# TNPU: An Efficient Accelerator Architecture for Training Convolutional Neural Networks

Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Xiaowei Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
{lijiajun,yan,luwenyan,jiangshuhao,gongshijun,wujingya,yanjunchao,lxw}@ict.ac.cn

## ABSTRACT

Training large scale convolutional neural networks (CNNs) is an extremely computation and memory intensive task that requires massive computational resources and training time. Recently, many accelerator solutions have been proposed to improve the performance and efficiency of CNNs. Existing approaches mainly focus on the inference phase of CNN, and can hardly address the new challenges posed in CNN training: the resource requirement diversity and bidirectional data dependency between convolutional layers (CVLs) and fully-connected layers (FCLs). To overcome this problem, this paper presents a new accelerator architecture for CNN training, called TNPU, which leverages the complementary effect of the resource requirements between CVLs and FCLs. Unlike prior approaches optimizing CVLs and FCLs in separate way, we take an alternative by smartly orchestrating the computation of CVLs and FCLs in single computing unit to work concurrently so that both computing and memory resources will maintain high utilization, thereby boosting the performance. We also proposed a simplified out-of-order scheduling mechanism to address the bidirectional data dependency issues in CNN training. The experiments show that TNPU achieves a speedup of 1.5× and 1.3×, with an average energy reduction of 35.7% and 24.1% over comparably provisioned state-of-the-art accelerators (DNPU and DaDianNao), respectively.

## CCS CONCEPTS

• **Computer systems organization → Neural networks**; **Data flow architectures**;

## KEYWORDS

Convolutional Neural Networks, Accelerator Architecture, CNN Training

**ACM Reference Format:**
Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Xiaowei Li. 2019. TNPU: An Efficient Accelerator Architecture for Training Convolutional Neural Networks. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3287624.3287641


*ASPDAC '19, January 21–24, 2019, Tokyo, Japan*

ACM ISBN 978-1-4503-6007-4/19/01…$15.00
https://doi.org/10.1145/3287624.3287641

## 1 INTRODUCTION

Convolutional neural networks (CNNs) have achieved great success in a broad range of applications such as image classification and speech recognition. However, training large scale CNNs is an extremely computation intensive task that requires massive computational resources and training time. For example, AlexNet takes up to six days to train on two GTX 580 3GB GPUs [8].

Training CNNs relies in general on the backpropagation algorithms that intrinsically supports fine-grained parallelism similar as the inference phase. However, accelerators dedicated for CNN inference usually cannot well handle the training phase because it poses new challenges compared to inference: the resource requirement diversity between convolutional layers (CVLs) and fully-connected layers (FCLs). Most of prior accelerator solutions mainly focused on CLVs because of their sophistication and high computation intensity but offer little insight into the optimizations on FCLs, since the resource requirements of FCLs are quite different from those of CVLs. Generally, FCLs require a relatively small amount of computation with a huge amount of memory accesses. On the other hand, CVLs require a massive of computation with a relatively small amount of memory accesses. Therefore, accelerators dedicated for CVLs could not well accommodate FCLs suffering high memory transaction costs and significant performance degradation. We observed that prior solutions inevitably incur resource underutilization of either computing resource or memory bandwidth, causing performance degradation when training CNNs. This can be confirmed by the experimental results of state-of-the-art CNN accelerators [2, 13, 15] demonstrated in Fig. 1, for example, the average training performance of DaDianNao [3] merely reaches no more than 50% compared to its performance on CVLs.

The high memory transaction cost of FCLs can be largely reduced by network compression. Previous studies [7] have proven that more than 90% of the parameters in FCLs can be pruned to zero without loss of accuracy. However, network compression is only effective in inference phase and it doesn't work in training, because it might not converge when training compressed CNNs.

Properly handling both CVLs and FCLs is the key to designing an efficient CNN training accelerator. The prevailing method is to customize the acceleration for CVLs and FCLs in separate way so that each customization reaches higher performance [14]. However, it significantly increases the hardware complexity and thereby overhead. Unlike prior approaches which trade complexity for flexibility, we take an alternative by orchestrating CVLs and FCLs in the same computing logic. We found that the resource requirement diversity between CVLs and FCLs can bring significant complementary effect to boost the performance of accelerators. For example, both computation and memory bandwidth resources will maintain high

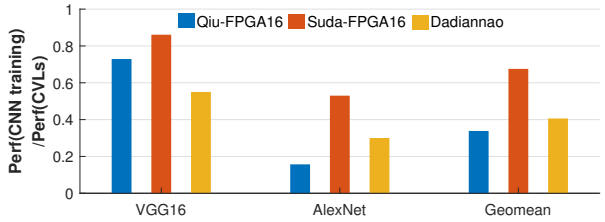



**Figure 1: The attainable CNN training performance of prior accelerators.**

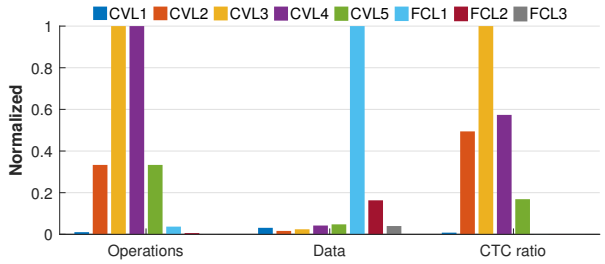



**Figure 2: The diverse resource requirements between CVLs and FCLs (normalized to the corresponding largest number).**

utilization by scheduling CVLs and FCLs to work concurrently. This work presents TNPU, an accelerator architecture for CNN training. TNPU partitions the PE array into two groups, one for processing CVLs and another for FCLs. The micro-architectural design of the PEs for CVLs and FCLs are homogeneous hence largely reduce the design complexity. TNPU exploits a simplified out-of-order scheduling scheme to ensure that the two groups of PEs can work concurrently, hence boosting the utilization of both computation and memory bandwidth resources.

The experimental results show that our accelerator achieves a speedup of 1.5× and 1.3×, with an average energy reduction of 35.7% and 24.1% over comparably provisioned state-of-the-art accelerators (DNPU and DaDianNao), respectively.

## 2 MOTIVATION

This section motivates TNPU by showing that: 1) FCLs exhibits quite different resource requirement compared to CVLs, which will cause performance degradation for accelerators dedicated for CVLs; 2) the data dependency in CNN training is more complicated than inference, which leads to a different design paradigm for CNN training accelerators.

### 2.1 Resource Requirement Diversity

The computational dominance of CVLs, has sparked significant interest in the design and optimization of accelerator structures for these layers [2, 4, 5, 9–11, 16]. In general, these accelerators can also handle the computations of FCLs because the computation of FCLs can also be represented as matrix operations similar as CVLs. However, they offer little insight in the optimization of FCLs because FCLs exhibits quite different resource requirements. This difference can be clearly identified by examining the metric *Computation to Communication (CTC)* ratio [16], which is defined as the computation operations per memory access: $CTC = N_{op}/N_d$, where $N_{op}$ denotes the total number of operations and $N_d$ denotes the amount

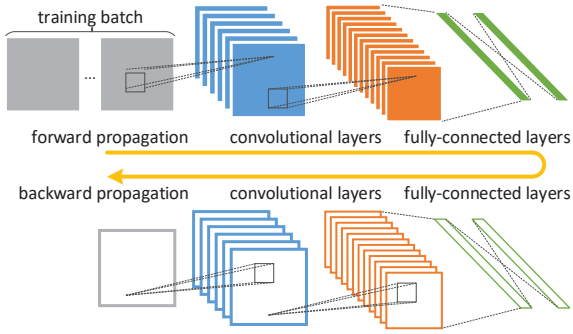



**Figure 3: Training convolutional neural networks based on backpropagation algorithms.**

of external data accesses in the same time span. Intuitively, greater CTC implies more intensive computation or less memory bandwidth requirement; by contrast, the smaller CTC is correlated with less computation but higher memory intensity. The former case is usually interpreted as computation-intensive patterns, while the later memory-intensive patterns. We find that CTC ratio of FCLs is greatly dwarfed by that of CVLs, as demonstrated by the example in Fig. 2. This result reveals that CVLs can do several orders of magnitudes more computing operations on each memory access than FCLs.

Prior solutions cannot well handle both CVLs and FCLs because of their intrinsically monolithic nature, i.e. both CVLs and FCLs share the computing resource in a time-division multiplexing manner and cannot be spatially separated. Specifically, in any time slot either CVLs or FCLs are processed by the same computing logic. In such scenario, CVLs and FCLs are fed with the same memory bandwidth, then to saturate the memory bandwidth, the computing capacity will be determined by CVLs because its CTC ratio is much larger than that of FCLs. In such case, efficiency loss is inevitable because the memory-intensive FCLs have not enough computations to saturate the computing capacity designed for the computing-intensive CVLs, thereby degrading the performance.

We found that the resource requirement diversity between CVLs and FCLs can bring significant complementary effect to boost the performance of accelerators. For example, both computation and memory resources will maintain high utilization by scheduling CVLs and FCLs to work concurrently.

### 2.2 Bidirectional Data Dependency

Another challenge for accelerating CNN training is that the data dependency in training is more complicated than inference. Training CNNs relies in general on the backpropagation algorithms which contain four phases: forward propagation (FP), backward propagation (BP), gradient descent computation (GD) and weight update (WU). The basic computation patterns of these phases are based on matrix operations, but the data dependency is quite different.

Fig. 3 presents the CNN training process based on backpropagation algorithms. In FP phase, the input data are firstly fed in to CVLs, followed by FCLs, which implies that the computation of FCLs could not begin until the processing of CVLs are finished. Then in BP phase, the errors are back propagated from FCLs to

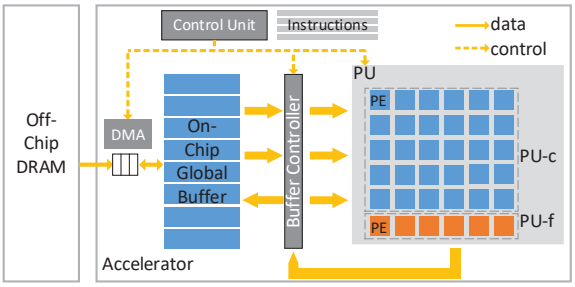



**Figure 4: Accelerator Architecture**

CVLs, i.e. FCLs are firstly processed, then CVLs, which implies a reversed data dependency to that in FP phase. By contrast, no explicit data dependency exists in GD and WU phase, either CVLs or FCLs can be processed first. Note that the pooling and normalization layers are not shown in the figure because there are relatively less computations and memory accesses in these layers.

The bidirectional data dependency in FP and BP phase increased the design complexity of CNN training accelerator. It poses a big challenge for the concurrent execution of CVLs and FCLs since there exists an explicit processing order between them.

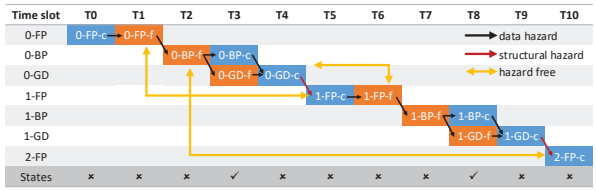
In a nutshell, these challenges motivate building an accelerator architecture to efficiently support CNN training.

# 3 TNPU ARCHITECTURE

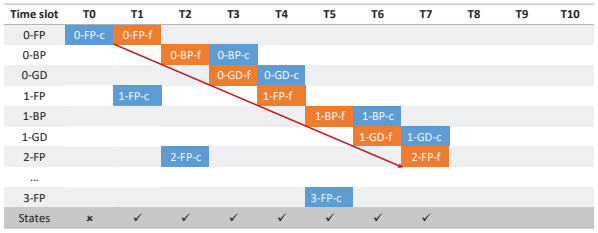
## 3.1 Overview

Fig. 4 presents the TNPU architecture, which contains a control unit, a buffer controller, on-chip global buffer, a direct memory access (DMA) module, a processing unit (PU) and off-chip DRAM. The PU contains an array of processing elements (PEs), which are separated into two groups, PU-c and PU-f, responsible for the computations of CVLs and FCLs, respectively. The PEs are connected with each other via a 2D mesh network on chip (NoC) and can support high compute parallelism to perform the massive matrix operations. All data for processing are stored in off-chip DRAM. The on-chip global buffer consists of neuron buffers and synapse buffers to temporarily store the data for computation.

Because of resource limitation, a small portion of the data are cached in on-chip neuron buffers and synapse buffers through DMA before being fed to PEs. Then, the buffer controller selects needed neurons and synapses for each PE from local neuron buffers and synapse buffers based on the loaded instructions which are decoded by the control processor, and transfers the selected data to PEs for neural computations. A key feature of the proposed architecture is that CVLs and FCLs share the computing resources in a space-division multiplexing manner. If computations are well scheduled, there will exist both CVL and FCL computations in most of the time slots, hence the complementary effect of resource requirement can be leveraged to boost the performance. Note that the PEs in PU-c and PU-f are homogeneous hence it will not induce extra complexity in PE micro-architectural designs.




(a) Conventional scheduling mechanism




(b) Simplified out-of-order scheduling mechanism

**Figure 5: Conventional scheduling mechanism vs. SOSM for CNN training**

## 3.2 Simplified OoO Scheduling Mechanism

One key challenge for the proposed architecture, is how to schedule CVLs and FCLs to process together as there exists explicit processing order between them, as claimed in Section 2.2. Specifically, in FP phase, as PU-f takes the output of PU-c as input, PU-f cannot start to work until PU-c finishes its computations. Then, PU-c stays idle until PU-f finishes its computations in both FP and BP phase.

This challenge can be identified by the example in Fig. 5(a), where PU-c and PU-f are considered as a two-stage pipeline. In the "pipeline", each stage is denoted as (x-y-z), where $x$ indicates the training instance id, $y$ indicates the phase type (FP, BP, GD) and $z$ indicates the layer type ("c" for CVLs, "f" for FCLs). We assume that the provided memory bandwidth is large enough to satisfy the computation thus the off-chip memory accesses is not shown in the figure. Each training instance contains three phases, FP, BP and GD. Note that the weight update phase is not included because: 1) weight update is only activated once in each training batch; 2) the computations in weight update phase is much less than other phases. Each phase contains computation of two layer types, for example, the FP phase for training instance 0 contains 0-FP-c and 0-FP-f. In Fig. 5(a), the data dependencies are indicated by the black arrow lines. In FP phases, the PU-c is invoked first to process convolutional computations, then PU-f takes over to process the FC computation. For example, in 0-FP phase, there exists data dependency between 0-FP-f and 0-FP-c, hence 0-FP-c is processed in time slot $T1$ instead of $T0$. The result is that in $T1$, PU-c is busy while PU-f is idle. In BP phase, because of the reversed data dependency compared with FP, PU-f is enabled first, then PU-c. As shown in 0-BP phase, 0-BP-c is processed after 0-BP-f. Furthermore, there exists data dependency between 0-FP-F and 0-BP-f, hence, 0-BP-f is scheduled in the time slot after that of 0-FP-f. By contrast, since no explicit data dependency exists in GD phase, either PU-c or PU-f can be enabled first, depending on the instant resource availability. In $T3$, since PU-c is already occupied by 0-BP-c, 0-GD-f is scheduled to $T3$ while 0-GD-c to the next time slot. In $T5$, the next training instance starts to process.

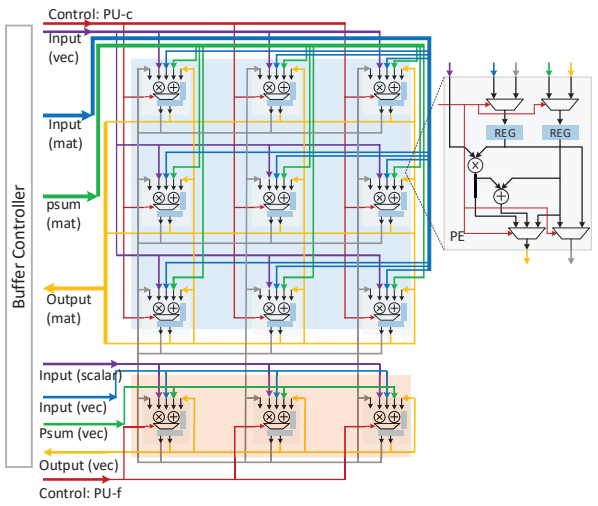



**Figure 6: Processing Unit (PU) and Processing Element (PE) architecture.**

The above example shows that the conventional scheduling mechanism cannot efficiently utilize the computing resources, causing idles of either PU-c or PU-f. Specifically, PU-c and PU-f are working concurrently only in $T3$, while in other time slots $T0$, $T1$, $T2$, $T4$, only one of them is busy, leading to low resource utilization thereby degrading the performance. To address this challenge, we apply a simplified out-of-order scheduling mechanism (SOSM) to keep both PU-c and PU-f working concurrently. The concept is similar to the out-of-order execution model in classical super-scalar processors.

SOSM is illustrated Fig. 5(b), where the PE idle slots are significantly reduced. The rationale is to leverage the input instance batch in CNN training. Since there is no data dependency between training instances in a batch, the CVLs of the next instance can be processed simultaneously with the FCLs of the current instances. The implementation of SOSM is much simpler than that in conventional super-scalar processors. Because the data dependency is simple and explicit and can be obtained in advance, it is unnecessary to implement a dedicated hardware such as scoreboards or reorder-buffers. As shown in Fig. 5(b), the FCL computations in PU-f is in-order, we need only pre-execute some FP-c computations of the training instances to fill in the idle time slots, among which there are no data dependencies. Through SOSM, we can keep both PU-c and PU-f busy without stalls. Specifically, we scheduled 1-FP-c to $T1$, 2-FF-c to $T2$, 3-FF-c to $T5$, then in all time slots (except for the warming up $T0$) both PU-c and PU-f can be kept in busy states. The time slots needed is significantly reduced, thereby boosting the performance.

### 3.3 Processing Unit and Processing Element

Fig. 6 demonstrates the architecture of the processing unit, which is composed of multiple PEs organized into a 2D mesh topology. The PU is designed for efficient computation of the core operations in both CVLs and FCLs, i.e. the matrix vector multiplication-accumulation operations. The PU is separated into two groups, PU-c (in the upper shaded box) and PU-f (in the lower shaded box), responsible for the computations of CVLs and FCLs, respectively. The PEs in PU-c is organized in a square pattern to efficiently support CVL computations in both FP and BP phases as they can be considered as symmetric phases. PU-f contains a row of PEs and can support vector multiplication-accumulation operations. The PEs in PU-f is much less than PU-c because: 1) FCLs computations can be also represented as vector multiplication-accumulation operations; 2) the computation amount in FCLs is usually much less than CVLs as mentioned in Section 2.1, a PE row provisioning is usually capable to handle the FCL computations. Without losing any generality, in Fig. 6 we consider a small design having $4 \times 3$ PEs, where $3 \times 3$ in PU-c and $1 \times 3$ in PU-f.

The PU can simultaneously read vectors and matrices from neuron buffers and synapse buffers through the interconnects, and distribute them to different PEs. Specifically, PU can support the main types of computation patterns in CNN training: matrix vector product, dot product and Cartesian product. To handle matrix vector product, PU-c firstly reads a vector and a matrix from the buffers. The vector is distributed in a row-sharing manner (PEs in a row shares the same data), while the matrix is distributed to the PEs in PU-c. Each PE performs the multiplication of the input elements and the results are propagated to the PEs of the first column for aggregation of the partial products. The PEs in the first row will hold the results of the matrix vector product. For Cartesian product which is the basic operation in GD phase, PU-c reads a row vector and a column vector from the buffers, which are distributed to the PEs in a row-sharing manner and column-sharing manner, respectively. Each PE performs the multiplication and holds the results of the Cartesian product. After performing computations, the PU will collect results from the PEs and sends them back to the buffers.

The PE micro-architecture is also shown in Fig. 6, which can execute fixed multiply-accumulation operations. Each PE has three inputs: one input for receiving the control signals, the other two receiving the multiplication operands. Each has two outputs: one output for writing computation results to buffers; one output for propagating locally-stored results to neighbor PEs, so that they can be aggregated for accumulation.

## 4 EXPERIMENTAL METHODOLOGY

### 4.1 Implementation

We implement TNPU in Synopsys design flow on TSMC 65nm technology: simulating with Synopsys Verilog Compile Simulator (VCS), synthesizing with Synopsys Design Compiler (DC), analyzing power with Synopsys PrimeTime (PT), and placing them with Synopsys IC Compiler (ICC). TNPU is equipped with a PU containing PE array size of $17 \times 16$, where PU-c and PU-f contain $16 \times 16$ and $1 \times 16$, respectively. TNPU is clocked at 400MHz. We use conventional 32-bit floating point arithmetic operators to insure the training accuracy. The detailed designs of the buffer organization is implemented to match [3] as closely as possible. The bandwidth of the external memory bus is limited to 17.1 GB/s since we use single-channel DDR3-2133.

### 4.2 Benchmarks

We collected five representative CNN models as the benchmarks: LeNet-5, MPCNN, AlexNet, VGG11 and VGG16. Since it takes days even weeks to train large neural networks such as VGGNet and

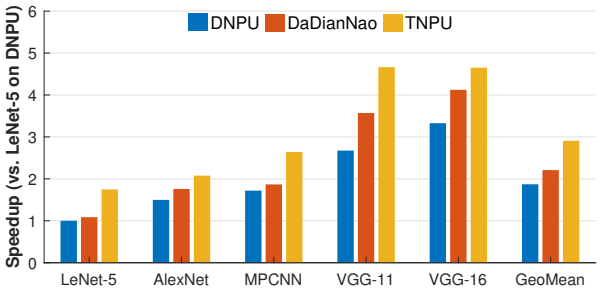



**Figure 7: CNN training performance comparison.**

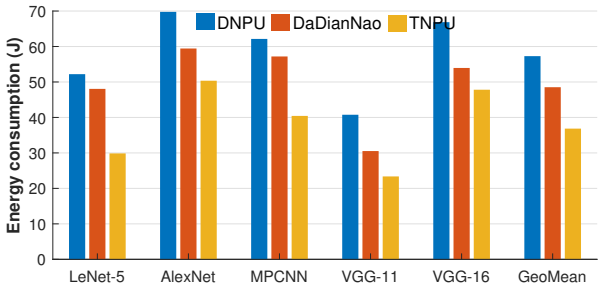



**Figure 8: Energy consumption comparison.**

AlexNet, we only clip an interval in the training process to make the time manageable. The training epoch is set as (50000,50000,100,10,10) respectively to make the training time of different models comparable. The training batch size is set as 50.

### 4.3 Baselines

We compare our design with two baselines, i.e. DaDianNao and DNPU. In DaDianNao, CVLs and FCLs share the computing resources in a time-division multiplexing manner. DNPU optimize CVLs and FCLs separately, using dedicated computation units, respectively. Because the DRAM bandwidth provisioning of prior solutions are widely ranged, it is unfair to make straightforward comparisons by simply comparing their experimental results. Moreover, there are many other optimization techniques in prior solutions which are hard to re-implement. Therefore, we created a cycle accurate simulator of the baseline accelerators following their design rationales. To make a fair comparison, we resized DNPU and DaDianNao to be equipped with the same number of multipliers with our design. 32-bit floating point operators are used in both of them and the off-chip memory bandwidth is also set as 17.1 GB/s. An energy model is used to evaluate the energy consumption. Because of significant differences in buffer sizing, buffer organization, and implementation choices, the baselines we evaluated cannot precisely represent the original proposals.

## 5 EXPERIMENTAL RESULTS

### 5.1 Performance

Fig. 7 summarizes the performance comparison between TNPU and the baselines which has been normalized to the performance of LeNet-5 on DNPU. Unsurprisingly, TNPU consistently outperforms the baselines on all of the benchmarks and is, on average 1.32× faster than DaDianNao and 1.55× faster than DNPU. The most striking result is that TNPU achieves a performance improvement of 74.8% and 60.1% on LeNet-5 over DNPU and DaDianNao, respectively. There are two reasons for the performance improvement. Firstly, TNPU can maintain high resource utilization by properly orchestrating CVLs and FCLs, leveraging the complementary effect of resource requirements of the two layer types. By contrast, DaDianNao shares the computing resource between CVLs and FCLs in a time-division multiplexing manner, which leads to underutilization of either computing resources or memory bandwidth resources. Specifically, DaDianNao incurs off-chip memory bandwidth under-utilization when processing CVLs because CVLs requires a relatively small amount of off-chip memory accesses, and incurs computing resource under-utilization when processing FCLs because FCLs only require a relatively small amount of computation with a huge amount of memory accesses, causing significant performance degradation. Secondly, TNPU efficiently eliminates the PE idles by SOSM for CNN training, maintaining high resource utilization. Although DNPU uses dedicated computation units for CVLs and FCLs, it incurs PE idles because of the bidirectional data dependency in CNN training, causing performance degradation.

The performance improvement of TNPU varies widely across the benchmarks. Specifically, TNPU performs only slightly better than the DaDianNao on VGG16 (13.1% improvement). The issue is that VGG is a CVL-dominant model, i.e. both the computation and memory access amount of FCLs are much less than CVLs. Hence, there exists little optimization opportunities by orchestrating CVLs and FCLs and conventional processor like DaDianNao can already achieve high resource utilization and performance. In other benchmarks, TNPU achieves consistently superior speedup higher than 1.3×.

### 5.2 Energy Consumption

In Fig. 8, we report the energy comparison of the architectures. The energy consumption we evaluated is an ideal version, where we assume that main memory accesses incur no energy cost. On average, TNPU reduces the energy consumption by 35.7% and 24.1% over DNPU and DaDianNao, respectively. Specifically, the energy reduction ranges from 27.8% (AlexNet) to 42.8% (LeNet-5) compared to DNPU, from 11.4% (VGG16) to 37.9% (LeNet-5) compared to DaDianNao. The high energy efficiency of TNPU stems from: 1) the improvement of performance, TNPU maintains high resource utilization and is free from performance degradation caused by bidirectional data dependency in CNN training; 2) low hardware overhead, only single computation unit is used. For comparison, DNPU uses two dedicated computation units which increases the design complexity, thereby overhead.

### 5.3 Scalability

Since off-chip memory bandwidth (BW) is an important metric for the performance on CNN training, we evaluate the scalability of the accelerators from the sensitivity to BW provisioning to study the scalability. Fig. 9 compares the geometric mean performance of the architectures with BW provisioning ranging from 11GB/s to 24GB/s. Note that since it is quite hard to precisely control the DRAM bandwidth provisioning, the results under different BW provision are derived simulation. With the decreasing of BW provisioning, TNPU maintains a stable high performance and consistently and significantly outperforms the baselines. By contrast, the performance of

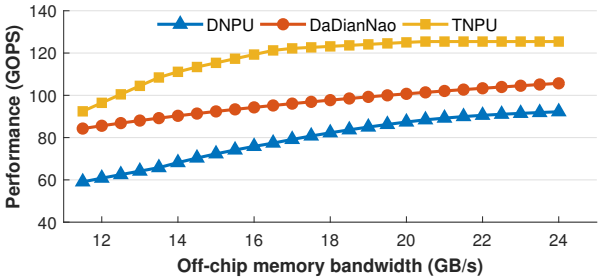



**Figure 9: Scalability to off-chip memory bandwidth.**

DNPU and DaDianNao degrade gradually and significantly. The reason is that the performance improvement of FCLs relies heavily on the BW provisioning. With surprisingly low BW provisioning, FCLs become the bottleneck. The results highlights that TNPU has good scalability.

## 6 RELATED WORK

In recent years, accelerating deep learning applications has been one of the hottest research tops in computer architecture. This section reviews some of these recent efforts.

Many neural network accelerators focused on optimizing CVLs from computing perspective or memory perspective. DianNao [2] designed a general neural network accelerator which focuses on memory bandwidth utilization. Work [16] focuses on the balance between computing resource and memory bandwidth for CVLs. Work [15] proposed a methodology to find the optimal accelerator design for any CNN model implementation. Eyeriss [4] is a recent ASIC CNN accelerator that couples a compute grid with a NoC, enabling flexibility in scheduling CNN computation. FlexFlow [11] proposed a flexible dataflow architecture which exploits the different parallelism schemes in CVLs. These solutions only optimize for CVLs, and offer little insight on the optimization of FCLs. They inevitably incur performance degradation when accommodating CNN training applications.

Sparse CNNs have emerged as an effective solution to reduce the amount of computation and memory accesses while maintaining the high accuracy. Cambricon-X [17] and Cnvlutin [1] removes the ineffective computations from zero weights and activations, respectively. SCNN [12] leverages the sparsity in both weights and activations. EIE [6] is an energy efficient inference engine that performs inference on the compressed FCLs and accelerates the resulting sparse matrix-vector multiplication. However, the pruning techniques can only be used after training CNNs, hence these solutions cannot address the challenges in CNN training.

## 7 CONCLUSION

In this paper, we describe and evaluate TNPU, an efficient accelerator architecture for CNN training that leverages the complementary effect of the resource requirements between CVLs and FCLs. Unlike prior approaches optimizing CVLs and FCLs in separate way, we take an alternative by smartly orchestrating the computation of CVLs and FCLs in single computing unit to work concurrently so that both computing and memory resources will maintain high utilization, thereby boosting the performance. We also proposed a simplified out-of-order scheduling mechanism to address the bidirectional data dependency problems in CNN training. Average on five representative benchmarks, our design is on average 1.5× and 1.3× faster than comparably provisioned DNPU and DaDianNao, respectively. Thanks to its high performance, TNPU achieves an average energy reduction of 35.7% and 24.1% over DaDianNao and DNPU, respectively.

## ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61572470, 61872336, 61532017, 61432017, 61521092, 61376043, and in part by Youth Innovation Promotion Association, CAS under grant No.Y404441000. The corresponding authors are Guihai Yan and Xiaowei Li.

## REFERENCES

[1] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *Proceedings of the 43rd ISCA*.

[2] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. [n. d.]. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th ASPLOS*.

[3] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proc. of MICRO*.

[4] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* (2017).

[5] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: shifting vision processing closer to the sensor.

[6] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd ISCA*.

[7] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*. 1135–1143.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*.

[9] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. 2018. CCR: A concise convolution rule for sparse neural network accelerators. In *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 189–194.

[10] J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li. 2018. SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators. In *2018 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 343–348.

[11] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A Flexible Dataflow Accelerator Architecture for Convolutional Neural Networks. In *2017 IEEE 23th HPCA*.

[12] Angshuman Parashar et al. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of 44th ISCA*. ACM.

[13] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proc. of FPGA*.

[14] D. Shin, J. Lee, J. Lee, J. Lee, and Hoi-Jun Yoo. 2017. An energy-efficient deep learning processor with heterogeneous multi-core architecture for convolutional neural networks and recurrent neural networks. In *2017 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. 1–2. https://doi.org/10.1109/CoolChips.2017.7946376

[15] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proc. of FPGA*.

[16] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 FPGA*.

[17] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Proc. of MICRO*.