NetPU-M: a Generic Reconfigurable Neural Network Accelerator Architecture for MLPs

Yuhao Liu, Shubham Rai, Salim Ullah, Akash Kumar
Chair of Processor Design, Center for Advancing Electronics Dresden (CfAED), TU Dresden, Germany
Email: {yuhao.liu1, shubham.rai, salim.ullah, akash.kumar}@tu-dresden.de

***Abstract*—Recent research widely deployed *Neural Networks* (NNs) in various scenarios, such as IoT systems, wearable devices, or smart sensors. However, the complex application scenarios cause the rapid extension of network model size and the requirement for higher-performance hardware platforms. Related works apply *Heterogeneous Streaming Dataflow* (HSD) and *Processing Element Matrix* (PEM) architectures as the most popular schemes for FPGA-based implementation of NN accelerator: 1) HSD architecture implements a complete network for given trained models on FPGA with simplified control but more hardware consumption; 2) PEM architecture implements reusable neuron structures controlled by runtime environments/drivers providing the generic acceleration supports for different network models. Our work explores a new hybrid architecture based on HSD and PEM to implement a reusable partial network structure on FPGA and achieve generic acceleration supports for different network models with simplified runtime control. This architecture supports scalable, mixable, quantized precision, and selectable activation functions, including *ReLU*, *Sigmoid*, *Tanh*, *Sign*, and *Multi-Thresholds*. Data stream transmission can reset the accelerator configuration in runtime without hardware implementation changes for different networks. This work has supported *Multi-Layer Perceptron* (MLP) in current.**

I. Introduction

FPGA-based Neural Network (NN) accelerator is a rapidly advancing subject in recent research. Although current widely-applied NN acceleration applications focus more on the deployments on CPU and GPU platforms, with the extension size and increasing energy consumption of large-scale networks, related researchers have begun to explore customized hardware and low-precision computation schemes to improve power efficiency and data throughputs. Compared with the high design complexity and expensive development costs of customized ASICs as NN accelerators, FPGA achieves a good trade-off between customizability, flexibility, and power efficiency, making it a feasible candidate as a NN acceleration platform.

Here, we classify the implementations of neural network structures on FPGAs as two hardware architectures: i) ***Heterogeneous Streaming Dataflow*** **(HSD)** architecture and ii) ***Processing Element Matrix*** **(PEM)** architecture.

- HSD architecture-based implementations explore the reconfigurability of FPGAs to support the customization of hardware design for one given network model. The related works based on this architecture choose to implement a complete and optimized neural network on FPGA for one target model, which simplifies the data stream control of model parameters and inference inputs and outputs. HSD architecture also allows the researchers to explore and apply state-of-the-art low-precision computation and model compression schemes in their works, such as quantization, binarization, approximation, model pruning, and folding, to reduce hardware resource utilization on FPGA. However, the limitation of on-chip hardware resources of FPGA is still the bottleneck for the deployment of large-scale networks. Researchers need to trade between the compressed model size and inference accuracy based on the optimization solutions mentioned above, especially for the low-power FPGA platforms on edge. Besides, for different networks, HSD architecture-based acceleration requires regeneration and re-optimization of hardware design for new target models.
- PEM architecture-based implementations can achieve relatively generic support for different network models compared with HSD architecture. This architecture essentially implements a few neuron structures on the FPGA, including a processing element array with/without systolic multiplier structure and a post-matric processing pipeline integrated with accumulator, activator, quantizer, pooling, etc. Therefore, to drive the above-described neuron structures, accelerating one complete neural network, the related PEM architecture-based FPGA or ASIC implementations require a runtime environment or driver to schedule and control the work of one or multiple PEMs. However, the runtime environment is a heavy payload for some lightweight systems, such as the low-power microcontroller of IoT or edge devices.

Considering the above discussion of HSD and PEM architectures, we assume the following scenario. For a lightweight edge device applying a low-power *Microcontroller Unit* (MCU) and middle-end FPGA development platform: 1) the on-chip hardware resources on the FPGA platform limit the available model size for HSD architecture. 2) low-power performance of MCU cannot support the execution of a complex runtime environment of PEM architecture well.

Therefore, we present a hybrid architecture, ***NetPU-M***, of HSD and PEM architectures in this paper. This architecture is designed to drive a reusable partial HSD-architecture-like implementation offering the generic neural network inference acceleration as PEM architecture with simplified runtime control. This work consists of a three-stage structure: ***Network Processing Unit*** **(NetPU)**, ***Layer Processing Unit*** **(LPU)**, and ***Transformable Neuron Processing Unit*** **(TNPU)**. NetPU reuses LPUs, and LPU reuses TNPUs to extend the flexibility of accelerators fitting different sizes of network model workloads. This accelerator only saves the required parameters of current accelerating layers in LPUs to on-chip memories, reducing the storage pressure for loading complete network parameters. This architecture supports scalable, mixable (the data precision of different layers can be different), quantized (1-8 bits) precision, and selectable activation functions, including *ReLU*, *Sigmoid*, *Tanh*, *Sign*, and *Multi-Thresholds*. The optimization of *Batch Normalization* (BN) folding is also a selectable function. Data stream transmission can reset all the above accelerator configurations in runtime without changing hardware for different networks.

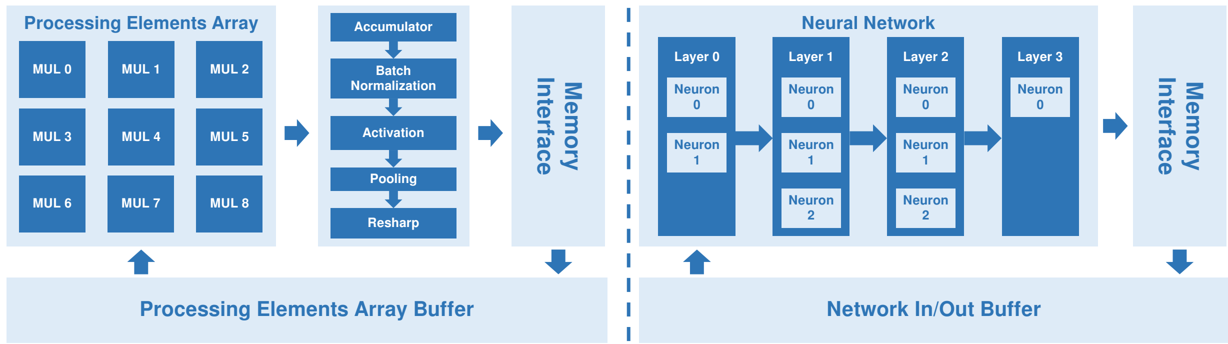

Fig. 1: PEM (left) and HSD (right) Architecture

A. Contributions

Here we highlight the major contributions in this works as follows, motivated by our poster in [20]:

- NetPU-M presents a **hybrid architecture** of HSD and PEM architecture offering the generic inference acceleration for different networks as PEM architecture with simplified runtime control like HSD architecture.
- NetPU-M configures the acceleration mentioned above settings through the data stream **without hardware regeneration for different network models** in runtime.
- NetPU-M only loads the required parameters, inputs, and weights for current inferring layers working in enabled LPUs, reducing the on-chip memory consumption to **support larger and deeper networks**.

B. Potential Application Scenarios

- Large-scale NN inference on lightweight edge devices based on low-power MCU and FPGA platforms.
- Generic NN inference IP core embedded in other FPGA projects with simplified control.
- Fast modeling and prototyping for low-precision NN models targeted to be accelerated on FPGA.
- Multiple FPGAs pipelined NN inference acceleration.

C. Organization

This manuscript is structured in the following way: **Section II** discusses the background of FPGA-based NN accelerators in recent research. **Section III** introduces the implementation of NetPU-M and the difference with existing architectures. **Section IV** shows the simulation and evaluation results of NetPU-M architecture based on the *Ultra96-V2* platform. **Section V** discusses the further potential improvement and optimization of NetPU-M architecture. **Section VI** concludes the contents of this paper.

II. Background

A. HSD and PEM Architectures

1) PEM Architecture: As shown in Figure 1 (left), this architecture focuses on designing a reusable neuron to offer generic support for different kinds of NN models, widely applying the *Processing Element* (PE) array and a post-matrices processing pipeline. This architecture is widely applied in recent industrial and research communities on FPGA and ASIC design, for instance: *DeepBurning* [33], *MP-OPU* [34], *DNNweaver* [26], *AngleEye* [13], *OpenVINO* (Intel) [11], *NVIDIA Deep Learning Accelerator* (NVDLA) [37], *Project BrainWave* (Microsoft) [10], *Vitis AI library* (Xilinx) [17], etc. The advantage of this architecture is the generality meeting the acceleration for different network models. However, because the implementation based on this architecture scheme is a single neuron processing unit, the related works require the cooperation of the runtime environment to drive the implemented hardware engines, such as the *User Mode Driver* and *Kernel Mode Driver* in NVDLA [37]. Additionally, the model compiler is also necessary to convert the trained network as executable data streams layer by layer, for example, the *NVDLA Loadable* [37].

2) HSD Architecture: As shown in Figure 1 (right), the works based on HSD architecture implement a complete network layer by layer on FPGA as hardware design. This architecture applies the heterogeneous layer design with different numbers of neuron processing units according to the given trained network to optimize the hardware resource utilization. Related works stream the data flow between layers and neurons under the hardware control modules, which can simplify the complexity of runtime environments. This architecture scheme is rapidly advancing in recent research, such as *FINN* [30], *FINN-R* [3], *LogicNets* [32, 31], *DeepFire* [2], *FixyFPGA* [21], *HLS4ML* [8, 12, 7], etc. However, the works based on this architecture need to regenerate the hardware design for the different networks, and the heterogeneous layer structure extends the complexity of hardware design. Therefore, automatic hardware generation tools are essential for related works meeting the simplification of hardware re-design. For instance, *FINN-R* [3] and *HLS4ML* [8, 12, 7] explored end-to-end automatic implementation generation by converting trained network models to hardware design.

B. Quantization

Large-scale network model leads to high hardware resource consumption, for example, requiring more DSP slices and on-chip memory in implementing model parameter storage and inference computation on FPGA. Therefore, previous research explored low-precision data representation schemes, such as *Quantization* [30, 3, 25, 32, 31, 8, 12, 7, 16, 35, 9], *Approximate Computing* [28], and *Posit Computing* [15, 14, 22].

Quantization is one of the most widely-applied low-precision solutions. Su et al. [27] explored the *Quantized Neural Network* (QNN) inference accuracy under different precision of 1/2/4/8/16-bits fixed-point format compared with 32-bits floating-point format. Results show that the low-precision schemes can achieve similar accuracy as the full precision. According to their evaluation, 2/4-bit schemes balance resource consumption

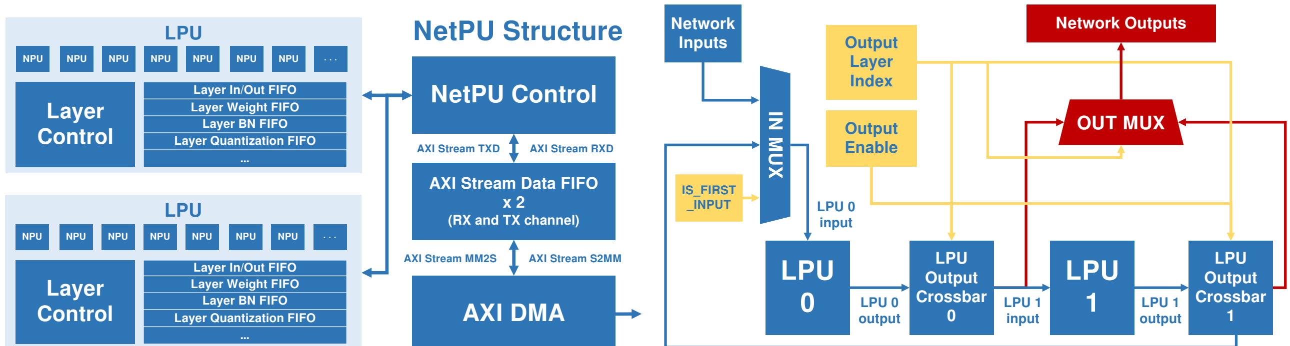

Fig. 2: Hardware Structure of NetPU-M (left) and LPU Recycle Design (right)

and inference accuracy better than others. 1-bit scheme requires a larger network to achieve similar accuracy as others.

Binarization is an extreme situation of quantization methods, which quantizes the activation and weight of a network model as 1-bit values. Because this method can further reduce the resource utilization of NN accelerators on FPGA, previous works widely explored the related implementation on theory and hardware design, such as *XNOR-Net* [24], *XNOR-Net++* [5], *FINN* [30], *FINN-R* [3], etc. As shown in Table I, XNOR gates replace the multiplier in *Binarized Neural Network* (BNN). Therefore, one XNOR operation can replace the multiplication operation of N binarized activation and weights. Because the '1' value in output represents the 1, and '0' represent the -1, applying the *Popcount* operation to count the number of '1' in XNOR outputs can get the sum of 1 outputs. The entire output width minus the above sum can get the sum of -1 outputs. The sum of N binarized activation and weight multiplied product will be the sum of 1 minus the sum of -1.

C. Activation and Batch Normalization

Batch Normalization (BN) can improve the inference accuracy and speed up the network training. Especially for low-precision neural networks, such as BNN and QNN, BN will be the necessary processing before activation. As shown in Equation 1 of BN computation, y_i and x_i refer to the output and input data separately. σ^2 and $\bar{x}$ are variance and mean of mini-batch. To be noticed, according to Equation 1, the computation of BN is full precision. Therefore, the hardware implementation of BN causes high resource consumption. According to the Equation 2, Krishnamoorthi et al. [18] present an optimization method to fold the BN into the weight and bias of *Convolution* (CONV) and *Fully Connected* (FC) Layers. *FINN* [30] applied another method to fold the BN into *Sign* activation as a threshold for BNN as shown in Equation 3.

$$\begin{cases} \hat{x_i} = \dfrac{x_i - \bar{x}}{\sqrt{\sigma^2 + \varepsilon}} & \text{// Normalize} \\ y_i \leftarrow \gamma \hat{x_i} + \beta \equiv BN_{\gamma,\beta}(x_i) & \text{// Scale and Shift} \end{cases} \quad (1)$$

TABLE I: XNOR: Binarized Multiplier

Signed			Unsigned		
Inputs		Output	Inputs		Output
1	1	1	1	1	1
1	-1	-1	1	0	0
-1	1	-1	0	1	0
-1	-1	1	0	0	1

$$\begin{aligned} BN_{\gamma,\beta}(Wx) &= \gamma \hat{Wx} + \beta = \gamma \frac{Wx - \bar{x}}{\sqrt{\sigma^2 + \varepsilon}} + \beta \\ &= \frac{\gamma W}{\sqrt{\sigma^2 + \varepsilon}} x + \left(\beta - \frac{\gamma \bar{x}}{\sqrt{\sigma^2 + \varepsilon}}\right) \end{aligned} \quad (2)$$

Activation, such as *ReLU*, *Sigmoid*, *tanh*, *Sign*, is widely-applied in recent research. However, for some non-linear activation functions, like *Sigmoid*, the hardware implementation consumes a large number of computing resources on FPGA. Therefore, the previous works [36, 1, 35, 4] explore the piecewise-linear function to approximate the nonlinear processing in the activation.

Half-wave Gaussian Quantization (HWGQ) [6][29] is another widely-used activation scheme in the implementation of FPGA-based NN accelerator. In other activation functions, such as *Sigmoid* and *tanh*, the output is the full-precision result, which needs the re-quantization to meet the input precision of the next layer. HWGQ method implements $2^N - 1$ thresholds and counts the number of thresholds as output, which is smaller than the input value. Therefore, the output of HWGQ (or Multi-Threshold) activation will be a quantized value as N-bits data, which folds the re-quantization into activation.

$$\begin{aligned} &Sign\left(BN_{\gamma,\beta}(i)\right) = \begin{cases} 1 & \gamma \frac{x_i - \bar{x}}{\sqrt{\sigma^2 + \varepsilon}} + \beta \geq 0 \\ -1 & \gamma \frac{x_i - \bar{x}}{\sqrt{\sigma^2 + \varepsilon}} + \beta < 0 \end{cases} \\ &\therefore Threshold = \bar{x} - \frac{\beta\sqrt{\sigma^2 + \varepsilon}}{\gamma} \Leftarrow \gamma \frac{x_i - \bar{x}}{\sqrt{\sigma^2 + \varepsilon}} + \beta = 0 \end{aligned} \quad (3)$$

Moreover, *Sign*, as shown in Equation 3, is the BNN-oriented activation. If the result of *Popcount* ≥ 0, the *Sign* output will be 1, else, the output will be 0. Equation 3 also explained how to fold the BN layer into *Sign* activation as thresholds [30].

III. Implementation

A. Introduction of NetPU-M Architecture

According to the summary about HSD and PEM architectures in section II. A, we compare the difference between these two schemes as Table II. The advantages of HSD architecture are the high optimization of hardware design for a given network model and simplified runtime dataflow streaming control. The shortages of this architecture are the requirement of hardware regeneration for different models and the high resource consumption for implementing a complete network on FPGA. PEM architecture offers generic support for different networks with a neuron processing engine. However, the dataflow streaming control and schedule will be complex.

Therefore, considering the promising application scenarios in section I, the target feature of NetPU-M architecture is: **Generic support for different network models meeting few resource consumption without hardware regeneration and complex runtime control.** All implementations in our NetPU-M architecture are written by Verilog. Moreover, we created a C++ program to generate the Verilog macro definitions as a hardware configuration file. Based on the generation block we widely applied in our Verilog codes, the NetPU-M project can easily build a suitable project for different FPGA platforms.

Therefore, the major differences in the hardware design of NetPU-M architecture compared with previous works are:

- To achieve the generic support for different network models without hardware regeneration, this work designed a *Transformable Neuron Processing Unit* (TNPU), which can reset the executing computing precision, activation, and BN folding optimization options in runtime. One crossbar module in TNPU controls the model resetting with the reconfiguration signals by the inner dataflow schedule.
- To simplify the runtime control, this work designed a three-stage structure as Figure 2 (left): A *Layer Processing Unit* (LPU) schedules multiple TNPUs, and A top *Network Processing Unit* (NetPU) schedules multiple LPUs. NetPU controls the data streaming of LPUs and resets LPUs as different kinds of NN layers, such as *Input Layer, Hidden Layer*, and *Output Layer*, and configures the input length and layer setting, including neuron number, activation, and BN folding options. *Input Layer* quantizes the high precision inputs to meet the precision in *Hidden Layers*. *Output Layer* finds the maximum value belongs to which output neurons to achieve the catalog classification. LPU configures and controls the TNPUs to process the given data stream.
- To reduce resource consumption, NetPU schedules the reuse of LPUs based on the recycling structure in Figure 2 (right). LPU controls the reuse of TNPU for inferring more neurons in one layer of a large-scale network model.

Some previous works implemented a similar reusable layer structure, such as the *Multi-layer Offload* in *FINN-R* [3]. However, the *Multi-layer Offload* in *FINN-R* implements some layers with the maximum size in a given trained network to infer the large-scale network models. Our work only implements a few TNPUs and LPUs to support larger network models by reusing TNPUs and LPUs, which can reduce the hardware consumption than *FINN-R* and does not require a given network for hardware generation.

TABLE II: HSD and PEM Architecture Comparison and the Target Features of NetPU-M

Architecture	HSD	PEM	NetPU-M
Generality for Different Networks	Needs Regeneration	Generic Supports	Generic Supports
Complexity of Runtime Environment	Simplified	High	Simplified
Hardware Resouce Consumption	High	Low	Middle
Automatic Generation Tools	Needs	No	No
Has Resource Limitation of Network Size	Yes	No	No

B. Implementation of NetPU-M Architecture

1) Transformable Neural Processing Units (TNPU): TNPU is the basic processing component for NetPU-M architecture. As shown in Figure 3, TNPU contains six submodules: i) ***Multiplier*** (MUL), ii) ***Accumulator*** (ACCU), iii) ***Batch Normalization*** (BN), iv) ***Activation*** (ACTIV), v) ***Quantization*** (QUAN), and vi) ***Crossbar***. The implementation of TNPU refers the works of *FINN* [30], *FINN-R* [3] and BN folding schemes described in section II.B and C.

- ***Multiplier:*** The supported precision of activation inputs and weights in the MUL submodule are from 1 to 8 bits, including N 8-bit integer multipliers ($N = 2^M$, $N \geq 32$) and N 8-bits binary multipliers, which contains four input ports (1. *MUL Inputs* ($8N$ bits), 2. *MUL Weights* ($8N$ bits), 3. *Input Precision Setting* (3 bits) 4. *Weight Precision Setting* (3 bits)) and one output port (*Multiplier Outputs* ($16N$ bits)). The precision of inputs and weights can be different. (*Exception*: When one of the inputs and weights precisions is 1-bit, another should also be 1-bit.) For 1-bit precision, the input and weight data represent eight 1-bit channels. One binary multiplier consists of one 8-bits XNOR gate and one *Popcount* as the implementation in FINN [30]. For 2-8bits precision, the input and weight are only one channel. Multipliers ignore the additional unused bits when precision is lower than 8 bits and higher than 1 bit.
- ***Accumulator:*** This submodule calculates the sum of MUL outputs, which contains two input ports (1. *Accumulator Input* ($16N$ bits), 2. *Bias Inputs* (8 bits)) and one output port (*Accumulator Output* (32 bits)). Only when the BN folding option is activated in the current inference task, *Bias Inputs* is available. 32 bits of ACCU output support computing the sum of at least 2^{16} ACCU inputs, which means our TNPU can support the neuron connection numbers for most datasets and network models by repeatedly reusing the TNPU and computing the sum of MUL results.
- ***Batch Normalization:*** BN submodule is available only when the BN folding option is disabled. This submodule has three input ports (1. *Activation Inputs* (32 bits), 2. *BN Scale* (32 bits), 3. *BN Offset* (32 bits)) and one output port (1. *Activation Outputs* (37 bits). *BN Scale* and *Offset* are two 32-bit fixed-point values, and the *Activation Outputs* is one 37-bit fixed-point value, which has 32 integer bits value and five fraction bits.

$$f(x) = \begin{cases} 1, & |x| \geq 5 \\ x \gg 5 + 0.84375 & 2.375 \leq |x| < 5 \\ x \gg 3 + 0.625 & 1 \leq |x| < 2.375 \\ x \gg 2 + 0.5 & 0 \leq |x| < 1 \end{cases}$$
$$Sigmoid_L(x) = \begin{cases} f(x) & x \geq 0 \\ 1 - f(x) & x < 0 \end{cases} \tag{4}$$

- ***Activation:*** The current supported activation functions in NetPU-M architecture are *ReLU*, *Sigmoid*, *tanh*, *Sign*, and *Multi-Threshold*. According to the implementation in *FINN* [30], *FINN-R* [3], *Sign* and *Multi-Threshold* require the trained thresholds. Therefore, this submodule contains four input ports (1. *ACTIV Inputs* (37 bits), 2. *Activation Selection* (3 bits), 3. *Sign Threshold* (32 bits) 4. *Multi-Thresholds* ($(2^M - 1) \times 32$ bits)) and one output ports (*ACTIV Outputs* (37 bits)). For *Sigmoid* and *tanh*, because $\tanh(x) = 2 \times Sigmoid(2x) - 1$, *tanh* can be converted from *Sigmoid*. Therefore, the ACTIV submodule has only implemented one shared *Sigmoid* for both two

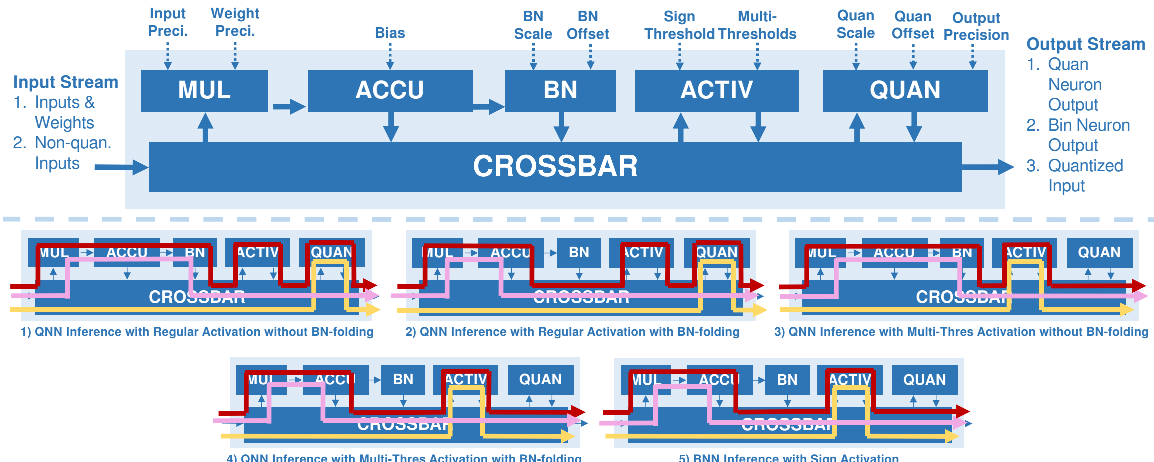

Fig. 3: I. Hardware Structure (top) of the TNPUs; II. Fig.1-5 Show the Data Stream Paths of TNPUs in Input/Output/Hidden Layers of BNN/QNN Models (Yellow Path Represents Input Layer. Pink Path Represents Output Layer. Red Path Represents Hidden Layer)

activation functions. Equation 4 present the principle of piecewise-linear approximate *Sigmoid* function [1], which significantly reduces the consumption of *Sigmoid* implementation avoiding the use of DSP slices. For *Multi-Thresholds* activation, this function requires $(2^M - 1)$ 32bits thresholds. M is the maximum supported precision of *Multi-Thresholds* activation in NetPU-M architecture. As shown in in Table IV, when $M = 8$, the *Multi-Thresholds* activation requires 256 32bits thresholds in one neuron, which causes a huge resource consumption ($\sim 28\%$ LUTs for one TNPU). Therefore, in our testing implementation, we limit the maximum supported precision of *Multi-Thresholds* activation as 4bits, which means at most 16 32bits thresholds reducing the LUTs consumption to $\sim 4\%$.

- ***Quantization:*** For the activated output of *ReLU*, *Sigmoid* and *tanh*, the full-precision value needs to be quantized to the required precision of the next layer. Therefore, this submodule has four input ports (1. *QUAN Inputs* (37 bits), 2. *Output Precision* (3 bits), 3. *QUAN Scale* (32 bits) 4. *QUAN Offset* (32 bits)) and one output port (*QUAN Outputs* (O bits)). O is the required input precision in the next layer. Therefore, in principle, NetPU-M architecture supports the mix-precision, which means the data precision in different layers can also be different. Moreover, this submodule is only available when the enabled activation is not *Sign* or *Multi-Thresholds*.
- ***Crossbar:*** This submodule schedules the dataflow in TNPU to execute the inference as the neuron in different layers. As shown in Figure 3, the crossbar controls the input and output dataflow to bypass some submodules in TNPU as the path of *Yellow* (As a neuron in input layers), *Pink* (As a neuron in output layers), and *Red* (As a neuron in hidden layers). When the BN folding option is enabled in the current inference task, the output of ACCU will not be passed into the BN submodule. When the inferring network model is BNN or applied activation is *Multi-Thresholds*, the output of ACTIV will not be passed into QUAN. When the inferring layer is an input layer, the inference input of the target dataset will be transmitted into ACTIV or QUAN submodule according to the applied activation. (If the current task applies *Multi-Thresholds* or *Sign*, the dataset input will be passed into ACTIV.) When the inferring layer is an output layer, the output of BN or ACCU will be passed to the TNPU output port as the final result of the current output neuron. Moreover, for the output layer, the current NetPU-M architecture implemented a *MaxOut* submodule to find the maximum result in the output layer. We will continue to complete this architecture to support the *SoftMax* in our further work.

2) Layer Processing Unit (LPU): LPU module controls multiple TNPUs to execute the inference of one layer in network model loading. As shown in Figure 2 (left), LPU consists of three parts: i) ***TNPU Cluster***, ii) ***Data Buffer Cluster***, iii) ***Layer Control***.

TNPU Cluster contains multiple TNPUs scheduled by LPU module. ***Data Buffer Cluster*** implemented the FIFOs as a buffer for data and parameter loading. Table III lists the buffer cluster we used in our testing implementation. Entire BRAM consumption is shown in Table V. Therefore, because the supported maximum precision in NetPU-M is 8bits, the max. input length and max. neuron number for data and parameter loading in one layer of the inferring network model is 8192, which can meet most MLP models. ***Layer Control*** schedules the TNPU to complete the inference of one layer. Because the number of TNPUs is smaller than the real neuron number of inferring network model, which limits the processing number of neurons in parallel, LPU divides the neurons into small batches to load the parameters and execute the inference of loaded neurons as one processing period. Therefore, for the MLP models, because every neuron requires the same inputs, to avoid repeated loading of layer activation inputs, we implemented an *Input Reload Buffer* to reuse the inputs as shown in Table III.

Figure 4 shows the working flow of LPUs in NetPU-M architecture, which contains three processing steps: *Layer*

TABLE III: Data Buffer Cluster in LPU

Buffer Name	Output Width	Depth	Buffer Name	Output Width	Depth
Layer Input	64 bits	1024	Input Reload	64 bits	1024
Layer Weight	64 bits	1024	Bias	64 bits	1024
BN Scale	128 bits	2048	BN Offset	128 bits	2048
Sign Threshold	128 bits	2048	Multi-Thresholds	128 bits	2048
QUAN Scale	128 bits	2048	QUAN Offset	128 bits	2048

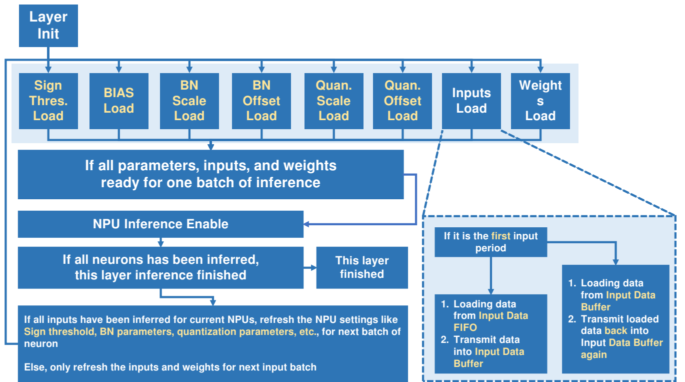

Fig. 4: Workflow in Layer Processing Units (LPUs)

Initialization, *Neuron Initialization*, and *Neuron Processing*:

- ***Layer Initialization***: LPU receives *Layer Setting Data* firstly to initial the layers, including layer type (*Input Layer*, *FC Layer*, *Output Layer*), activation type (*Sign, Sigmoid, tanh, ReLU, Multi-Thresholds*), BN folding option, neuron input precision, weight precision, neuron output precision, neuron number, neuron input length.
- ***Neuron Initialization***: According to the initialized information of neurons, LPU loads the required neuron inputs and parameter data from *Buffer Cluster* into TNPUs, such as the BN parameter, Sign/Multi-thresholds, and QUAN parameter.
- ***Neuron Processing***: This step loads weights from *Buffer Cluster* and executes the neuron inference processing in TNPUs. LPU repeats this step until all weights are processed. The loaded neurons in TNPU finish the inference. If all neurons in the current inferring layer have finished, LPU sets the finished signal. If not, LPU turns to ***Neuron Initialization*** step and initials new neurons.

3) Network Processing Unit (NetPU): Figure 2 (left) shows the hardware design in NetPU, which contains a *LPU Cluster*, *NetPU FIFO Cluster*, and *In/Output Control*. According to our above design, TNPU achieves support for different layers, meeting a large number of inputs and various activation. LPU schedules the TNPU to support the inference of most 8192 neurons in one layer, which meets most MLP models. Therefore, as the top control module, the NetPU module controls the LPUs to execute the inference with the *Recycling Layer Structure* to support a very deep MLP model. Figure 2 (right) shows the hardware design of *Recycling Layer Structure*: LPUs connect each other as a loop. The input of inferring dataset is loaded into the first LPU. The outputs of each LPU will be connected to the next LPU as input and an *Output Multiplexer* (OM). When the inference of the entire network model is working on the last layer, OM connects the LPU that infers the output layer to *Network Output FIFO*.

The working flow in NetPU contains the following steps: 1) ***NetPU Initialization***, 2) ***LPU Initialization***, 3) ***LPU Processing***, 4) ***LPU Resetting***.

- ***NetPU Initialization***: In this step, NetPU loads the layer number of the target network model. Then, loading all layer settings from *Network Input FIFO* into a *Layer Setting FIFO*.
- ***LPU Initialization***: NetPU initials the LPUs to execute the first period of processing. First, NetPU loads the target dataset input into the first LPU. Second, all LPUs load the layer setting from *Layer Setting FIFO*. Then, all LPUs load the required parameters to initial their processing layers according to the loading layer setting.
- ***LPU Processing***: All LPUs start processing and loading the weights from *Network Input FIFO*. If one LPU finishes the inference, NetPU checks if there is an unprocessed layer in the target network model. If yes, it turns to the next step. If no, NetPU waits for all LPUs to finish the processing and sets the network model inference finished.
- ***LPU Resetting***: Loading the new layer setting and parameters to reset the finished LPU. Then NetPU turns to step ***LPU Processing*** and continues the inference.

According to the above processing control in NetPU, we can find that the order of data loading can be predicted, which means the data loading order from *Network Input FIFO* is always: *(1)* Layer Number N, *(2)* All Layer Settings, *(3)* Dataset Inputs, *(4)* Parameter of Layer.0, *(5)* Parameter of Layer.1, *(6)* Weights of Layer.0, *(7)* Parameter of Layer.2, *(8)* Weights of Layer.1, ..., *(i)* Parameter of Layer.$(N-2)$, *(i+1)* Weights of Layer.$(N-3)$, *(i+2)* Parameter of Layer.$(N-1)$, *(i+3)* Weights of Layer.$(N-2)$, *(i+4)* Weights of Layer.$(N-1)$. Therefore, if we pre-package all inputs and network models based on the above order, the runtime control of NetPU-M architecture is only the data streaming. Because NetPU loads the outputs of each layer into the on-chip memory (*Layer Input Buffer*), this design requires no additional memory access in the processing until the inference is finished, which can highly simplify the runtime control.

IV. Evaluation

We synthesized and simulated four instances of TNPU and one instance of NetPU-M and measured this NetPU-M implementation with six network models on *Ultra96-V2 Evaluation Platform* (Xilinx Zynq UltraScale+ MPSoC). Table IV and Table V show the simulation and synthesis results of four single TNPUs and the entire NetPU-M with different inference configurations. Therefore, we can summarize some features of NetPU-M architecture:

For TNPU, as shown in Table IV, we synthesized four instances supporting eight pairs of 8-bit activation and weights which can be quantized to $1 \sim 8$ bits precision. Therefore, one TNPU consists of eight XNOR multipliers for binarized inputs (1 bit) and eight integer multipliers for quantized inputs (> 1 bit). Moreover, these TNPU instances can support the runtime reconfiguration of *Sign*, *ReLU*, *Sigmoid*, *tanh*, and *Multi-Threshold* activation functions. Because *Multi-Threshold* activation function requires $2^n - 1$ thresholds for n-bits quantization, which leads to high resource consumption, the four instances we implemented limit the supporting quantization precision as $1 \sim 4$ bits and $1 \sim 8$ bits separately requiring 15 and 255 thresholds for one TNPU. According to the resource utilization on *Ultra96-V2* platform shown on Table IV, *Multi-Threshold* activation is not a suitable activation function for the high-precision quantization. The two TNPU instances supporting the maximum 8-bit quantization consume more than 27% LUTs on *Ultra96-V2* platform. Therefore, for the NetPU-M instances we implemented in this work, we limit the supported quantization precision range of Multi-Threshold activation to $1 \sim 4$ bits. For the other selectable runtime-reconfigurable activation functions, such as ReLU and Sigmoid, the supported precision range is $1 \sim 8$ bits. Furthermore, based on the macro definition and generation block we applied in our Verilog codes, we explored

TABLE IV: Resource Utilization of Single TNPU on Ultra-96 V2

Precision	Activation	Input Number	Input WIdth	Entire Inputs Width	XNOR Mul Number	DSP Mul Number	LUT Mul Number	Max. Multi-Thres. Supported bits	BN Mul Mode	LUTs Utilization		DSPs Utilization		FF Utilization	
										Number	Rate	Number	Rate	Number	Rate
1-8	All	8×2	8×2	64×2	8	8	0	8	DSP	19049	27.00%	16	4.44%	32	0.02%
									LUT	20138	28.54%	12	3.33%	32	0.02%
								4	DSP	2705	3.83%	16	4.44%	32	0.02%
									LUT	3794	5.38%	12	3.33%	32	0.02%
								Total Resource Number:		70560	-	360	-	141120	-

TABLE V: Simulation and Resource Utilization of NetPU-M Architecture on Ultra-96 V2 based on 100MHz Clock

LPU Number	TNPU Num in LPU	Applied Activation	BN Folding	LUTs Utilization		DSPs Utilization		FF Utilization		BRAM Utilization		Inference Latency in Simulation		
				Number	Rate	Number	Rate	Number	Rate	Number	Rate	TFC (63x3)	SFC (256x3)	LFC (1024x3)
2	8	Multi-Thres	Yes	59755	84.69%	256	71.11%	14601	10.35%	129.5	59.95%	172.165us	882.085us	7408.225us
			No									175.805us	895.805us	7462.205us
		Sign	-									38.745us	133.785us	974.745us
		Total Resource Number:		70560	-	360	-	141120	-	216	-			

TABLE VI: Comparison between NetPU-M and FINN

Work	Implementation	Target Platform	Clock (MHz)	Resource Utilization			Precision	Latency (us)					
				LUT	BRAM	DSP		TFC (64X3)	P_{wall} (W)	SFC (256X3)	P_{wall} (W)	LFC (1024X3)	P_{wall} (W)
NetPU	CGM-64	Ultra96-V2	100	66,494	126.5	256	W1A1	44.64	6.94	139.75	6.86	980.63	6.99
							W2A2	178.180	7.05	888.000	6.90	-	-
							W1A2	-	-	-	-	7414.13	6.88
FINN	SFC-max	Zynq7000	200	91,131	4.5	-	W1A1	-	-	0.31	21.2	-	-
	LFC-max			82,988	396	-	W1A1	-	-	-	-	2.44	22.6
	SFC-fix			5,155	16	-	W1A1	-	-	240	8.1	-	-
	LFC-fix			5,636	114.5	-	W1A1	-	-	-	-	282	7.9

the different multiplier computation resource settings in these four TNPU instances. When Verilog macro definitions make the generation block create the BN and MUL submodules shown in Figure 3 applying LUT/DSP-based multipliers, four TNPU instances consume different hardware resources, which can be used to suit different FPGA platforms meeting their resource limitation. In the final applied hardware setting for our NetPU-M instance, BN and MUL submodules apply pure DSP multipliers to balance the LUT resource consumption.

Based on the TNPU setting mentioned above, we implemented a NetPU-M instance for simulation and actual measurement on *Ultra96-V2* platform. Table V shows the synthesis and simulation of this instance. We explored the inference of six pre-trained 1/2bits quantized MLP models from FINN [30] and Brevitas [23] in this NetPU-M instance, *TFC-w1a1, TFC-w2a2, SFC-w1a1, SFC-w2a2, LFC-w1a1, LFC-w1a2*. These models apply the MNIST dataset (28×28 handwritten digit images) [19] and contain one input layer for input quantization, three hidden layers with 64 (TFC), 256 (SFC), and 1024 neurons (LFC), separately, and one output layer based on the *MaxOut* activation. $wnam$ means the weights and activation inputs in this model have been quantized to n and m bits. For the binarized network models (*TFC-w1a1*, *SFC-w1a1*, and *LFC-w1a1*), we configure the NetPU-M instance to apply the Sign activation and folding the BN into the Sign threshold like the BNN models in *FINN* works [30]. For the other network models, we configured the NetPU-M instance working on Multi-Threshold activation and explored the different latency when these models fold the BN layer or not. The simulation results in Table V show that our NetPU-M instance can infer all six network models with different precision, activation functions, and BN folding options, without the hardware regeneration. Binarized network models reduce the inference latency compared with 2-bit quantized network models. Folding the BN layer into thresholds can also speed up the inference.

Furthermore, we measured the actual inference latency of these six models on our NetPU-M instance and compared the results with *FINN* [30] instances. In the actual measurement, the 2-bit quantized network models have been configured as BN folding mode. As shown in Table VI, the measured latency has a slight increase compared to the simulation results, caused by the *DMA* transmission and *Processing System* (PS) control of *Zynq Ultrascale+* system. *FINN* [30] work explored four instances optimized for *SFC-w1a1* and *LFC-w1a1* models. *max* instances consume more resources for higher performance, *fix* instances can highly reduce resource consumption but cause a high inference latency. Compared with four *FINN* instances, our NetPU-M instance can infer all six models in one implementation. And in principle, our current implementation can support a maximum of 4096 neurons in each hidden layer for one quantized MLP model. Moreover, our implementation can use the DSP resource on FPGA to reduce the consumption of LUTs for large instances in a lightweight FPGA platform. As a trade-off between achieving generic support for different MLP models, our implementation has a higher inference latency compared with most *FINN* instances. Moreover, Table VI also listed the power consumption of different models inferring on our instance compared with four FINN instances. The P_{wall} means the measurement results from a wall power meter. Our implementation requires less power than four FINN instances.

V. Further Works

A. Optimization of Hardware Design

Compared with previous works, our design can achieve a high generality, but the NN inference is slower than related works. To extend and optimize our designs, we have four major ongoing improvements for NetPU-M architecture in the current:

- According to the Table V and the analysis in section IV, the bottleneck of parameter loading causes most of the inference latency. Therefore, optimizing the data loading schemes is necessary for NetPU-M to improve the module's efficiency.
- Moreover, we will also optimize the buffer implementation in LPU. For example, when LPU is working in BN folding,

the BN scale and offset buffer is not working, which causes the waste of hardware resource. We are exploring buffer reuse for different network models.

- For the current design, when the precision of inferring network model is 2-8bits, NetPU-M needs to load the data as 8bits for each layer. This means, for example, for 2-bit data, there are 6 bits as the placeholder, which causes the lower data streaming speed and the waste of hardware resources and power consumption. Therefore, in the next step, we will explore the multi-channel schemes for low precision ($< 8bits$) situations to speed up the data loading and computing efficiency.
- NetPU-M architecture supports the inference of MLP models now. Therefore, the following steps of our work will extend the network support range of NetPU-M architecture to meet the acceleration of *CNN*, *ResNet*, *LSTM*, *SNN*, etc.

VI. Summary and Conclusion

Our work explored a new hybrid architecture for generic MLP inference, NetPU-M, based on widely-applied HSD and PEM architecture for the implementation of FPGA-based NN inference accelerators. This architecture achieved the generic support for different network models meeting few resource consumption without hardware regeneration and complex runtime control. The current design supports the 1-8bit quantization and mix-precision model. According to the measurement and simulation, our work can achieve a more generic acceleration of different MLP models with limited hardware resources compared with the implementation in *FINN* [30].

References

[1] Hesham Amin, K Memy Curtis, and Barrie R Hayes-Gill. "Piecewise linear approximation applied to nonlinear function of a neural network". In: *IEE Proceedings-Circuits, Devices and Systems* 144.6 (1997), pp. 313–317.

[2] Myat Thu Linn Aung et al. "DeepFire: Acceleration of Convolutional Spiking Neural Network on Modern Field Programmable Gate Arrays". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 28–32.

[3] Michaela Blott et al. "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–23.

[4] Safa Bouguezzi et al. "An Efficient FPGA-Based Convolutional Neural Network for Classification: Ad-MobileNet". In: *Electronics* 10.18 (2021), p. 2272.

[5] Adrian Bulat and Georgios Tzimiropoulos. "XNOR-Net++: Improved Binary Neural Networks". In: *CoRR* abs/1909.13863 (2019). arXiv: 1909.13863.

[6] Zhaowei Cai et al. "Deep Learning with Low Precision by Half-wave Gaussian Quantization". In: *CoRR* abs/1702.00953 (2017). arXiv: 1702.00953.

[7] Farah Fahim et al. "hls4ml: An Open-Source Co-Design Workflow to Empower Scientific Low-Power Machine Learning Devices". In: *Research Symposium on Tiny Machine Learning*. 2021.

[8] Farah Fahim et al. "hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices". In: *CoRR* abs/2103.05579 (2021). arXiv: 2103.05579.

[9] Julian Faraone et al. "Syq: Learning symmetric quantization for efficient deep neural networks". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4300–4309.

[10] Jeremy Fowers et al. "A Configurable Cloud-Scale DNN Processor for Real-Time AI". In: *Proceedings of the 45th International Symposium on Computer Architecture, 2018*. ACM, 2018.

[11] Yury Gorbachev et al. "OpenVINO deep learning workbench: Comprehensive analysis and tuning of neural networks inference". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 2019, pp. 0–0.

[12] Giuseppe Di Guglielmo et al. "Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML". In: *CoRR* abs/2003.06308 (2020). arXiv: 2003.06308.

[13] Kaiyuan Guo et al. "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.1 (2018), pp. 35–47.

[14] Gustafson and Yonemoto. "Beating Floating Point at Its Own Game: Posit Arithmetic". In: *Supercomput. Front. Innov.: Int. J.* 4.2 (June 2017), 71–86.

[15] John L Gustafson. "Posit arithmetic". In: *Mathematica Notebook describing the posit number system* 30 (2017).

[16] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018.

[17] Vinod Kathail. "Xilinx Vitis Unified Software Platform". In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, 173–174.

[18] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[19] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[20] Yuhao Liu et al. "NetPU: Prototyping a Generic Reconfigurable Neural Network Accelerator Architecture". In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. 2022, pp. 1–1.

[21] Jian Meng et al. "FixyFPGA: Efficient FPGA Accelerator for Deep Neural Networks with High Element-Wise Sparsity and without External Memory Access". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 9–16.

[22] Suresh Nambi et al. "¡italic¿ExPAN(N)D¡/italic¿: Exploring Posits for Efficient Artificial Neural Network Design in FPGA-Based Systems". In: *IEEE Access* 9 (2021), pp. 103691–103708.

[23] Alessandro Pappalardo. *Xilinx/brevitas*.

[24] Mohammad Rastegari et al. "Xnor-net: Imagenet classification using binary convolutional neural networks". In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[25] Vladimir Rybalkin et al. "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs". In: *2018 28th international conference on field programmable logic and applications (FPL)*. IEEE. 2018, pp. 89–897.

[26] Hardik Sharma et al. "From high-level deep neural models to FPGAs". In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pp. 1–12.

[27] Jiang Su et al. "Accuracy to throughput trade-offs for reduced precision neural networks on reconfigurable logic". In: *International Symposium on Applied Reconfigurable Computing*. Springer. 2018, pp. 29–42.

[28] Salim Ullah et al. "High-Performance Accurate and Approximate Multipliers for FPGA-based Hardware Accelerators". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), pp. 1–1.

[29] Yaman Umuroglu and Magnus Jahre. "Streamlined Deployment for Quantized Neural Networks". In: *CoRR* abs/1709.04060 (2017). arXiv: 1709.04060.

[30] Yaman Umuroglu et al. "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.

[31] Yaman Umuroglu et al. "High-throughput dnn inference with logicnets". In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2020, pp. 238–238.

[32] Yaman Umuroglu et al. "LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications". In: *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2020, pp. 291–297.

[33] Ying Wang et al. "DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family". In: *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2016, pp. 1–6.

[34] Chen Wu et al. "MP-OPU: A Mixed Precision FPGA-based Overlay Processor for Convolutional Neural Networks". In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE. 2021, pp. 33–37.

[35] Jiwei Yang et al. "Quantization Networks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[36] Ming Zhang, Stamatis Vassiliadis, and Jose G. Delgado-Frias. "Sigmoid generators for neural computing using piecewise approximations". In: *IEEE transactions on Computers* 45.9 (1996), pp. 1045–1049.

[37] Gaofeng Zhou, Jianyang Zhou, and Haijun Lin. "Research on NVIDIA Deep Learning Accelerator". In: *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 2018, pp. 192–195.