T-DLA: An Open-source Deep Learning Accelerator for Ternarized DNN Models on Embedded FPGA

Yao Chen[1], Kai Zhang[1,2], Cheng Gong[2], Cong Hao[3], Xiaofan Zhang[3], Tao Li[2], Deming Chen[3]
[1]Advanced Digital Sciences Center, Singapore, [2]Nankai University, Tianjin, China
[3]University of Illinois at Urbana-Champaign, IL, USA
yao.chen@adsc-create.edu.sg, {kai.zhang,cheng-gong}@mail.nankai.edu.cn
litao@nankai.edu.cn,{congh, xiaofan3, dchen}@illinois.edu

***Abstract*—Deep Neural Networks (DNNs) have become promising solutions for data analysis especially for raw data processing from sensors. However, using DNN-based approaches can easily introduce huge demands of computation and memory consumption, which may not be feasible for direct deployment onto the Internet of Thing (IoT) devices, since they have strict constraints on hardware resources, power budgets, response latency, and manufacturing cost. To bring DNNs into IoT devices, embedded FPGA can be one of the most suitable candidates by providing better energy efficiency than GPU and CPU based solutions, and higher flexibility than ASICs. In this paper, we propose a systematic solution to deploy DNNs on embedded FPGAs, which includes a ternarized hardware Deep Learning Accelerator (T-DLA), and a framework for ternary neural network (TNN) training. T-DLA is a highly optimized hardware unit in FPGA specializing in accelerating the TNNs, while the proposed framework can significantly compress the DNN parameters down to two bits with little accuracy drop. Results show that our training framework can compress the DNN up to $14.14\times$ while maintaining nearly the same accuracy compared to the floating point version. By illustrating our proposed design techniques, the T-DLA can deliver up to 0.4TOPS with 2.576W power consumption, showing $873.6\times$ and $5.1\times$ higher energy efficiency (fps/W) on ImageNet with Resnet-18 model comparing to Xeon E5-2630 CPU and Nvidia 1080 Ti GPU. To the best of our knowledge, this is the first instruction-based highly efficient ternary DLA design reported from the literature.**

I. Introduction

Deep neural networks (DNNs) are becoming attractive solutions for many machine learning applications. Their capability of effective feature extraction for raw data from sensors on IoT devices is a big advantage of DNN based algorithms. Meanwhile, with the increase of the size and complexity of neural networks, deploying a DNN with a large number of parameters and complex data transmission on a small and low power device becomes increasingly difficult [1], [2].

Generally, floating point data are adopted to ensure the accuracy of the data representation during the processes of DNN training. During DNN inference, however, it shows that accuracy is less sensitive to data representation [3], [4]. In order to implement DNNs on IoT devices, a number of DNN model compression methods are proposed [1], [5], [6]. DNN model compression with pruning is an effective way to reduce the number of weights in a DNN model as well as the data bit-width in a DNN model, with an extreme case that discretized the weights down to binary. These methods can dramatically reduce the network size as well as number of the multiplications in the kernel computation. However, they also can cause significant degradation on the accuracy of the output of the models especially when the data in the DNN models are binarized. In this work, we focus on DNNs with ternarized weights. Our hypothesis is that ternarized DNNs have a great potential to maintain high accuracy while still being compact in size with high execution speed and energy efficiency.

Besides general purpose processors (GPP) and digital signal processors (DSP), FPGA is becoming an attractive platform to achieve efficient DNN processing [7]–[9]. Especially modern SoC FPGA contains on-chip low power processor and sufficient interfaces that support the widely used IoT sensors. FPGA also provides the flexibility to be configured as Domain Specific Architecture (DSA) design that enables realization of diverse DNN models for applications. Several design flows for embedded FPGA based Deep Learning Accelerator (DLA) are proposed, such as TVM and CHaiDNN [10], [11]. However, the advantage of the finer granularity logic control of FPGA and low bitwidth is not well explored in these previous works. To enable a low-bitwidth DLA design while maintaining high accuracy of the DNN models running on the DLA would require a deep fusion of the DNN training with the specific DLA design, which in general can be a very challenging task.

In this paper, we propose a highly efficient T-DLA especially designed for terarized DNN models, together with a DNN model training framework for a complete system solution. Our solution provides training and quantization of the input DNN model into ternarized weights with quantized activations, and also provides a specific DLA for the execution on the targeted embedded FPGA platform. To summarize, the detailed contributions of this work are:

- Caffe based open-source flow for scalable and effective coefficient ternarization and quantization for DNN training which achieves floating-point free DNN inference.
- A fully customizable and inner module pipelined DLA architecture with specialized instruction set for DNN acceleration on embedded FPGAs.
- Multi-clock domain and pipelined adder tree design to further explore the low-power consumption and high energy-efficiency potential of the embedded FPGA platform.
- The T-DLA system delivers up to 0.4TOPS with 2.576W power consumption and shows $873.6\times$ and $5.1\times$ energy efficiency on ImageNet with the Resnet-18 model comparing to Xeon E5-2630 CPU and Nvidia 1080 Ti GPU.

The rest of this paper is organized as follows. Section II

978-1-7281-3391-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ISVLSI.2019.00012

presents the ternary quantization method during DNN model training. Section III explains the overall T-DLA architecture and its individual modules in detail. Section IV shows the experimental results and followed by conclusions in Section V.

II. Training For Ternary Quantization

The design of a high performance and energy-efficiency DNN accelerating system involves both model training and hardware implementation [4]. In order to achieve a high performance system with constrained hardware resource, we start from the training of the DNN model but with careful consideration of the FPGA implementation requirements. One key advantage of FPGA is its full support of flexible bit-width control and customization. Thus, during the training, we compress all the weights of convolutional layers and as much as the fully connected layers in the model into ternary representation, and quantize all the other parameters for the rest of the layers or operations into fixed bit-width representations to achieve a low bitwidth and floating-point free inference of the DNN model.

A. Weight Approximation

We adopt the same threshold-based weight approximation method proposed in [5] to compress the weight w^f from floating point to ternary weight $w^t(\{-\alpha, 0, \alpha\})$ with a scaling factor α:

$$w^t = \begin{cases} \alpha & : w^f > \Delta \\ 0 & : |w^f| \leq \Delta \\ -\alpha & : w^f < -\Delta \end{cases} \tag{1}$$

where $\pm\Delta$ are symmetric thresholds. The value of Δ and α are approximated as:

$$ternarize : \begin{cases} \Delta = 0.7E(|w^f|) \\ \alpha = E_{i \in \{i||w^f(i)|>\Delta\}}(|w^f|) \end{cases} \tag{2}$$

B. Scale factor and Activation Quantification

Although the existing work has trained the weights of convolutional layers into ternary, there is still a requirement of α value to be floating point to remain the accuracy of the processing results [5]. In order to achieve a floating point free implementation of the DNN model on IoT devices, we propose a dynamic quantification method to compress the scaling factor and the output of activation functions into lower bitwidth.

Any floating point number f can be expressed as:

$$f = (-1)^s \cdot 2^{-p} \cdot Q \tag{3}$$

where s is the sign, p is the resolution of the data, and Q is the quantized integer value. With a sufficient number of bits, the floating point number f can be represented by an integer Q and its point position p and sign s precisely.

In fact, the number of bits is always limited. In order to minimize the loss, it is most desirable to retain the front bits. Especially for neural networks, larger values play a more important role than smaller ones [1], [12], so it is appropriate to maintain the front bits:

$$\begin{cases} p = \lfloor \log_2(|f|) \rfloor - (b-1) \\ Q = Round(\frac{f}{2^p}) \end{cases} \tag{4}$$

where b is the maximum number of bits that are limited, and $Round()$ is the rounding function. As a single floating point value, α in Eq. (2) can be quantized according to Eq. (4).

The output of the activation layer is an array, denoted as A. It is necessary to find a shared point position p that minimizes the total quantization error:

$$p^*, Q^* = \arg\min_{p,Q} \|Q - A\|_2^k \tag{5}$$

where k is a positive integer, and the cost function in Eq. (5) is designed to illustrate that larger values in the output data are more important and will be kept. Show as Eq. (6), the bit location stands for the most value that should be reserved.

$$quantize : \begin{cases} p = \lfloor \log_2(\max(|A|)) \rfloor - (b-1) \\ Q = Round(\frac{A}{2^p}) \end{cases} \tag{6}$$

C. Model Training

In forward propagation of layer l, we first ternarize w_l^f to w_l^t and quantize α_l at the same time. Then, we use w^t and the quantized activation Q_{l-1} of the previous layer to calculate the activation value A_l. Finally, we get the quantized activation Q_l of the current layer from A_l and forward it to the next layer.

During back propagation, because the values from Eq. (2) are not differentiable, derivatives of w^t are computed instead, yielding the identity function:

$$g = \frac{\partial J(w^t, Q)}{\partial w^t} = \frac{\partial J(w^t, Q)}{\partial w^f} \tag{7}$$

To deploy a DNN model on any device for inference, we only need to save the ternary-valued weights and the scaling factors together with other quantized network parameters.

III. T-DLA System Design

To optimally explore the computation capacity with the trained ternary weighted DNN model, we propose T-DLA, a systematic solution in consideration of both performance and resource limitation, to efficiently execute the computation intensive operations with ternarized weights in DNN inference. The key features of the proposed T-DLA system are:

- Pre-defined customizable high performance components that execute based on the specialized instruction set to support various operations for an DNN model.
- Contains instruction controlled variable-length line buffer for lower latency.
- A highly optimized computation unit which contains a ternary computation array and a fully pipelined adder tree.
- All the components inside the accelerator are fully pipelined.

In order to provide enough flexibility for system integration, the T-DLA is designed with memory interfaces for both data and instructions, which eases the control and integration with an embedded ARM core through a general memory mapped bus system.

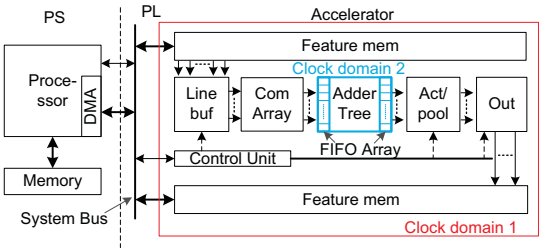

Fig. 1: Accelerator and system architecture.

A. Overall Architecture

As shown in Figure 1, the T-DLA system is constructed with 1) a memory sub-system that contains a feature memory and line buffer, 2) a computation core that is constructed with an computation array and an adder tree, 3) an activation and pooling logic unit, 4) an output transfer module and 5) the corresponding on-the-fly control unit. The connection to the processor is presented in this figure to show the flexibility of integration of our T-DLA with embedded processors. The input and output data ports are mapped to different address space in the bus system and the DMA through the bus system is enabled. The accelerator works in an instruction mode and receives instructions from its instruction register in the control unit. The input and output feature interfaces are designed as general bus interfaces that could be easily connected to a system bus. The DMA is controlled by the embedded processor core to provide control flexibility at runtime.

B. Memory Sub-system

The T-DLA contains two levels of cache-like memories: feature memory for the feature data storage, and a flexible variable-length line buffer to feed the feature data to the computational array in parallel.

a) Feature memory: The T-DLA has its own local feature memory to enable the data to be streamed to it from the external memory and temporarily store the input and output feature data. Both the input and output data transfers are double buffered. The feature memory is designed using dual-port BRAM. The depth of the BRAM is set as a parameter in the feature memory template and is configured according to the on-chip memory capacity during the system integration.

b) Variable-length line buffer: Due to the computation pattern of convolution, the data in the feature memory need to be provided to the computation array in a shifted and paralleled manner. Line-buffer is a general way to read and provide the data to the paralleled computation module. However, because of the various sizes of kernels and output features that are required by a single CNN, a simple line buffer with a fixed output size and buffer depth could not satisfy the requirement of different layers. To solve this problem, the line buffer unit is designed with a variable kernel size K_{lbuf} and variable depth D_{lbuf} which are controlled by the kernel size logic and buffer depth logic, as shown in Figure 2. The control logic is implemented as a switch. With the value passed to the control register, the corresponding shift lines are connected and output channels are selected. The maximum supported kernel size L_K

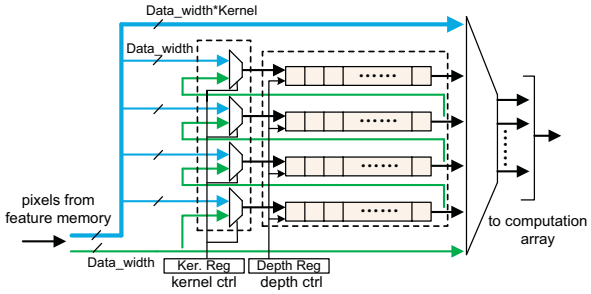

Fig. 2: Variable-length line buffer.

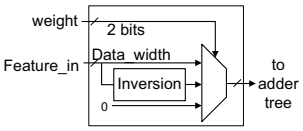

Fig. 3: Ternary computation unit.

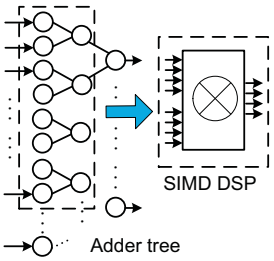

Fig. 4: Adder tree.

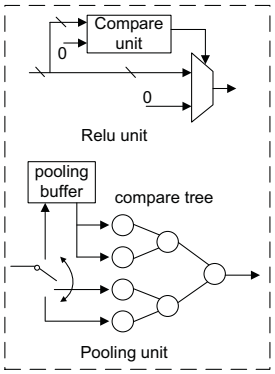

Fig. 5: Relu and pooling unit.

and depth of the line buffer L_D are set to be fixed and are configured with the consideration of both the on-chip resource capacity and the input DNN model.

C. Computation Modules

The computation of convolution and pooling in our T-DLA are illustrated by 1) a ternary computation array, 2) an adder tree, 3) an activation unit and 4) a pooling module. The control logic extracts the instruction and control the listed modules.

1) Ternary computation array: With our TNN model training, the weights are represented with 2 bits using two's complement encoding, thus the computation unit in the convolution layer is simplified to a selection and inversion logic as shown in Figure 3. Due to the simplified logic, the number of the LUT and FF consumption for each computation module is the same and formulated as Equ. 8. The required maximum feature data width D_w is extracted during the training in Section II.

$$N_{LUT} = N_{FF} = D_w \tag{8}$$

The entire computation array is constructed by a group of computation modules. Benefiting from the resource efficiency of the computation unit, we extend the parallelism on input channel, output channel and kernel dimension, hence the computation array is constructed by $T_n \times T_m \times L_K^2$ computation units, which could process these numbers of input data simultaneously. T_n, T_m represents the maximum input and output channel number of our T-DLA for a single call of the computation array. The configurations of T_n, T_m, L_K are

based on the on-chip resource availability. The data processing in the computation array takes place in a single clock cycle then all the results are output to the following adder tree unit.

2) Pipelined adder tree: With the previous computation array, $T_n \times T_m \times L_K^2$ number of processed data are provided to the following adder tree in every clock cycle. Because the computation array is constructed with the ternary computation units which only consume LUTs and FFs. This allows us to use DSP to construct the adder tree. In order to process the data provided from the computation array in a streaming manner, two methods have been utilized to the DSPs in the adder tree: 1) input data squeeze and 2) independent clock-domain setting.

a) Data squeeze: A DSP in FPGA chip could be configured to Single Instruction Multiple Data (SIMD) mode for addition operation. The input/output of modern DSPs in an FPGA chip can be split into smaller data segments in SIMD mode where the internal carry propagation between segments is blocked to ensure independent operation for all segments. Our TNN training process allows us to constrain the features to be less than 12 bits. In order to incorporate the advantage of our TNN-specific training solution, we split the original 48 bits input of the DSP into 4 independent accumulation channels to achieve a higher throughput for the data processing, as is shown in Figure 4. Thereby, a single DSP completes addition operations for 8 pieces of input data and provides 4 outputs. With the advantage of SIMD mode, the DSP provides output results in every single clock cycle after the internal register lines are fully filled up.

b) Clock-independent design: Generally, DSP units could be configured to a higher clock frequency compared to other resources such as IO and LUT . In order to fully explore the efficiency of the DSP, the adder tree unit is designed with a separate clock input. Thereby, the input and output of the adder tree are buffered with asynchronous FIFO arrays. The FIFOs in the input FIFO array for the adder tree are configured with a slower input clock but wider input data width, and with a faster output clock with narrower output data width. The FIFOs in the output FIFO array are configured as the opposite. In this way, the throughput of the input/output FIFOs can be balanced and the DSP units can run at a high clock frequency. The number of the FIFOs and their depths are configured during the system integration based on the final implementation clock frequency and the on-chip resource limitation. Our clock-independent design could enable the DSPs to operate at a higher frequency and throughput without the impact of other slower logic components.

3) Activation and pooling unit: A ReLU based activation module applies the non-linear activation function to the output data from the previous layer. A Max Pooling module is designed to utilize buffers to apply a specific 2×2 sliding window to the input data and outputs the maximum value. The activation and pooling modules are shown in Figure 5.

D. On-the-fly Control Unit and Instruction Set

In each data processing stage, the control unit decodes the input instruction, generates the control signals and passes the variables in the instruction word to each of the variable registers for different modules. The control signals to different modules are listed as follows.

TABLE I: Instruction word

Bits	63:57	56:49	48:47
Description	Reserved	Feature size	48 - out buffer sel.
			47 - in buffer sel.
Bits	46:4		3:0
Description	46:31 - w_mem_addr		3 - Output
	30:23 - s_mem_addr		2 - Activation
	22:7 - cycle counter		1 - Pooling
	6:4 - kernel size		0 - line buffer

- **In/out buffer select** indicates the current activated buffer for the input/output feature buffer.
- **Compute enable** is the start signal for the line buffer to start reading data from the feature memory and stream the data to the computation array.
- **Activation/pooling enable/bypass** is used to enable/bypass the activation/pooling unit.
- **Output select** indicates the output data is available and starts the data output logic to transfer data out from the buffer of the T-DLA.

The computation array and adder tree are driven by the input data from the line buffer, so there is no control signals required by the computation array and the adder tree.

To ease the task scheduling for the application running on the embedded processor, the T-DLA is designed to operate based on a small set of specialized instructions. Corresponding control signals and variables for different function units in the accelerator are generated and transferred based on the instruction opcode. They are designed to be as simple as possible but are able to express the operations in a large varieties of CNNs. Each instruction is defined as a 64-bit word and the format is shown in Table I. The detailed control bits and variables are customized based on the input DNN model according to the accelerator tasks.

E. Inter Accelerator Pipeline

The feature memory is double buffered to enable an efficient data transfer from the host to the accelerator. The computation array and the adder tree are driven by the data from the line buffer. It takes only one clock cycle for the data to be processed and sent to the input FIFO array of the adder tree, which is the data input stage. The output FIFO array of the adder tree buffers the results for the following modules to process. The following activation and pooling unit also contains their own buffers to store the output and the start of the unit is controlled by the status of the output FIFO of adder tree, which is the activation and pooling stage. The output unit processes the output function when the output signal is enabled, which is the data output stage. The three stages in the accelerator are designed to run in a pipelined manner with properly pre-defined buffer sizes.

IV. Evaluations

To evaluate the effectiveness of our proposed solution, we provide the detailed evaluation results from the DNN model training to the final system implementation.

A. Experimental Settings

We choose the most popular and representative datasets for evaluation: MNIST, Cifar10 and ImageNet. The corresponding DNN models that are trained and tested on these data sets are Lenet-5, Cifarnet, a VGG-like network model that contains 64 channels [13] and Resnet-18. For fair comparison, we follow the same training and testing processes. The ternary training and quantization algorithms are implemented in C++ and merged into the original Caffe [14] flow. The measurement of accuracy and frame per second (fps) of the original Caffemodel is on a server with two Intel Xeon E5-2630 v3 CPU and an Nvidia 1080 Ti GPU. The accelerator system for the DNN models are implemented on a Xilinx Zedboard FPGA platform that is suitable for edge applications with very limited logic resources. It is equiped with a Xilinx Zynq-7000 SoC chip XC7Z020-CLG484-1 that contains an on-chip dual-core ARM Cortex A9 together with 53.2K LUTs, 106.4K FFs, 140 BRAM blocks with 36Kb and 220 DSPs. Vivado System Design Suite 2018.1 is used for system implementation.

B. Training Performance

We first evaluate our training flow in terms of classification accuracy and model size reduction, and compare the results to the original 32-bit floating point ones.

1) Classification accuracy: The classification accuracy of our tested models on different data sets are shown in Table II. For simplicity, we only show the top-1 accuracy. Comparing to floating point (Floating in the Table), the classification accuracy under ternary weight and quantified scale and activation shows a very small degradation. We also show comparable accuracy comparing to several recent works [4], [5], in which only weights are ternarized but the quantization of scale factor and activation are not applied. Our proposed method shows better accuracy for Resnet-18 on ImageNet data set, which is a larger network and a larger data set with higher image resolution. This result demonstrates the scalability and stability of our training method. Our training and quantization enable the DNN model to process the computation intensive convolution operations with multiplication free logic while maintaining the classification accuracy.

2) Model size reduction: Our method also greatly reduces the memory footprint (Mem. Reduc.) of the DNN models, as shown in Table II. Ternary weight occupies only 2 bits but the original floating point data requires 32 bits memory space. Our training solution enables the original DNN models to be compressed into much smaller size, and therefore they could be implemented on IoT devices with limited storage capacity. As shown in Table II, for convolution layers, our training method could compress the parameter size to the theoretical limit (16× reduction). For fully connected (FC) layers, it is observed in the experiments that the last FC layer greatly affects the accuracy, so we apply 12-bit fixed point quantization on the last FC layer instead of ternarization. Hence, the networks with less or no FC layers have better compression rate, such as Cifarnet and Resnet-18. Our training solution reduces up to 92.93% (14.14×) of the size of Resnet-18 with floating point data. The quantified scale and activation further reduces the bitwidth of the feature data. As a result, the ternary weights together with the quantization method also reduces the data transmission latency and power consumption during the model inference due to reduced size of both the weight and the feature data.

TABLE II: Training Evaluations

Dataset	MNIST	CIFAR-10	CIFAR-10	ImageNet
Model	Lenet-5	Cifarnet	VGG-like	Resnet-18
Top-1 Classification Accuracy				
Floating	**99.41**	**80.54**	89.24	65.44
Ours	99.2	78.7	89.08	**65.6**
[4]	98.33	-	87.89	-
[5]	99.35	-	**92.56**	61.8
Model Size				
Param. Total (M)	0.43	0.279	5.35	11.69
Param. Conv (M)	0.025	0.258	1.114	11.177
Floating (MB)	1.644	1.065	20.408	44.594
Ours (MB)	0.393	0.081	4.284	3.154
Mem.Reduc.(%)	76.09	92.39	79.01	92.93

C. Hardware Resource and Power Evaluation

After the DNN model is trained, we configure the T-DLA system to execute model inferences. Two representative accelerator configurations and the corresponding system resource utilization, power consumption and execution performance are shown in Table III. We only show the most important accelerator configuration parameters (Acc. Config. Par.) which are T_n, T_m, L_K, L_D and the quantized feature data width D_w as they have been mentioned in the above sections. The feature data width of the model is decided during the training process and the accelerator could be configured based on it. We only show the resource utilization of the feature data customized as 12-bit and 8-bit for the accelerator to simplify the presentation.

As is shown in Table III, using parameterized component template, our T-DLA could be easily customized by different configuration parameters. The LUT resource is dominated by the feature data width D_w since our computation array constructed with ternary computation unit is the most LUT and FF consuming module and each of them costs D_w numbers of LUT and FF. The adder tree is the most critical component that requires DSP resources. When we scale up the maximum output channel number to 16, 91.82% of the on-chip DSPs are utilized. Our targeted platform supports a highest frequency at **250MHz** for the customizable logic including DSPs. Our design could fully utilize it with different customizations without any timing issues by taking advantage of our ternary computation array and the clock independent design for the DSPs, hence fully boosting the execution capacity of the platform.

Although we test different customizations on the same platform, the T-DLA could be easily migrated to different platforms with different resource capacities. The T-DLA could scale to different sizes based on the resource capacity of the platform, and the peak performance is decided by both the supported frequency and the resource capacity of the platform.

D. Performance Comparison

We compare the performance of our system in terms of accuracy, frame per second and power consumption to other

TABLE III: Accelerator Resource and Performance

Acc. Config. Par. $< T_n, T_m, L_K, L_D, D_w >$	$< 4, 8, 5, 32, 12 >$				$< 4, 16, 5, 32, 8 >$			
Res. Util. $< LUT/FF/BRAM/DSP >(\%)$	78.71 / 37.76 / 75.00 / 49.55				71.28 / 47.47 / 68.93 / 91.82			
Clock Freq. Logic / Adder (MHz)	125 / 250				125 / 250			
Power(Watt)	2.276				2.576			
Peak Performance (GOPS)	200				400			
DNN Model	Lenet-5	Cifarnet	VGG-like	Resnet-18	Lenet-5	Cifarnet	VGG-like	Resnet-18
fps (images/S)	53498.8	8501.5	230	10.3	62051.1	15792.8	457	20.48
uJ/image	42.5	267.7	9.9×10^3	0.22×10^6	41.5	163.1	5.6×10^3	0.13×10^6

TABLE IV: Comparison with State-of-the-art implementations.

Dataset	Design	Model	Accuracy(%)	Fea. quan.	W. quan.	fps	Power(W)	fps/W	platform
MNIST	[15]	MFC-max	97.69(2.31)	1bit	1bit	6238000	11.3	**552000**	ZC706
MNIST	[16]	Lenet-5	-	8bit	3bit	70000	4.98	1405.6	ZC706
MNIST	**Ours**	Lenet-5	**99.2**	8bit	2bit	62051.1	2.576	24088.2	Zedboard
CIFAR 10	[15]	VGG-like	80.1(19.9)	24bit	1bit	21900	3.6	6080	ZC706
CIFAR 10	[17]	VGG-like	81.8(18.2)	1bit	1bit	420	2.3	182.6	Zedboard
CIFAR 10	[13]	VGG-like	86.71(13.29)	8bit	2bit	27043	6.8	3976	VC709
CIFAR 10	[18]	VGG-like	88.68(11.32)	1bit	1bit	168	4.7	35.8	Zedboard
CIFAR 10	**Ours**	VGG-like	**89.08**	8bit	2bit	457	2.576	177.4	Zedboard
CIFAR 10	**Ours**	Cifarnet	78.7	8bit	2bit	15792.8	2.576	**6130.7**	Zedboard
ImageNet	[5]	Resnet-18	65.44	FP32	FP32	1.545	85*2	0.0091	Xeon E5-2630 v3
ImageNet	[5]	Resnet-18	65.44	FP32	FP32	387.597	250	1.55	Nvidia 1080Ti
ImageNet	**Ours**	Resnet-18	**65.6**	8bit	2bit	20.48	2.576	**7.95**	Zedboard

designs either with the same DNN model or on the same data set. The results are shown in Table IV. For **MNIST** data set, the design in [15] shows better frame per second (fps) and fps per Watt (fps/W) since the DNN model they implemented is relatively simple and their targeted ZC706 platform has almost $4\times$ more resources in all aspects compared to ours. However, we outperform a similar design [16] with 3-bit weight quantization on the same platform by $17\times$ in terms of fps/W. On **CIFAR10** data set, we report 2 different network models, VGG-like and Cifarnet. Our design shows dominating accuracy advantage among all the VGG-like models. Our Cifarnet implementation shows the best results in terms of fps and fps/W. On **ImageNet** data set, we could not find an implementation of Resnet-18 on FPGA, so we directly compare the results of our system to the floating point version on our servers. Our acceleration system shows a longer processing latency than GPU but outperforms CPU by $9.2\times$. Our accelerator shows better fps/W comparing to CPU and GPU implementations, reaching $873.6\times$ and $5.13\times$, respectively.

V. Conclusion and Future Work

Our T-DLA solution is advantageous in terms of both maintaining high accuracy and high inference performance when compared to other designs. In the future, dynamic fixed point quantization will also be considered. We have implemented our T-DLA with an RTL library which could be synthesized for both FPGA and ASIC. Exploring ASIC implementations is a future task. Finally, our Caffe based DNN model training flow with ternarization and the T-DLA hardware designs will be open-sourced in the near future. Our current release could be found at https://github.com/microideax/T-DLA.git.

VI. Acknowledgment

This work is partly supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, the IBM-Illinois Center for Cognitive Computing System Research (C3SR) - a research collaboration as part of IBM AI Horizons Network. It is also partially supported by the National Natural Science Foundation of China (61872200) and the Natural Science Foundation of Tianjin (18YFYZCG00060).

References

[1] S. Han *et al.*, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv*, 2015.
[2] X. Zhang *et al.*, "Machine learning on FPGAs to face the IoT revolution," in *ICCAD*, 2017.
[3] J. Wang *et al.*, "Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA," in *FPL*, 2018.
[4] H. Alemdar *et al.*, "Ternary neural networks for resource-efficient AI applications," in *IJCNN*, May 2017.
[5] F. Li and B. Liu, "Ternary weight networks," *arXiv*, 2016.
[6] M. Courbariaux *et al.*, "Binarynet: training deep neural networks with weights and activations constrained to +1 or -1," *arXiv*, 2016.
[7] S. Liu *et al.*, "Real-time object tracking system on FPGAs," in *SAAHPC*, 2011.
[8] X. Zhang *et al.*, "DNNBuilder: an automated tool for building high-performance Dnn hardware accelerators for FPGAs," in *ICCAD*, 2018.
[9] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *FPGA*, 2016.
[10] T. Chen *et al.*, "TVM: end-to-end optimization stack for deep learning," *CoRR*, 2018.
[11] Xilinx, "https://github.com/xilinx/chaidnn," 2018.
[12] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.
[13] A. Prost-Boucle *et al.*, "Scalable high-performance architecture for convolutional ternary neural networks on FPGA," in *FPL*, 2017.
[14] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *MM*, 2014.
[15] Y. Umuroglu *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," in *FPGA*, 2017.
[16] J. Park and W. Sung, "FPGA based implementation of deep neural networks using on-chip memory only," *CoRR*, 2016.
[17] H. Nakahara *et al.*, "A fully connected layer elimination for a binarizec convolutional neural network on an FPGA," in *FPL*, 2017.
[18] R. Zhao *et al.*, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *FPGA*, 2017.