3U-EdgeAI: Ultra-Low Memory Training, Ultra-Low Bitwidth Quantization, and Ultra-Low Latency Acceleration

Yao Chen
yao.chen@adsc-create.edu.sg
Advanced Digital Sciences Centre, Singapore

Cole Hawkins
colehawkins@ucsb.edu
University of California, Santa Barbara, USA

Kaiqi Zhang
kzhang70@ucsb.edu
University of California, Santa Barbara, USA

Zheng Zhang
zhengzhang@ece.ucsb.edu
University of California, Santa Barbara, USA

Cong Hao
callie.hao@ece.gatech.edu
Georgia Institute of Technology

ABSTRACT

The deep neural network (DNN) based AI applications on the edge require both low-cost computing platforms and high-quality services. However, the limited memory, computing resources, and power budget of the edge devices constrain the effectiveness of the DNN algorithms. Developing edge-oriented AI algorithms and implementations (e.g., accelerators) is challenging. In this paper, we summarize our recent efforts for efficient on-device AI development from three aspects, including both training and inference. First, we present on-device training with **ultra-low memory** usage. We propose a novel rank-adaptive tensor-based tensorized neural network model, which offers orders-of-magnitude memory reduction during training. Second, we introduce an **ultra-low bitwidth** quantization method for DNN model compression, achieving the state-of-the-art accuracy under the same compression ratio. Third, we introduce an **ultra-low latency** DNN accelerator design, practicing the software/hardware co-design methodology. This paper emphasizes the importance and efficacy of training, quantization and accelerator design, and calls for more research breakthroughs in the area for AI on the edge.

CCS CONCEPTS

• **Hardware → Reconfigurable logic and FPGAs**; • **Computing methodologies → Machine learning**.

KEYWORDS

Edge AI, on-device training, DNN quantization, DNN acceleration

ACM Reference Format:
Yao Chen, Cole Hawkins, Kaiqi Zhang, Zheng Zhang, and Cong Hao. 2021. 3U-EdgeAI: Ultra-Low Memory Training, Ultra-Low Bitwidth Quantization, and Ultra-Low Latency Acceleration . In *Proceedings of the Great Lakes Symposium on VLSI 2021 (GLSVLSI '21), June 22–25, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3453688.3461738

GLSVLSI '21, June 22–25, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8393-6/21/06…$15.00
https://doi.org/10.1145/3453688.3461738

1 INTRODUCTION

Deep neural networks (DNNs) are becoming attractive solutions for many edge AI applications and have made remarkable progress in various areas such as computer vision, natural language processing, health care, autonomous driving, and surveillance. Meanwhile, with the increase of the size and complexity of the neural networks, training and deploying a DNN with a large number of parameters and complex data transmission on small and power-constrained edge devices, such as smart phones and wearable devices, becomes increasingly challenging [12, 14, 41]. In this work, we focus on three primary challenges: **ultra-low memory** training, **ultra-low bitwidth** quantization, and **ultra-low latency** acceleration, and discuss our solutions for each of them.

First, there is an increasing demand for on-device machine learning model training, to preserve data privacy, enable model personalization and lifelong learning, and to improve energy efficiency to avoid the massive data transmission to the cloud [34, 38]. However, model training has a much larger memory requirement than inference, exposing additional challenges for on-device training, where the edge-devices are usually equipped with limited memory capacity. Therefore, **ultra-low memory** training method must be explored to enable on-device training. To this end, we present *an end-to-end low-precision tensorized neural network training framework* with orders-of-magnitude memory reduction [40]. The rank-adaptive tensorized training method employs a Bayesian method for automatic tensor rank determination and model compression in the training process.

Second, to implement DNNs on the memory-constrained edge devices, pruning and quantization are promising to reduce the number of weights and the data bit-width in DNN models, with an extreme case that quantizes the weights down to binary/ternary representations [7, 12, 24]. These methods can dramatically reduce the network size as well as number of the multiplications during the execution of the model. Given the tight memory and computing resource budget on the edge, **ultra-low bitwidth** quantization methods are especially attractive. However, ultra-low bitwidth quantization can easily cause significant degradation on the model accuracy, making such aggressive quantization methods challenging. To address such challenges, we present *a novel ternary weight quantization method by proposing a vectorized loss function*, achieving the state-of-the-art accuracy under the same compression ratio [10].

Third, for efficient DNN deployment on the edge-devices, FPGAs are becoming attractive platforms comparing with CPUs, GPUs

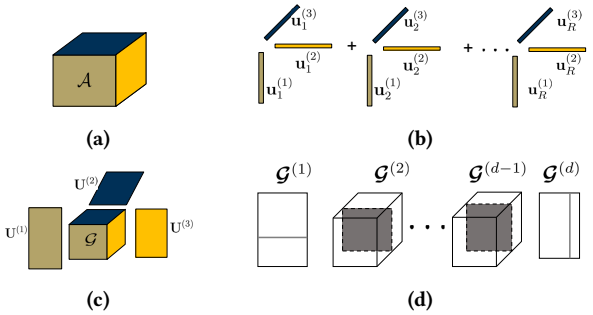

Figure 1: (a): An order-3 tensor. (b) and (c): CP and Tucker representations, respectively. (d): TT representation, where the gray lines and squares indicate a slice of the TT core by fixing its mode index. This figure is reproduced from [15].

and digital signal processors (DSPs) [26, 33, 42]. FPGAs can provide the flexibility to be configured as domain specific architecture that can meet various implementation requirements such as **ultra-low latency** on the edge-devices. In addition, modern SoC FPGAs integrate low power processors and sufficient interfaces that can support widely used sensors for Internet-of-things (IoT) applications. We present *the first instruction based ternarized low-latency deep learning accelerator* with high performance, low resource utilization, and high flexibility for different DNN models [5].

The remaining of this paper is organized as follows. Section 2 introduces our low-memory rank-adaptive on-device training framework; Section 3 introduces our low-bitwidth DNN quantization solution; Section 4 introduces our low-latency DNN accelerator design. In Section 5 we demonstrate the effectiveness of our proposed methods, followed by the conclusions and future work in Section 6.

2 ULTRA-LOW MEMORY TRAINING

The large amount of model parameters consume massive computing and memory resources, which prevents direct training of neural networks on edge devices. A promising technique of reducing model parameters is low-rank tensor decomposition [20, 30]. This method has achieved great success in post-training compression and fixed-rank training [3, 9, 21, 29, 35, 39, 44]. However, several fundamental issues need to be addressed in on-device one-shot training:

- Firstly, a rank-adaptive training framework is needed to avoid combinatorial search of tensor ranks and multiple training runs.
- Secondly, hardware-friendly tensor algorithms should be developed to facilitate their implementation on edge devices.

In this section, we summarize our recent work on the algorithm [15, 16] and hardware [40] levels to address these challenges.

2.1 Bayesian Tensorized Training Models

2.1.1 Low-rank tensor representation. In many cases we can describe a neural network with much less parameters via low-rank tensors. Consider a weight matrix $\mathbf{W} \in \mathbb{R}^{J\times I}$ for example (and other parameters such as convolutional filters and embedding tables can be handled similarly). We can firstly fold $\mathbf{W}$ to a high-dimensional tensor $\mathcal{W}$ of size $J_1 \times \cdots \times J_d \times I_1 \times \cdots \times I_d$, where $I = \prod_{n=1}^{d} I_n, J = \prod_{n=1}^{d} J_n$. Then, we can describe the tensor $\mathcal{W}$ with some low-rank tensor factors $\mathbf{\Phi}$. This can be done with various low-rank tensor decomposition formats as shown in Fig. 1 [15]. In various tensor decompositions, $\mathbf{\Phi}$ denotes the associated tensor factors. For large fully connected layers and embedding tables, the tensor-train matrix (TTM) format turns to be highly effective [15]. In the TTM format, $\mathbf{\Phi} = \{\mathcal{G}^{(n)}\}_{n=1}^{d}$, and each $\mathcal{G}^{(n)} \in \mathbb{R}^{R_{n-1}\times J_n \times I_n \times R_n}$ is an order-4 TTM core. The vector $\mathbf{R} = (R_0, R_1, \cdots, R_d)$ with $R_0 = R_d = 1$ is the tensor ranks that determine the model complexity. With low-rank tensors, one may reduce the number of model parameters from an exponential function of d to a linear one.

2.1.2 Bayesian Tensorized End-to-End Training. Despite the high compression ratio via tensor methods, determining the tensor rank in advance is very hard [17]. This is further complicated by the nonlinear forward model in neural networks, which has prevented tensorized one-shot on-device training in previous works. We have developed two Bayesian models to address this issue:

- **Stein Variational Inference for TTM Format.** In [16], we have considered TTM format. We model each slice of $\mathcal{G}^{(k)}$ with a zero-mean Gaussian prior density. We further control the variance by two tunable Gamma hyper-priors to enforce low tensor ranks. The actual tensor rank is decided jointly by the training data and rank-controlling hyper-parameters. Starting from an initial rank parameter R_k, we can learn an actual rank $\hat{R}_k \leq R_k$, leading to further model compression in the training process. This method uses a Stein variational inference [25] to compute the posteior density for small- or medium-size neural networks.
- **Scalable SVI for One-Shot Tensorized Training.** In [15], we have developed a more generic and efficient Bayesian model for tensorized training. This work can handle CP, Tucker, TT and TTM formats. It uses Gaussian priors to model low-rank tensor factors, and uses Half-Cauchy or Log-Uniform hyper-priors to control tensor ranks. We have improved the stochastic variational inference (SVI) [18] by two steps. Firstly, we simplify the posterior density of rank-controlling hyper-parameters to a Delta function to avoid gradient explosion. Secondly, we use a hybrid numerical/analytical update rule inside SVI. This highly scalable method can perform one-shot training of very large-scale neural networks with billions of model parameters.

2.1.3 Performance Summary.

- Our first method [16] has been tested on a two-layer fully connected neural network, a 6-layer CNN and a 110-layer residual neural network. Our work has produced 7.4× to 137× more compact neural networks directly from the training with little or no accuracy loss.
- Our recent work [15] has been tested on a practical CNN, a large-scale NLP model [19] and an extremely large deep learning recommendation model (DLRM) [28] from Facebook. Orders-of-magnitude parameter reduction has been achieved in the training process. As shown in Table 1, training the DLRM with a standard method involves 4.25×10^9 variables. Our proposed method only trains 2.36×10^6 variables due to low-rank tensorization, and it further reduce the model parameters to 164K in the training process due to the automatic rank determination. The overall parameter reduction ratio in the training process is 2.6×10^4.

Table 1: Performance of our tensorized training [15] on the Facebook DLRM model.

	standard	tensorization	rank-adaptive training
# parameters	4.25B	2.36M	164K
compression	N/A	1,800×	26,000×

2.2 One-Shot On-Device Tensorized Training

To demonstrate on-device training, we have developed a low-precision tensorized training algorithm and its FPGA prototype [40].

2.2.1 Low-Precision Tensorized Training. We consider the maximum a posteriori probability (MAP) estimate of the Bayesian model [15]. In this case, the training loss function includes two parts: the cross-entropy loss of a neural network classifier dependent on TTM factors $\{\mathcal{G}^{(k)}\}_{k=1}^{d}$, and a regularization term caused by the Gaussian priors of TTM factors as well as the Log-Uniform hyper-priors for rank-controlling parameters $\boldsymbol{\lambda}_k$'s. In the training process, both TTM factors and rank-controlling parameters will be computed. To reduce the training cost on hardware, a low-precision tensorized training algorithm is developed based on the following key ideas:

- We use BinaryConnect [8] to compute low-precision TTM factors. BinaryConnect keeps the real values of all low-precision parameters in a buffer. In each iteration, the gradients are accumulated in the buffer, and the low-precision parameters are updated by quantizing the buffer. To handle the non-differentiable quantization function in the training process, we use the straight-through estimator (STE) [2] to approximate its gradient.
- We use different precisions for different variables in the training process. Specifically, we use 4 bits to represent TT factors, 8 bits for activations and bias, and 16 bits for the gradients.

2.2.2 On-FPGA Training. To demonstrate our training algorithms on edge devices, we have implemented an FPGA accelerator as shown in Fig. 2 for the low-precision tensorized training framework.

- Since our low-rank tensorization can greatly reduce the training variables, all model parameters may be stored in the on-chip BRAM. The data samples, activations, and gradients are stored in the off-chip DRAM during the training process.
- The forward and backward propagations are run on the FPGA programmable logic. The TTM factors and rank-controlling parameters are updated on the embedded ARM core.
- Three processing elements (PEs) are designed for the forward and backward propagation. PE1 and PE2 are shared by the forward and backward propagations, and they handle tensor contractions. PE1 is used for a two-index tensor contraction which contains the last dimension of two tensor variables. In contrast, PE2 performs a tensor contraction along a single dimension that is not the last. PE3 computes the outer products in a backward propagation.

3 ULTRA-LOW BITWIDTH QUANTIZATION

Neural network quantization employs low precision (bitwidth) data for efficient model execution. Especially, ultra-low bitwidth quantization leads to much less memory usage, lower complexity of the multiply-accumulate operations, and higher efficiency of model execution, making it an appealing technology for enabling AI at

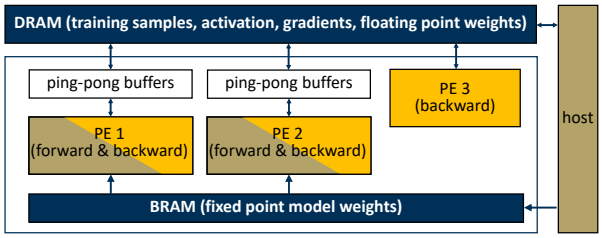

Figure 2: Our FPGA accelerator for the end-to-end tensorized training. Reproduced from [40].

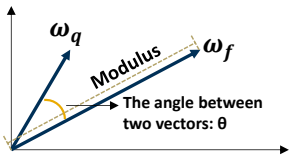

Figure 3: The quantization angle between ω_f and ω_q.

edge devices. However, aggressively lowering the data bitwidth (e.g., lower than 4-bit) is very challenging:

- It can easily result in large accuracy degradation [5, 11, 13], requiring a careful balance between the computing efficiency and the final model accuracy.
- Minimizing the quantization loss, i.e., the L2 distance between the original and the quantized values, is an appealing method [6, 12, 22–24, 37] but have major drawbacks such as easily falling into local optima and neglecting the distribution and correlations of the weights [10].

To address such challenges and achieve high-accuracy ultra-low bitwidth quantization, we have proposed a quantization method, namely **VecQ** [10], with a novel *vectorized loss function* and an open-sourced training flow. VecQ can quantize the model weights into 1-bit to 16-bit and shows exceptional performance especially under ultra-low bitwidth, e.g., ternary values.

Vectorized Loss Function. We organize the weights within one DNN layer into a vector $\omega \in \mathbb{R}^l$ where l is the number of weights, and denote the original floating point weight vector by ω_f and the quantized weight vector by ω_q. Typically, there is a scaling factor α such that $\omega_q = \alpha \boldsymbol{v}$, where each element in $\boldsymbol{v}$ is a low-bitwidth representation. Based on the vector representations, we define the *quantization angle* between ω_f and ω_q, denoted by θ. Figure 3 shows an example when $l = 2$. The objective is to find optimal α and $\boldsymbol{v}$ such that ω_q is as close to ω_f as possible.

We propose the *vectorized loss* to describe the quantization loss, denoted by J_v, and we minimize J_v during the quantization. J_v is defined as the summation of the *orientation loss*, denoted by J_o, and the *modulus loss*, denoted by J_m as follows:

$$\begin{aligned} J_o &= 1 - \cos\theta, \quad \text{where } \cos\theta = \frac{\alpha\boldsymbol{v} \cdot \omega_f}{||\alpha\boldsymbol{v}|| \cdot ||\omega_f||} \\ J_m &= ||\omega_f - \alpha\boldsymbol{v}||_2^2 \\ J_v &= J_o + J_m \end{aligned} \tag{1}$$

The orientation loss describes the angle between two vectors, while the modulus loss describes the squared distance between ω_f and ω_q. Notably, by minimizing J_v, we usually achieve lower quantization

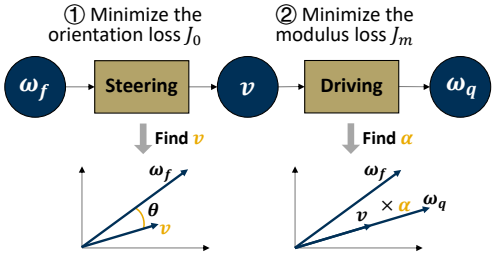

Figure 4: The data quantization flow of VecQ, reproduced from VecQ [10].

Table 2: The format of the instruction word in T-DLA.

Byte Idx	7	6	5	4	3	2	1	0
Load	OP	FS	SAM	SAL	DAM	DAL	KS	CC

[1]OP: operation code [2]FS: input feature size
[3]SAM/SAL: source address most/least significant byte
[4]DAM/DAL: destination address most/least significant byte
[5]KS: kernel size [6]CC: in/out/activation/pooling selection

loss comparing with directly minimizing J_m. More details can be found in the VecQ paper [10].

Vectorized Loss Minimization. We minimize J_v in two steps, namely steering and driving, as shown in Figure 4. First, the steering step minimizes the orientation loss J_o to find the best $\boldsymbol{v}$, since J_o is independent of α. Second, the driving step minimizes the modulus loss J_m to find the best scaling factor α. For a convolution layer, all the weights within the same layer share the same scaling factor; for a depth-wise convolution layer, each kernel has its own scaling factor for a better representation of the less number of weights for it. We quantize the activations to fixed point values during the training to further reduce the memory utilization.

Training Flow Integration. We integrate our VecQ solution into the Tensorflow and Pytorch DNN training frameworks. For each layer, in the forward propagation, we first quantize the weights from ω_f to ω_q, and then use ω_q to compute the output activations, which is also quantized into fixed point. In the backward propagation, the gradients are also calculated using ω_q to update ω_f.

4 ULTRA-LOW LATENCY ACCELERATION

The effectiveness of a dedicated FPGA accelerator for DNN models have been widely demonstrated [4, 33]. However, ultra-low latency accelerators for edge devices with an extremely limited resource budget still require careful design considerations.

Benefiting from our quantization solution VecQ for ultra-low bitwidth, we have proposed **T-DLA**, a light-weight ternarized accelerator overlay under strict resource constraint, to achieve ultra-low latency on edge devices [5]. The key features of T-DLA include:

- An optimized and expressive single instruction multiple data (SIMD) instruction set.
- A novel memory sub-system supporting effective data access of the computation modules.
- An efficient execution pipeline with low-latency computation modules.

SIMD Instruction Set. To support the task scheduling for various DNN models, the instruction set of T-DLA is designed as simple

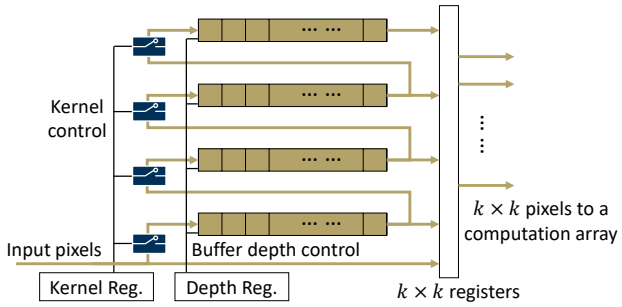

Figure 5: The variable-length line buffer. The kernel size and the depth of the buffers are both configurable.

yet expressive enough for a large variety of DNNs. Each instruction is a 64-bit word (8 bytes) with the format shown in Table 2. The payloads of the different bytes are generated according to the layer configurations.

Memory Sub-system. The memory subsystem contains two levels of storage to provide low latency data fetching to the computation units: a simple input buffer and a variable-length line buffer. The simple input buffer is a BRAM buffer for temporary input feature storage; the variable line buffer serves for the efficient data streaming into the ternary computation array, as shown in Figure 5. It is designed to support variable kernel size k and variable buffer depth d, which are specified by the instruction to reduce the data transmission latency caused by fixed hardware paths. Once configured by the instruction, it provides an output of $k \times k$ data each clock cycle to the computation array.

Execution Pipeline and Computation Modules. T-DLA has four major computation modules: 1) a ternary computation array, 2) a set of adder trees, 3) activation and scaling modules, and 4) pooling modules.

Ternary computation array. With our ternarized model training via VecQ, the weights are represented by 2 bits using two's complement encoding, so that the multiplication in the convolution layer is simplified to selection and inversion logic. Benefiting from such simplified logic, we can achieve parallelism along the input channel, the output channel, and the kernel dimensions. The computation array is constructed by $T_n \times T_m \times L_K^2$ computation units, which can process this number of input data simultaneously. T_n and T_m are the maximum numbers of the input and output channel that can be processed by the computation array, and L_K is the pre-defined maximum allowable kernel size. The values of T_n, T_m, L_K and the length of the line buffer L_D are all configurable and could be determined based on the on-chip resource availability.

Adder tree. Since the computation array is built only using LUTs and FFs, we use DSPs to construct adder trees. We take advantage of the SIMD mode of the DSPs where the internal carry propagation between segments is blocked to ensure independent operation. Therefore, we split the 48-bit input of a DSP into four 12-bit independent accumulation channels, so that a single DSP can perform addition for 8 pieces of input data and provide 4 outputs. Benefiting from the SIMD mode, the DSPs can provide outputs in every single clock once the internal register lines are filled up. Furthermore, the clock frequency of the DSPs are configured to be higher than

Table 3: The on-device ultra-low memory training method on the Fashion MNIST dataset.

Method	Training accuracy	Testing accuracy	Model parameters	Memory in bits	Memory reduction
Vanilla	95.75%	89.27%	4.67×10^5	1.49×10^7	N/A
Floating, w/o prior	92.54%	88.03%	1.48×10^4	4.74×10^5	31.4×
Fixed, w/o prior	88.31%	86.67%	1.48×10^4	6.13×10^4	243×
Floating, w/ prior	90.17%	87.88%	1.08×10^4	3.46×10^5	43.1×
Fixed, w/ prior	85.45%	84.86%	1.22×10^4	5.11×10^4	292×

Table 4: VecQ: top-1 classification accuracy

Dataset	MNIST	CIFAR10	CIFAR10	ImageNet
Model	Lenet-5	Cifarnet	VGG-like	Resnet-18
Floating	99.41	80.54	93.49	69.60
IJCNN'17 [1]	98.33	-	87.89	-
NIPS'16 [24]	99.35	-	92.56	61.8
Ours	99.5	78.7	92.94	68.23

Table 5: VecQ: model size reduction

Model	Lenet-5	Cifarnet	VGG-like	Resnet-18
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

other logic parts with the help of input/output asynchronous FIFOs, which further reduces the processing latency.

Other modules. The ReLU activation module, the linear scaling module, and the max pooling module are all designed to process in a single clock cycle to reduce the depth of the execution pipeline.

5 EXPERIMENTAL RESULTS

In this section we demonstrate the effectiveness of our methods, including the ultra-low memory training framework, the ultra-low bitwidth VecQ quantization, and the ultra-low latency T-DLA design. Notbly, all these works are open-sourced.

5.1 On-device Training

We implement our low-precision rank-adaptive tensorized training on an Avnet Ultra96-V2 FPGA board and use it to train a two-layer neural network for a classification task on the FashionMNIST dataset. There are 512 neurons in the hidden layer, folded into $4 \times 4 \times 2 \times 16$ for the first layer, and 32×16 for the second layer. We use the Pytorch and Tensorly modules to implement our training algorithm on the embedded processor. For the FPGA we set the clock rate to 100MHz. We compare the training methods with or without the low rank TT priors. As shown in Table 3, our method achieves 294× memory reduction for the model parameters compared with the standard non-tensorized training. This on-FPGA training has achieved 59× speedup and 123× energy reduction than the training on an embedded CPU.

5.2 Ultra-low Bitwidth Quantization

We use MNIST, Cifar10 and ImageNet to evaluate the ultra-low bitwidth quantization, VecQ. The evaluated DNN models include Lenet-5, Cifarnet, a VGG-like network [32], and Resnet-18.

Table 6: T-DLA resource and performance

Configuration Parameters $< T_n, T_m, L_K, L_D, D_w >$	$< 4, 16, 5, 32, 12 >$			
Resource Utilization(%) $< LUT/FF/BRAM/DSP >$	79 / 47.47 / 68.93 / 91.82			
Clock Frequency Logic / Adder (MHz)	125 / 250			
Peak Performance (GOPS)	400			
DNN Model	Lenet-5	Cifarnet	VGG-like	Resnet-18
latency(ms)	0.016	0.063	2.12	48.8

Table 7: T-DLA: Comparison with the state-of-the-art implementations.

Dataset	Design	Model	Acc.(%)	F., W. (bits)	fps	platform
MNIST	[36]	MFC-max	97.69	1, 1	6238000	ZC706
MNIST	[31]	Lenet-5	-	8, 3	70000	ZC706
MNIST	**Ours**	Lenet-5	**99.5**	8, 2	62051.1	Zedboard
CIFAR10	[36]	VGG-like	80.1	24, 1	21900	ZC706
CIFAR10	[27]	VGG-like	81.8	1, 1	420	Zedboard
CIFAR 10	[32]	VGG-like	86.71	8, 2	27043	VC709
CIFAR 10	[43]	VGG-like	88.68	1, 1	168	Zedboard
CIFAR 10	**Ours**	VGG-like	**89.08**	8,2	457	Zedboard
ImageNet	[24]	Resnet-18	65.44	FP32,FP32	1.545	Xeon[1]
ImageNet	[24]	Resnet-18	65.44	FP32,FP32	387.597	1080Ti[2]
ImageNet	**Ours**	Resnet-18	**68.23**	8, 2	20.48	Zedboard

[1]Xeon: Xeon E5-2630 v3; [2]1080Ti: Nvidia 1080Ti

5.2.1 Classification accuracy. The classification accuracy on different datasets are shown in Table 4. For simplicity, we only show the top-1 accuracy. Comparing to the floating point models (Floating in the table), the classification accuracy using ternary weights and quantized scalars and activations shows negligible degradation. VecQ also achieves superior accuracy comparing to the recent works [1, 24], in which only the weights are ternarized but not the scalars and activations. Our proposed method shows better accuracy for Resnet-18 on ImageNet data set. This result demonstrates the scalability and stability of VecQ, especially in aggressive low-bitwidth quantization scenarios.

5.2.2 Model size reduction. VecQ also greatly reduces the memory footprint (Mem. Reduc.) as shown in Table 5. Ternary weight occupies only 2 bits whereas the original floating point requires 32 bits. As shown in Table 5, for convolution layers, VecQ compresses the parameters nearly to the theoretical limit (almost 16× reduction). We quantize the last FC layer to 12-bit to maintain accuracy, so that the networks with less or no FC layers have higher compression ratio, such as Cifarnet and Resnet-18. Specifically, VecQ reduces up to 92.93% (14.14×) size of Resnet-18 in floating point.

5.3 Ultra-low Latency Acceleration

We use the models quantized by VecQ to evaluate our T-DLA accelerator design in terms of accuracy and frame per second (fps). The measurements of the original models are on a server with two Intel Xeon E5-2630 v3 CPUs and one Nvidia 1080 Ti GPU. T-DLA is implemented on a Xilinx Zedboard FPGA, which is suitable for edge applications with very limited logic resources. It has an on-chip dual-core ARM Cortex A9, and has 53.2K LUTs, 106.4K FFs, 140 BRAM blocks of 36Kb each, and 220 DSPs. Vivado System Design Suite 2019.2 is used for system implementation.

5.3.1 Hardware Resource and Processing Latency Evaluation. We choose an accelerator configuration that fully utilizes the given resources, shown in Table 6, together with the execution latency of the different models with this configuration. We only show the most important configuration parameters including T_n, T_m, L_K, L_D, and the quantized bitwidth of the activations (D_w). As can be seen in Table 6, T-DLA with customized configuration can almost use all the resources, especially the DSPs. The targeted FPGA can support up to 250MHz for the DSPs, which is twice of the frequency of other logic benefiting from the ternary computation array and independent clock design of the adder trees.

5.3.2 Performance Comparison. We compare T-DAL in terms of accuracy and fps with existing designs, either using the same DNN model or the same dataset. The results are shown in Table 7. For MNIST dataset, the design in [36] shows higher fps because of the DNN model they used is simpler and the ZC706 platform has almost 4× more resources than ours. However, our implementation on Zedboard has a comparable fps (62051) to a design [31] (70000) with 3-bit weights on the ZC706 platform. On CIFAR10 dataset, our design shows dominating accuracy advantage among all the VGG-like models. On ImageNet dataset, we directly compare our results with the floating point version. T-DLA shows longer execution latency than the GPU but outperforms the CPU by 9.2×.

6 CONCLUSIONS

In this paper, we summarized our recent efforts for efficient on-device AI development including both training and inference. We mainly focused on three major challenges of edge AI development. First, we presented on-device training with ultra-low memory usage by proposing a novel rank-adaptive tensor-based tensorized neural network model, which offers orders-of-magnitude memory reduction during training. Second, we introduced VecQ, a novel quantization method that supports ultra-low bitwidth quantization with negligible accuracy degradation. Third, we presented T-DLA, an ultra-low latency DNN accelerator design for ternarized DNNs achieving the state-of-the-art performance. On top of the achievements in this paper, we expect more research breakthroughs to boost the development and deployment for the edge AI.

REFERENCES

[1] H. Alemdar et al. 2017. Ternary neural networks for resource-efficient AI applications. In *IJCNN*.
[2] Yoshua Bengio et al. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).
[3] Giuseppe G Calvi et al. 2019. Tucker tensor layer in fully connected neural networks. *arXiv preprint arXiv:1903.06133* (2019).
[4] Yao Chen et al. 2019. Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs. In *FPGA*.
[5] Yao Chen et al. 2019. T-DLA: An Open-source Deep Learning Acceleratorfor Ternarized DNN Models on Embedded FPGA. *ISVLSI* (2019).
[6] Gong Cheng et al. 2019. μL2Q: An Ultra-Low Loss Quantization Method for DNN Compression. (2019).
[7] M. Courbariaux et al. 2016. Binarynet: training deep neural networks with weights and activations constrained to +1 or -1. *arXiv* (2016).
[8] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.
[9] Timur Garipov et al. 2016. Ultimate tensorization: compressing convolutional and fc layers alike. *arXiv preprint arXiv:1611.03214* (2016).
[10] Cheng Gong et al. 2020. VecQ: Minimal loss dnn model compression with vectorized weight quantization. *IEEE Trans. Comput.* (2020).
[11] Philipp Gysel and other. 2016. Hardware-oriented Approximation of Convolutional Neural Networks. *CoRR* abs/1604.03168 (2016).
[12] Song Han et al. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. (2016).
[13] Cong Hao et al. 2019. FPGA/DNN Co-Design: An Efficient Design Methodology for IoT Intelligence on the Edge. In *DAC*.
[14] C. Hao et al. 2021. Enabling Design Methodologies and Future Trends for Edge AI: Specialization and Co-design. *IEEE Design Test* (2021), 1–1.
[15] Cole Hawkins, Xing Liu, and Zheng Zhang. 2020. Towards Compact Neural Networks via End-to-End Training: A Bayesian Tensor Approach with Automatic Rank Determination. *arXiv preprint arXiv:2010.08689* (2020).
[16] Cole Hawkins and Zheng Zhang. 2019. Bayesian Tensorized Neural Networks with Automatic Rank Selection. *arXiv preprint arXiv:1905.10478* (2019).
[17] Christopher J Hillar and Lek-Heng Lim. 2013. Most tensor problems are NP-hard. *Journal of the ACM (JACM)* 60, 6 (2013), 45.
[18] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. 2013. Stochastic variational inference. *The Journal of Machine Learning Research* 14, 1 (2013), 1303–1347.
[19] Valentin Khrulkov, Oleksii Hrinchuk, Leyla Mirvakhabova, and Ivan Oseledets. 2019. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787* (2019).
[20] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.
[21] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. 2014. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553* (2014).
[22] Cong Leng et al. 2018. Extremely Low Bit Neural Network: Squeeze the Last Bit Out With ADMM. In *AAAI*.
[23] Cong Leng et al. 2018. Extremely low bit neural network: Squeeze the last bit out with admm. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
[24] F Li and B Liu. 2016. Ternary Weight Networks. *Proc. NIPS Workshop Efficient Methods Deep Neural Network* (2016).
[25] Qiang Liu and Dilin Wang. 2016. Stein variational Gradient descent: a general purpose Bayesian inference algorithm. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*. 2378–2386.
[26] S. Liu et al. 2011. Real-time object tracking system on FPGAs. In *SAAHPC*.
[27] H. Nakahara et al. 2017. A fully connected layer elimination for a binarizec convolutional neural network on an FPGA. In *FPL*.
[28] Maxim Naumov et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).
[29] Alexander Novikov et al. 2015. Tensorizing neural networks. In *Advances in neural information processing systems*. 442–450.
[30] Ivan V Oseledets. 2011. Tensor-train decomposition. *SIAM Journal on Scientific Computing* 33, 5 (2011), 2295–2317.
[31] Jinhwan Park and Wonyong Sung. 2016. FPGA Based Implementation of Deep Neural Networks Using On-chip Memory Only. *CoRR* (2016).
[32] A. Prost-Boucle et al. 2017. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *FPL*.
[33] Jiantao Qiu et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*. 26–35.
[34] Surat Teerapittayanon et al. 2017. Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices. In *ICDCS*. 328–339.
[35] Andros Tjandra et al. 2017. Compressing recurrent neural network with tensor train. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 4451–4458.
[36] Yaman Umuroglu et al. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *FPGA*.
[37] Peisong Wang et al. 2018. Two-step quantization for low-bit neural networks. In *CVPR*.
[38] Yue Wang et al. 2019. E2-Train: Training State-of-the-art CNNs with Over 80% Energy Savings. In *NeurIPS*.
[39] Miao Yin et al. 2020. Compressing Recurrent Neural Networks Using Hierarchical Tucker Tensor Decomposition. *arXiv preprint arXiv:2005.04366* (2020).
[40] Kaiqi Zhang et al. 2021. On-FPGA Training with Ultra Memory Reduction: A Low-Precision Tensor Method. *ICLR Workshop of Hardware Aware Efficient Training* (2021).
[41] Xiaofan Zhang et al. 2017. Machine learning on FPGAs to face the IoT revolution. In *ICCAD*.
[42] Xiaofan Zhang et al. 2018. DNNBuilder: an automated tool for building high-performance Dnn hardware accelerators for FPGAs. In *ICCAD*.
[43] Ritchie Zhao et al. 2017. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In *FPGA*.
[44] Mingyi Zhou et al. 2019. Tensor rank learning in CP decomposition via convolutional neural network. *Signal Processing: Image Communication* 73 (2019), 12–21.