model is neural network friendly [1]. Although FPGA-based DNNs have found their way to datacenters, they are still not popular for edge computing due to their cost, power consumption and development time. Hybrid FPGA platforms such as Xilinx Zynq and Intel Cyclone V are promising in this regard as they are relatively cheaper, low power consuming and can support a hardware-software co-design approach for better flexibility.

Another stumbling block for FPGA-based neural computing is the lack of programmer friendly development environment. Although FPGA vendors are providing AI/ML platforms [2], [3], they often target high-end devices for implementation and toolflows are difficult to follow. In this work we introduce ZyNet, a Python package which not only generates synthesizable DNN code, but automatically integrates it with necessary peripherals to provide a complete system solution. The target implementation platforms are low-cost hybrid FPGA platforms such as ZedBoard and MicroZed, but the solution can be equally extended to any other FPGA platform. The generated DNN RTL code are vendor agnostic and can be implemented with any logic synthesize tool.

The remainder of this paper is organized as as the following, Section II discusses the motivation, Section III discusses the architecture of ZyNet, Section IV discusses the implementation results and Section V concludes the paper and recommends future research directions.

II. Motivation

The code snippet below shows the implementation of a 5-layer DNN in TensorFlow with 4 hidden layers with 30, 30, 10 and 10 neurons each. Each layer uses ReLU-based activation function and an output *softmax* layer is used to determine the output neuron with maximum value. The programing model is very intuitive, flexible, scalable and programmer friendly.

```
import tensorflow as tf
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(30,activation=relu))
model.add(tf.keras.layers.Dense(30,activation=relu))
model.add(tf.keras.layers.Dense(10,activation=relu))
model.add(tf.keras.layers.Dense(10,activation=relu))
model.add(tf.keras.layers.Dense(10,activation=softmax))
model.compile(optimizer='adam',metrics=['accuracy'],
loss='sparse_categorical_crossentropy')
```

Our aim is to develop a platform, which is like TensorFlow but generates efficient RTL code targeting FPGAs. The snippet below describes the same network as before, but generates Verilog RTL code, which can be synthesized and implemented using any target FPGA vendor tools. Since hardware implementation provides bit-level precision of, three additional parameters are provided with the *compile* method. Details of ZyNet methods are explained in Section III-D.

```
import zynet as zn
model = zn.zynet.model()
```

978-1-7281-2943-3/19/$31.00 ©2019 IEEE
DOI 10.1109/ICFPT47387.2019.00058

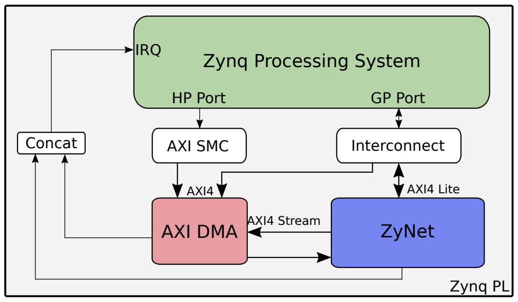


Fig. 1. System Architecture

```
model.add(zn.zynet.layer("flatten",784))
model.add(zn.zynet.layer("Dense",30,"relu"))
model.add(zn.zynet.layer("Dense",30,"relu"))
model.add(zn.zynet.layer("Dense",10,"relu"))
model.add(zn.zynet.layer("Dense",10,"relu"))
model.add(zn.zynet.layer("Dense",10,"hardmax"))
model.compile(pretrained='No',dataWidth=16,
    weightIntSize=4,inputIntSize=1)
```

The *dataWidth* specifies total bit-width of input data, bias and weight values. *InputIntSize* and *weightIntSize* specifies the number of bits used to represent the integer portion of input and weight values. Since ZyNet implementations use fixed point representation, these parameters are required for internal hardware implementation. For *pretrained* networks, where weight and bias values are available before network synthesis, these parameters can be automatically determined by the tool by analyzing the maximum and minimum values.

III. Architecture

The following section discusses the architecture of the DNNs generated by ZyNet, its parameters and application interface.

A. System Architecture

Fig. 1 depicts the complete system architecture generated by ZyNet when targeting the Zynq platform. It packages the DNN as an IP core (called ZyNet itself) and automatically integrates with other peripherals. The AXI4-Lite interface of ZyNet is connected to the GP0 (General Purpose 0) interface of Zynq PS (processing system). This interface is used for configuration in case of untrained networks and for reading out the final network output in case of both pre-trained and on-board trained network.

The AXI4 stream interface from ZyNet is connected to a DMA controller, which in turn is connected to the external memory through the Zynq HP0 (high performance 0) interface. This enables training and test data to be directly streamed from external memory to ZyNet. The DMA controller is also interfaced with the Zynq GP0 port for configuration. Interrupt signals from both ZyNet and DMA controller are connected to the PS interrupt interface.

Table I lists the register file of ZyNet which is accessible through the AXI4-Lite interface. For an on-board trained network, software running on the Zynq processing system (ARM processor core) can access individual weight and bias values by initially setting the *Layer Number* and *Neuron Number* registers and then writing to the *Weight/Bias* registers. To detect the output neuron with maximum value, it is enough to read from offset address 0x8. To access the output from each neuron in the output layer, software can read from address 0x14 (Output Neuron Register). Initial read from this register corresponds to first output neuron. For each read from this register, the output neuron number is auto incremented, and the corresponding data can be accessed. *Control* register is used for issuing control signal such as soft reset to the network and the *Status* register gives the network status such as availability of valid output data. The interrupt signal of ZyNet is directly wired to this Status register bit.

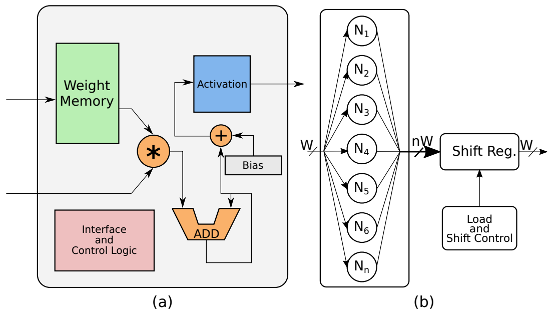


Fig. 2. (a) Neuron Architecture (b) Layer Architecture

B. Neuron

The architecture of a single artificial neuron used by ZyNet is as shown in Fig. 2(a). Each neuron has independent interfaces for configuration (weight, bias etc.) and data. Irrespective of number of predecessors, each neuron has a single interface for accepting data. This enables scalability of the network and improves clock performance by compromising on latency.

An internal memory, whose size is decided by the number of inputs to the neuron, is used to store the weight values corresponding to each input. Depending on whether the network is configured as *pre-trained* or not, either a RAM with read and write interface or a ROM initialized with weight values are instantiated. As inputs are streamed into the neuron, a control logic reads the corresponding weight value from the memory. Inputs and corresponding weight values are multiplied and accumulated (MAC) and finally added with the *bias* value. Like weights, bias values are stored in registers at implemen-

TABLE I
Register Address Map for ZyNet AXI4-Lite Interface

Offset Address	Access	Function
0x00	RW	Weight Register
0x04	RW	Bias Register
0x08	R	Final Output Register
0x0C	W	Layer Number Register
0x10	W	Neuron Number Register
0x14	R	Output Neuron Register
0x18	R	Status Register
0x1C	RW	Control Register

tation time if the network is *pre-trained* or configured at run-time from software. The output from the MAC unit is finally applied to the activation unit. Based on the type of activation function configured (Sigmoid, ReLU, hardMax etc.), either a look-up-table based (for Sigmoid) or a circuit-based function is implemented by the tool. The type of function chosen has a direct impact on the accuracy of the network and the total resource utilization and clock performance. The depth of the LUT for Sigmoid function can be optionally specified by the user or the tool can automatically determine it.

C. Layer

Each layer instantiates user specified number of neurons and manages data movement between the layers. Since each neuron has a single data interface and a fully connected layer requires connection to every neuron from the previous layer, data from each layer is initially stored in a shift register. It is then shifted to the next layer one per clock cycle as shown in Fig. 2(b). Connection between layers and integration with input and output AXI interfaces are automatically implemented by the tool.

D. ZyNet Methods

Table II describes the most important methods available in the ZyNet package. When the *add* method is used to add a new layer to the network, user has to specify its type (such as flatten or dense), number of neurons in the layer and the activation function type. *Flatten* type is applicable to only input layer and presently all hidden and output layer should be *dense* type. Presently three kind of activation functions are supported such as Sigmoid, Rectified Linear Unit(ReLU) and hardMax. If the *pretrained* attribute is set to *True* with the compile method, the user should provide the weight and bias values as Python lists. The DNN generated by the *compile* method is FPGA architecture independent, but the methods for IP packaging, and integration with the processor are currently supported only on Xilinx Zynq devices. In addition to the above listed methods, several helper methods are available to the user to determine the optimal fixed-point representation, extraction of weights and biases from software trained model, generating simulation data etc.

TABLE II
IMPORTANT METHODS AVAILABLE IN THE ZYNET PACKAGE

Methods	Action
zynet.model()	Creates a zynet DNN object
model.add()	Adds a new layer to the DNN
model.compile()	Generates the DNN RTL code
zynet.makeXilinxProject()	Generates a Xilinx project with DNN as the top module
zynet.makeIP()	Packeges the DNN into IP-XACT format
zynet.makeSystem	Creates a block design with the generated IP block, Zynq processor system, DMA controller and other peripherals

IV. RESULTS AND DISCUSSIONS

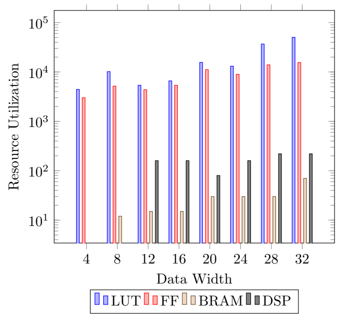
In this section we discuss the implementation results and performance of ZyNet based DNNs. Multiple DNNs targeting the popular MNIST dataset is implemented and evaluated through both simulation and hardware validation. MNIST dataset for handwritten digit recognition uses 30000 images for training and 10000 images for testing. The weights and biases for the network were initially determined from software implementation and used for pre-training hardware implementation. The software implementation after 30 epoch training provides 96.52% detection accuracy for the testing set.

All implementations follow a 5 layer architecture with 784 neurons in input layer, 2 hidden layers with 30 neurons each and 1 hidden layer with 10 neurons and an output layer with 10 neurons. The output layer is connected to a *hardmax* module to detect the neuron with maximum output value. All designs are simulated and implemented with Xilinx Vivado 2018.3 version and hardware validated on ZedBoard having an xc7z020clg484-1 SoC and 512MB external DDR3 memory. All implementations use Vivado default settings and do not apply any optimization (timing, area or power).

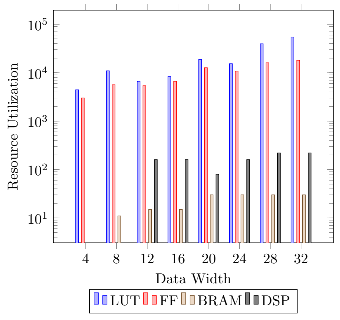
The DNN implementation was evaluated for both Sigmoid and ReLU activation functions for varying datawidth. Fig.3(d) shows the relation between the detection accuracy and datawidth. It could be seen that for very small datawidth (such as 4 and 8 bits), Sigmoid-based function implementation outperforms ReLU-based implementation. As the width increases, ReLU has slight advantage over Sigmoid implementation and accuracy of both implementations becomes constant beyond 12-bits. Sigmoid implementation gives a maximum of 94.86% detection accuracy and ReLU gives a maximum of 95.87%. The degradation in the result compared to software implementation can be attributed to the error introduced due to fixed-point representation of weights, biases and input data. Still the approximation causes less than 1% error but gives considerable advantage in terms of resource utilization and clock performance.

Figures 3(a) and 3(b) compares the resource utilization of the DNNs for different data widths in terms of LUTs, flip-flops, Block RAMs (BRAMs) and DSP slices while using the two different activation functions. Since the RTL code generated by ZyNet does not explicitly instantiates any IP cores, for smaller designs the implementation tool (Vivado) automatically maps the multipliers and weight memory blocks into LUTs and flip-flops. For larger data size, the lookup table used for implementing Sigmoid function are mapped to Block RAMs, which considerably increases the BRAM utilization. For example, for 32-bit implementation, Sigmoid-based implementation requires 50350 LUTs, 15544 flip-flops, 70 BRAMs and 220 DSP slices. At the same time ReLU-based implementation requires 54559 LUTs, 18074 flip-flops, 30 BRAMs and 220 DSP slices. These numbers roughly maps to 94.6% LUTs, 17% flip-flops, 21.4% BRAMs and 100% DSP slices of the chip. It should be noted that 16-bit implementation also consumes 220 DSP slices and as discussed before for larger networks the tool automatically maps the multipliers into LUTs and flip-flops. Thus the largest network size is constrained by the number of LUTs available in the device.

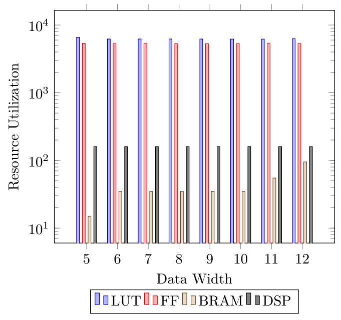
Another interesting result is the size of the Sigmoid look-up-



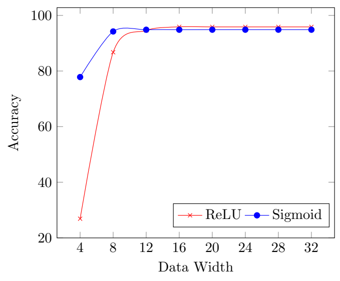
(a) Resource utilization for sigmoid Activation function



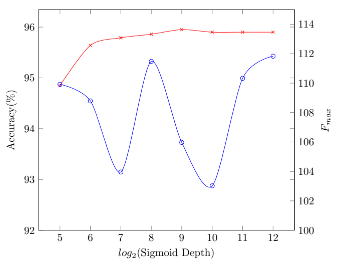
(b) Resource utilization for ReLU Activation function



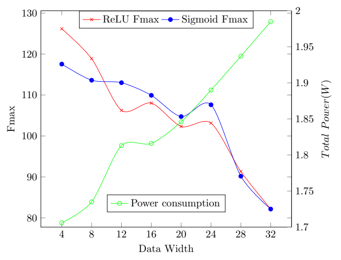
(c) Resource utilization for sigmoid Activation function with varying depth



(d) Comparison of accuracy for Sigmoid and ReLU activation functions



(e) Detection accuracy and varying sigmoid memory depth



(f) Datawidth vs Maximum Frequency and Power consumption

Fig. 3. Evaluation results in terms of resource utilization, power consumption and maximum clock performance for different ZyNet neural networks

table and the detection accuracy. Fig. 3(e) shows the relation between the number of address bits used for Sigmoid LUT (sigmoid memory depth is $2^{address\_bits}$) and detection accuracy. Here the datawidth is kept at 16 bits. Simulations showed that the accuracy is quite low when the number of address bits is less than the sum of integer bits used to represent input and weight values. Here the minimum value is set as 5 bits, since the implementation uses 1-bit integer value for input and 4-bits for weight. It could be seen that the Accuracy initially increases and remains constant after 8 bits. Comparing it with Fig. 3(c), resource utilization remains constant between LUT address width 6 to 10. At the same time detection accuracy increases from 95.64% to 95.95% between these values. The constant resource utilization is attributed the granularity of BRAM available in the device, which is 18Kb for 7-series Xilinx FPGAs. Beyond 10-bits, resource utilization increases due to more BRAMs are required for implementing Sigmoid LUTs. The detection accuracy also slightly reduces (95.90%) and remains constant. The maximum clock frequency varies between 103 MHz and 111 MHz due to placement algorithm of the implementation tool.

Fig. 3(f) shows the maximum clock performance of the DNNs for varying datawidth with two activation functions. In most cases Sigmoid outperforms ReLU, but on Zynq platforms most other design components run at 100 MHz, which is achieved by both implementations up to 24-bit datawidth. The plot also shows the power consumption of ZedBoard for Sigmoid-based network when running at 100 MHz clock frequency. It varies between 1.706 W and 1.985 W depending on the datawidth (resource utilization). Comparing it with the other popular edge device Raspberry-Pi 3, the power consumption is in the range of 2.7W to 3.1W while running TensorFlow for MNIST dataset.

Finally comparing the throughput, ZyNet based network on ZedBoard running at 100 MHz has a detection speed of 7.84 us/sample. On a Raspberry Pi 3 using tensor flow, the throughput is 266 us/sample. On a standard computer with Intel i7 processor and 16GB RAM, the throughput for the same network is 30 us/sample. This clearly demonstrates the advantage of reconfigurable platforms for edge computing in terms of throughput and power consumption.

V. Conclusion and Future

In this work we discussed the implementation of ZyNet, a Python package for automating deep neural network implementation on low-cost FPGA-based edge computing devices. Results show that the implementation are capable of achieving high accuracy and clock performance with low resource utilization. The proposed platform is capable of outperforming pure software implementation by several orders compared to other embedded platforms by consuming lesser power. The package is available in the public domain through the Python Package Index (PyPI) repository and example designs are available through the git repository [4].

References

[1] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.
[2] *DNNDK User Guide*, Xilinx Inc., Jun. 2019.
[3] *The OpenVino Toolkit*, Intel, 2019.
[4] ZyNet git repository. [Online]. Available: https://github.com/dsdnu/zynet