A Reconfigurable DNN Training Accelerator on FPGA

Jinming Lu, Jun Lin, Zhongfeng Wang
School of Electronic Science and Engineering, Nanjing University, Nanjing, China
Email: jmlu@smail.nju.edu.cn, jlin@nju.edu.cn, zfwang@nju.edu.cn

***Abstract*—In recent years, deep neural networks (DNNs) have been widely applied in various tasks, demonstrating outstanding performance. To further outspread in practical applications, the efficient hardware implementation of DNNs is becoming a critical issue. With the rise of online learning, training DNNs on resource-constrained platforms has attracted more attention most recently. In this paper, we propose an FPGA-based accelerator for efficient DNN training. First, a reconfigurable processing element is designed, which is flexible to support various computation patterns during training in a unified architecture. Second, a well optimized architecture is presented to perform the computation of batch normalization layers in different stages. Finally, a prevailing model (ResNet-20) for CIFAR-10 dataset is implemented on Xilinx VC706 platform with our framework. Experimental results show that our design achieves 421 GOPS and 43.18 GOPS/W in terms of throughput and energy efficiency, respectively. The comparison results illustrate that our accelerator significantly outperforms prior works.**

***Index Terms*—Deep neural networks, hardware accelerator, training, batch normalization, FPGA**

I. Introduction

Recently, Deep Neural Networks (DNNs) have shown the state-of-the-art performance in various tasks, such as computer vision [1], speech recognition [2], and natural language processing [3]. The success of DNNs benefits from the enormous accessible data, the improvement of computing capacity, and the evolving algorithms. However, there is no free lunch. Traditionally, due to the intensive computation and a large amount of data access, DNNs can only be deployed on high-end graph processing units (GPUs) with powerful computing capacity and high energy consumption, which has caused considerable economic costs. As a result, deploying DNNs in a power-efficient and resource-constrained platform is a critical problem that hinders the further widespread application of DNNs.

Prior efforts have been made to improve the efficiency of DNNs. Some of them aimed to reduce DNNs complexity from the perspective of algorithms, such as eliminating the redundancy inherent in models [4], utilizing low-precision representation [5], and designing more compact model architectures [6]. Besides, there are also some specialized accelerators proposed for DNNs, reducing the computational overhead by exploiting the data reuse and parallelism [7], [8].

Most works mentioned above mainly focused on the optimization for the inference phase of DNNs, which is not suitable for model training. People usually collect massive data and train DNNs on GPU clusters. However, for users take privacy more seriously, training DNNs on embedded devices is becoming an urgent demand [9]. Compared to inference, training has very different computation patterns and memory requirements. For inference phase, sampled data are processed by a well-trained model in a forward propagation (FP) way, exporting predicted results. For supervised training, a large amount of labeled data are fed in a DNN until convergency in an iterative procedure, in which each iteration involves forward propagation (FP), backward propagation (BP), weight gradients generation (WG). Therefore, roughly more than 3 times operations are executed for one sample. Besides, the intermediate activations have a longer life period duo to following gradients computation in the BP, which causes a dramatic increase in storage requirements. As a result, the design of training accelerators faces greater challenges.

Some efforts tried to optimize the training process most recently. On the one hand, low-bit formats like FP16 [10], INT8 [11], LOG [12], and Posit [13], [14], are introduced in DNN training, reducing computation and memory demands. On the other hand, several DNN training accelerators are developed based on ASIC or FPGA platforms. F-CNN [15] proposed a training framework by reconfiguring a stream datapath at runtime, where various customizable kernels are designed for basic operations. Liu et al. [16] presented a uniform computation engine for various operations. A multi-FPGA cluster to mapping a whole DNN training program was utilized in FP-Deep [17]. By exploiting the intra-layer and inter-layer parallelism, the lifetime of immediate activations can be minimized. However, their optimization methods will face challenges on models with batch normalization (BN) layers since the data dependency between adjacent layers is hard to be broken. Dey et al. [18] implemented a reconfigurable architecture for training a sparse network, whose connectivity is pre-defined. In [19], a compressed CNN training framework was proposed using the filter-wise pruning and fixed-point quantization method. Both operator-sparse and result-sparse patterns were utilized in their dedicated processing elements. But these two works [18], [19] can only work in several limited situations, such as fine-tune stages and simple tasks, owing to their pre-determined sparsity. An automatic compiler for CNN training accelerator was developed in [9], and the batch-level parallelism was explored in [20].

This work was supported in part by the National Natural Science Foundation of China under Grants 61604068 and in part by the Fundamental Research Funds for the Central Universities under Grant 021014380065

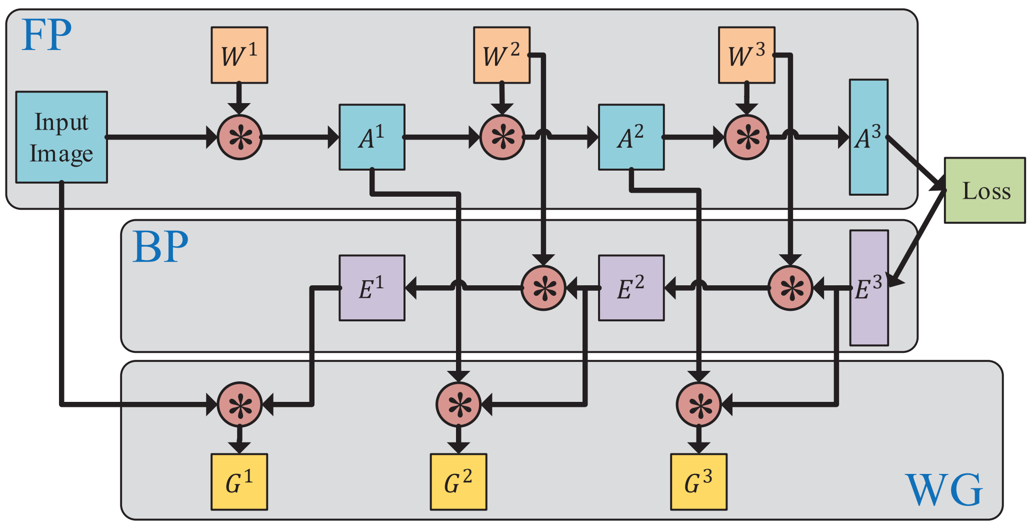


Fig. 1. Overall dataflow in DNN training

However, some issues are not thoroughly investigated yet.

- During different phases, the computation patterns show high diversity. During FP, BP, and WG phases, convolutional (CONV) layers all can be formulated in convolutional operation, their characteristics vary dramatically. It will easily incur the under-utilization of processing elements (PEs), owing to lack of flexibility.
- Most prior works took only the traditional CONV layers into account and used a pooling layer to reduce feature size. But current prevailing models [1], [21] usually adopt a CONV layer with stride being 2 to down-sample features, which brings more computational diversity in the training process. Besides, there are a lot of dummy operations in these non-unit-stride CONV layers.
- Batch normalization (BN) layers play an indispensable role in present DNN models [22]. In works concentrating on inference, a BN layer can be folded into the adjacent CONV layer, since it just performs a simple linear transformation. Furthermore, a layer fusion method can be used for reducing data traffic. However, the situation is quite different for training. The normalization operation can not be completed unless the front CONV layer finishes processing whole mini-batch features. Therefore, the implementation of BN layers has a major impact on overall latency and efficiency, even though the number of operations in BN layers is much less than that in CONV layers.

In this work, we mainly concentrate on addressing the above issues, then propose a reconfigurable hardware accelerator for DNN training. Our contributions are summarized as follows:

- A reconfigurable PE is proposed to efficiently support various computation patterns in training, therefore dummy operations can be eliminated for non-unit-stride CONV layers, and the PE utilization can be kept at a high level all the time.
- We optimize the computation flow of BN layers during FP and BP phases, reducing the latency of normalization operation. Then we present a unified architecture of BN layers for both FP and BP phases .
- A reconfigurable accelerator is designed for DNN training. Implemented on Xilinx VC706 platform, our design achieves 421 GOPS and 43.1 GOPS/W in terms of throughput and energy efficiency. The comparison results illustrate our accelerator outperforms prior works and has a 4X better energy efficiency than GPU.

The organization of the rest of this paper is as follows: Section II gives an overview of the DNN training algorithm. Section III presents our DNN training architecture. Section IV reports experimental results. Section V draws the conclusion.

II. Background

In this section, we briefly introduce the overall computation flow in DNN training. The training dataflow is described in Fig. 1 for a 3-layer neural network, which comprises of two CONV layers and a fully-connected (FC) layer. All involved data are categorized into weights (W), activations (A), errors (E), and gradients (G), whose superscripts indicate layer indices. The symbol '$*$' indicates the convolutional or matrix-multiplication operation. We divide the training process into three phases, including forward propagation (FP), background propagation (BP), weight gradient (WG).

- **Forward Propagation** : First, input images are fed into the model. Then, based on the layer type, activations (A^l) of each layer are successively calculated by corresponding operations. For a CONV layer, the output value $A^l[m, x, y]$

is calculated by convolving weights (W^l) with the activations of the previous layer (A^{l-1}).

$$A^l[m,x,y] = \sum_{c=0}^{C-1}\sum_{k_x=0}^{K_x-1}\sum_{k_y=0}^{K_y-1} W^l[m,c,k_x,k_y]\times A^{l-1}[c,x+k_x,y+k_y]. \tag{1}$$

For an FC layer, the computation can be formulated as a simple matrix-vector multiplication. If its previous layer is a CONV layer as shown in Fig. 1, a flattening operation is needed.

$$A^l = W \times A^{l-1}. \tag{2}$$

At the end of the FP phase, a predicted label is compared with the golden label, and the loss is computed.

- **Backward Propagation** : Errors are evaluated and back-propagated through each layer. The computation pattern is similar to that of FP phase. Nevertheless, the kernels need a rotation of 180 degrees and a transposition between in-channel and out-channel dimensions for CONV layers, and the weight matrixes need to be transposed for FC layers. The computation in a CONV layer during BP phase is shown as :

$$E^{l-1}[c,x,y] = \sum_{m=0}^{M-1}\sum_{k_x=0}^{K_x-1}\sum_{k_y=0}^{K_y-1} W^l[m,c,k_x,k_y]\times E^l[c,x+K_x-1-k_x,y+K_y-1-k_y]. \tag{3}$$

For an FC layer, the errors are obtained by :

$$E^{l-1} = E^l \times (W^l)^T. \tag{4}$$

- **Weight Gradients** : Weight gradients are the derivatives of the loss with respect to weights. For a CONV layer, weight gradients are obtained by convolving activations of the previous layer with errors of the present layer.

$$G^l[m,c,k_x,k_y] = \sum_{m=0}^{M-1}\sum_{x=0}^{X-1}\sum_{y}^{Y-1} E^l[m,x,y]\times A^{l-1}[c,x+k_x,y+k_y]. \tag{5}$$

For an FC layer, they are completed by an outer production operation.

$$G^l = A^{l-1} \circ E^l. \tag{6}$$

III. DNN Training Architecture

In this section, we describe the proposed DNN training accelerator.

A. A Reconfigurable Processing Element

As shown in Fig. 2, a reconfigurable PE is designed, which contains 3 multiply-accumulate units (MACs), 4 multiplexers, and a partial sum (Psum) regfile. The key features of this unit are :

- In the PE level, a CONV layer is processed in a row-by-row fashion, and an FC layer can be treated as a CONV layer whose kernel size and feature size both are 1x1.

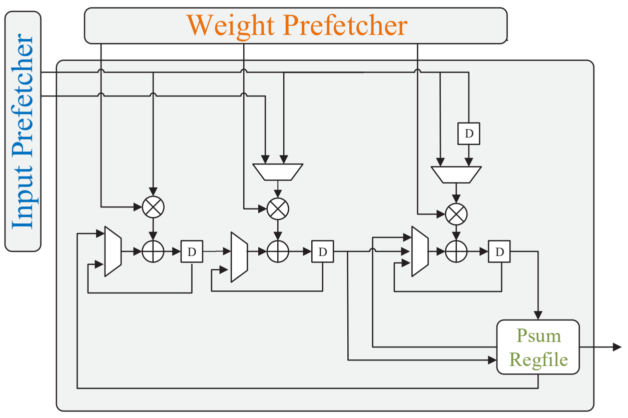


Fig. 2. The architecture of processing element

- In the MAC level, the computation can be executed in a weight-stationary [23] dataflow during FP and BP phases for CONV layers owning 3x3 kernels, and in an output-stationary dataflow for other computation patterns.
- The dummy operations can be eliminated in CONV layers whose strides equal 2.
- The partial sums are locally reused until the final outputs are obtained.

First, we take a 1x3 convolution for example, and illustrate how our PE works on various computation patterns in a CONV layer.

1. **FP/BP, stride=1**: While stride=1, the computation patterns at spatial dimensions are the same during FP and BP phases. From Fig. 3(a) we can see that the PE works like a typical 1-D convolver in a weight-stationary dataflow.
2. **WG, stride=1**: From Eq. (5), the computation during WG can be formulated into convolution operation. However, the kernel sizes, which actually are the feature sizes of errors, are relatively large and vary in different layers. Meanwhile, the gradients have the same sizes as weights. Considering this case, we configure the PE in an output-stationary fashion as shown in Fig. 3(b).
3. **FP, stride=2**: While stride=2, input-skip operations are required during the FP phase. Therefore, using the 1-D convolver directly causes that half operations are invalid, leading to unnecessary computational overhead. In our design, input data are divided into two groups based on coordinates, *i.e.* even input group and odd input group, which are sent to different MACs in parallel. As a result, dummy operations can be eliminated. The PE is configured as Fig. 3(c).
4. **BP, stride=2**: Different from FP, the computation in this case requires zero-inserted operations to up-sample errors. As indicated in Fig. 3(e), to skip the zero-input operations, we split the output data into two groups based on output coordinates, *i.e.* even output group and odd output group, which are received from different MACs in parallel.
5. **WG, stride=2**: The zero-inserted operations are also

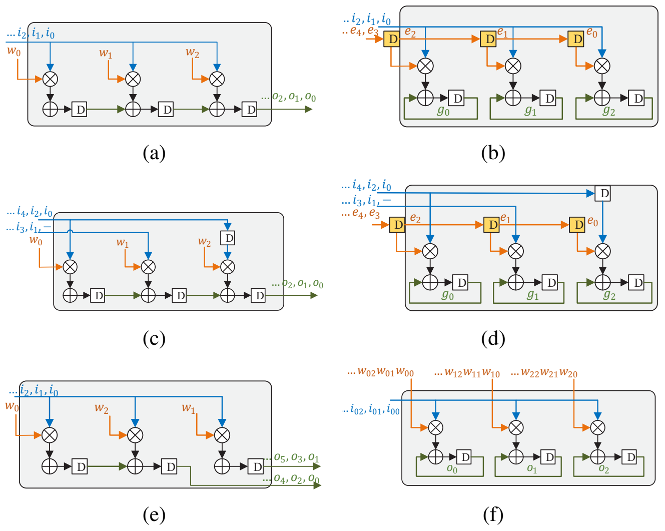


Fig. 3. The computation patterns of PE

needed for WG while stride=2. Fig. 3(d) shows that we adopt the input division method to avoid zero-input operations.

Then, we briefly describe how the PE works on 1x1 CONV and FC layers. Since input-skip and zero-input operations can be easily implemented while kernel size equals 1, we do not discuss those cases where stride=2. As depicted in 3(f), we utilize output-stationary dataflow for all phases. The input data are broadcasted to 3 MACs, and different weights are transmitted to 3 MACs in parallel. Here the coordinates (m, n) of $w_{m,n}$ denotes output-channel and input-channel, respectively, and the coordinate k of $i_{k,n}$ represents batch-index or spatial coordinate.

B. A Unified Batch Normalization Unit

The BN layer has been an essential component in current prevailing DNNs, but not been thoroughly investigated in prior training accelerators. Though the number of operations in BN layers occupy a small percentage of the whole model, a vanilla implementation will cause unnoticeable resource wasting and latency increasing. We optimize the training process of BN layers using algorithm and hardware co-design. The overall computation processes can be divided into 3 steps during both FP and BP phases : statistics computation, local variables computation, and linear transformation.

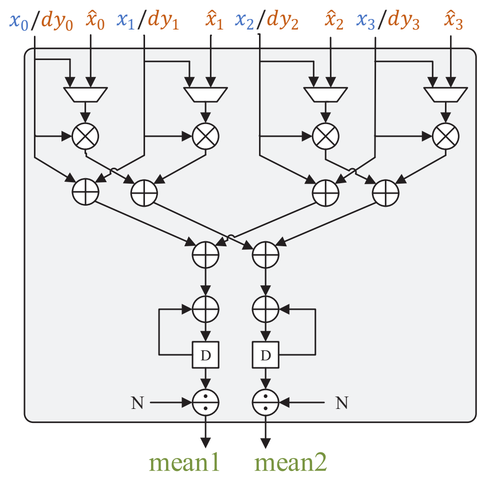
Statistics Computation: Benefiting from the transformation in Eq. 7, the calculation of var does not need to wait for finishing the calculation of μ. $\mathbb{E}[X^2]$ and $\mathbb{E}^2[X]$ can be obtained in parallel, called *mean1* and *mean2* in line 1-2 in Algorithm 1. A similar process occurs in the BP phase as shown in line 10-13 in Algorithm 1, except that *mean2* is the expectation of $\mathrm{d}y_i \times \hat{x}_i$. Therefore, we implement statistic computation for FP and BP phases in a unified module, named BN STAT, which contains two adder trees, a set of multipliers, a set of multiplexers, two accumulators, and two dividers. Note that the divider can be replaced by a simpler shifter operation when the batch-size N is a power of 2. An example of BN STAT with a size of 4 is depicted in Fig. 4.

$$\mathrm{Var}[X] = \mathbb{E}\left[(X - \mathbb{E}[X])^2\right] = \mathbb{E}[X^2] - \mathbb{E}^2[X]. \quad (7)$$

Local Variables Computation: During both FP and BP phases, some variables only require to be computed once in one layer, such as λ in line 5 and $\gamma\lambda$ in line 14 of Algorithm 1. Hence we integrate their implementations into the linear transformation module, called BN Core, in a time-multiplexed

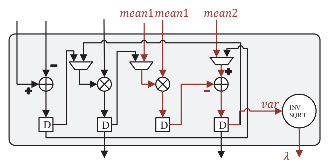
Algorithm 1 Batch Normalization Training

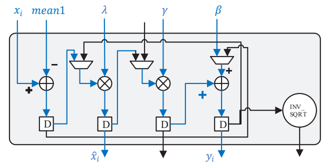
```
    1. Forward Propagation:
Require: A mini-batch of input features $\{x_1, x_2, ..., x_N\}$, where N is mini-batch size, D is the feature numbers; learnable parameters $\gamma, \beta$;
 1: mean1 $\leftarrow \frac{1}{N}\Sigma_{n=1}^{N} x_i$
 2: mean2 $\leftarrow \frac{1}{N}\Sigma_{n=1}^{N} x_i^2$
 3: $\mu \leftarrow$ mean1
 4: $var \leftarrow$ mean2 $- \mu^2$
 5: $\lambda \leftarrow 1/\sqrt{var + \epsilon}$
 6: $\hat{x}_i \leftarrow (x_i - \mu) \times \lambda$
 7: $y_i \leftarrow \hat{x}_i \times \gamma + \beta$
 8: Passing $\{y_i\}$ to the next layer;
 9: Saving $\lambda, \hat{x}_i$ for backward;
    2. Backward Propagation:
Require: A mini-batch of errors $\{\mathrm{d}y_1, \mathrm{d}y_2, ..., \mathrm{d}y_N\}$
10: $\mathrm{d}\beta \leftarrow \Sigma_{n=1}^{N} \mathrm{d}y_i$
11: $\mathrm{d}\gamma \leftarrow \Sigma_{n=1}^{N} \mathrm{d}y_i \times \hat{x}_i$
12: mean1 $\leftarrow \frac{\mathrm{d}\beta}{N}$
13: mean2 $\leftarrow \frac{\mathrm{d}\gamma}{N}$
14: $\gamma\lambda \leftarrow \gamma \times \lambda$
15: temp1$_i \leftarrow \hat{x}_i \times$ mean2
16: temp2$_i \leftarrow \mathrm{d}y_i -$ mean1
17: $\mathrm{d}x_i \leftarrow \gamma\lambda \times ($temp2$_i -$ temp1$_i)$
18: Passing $\{\mathrm{d}x_i\}$ to the previous layer.
```

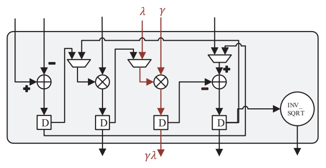


x_0/dy_0 $\hat{x}_0$ x_1/dy_1 $\hat{x}_1$ x_2/dy_2 $\hat{x}_2$ x_3/dy_3 $\hat{x}_3$
D
N
mean1
mean2

Fig. 4. The architecture of BN STAT


(a) Local variables in FP


(b) Linear transformation in FP


(c) Local variables in BP

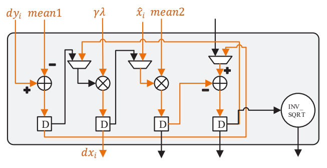

(d) Linear transformation in BP

Fig. 5. The workload of BN Core

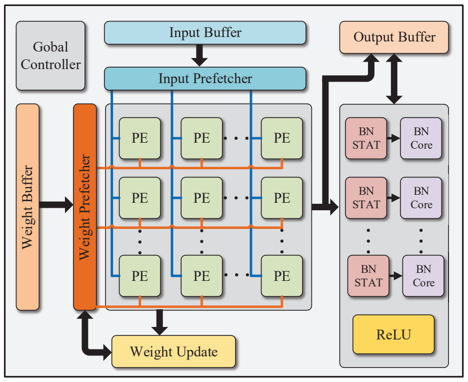


Fig. 6. The architecture of DNN training system

manner, as shown in Figs. 5(a) and 5(c). The results are locally stored in registers for subsequent use in linear transformation.

Linear Transformation: This step corresponds to line 6-7 in the FP phase and line 15-17 in the BP phase, completed in a BN Core. A BN Core comprises two adders, two multipliers, three multiplexers, and an INV-SQRT function unit, and it works in a fully pipelined manner during this step. As illustrated in Figs. 5(b) and 5(d), input data are streamed in the BN Core, then output data are obtained and written to the data buffer, preparing for the computation of the subsequent CONV layer.

C. Overall System

The overall architecture of the proposed DNN training accelerator is illustrated in Fig. 6, containing a global controller, a PE array, a BN unit, and several multi-bank buffers.

The global controller is responsible to ensure that computations are executed in the right order, and configure PEs and BN Cores based on different phases. The PEs are organized in a 2-D systolic array with a size of 16x32. where each row shares the same weight and each column shares the same input. Inputs are stored in a multi-bank buffer, and each bank corresponds to one PE column. For different PE columns, inputs are unrolled along batch or row dimensions. To deal with the transposition of weights during the BP phase, weights are arranged in a cyclic fashion as introduced in [9]. For different PE rows, weights are unrolled along output-channel (during FP) or input channel (during BP). The BN unit contains a set of BN STAT, each with a size of 32, and BN Core pairs, each of which receives intermediate results from a PE row. Moreover, a ReLU module is integrated in the BN unit to perform non-linear transformation. We explore the data formats configuration and determine to use the 8-bit fixed-point in CONV layers and half-precision floating-point (FP16) in BN layers, for all kinds of data types.

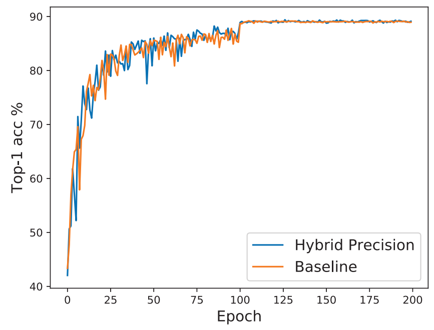


Fig. 7. Accuracy curves of hybrid precision strategy and baseline

IV. Experimental Results

To evaluate the feasibility of our framework, we trained a ResNet-20 model for CIFAR-10 using a hybrid precision strategy, which is consistent with our hardware design, where CONV layers are calculated using 8-bit fixed-point and BN layers are calculated using FP16. Here we adopt a vanilla SGD optimization algorithm without any regularization technique. The simulation is implemented on PyTorch framework. The accuracy curves of our training method and the baseline are plotted in Fig. 7. We can see that there is no obvious accuracy gap between them.

TABLE I
FPGA Resource Utilization

Resource	LUT	LUT RAM	FF	BRAM	DSP
Utilization	141377	17328	120009	912	2176
Available	433200	174200	866400	1470	3600
Ratio	32.64%	9.95%	13.85%	62.04%	60.44%

The accelerator design is coded in Verilog and synthesized using Vivado 2018.3. The target hardware platform is Xilinx VC709 (Virtex7 XC7VX690T). TABLE I reports the hardware resource utilization. Working at 200MHz, our design achieves 421 GOPS and 43.18 GOPS/W in terms of performance and energy efficiency.

The comparisons between prior training accelerators and our design are summarized in TABLE II, which reveals that our design has 4X better energy efficiency than GPU, and outperforms existing training accelerators.

TABLE II
PERFORMANCE COMPARISON BETWEEN OUR DESIGN AND PRIOR TRAINING ACCELERATORS AND GPU

	[15]	[16]	[9]	[17]	[19]	Ours	GPU
Device	Maxeler MPC-X	ZU19EG	Stratix 10	VC709	KCU1500	VC709	P100
Precision	FP 32	FP 32	Fixed 16	Fixed 16	Fixed 8/24	Fixed 8 / FP 16	FP 32
Performance (GOPS)	7.01	86.12	163	1022	641.1	421	1384
Power (W)	27.3	14.2	20.6	32	26.8	9.75	130
Energy Efficiency (GOPS/W)	0.27	6.05	7.90	31.97	24.01	43.18	10.65

V. CONCLUSION

In this paper, we propose a reconfigurable hardware accelerator for DNN training with a hybrid precision strategy. The reconfigurable PE can support various computation patterns and eliminate the dummy computation in non-traditional CONV layers. The BN unit reduces the latency from the normalization operation, and executes the FP/BP phase in a unified hardware architecture. As a result, our design is able to support the training of prevailing DNNs like ResNet. The implementation results demonstrate that our design achieves 421 GOPS performance and 4X better energy efficiency than GPU.

REFERENCES

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770–778, 2016.

[2] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen et al., "Deep speech 2: End-to-end speech recognition in english and mandarin," in International conference on machine learning, 2016, pp. 173–182.

[3] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[4] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in Advances in Neural Information Processing Systems 29, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds. Curran Associates, Inc., 2016, pp. 2074–2082. [Online]. Available: http://papers.nips.cc/paper/6504-learning-structured-sparsity-in-deep-neural-networks.pdf

[5] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in European conference on computer vision. Springer, 2016, pp. 525–542.

[6] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan et al., "Searching for mobilenetv3," in Proceedings of the IEEE International Conference on Computer Vision, 2019, pp. 1314–1324.

[7] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," IEEE journal of solid-state circuits, vol. 52, no. 1, pp. 127–138, 2016.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," ACM SIGARCH Computer Architecture News, vol. 42, no. 1, pp. 269–284, 2014.

[9] S. K. Venkataramanaiah, Y. Ma, S. Yin, E. Nurvitadhi, A. Dasu, Y. Cao, and J. sun Seo, "Automatic compiler based fpga accelerator for cnn training," 2019 29th International Conference on Field Programmable Logic and Applications (FPL), pp. 166–172, 2019.

[10] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," ArXiv, vol. abs/1710.03740, 2017.

[11] S. Wu, G. Li, F. Chen, and L. Shi, "Training and inference with integers in deep neural networks," ArXiv, vol. abs/1802.04680, 2018.

[12] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," arXiv preprint arXiv:1603.01025, 2016.

[13] J. Lu, C. Fang, M. Xu, J. Lin, and Z. Wang, "Evaluations on deep neural networks training using posit number system," IEEE Transactions on Computers, 2020.

[14] H. F. Langroudi, Z. Carmichael, D. Pastuch, and D. Kudithipudi, "Cheetah: Mixed low-precision hardware & software co-design framework for dnns on the edge," arXiv preprint arXiv:1908.02386, 2019.

[15] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," 2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp. 107–114, 2016.

[16] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, "An fpga-based processor for training convolutional neural networks," 2017 International Conference on Field Programmable Technology (ICFPT), pp. 207–210, 2017.

[17] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Xu, R. Patel, and M. Herbordt, "Fpdeep: Acceleration and load balancing of cnn training on fpga clusters," in 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018, pp. 81–84.

[18] S. Dey, D. Chen, Z. Li, S. Kundu, K.-W. Huang, K. M. Chugg, and P. A. Beerel, "A highly parallel fpga implementation of sparse neural network training," 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1–4, 2018.

[19] K. Guo, S. Liang, J. Yu, X. Ning, W. Li, Y. Wang, and H. Yang, "Compressed cnn training with fpga-based accelerator," in Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2019, pp. 189–189.

[20] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, "Towards efficient deep neural network training by fpga-based batch-level parallelism," 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 45–52, 2019.

[21] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4510–4520, 2018.

[22] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," arXiv preprint arXiv:1502.03167, 2015.

[23] Y. Lin and T. S. Chang, "Data and hardware efficient design for convolutional neural network," IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 65, no. 5, pp. 1642–1651, 2018.