2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

PERMDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices

Chunhua Deng*+
City University of New York
chunhua.deng@rutgers.edu

Siyu Liao*+
City University of New York
sl1583@scarletmail.rutgers.edu

Yi Xie+
City University of New York
yx238@scarletmail.rutgers.edu

Keshab K. Parhi
University of Minnesota, Twin Cities
parhi@umn.edu

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

Bo Yuan+
City University of New York
bo.yuan@soe.rutgers.edu

***Abstract*—Deep neural network (DNN) has emerged as the most important and popular artificial intelligent (AI) technique. The growth of model size poses a key energy efficiency challenge for the underlying computing platform. Thus, model compression becomes a crucial problem. However, the current approaches are limited by various drawbacks. Specifically, network sparsification approach suffers from irregularity, heuristic nature and large indexing overhead. On the other hand, the recent structured matrix-based approach (i.e., CIRCNN) is limited by the relatively complex arithmetic computation (i.e., FFT), less flexible compression ratio, and its inability to fully utilize input sparsity.**

To address these drawbacks, this paper proposes PERMDNN, a novel approach to generate and execute hardware-friendly structured sparse DNN models using permuted diagonal matrices. Compared with unstructured sparsification approach, PERMDNN eliminates the drawbacks of indexing overhead, non-heuristic compression effects and time-consuming retraining. Compared with circulant structure-imposing approach, PERMDNN enjoys the benefits of higher reduction in computational complexity, flexible compression ratio, simple arithmetic computation and full utilization of input sparsity. We propose PERMDNN architecture, a multi-processing element (PE) fully-connected (FC) layer-targeted computing engine. The entire architecture is highly scalable and flexible, and hence it can support the needs of different applications with different model configurations. We implement a 32-PE design using CMOS 28nm technology. Compared with EIE, PERMDNN achieves $3.3\times \sim 4.8\times$ higher throughout, $5.9\times \sim 8.5\times$ better area efficiency and $2.8\times \sim 4.0\times$ better energy efficiency on different workloads. Compared with CIRCNN, PERMDNN achieves $11.51\times$ higher throughput and $3.89\times$ better energy efficiency.

***Index Terms*—Deep Learning, Model Compression, VLSI**

I. INTRODUCTION

Starting their resurgence from Hinton's seminal paper [1], neural networks [2] have emerged as today's most important and powerful artificial intelligence (AI) technique. Thanks to the availability of unprecedentedly abundant training/test data and the significant advances in computers' processing speed, large-scale deep neural networks (DNNs) have been able to deliver record-breaking accuracy results in many tasks that demand intelligence, such as speech recognition [3], object recognition [4], natural language processing [5] etc.

The extraordinary performance of DNNs with respect to high accuracy is mainly attributed to their very large model sizes [6]–[8]. As indicated in a number of theoretical analysis [9] [10] and empirical simulations [11]–[13], scaling up the model sizes can improve the overall learning and representation capability of the DNN models, leading to higher classification/predication accuracy than the smaller models. Motivated by these encouraging findings, the state-of-the-art DNNs continue to scale up with the purpose of tackling more complicated tasks with higher accuracy.

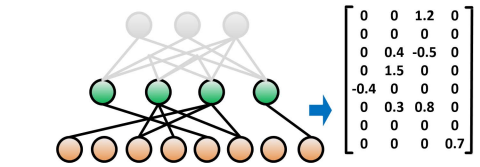
On the other hand, from the perspective of hardware design, the continuous growth of DNN model size poses a key energy efficiency challenge for the underlying computing platforms. As pointed out in [14], since the size of on-chip SRAM is usually very limited, placing the large-scale DNN models on the off-chip DRAM, which has more than 100 times higher energy cost than SRAM, is a bitter but inevitable choice. Consequently, high energy cost incurred by the frequent access to DRAM makes the energy-efficient deployment of DNN systems very challenging.

To improve energy efficiency, efficient model compression has emerged as a very active topic in AI research community. Among different types of DNN compression techniques [15]–[18], two approaches show promising results. First, *network sparsification* is believed to be the most popular and state-of-the-art strategy because of its good balance between compression ratio and test accuracy. To date, many methods [15] [16] [18] have been proposed to perform efficient sparsification on different DNN models. Echoing the importance and popularity of this approach, several sparse model-oriented hardware architectures [19]–[22] have been proposed .

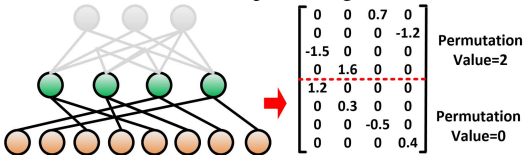
However, the current network sparsification methods suffer from the inherent drawbacks of *irregularity*, *heuristic nature* and *indexing overhead*. Consequently, despite the encouraging compression ratio, the *unstructured* sparse DNN models cannot achieve optimal performance on the current computing platforms, especially DNN hardware accelerators. Essentially, the inefficient execution and hardware implementation are

*Chunhua Deng and Siyu Liao contribute equally to the paper.
+Chunhua Deng, Siyu Liao, Yi Xie and Bo Yuan are now with Rutgers University.

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00024



(a) Unstructured sparse weight matrix.



(b) Block-permuted diagonal weight matrix.

Fig. 1: Weight representation by using (a) conventional unstructured sparse matrix. (b) block-permuted diagonal matrix.

caused by the *non-hardware-friendly* sparse models.

To overcome the drawbacks of the irregularity, an alternative approach is to directly represent the network with *structured matrices* [23]. A notable recent work is CIRCNN [24], which represents weights using block-circulant matrices and partitions the weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ into blocks of square circulant sub-matrices (e.g., $\mathbf{W}_{ij}$ of size $k \times k$). Due to circulant structure, only k (instead of k^2) weights need to be stored for each sub-matrix. The calculation of $\mathbf{W}_{ij}\mathbf{x}_j$ can be performed as $\text{IFFT}\big(\text{FFT}(\mathbf{w}_{ij}) \circ \text{FFT}(\mathbf{x}_j)\big)$, where $\circ$ denotes element-wise multiplications and $\mathbf{w}_{ij}$ is the first row of $\mathbf{W}_{ij}$. Utilizing FFT-based fast multiplication, it simultaneously reduces computational cost and storage cost with small accuracy loss.

Due to the hardware-friendly model representation, CIRCNN demonstrated the efficient hardware implementation and better performance. However, it is still limited by the relatively complex arithmetic computation (i.e., FFT), less flexible compression ratio, and inability to fully utilize input sparsity.

In this paper, we propose PERMDNN, a novel approach that generates and executes hardware-friendly structured sparse DNN models using *permuted diagonal matrices*. As illustrated in Fig. 1(b), permuted diagonal matrix is a type of structured sparse matrix that places all the non-zero entries in the diagonal or permuted diagonal. When the weight matrices of DNNs can be represented in the format of multiple permuted diagonal matrices (so-called *block-permuted diagonal matrices*), their inherent strong structured sparsity leads to great benefits for practical deployment. Specifically, it eliminates indexing overhead, brings non-heuristic compression effects, and enables re-training-free model generation.

While both methods impose the structure on the construction of weight matrices, PERMDNN offers three advantages over CIRCNN: 1) *Simpler arithmetic computation*. Unlike CIRCNN, which inherently requires complex number operation (complex multiplication and addition) because of the use of FFT, PERMDNN is purely based on real number arithmetic, thereby leading to low hardware cost with the same compression ratio; 2) *Significantly better flexibility*. PERMDNN hardware can freely support different sizes of permuted diagonal matrices, while CIRCNN hardware is limited to only support 2^t-size circulant matrix because most of FFT hardware is 2^t-point-based. This means that PERMDNN is much more flexible for the choice of compression ratio; 3) *Full utilization of input sparsity*. PERMDNN can fully utilize the dynamic sparsity in the input vectors but CIRCNN cannot. The reason is that CIRCNN can only process frequency-domain input vectors that lose important time-domain sparsity. It allows PERMDNN to achieve additional improvements in throughput and energy efficiency over CIRCNN.

Based on PERMDNN, we develop an end-to-end training scheme that can generate a high-accuracy permuted diagonal matrix-based DNN models from scratch. We also develop the corresponding low-complexity inference scheme that executes efficiently on the trained structured sparse DNN models. Experimental results on different datasets for different application tasks show that, enforced with the strong structure, the permuted diagonal matrix-based DNNs achieve high sparsity ratios with no or negligible accuracy loss.

To accelerate inference, we propose PERMDNN architecture, a high-performance permuted diagonal matrix-based inference engine that targets the fully-connected (FC) layers of DNN models. The PERMDNN architecture is designed to fully reap the benefits of permuted diagonal matrix-based structure. Unlike EIE [14], the state-of-the-art DNN architecture targeting FC layer, PERMDNN architecture does not incur index overhead and load imbalance due to the irregularity of conventional unstructured sparse DNN models, thereby leading to significant improvement in hardware performance. With an array of processing elements (PEs), PERMDNN architecture is an elastic and scalable architecture that can adapt to different sizes of DNN models, size of component permuted diagonal matrices, and number of PEs. This allows the architecture to be deployed in various application scenarios that have different requirements on power, area and throughput.

To demonstrate the advantages of PERMDNN architecture, we implement a 32-PE design using CMOS 28nm technology. Operating on 1.2GHz clock frequency, the PERMDNN implementation consumes 703.4mW and 8.85mm^2. Meanwhile, equipped with 8 multipliers in each PE, the processing power of 32-PE PERMDNN achieves 614.4GOPS for a compressed DNN model, which approximately corresponds to 14.74TOPS on an uncompressed network. Compared to EIE, PERMDNN achieves $3.3\times \sim 4.8\times$ higher throughout, $5.9\times \sim 8.5\times$ better area efficiency and $2.8\times \sim 4.0\times$ better energy efficiency on different workloads. Compared to CIRCNN, PERMDNN achieves $11.51\times$ higher throughput and $3.89\times$ better energy efficiency.

II. MOTIVATION

A. Importance of FC Layer-targeted Architecture

Similar to EIE, PERMDNN is a customized DNN architecture that targets for FC layers. Specifically, PERMDNN accelerates the sparse matrix-vector multiplication ($M \times V$),

which is the kernel computation of any FC layers. We justify the importance of FC layer with the following three arguments.

First, FC layer has a wide spectrum of application scenarios. As summarized in Table I, besides being used in convolutional neural network (CNN) for computer vision tasks, FC layer is also the dominant layer in many other types of DNNs (e.g., recurrent neural network (RNN[1]) and multi-layer perceptrons (MLP) for speech recognition and natural language processing tasks. Therefore, optimizing FC layer can universally benefit a full spectrum of AI tasks.

Second, real-world industrial needs call for efficient FC layer-targeted DNN architecture. As revealed in Googles seminal TPU paper [25], *more than 95%* workload in Google's datacenters are processed by those FC layer-centered DNNs, such as RNNs and MLPs. Therefore, [25] called for more research efforts on FC layers, which is currently far less active than the effort for convolutional (CONV) layers.

Finally, optimized FC layer-targeted architecture can improve the overall performance of CONV layer-centered models (e.g., CNNs). As analyzed in [20], forcing a CONV layer-targeted architecture to execute FC layers will cause noticeable performance degradation. We believe that, to realize an efficient architecture for CNNs with both CONV and FC layers, a specialized design that optimizes for FC layer is a preferable strategy to maximize hardware performance.

TABLE I: Types of DNN models used in practical AI fields.

Applications	DNN Models	Component Layer
Computer Vision	CNN	CONV layer (Main) FC layer
Speech Recognition	RNN, MLP	FC layer
Natural Language Processing	RNN, MLP	FC layer

B. Drawbacks of Unstructured DNN Sparsification

Due to the large memory requirement and the inherent redundancy, FC layers are usually compressed with network sparsification, such as heuristic pruning [15] and conducting certain regularization [18]. The downside of the approach is that the unstructured sparse models are not friendly to be implemented in DNN hardware accelerators.

First, the structure of their generated sparse DNNs is usually highly irregular [15] or only exhibits weak regularity [18]. It incurs significant space/computation overhead due to indexing the irregular sparse weight matrices. For instance, in EIE, each weight requires 4-bit virtual weight tag to represent its actual value and additional 4 bits to record its relative position in the entire weight matrix. Therefore, the overall storage cost for one weight is actually 8 bits instead of 4 bits, significantly limiting the achievable performance.

Second, the compression ratio of DNN sparsification is typically heuristic. This is because the inherent process of weight/neuron pruning or regularization-caused sparsification is usually uncontrollable and unpredictable. Consequently, in the scenario of compression effect is pre-defined, precisely controlling the compression ratio to satisfy the design specification becomes challenging.

Moreover, the existing DNN sparsification approaches introduce large training overhead. Operating on the pre-trained dense model, weight/neuron pruning and retraining are performed iteratively to ensure the original accuracy of the dense model can also be achieved by the sparse model. Clearly, the process is time-consuming, — sometimes it may even take longer time to re-train sparse models than the dense models. For instance, as reported in [26], in order to get a fair compression rate on a pre-trained AlexNet model, it takes 4800K and 700K iterations by using the pruning methods in [15] and [26], respectively, while training this pre-trained model only takes 450K iterations.

C. Drawbacks of Circulant Matrix-based Compression

An alternative approach to achieve network compression is to directly represent the network with *structured matrices*. As an example, CIRCNN utilizes the mathematical property of circulant matrix to compress DNN models that can be efficiently implemented in hardware. While avoiding the irregularity, CIRCNN suffers from a different set of drawbacks. First, it requires high-cost arithmetic operation. The training and inference algorithms of CIRCNN are based on FFT computation, which is inherently involved with complex multiplication and addition. Unfortunately, arithmetic operations on complex numbers incur much higher cost than their counterparts on real numbers. For instance, one complex multiplication requires four real multiplications and two real additions.

Moreover, it lacks flexibility on compression ratio. The compression ratio of CIRCNN is determined by the size of component circulant matrix. However, since this size also determines the length of FFT, in order to facilitate FFT hardware design that mostly uses 2^t-point FFT[2], the compression ratio of each layer of CIRCNN has to be selected among the values of 2^t. Obviously, such restriction severely limits the potential applications of CIRCNN.

Finally, it loses the opportunity of utilizing input sparsity. As discussed in [14] [21], utilizing the dynamic input sparsity is an important technique to reduce the power consumption and computational time of DNN hardware. However, since CIRCNN processes the input vector in the frequency domain, its original abundant sparsity in time domain is completely lost. It prevents CIRCNN from utilizing the important input sparsity for performance improvement.

III. PERMDNN: ALGORITHMS & BENEFITS

A. PERMDNN *Representation*

To address the drawbacks of the current sparsification-based and structured matrix-based methods, we propose PERMDNN, a new structured sparse DNN representation, shown in Fig. 1(b). Specifically, we enforce that the weight matrices of

[1]In this paper the FC in RNN specifically means the component weight matrices in RNN.

[2]non-2^t-point FFT has much more complex hardware than 2^t-point FFT.

the DNN models consist of multiple permuted diagonal sub-matrices, where all the non-zero entries of the permuted diagonal matrices locate in the diagonals or permuted diagonals. In general, the m-by-n weight matrix of one layer of PERMDNN contains mn/p^2 p-by-p permuted diagonal sub-matrices[3]. As it will be shown later, this type of *block-permuted diagonal weight matrix* exhibits strong spatial structure that brings significant reduction in space cost and computational cost.

B. Inference and Training Schemes of PERMDNN

Based on the representation of PERMDNN, we develop the computation procedures of forward and backward propagation that are the key parts for inference and training process.

In general, we assume that the weight matrix of a fully-connected layer of PERMDNN is an m-by-n block-permuted diagonal matrix $\mathbf{W} \in \mathbb{R}^{m\times n}$. Here $\mathbf{W}$ consists of multiple p-by-p permuted diagonal sub-matrices. Considering there are $(m/p) \times (n/p)$ sub-matrices in total and each has its own permutation parameter, we index all the sub-matrices from 0 to $(m/p)\times(n/p)-1$ and denote k_l as the value of permutation parameter for the l-th permuted diagonal sub-matrix. Then, for arbitrary entry w_{ij} of $\mathbf{W}$, its value can be represented as:

$$w_{ij} = \begin{cases} \mathrm{q}_{k_l\times p+c} & \text{if } (c+k_l) \mod p \equiv d \\ 0 & \text{otherwise} \end{cases}, \tag{1}$$

where $c \equiv i \mod p$, $d \equiv j \mod p$ and $l = (i/p) \times (n/p) + (j/p)$. Here $\mathbf{q} = (q_0, q_1, \ldots, q_{mn/p-1})$ is the vector that contains all the non-zero entries of $\mathbf{W}$.

Forward Propagation for Inference. Recall that the forward propagation during inference phase for FC layer is performed as $\mathbf{y} = \psi(\mathbf{a}) = \psi(\mathbf{W}\mathbf{x})$[4], where $\psi(\cdot)$ is the activation function and $\mathbf{a} = \mathbf{W}\mathbf{x} = (a_1, a_2, \ldots, a_m)^T$. In addition, $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ and $\mathbf{y} = (y_1, y_2, \ldots, y_m)^T$ are the input and output vectors of FC layer, respectively. Accordingly, when $\mathbf{W}$ is the trained block-permuted diagonal weight matrix with entries that are represented by Eqn. (1), the calculation of a_i, as the key computation in the forward propagation, can now be simplified as $a_i = \sum_{g=0}^{n/p-1} w_{ij}x_j$, where $j \equiv (i + k_l) \mod p + gp$ and $l = g + (i/p) \times (n/p)$. Notice that here for weight storage, only the mn/p -length vector $\mathbf{q}$ that contains all the non-zero entries of W needs to be stored in the inference phase. Also, the calculation of a_i is significantly simplified from the original computation as $a_i = \sum_{g=0}^{n-1} w_{ij}x_j$.

Backward Propagation for Training. When the FC layer is imposed with permuted diagonal structure, we need to ensure the FC layer of PERMDNN always exhibits permuted diagonal structure in each iteration of training phase. Recall that the forward propagation for FC layer is performed as $\mathbf{y} = \psi(\mathbf{a})$ where $\mathbf{a} = (a_1, a_2, \ldots, a_m)^T$. Then, the gradient calculation, which is the key step of the backward propagation, is performed as $\frac{\partial J}{\partial w_{ij}} = x_j \frac{\partial J}{\partial a_i}$, where J is the loss function [27] of the neural network. Notice that according to the principle of backpropagation [27], x_i in the current layer will be backward propagated to the previous layer as a_i. Therefore, together with the above equation, the weight updating rule for the FC layer of PERMDNN can be derived as:

$$w_{ij} \leftarrow w_{ij} - \epsilon x_j \frac{\partial J}{\partial a_i}, \text{for any } w_{ij} \neq 0, \tag{2}$$

$$\frac{\partial J}{\partial x_j} = \sum_{g=0}^{m/p-1} w_{ij} \frac{\partial J}{\partial a_i}, \tag{3}$$

where $i \equiv (j+p-k_l) \mod p+gp$, $l = gn/p+j/p$ and ϵ is the learning rate. Here in Eqn. (2) the calculation of $\frac{\partial J}{\partial a_i}$ is aided with the computation in Eqn. (3) since each x_i in the current layer will be backward propagated to the previous layer as a_i. Notice that here the weight update scheme described in Eqns. (2) (3) theoretically guarantees the trained sparse network always exhibits block-permuted diagonal structure and hence provides attractive end-to-end training solution.

C. Extension to Convolutional Layer

Forward Propagation for Inference on CONV Layer. The idea of imposing permuted diagonal structure on the weight matrix of FC layer can be further generalized and applied to the weight tensor of CONV layer. Recall that the weight tensor of a CONV layer is a 4D tensor that can be viewed as a "macro" matrix with each entry being a filter kernel; therefore, as illustrated in Fig. 2, the permuted diagonal structure can be imposed on the input channel and output channel dimensions of the weight tensor. Then, similar to the case for FC layer, the forward propagation for inference on CONV layer using permuted diagonal matrix can be described as follows:

$$\mathcal{Y}(i,x,y) = \sum_{g=0}^{c_2/p-1} \sum_{w=0}^{w_1-1} \sum_{h=0}^{h_1-1} \mathcal{F}(i,j,w,h)\mathcal{X}(j,x-w,y-h), \tag{4}$$

where $\mathcal{X} \in \mathbb{R}^{c_0\times w_0\times h_0}$, $\mathcal{Y} \in \mathbb{R}^{c_2\times w_2\times h_2}$, $\mathcal{F} \in \mathbb{R}^{c_0\times c_2\times w_1\times h_1}$ represent the input, output and weight tensors of convolutional layer, respectively. Here w_i and h_i for i=0,1,2 are the width and height of the input, kernel and output tensor, respectively. c_0 and c_2 are the numbers of input channels and output channels. In addition, $j \equiv (i + k_l) \mod p + gp$ and $l = g + (i/p) \times (n/p)$.

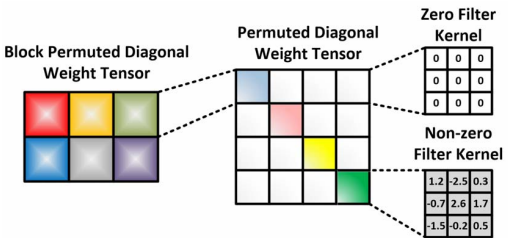


Fig. 2: Block-permuted diagonal weight tensor of CONV layer.

Backward Propagation for Training on CONV Layer. Similar to the FC layer, in order to ensure the convolutional layer of PERMDNN always exhibits the permuted diagonal structure during the training phase, the corresponding weight

[3]When m or n is not divided by p, zeroes are padded. It will not cause extra overhead since padded zeroes are not involved in computation/storage.

[4]Bias is combined with $\mathbf{W}$ for simplicity.

updating procedure needs to be re-designed. Following the similar steps of deriving training procedure for FC layer, the weight update rule in the back propagation on CONV layer can be described as follows:

$$\mathcal{F}(i,j,w,h) \leftarrow \mathcal{F}(i,j,w,h) - \epsilon \sum_{x=0}^{w_2}\sum_{y=0}^{h_2} \mathcal{X}(i,x-w,y-h) \times \frac{\partial J}{\partial \mathcal{Y}(i,x,y)}, \text{for any } \mathcal{F}(i,j,w,h) \neq 0, \quad (5)$$

$$\frac{\partial J}{\partial \mathcal{X}(i,j,x)} = \sum_{g=0}^{c_0/p-1}\sum_{w=0}^{w_1-1}\sum_{h=0}^{h_1-1} \mathcal{F}(i,j,w,h)\frac{\partial J}{\partial \mathcal{Y}(i,x+w,y+h)}, \quad (6)$$

where $i = (j+p-k_l) \mod p + gp$ and $l = gn/p + j/p$. Notice that here in Eqn. 5 the calculation of $\frac{\partial J}{\partial \mathcal{Y}(i,x,y)}$ is aided with the computation in Eqn. 6 since each $\mathcal{X}(j,x,y)$ in the current layer will be backward propagated to the previous layer as $\mathcal{Y}(j,x,y)$. Again, the weight update scheme described in Eqns. (5) (6) theoretically guarantees the trained sparse network always exhibits block-permuted diagonal structure.

D. Test Accuracy and Compression Ratio

By leveraging the forward and backward propagation schemes in Eqns. (2)-(6), the PERMDNN models can be trained from scratch and tested. Table II - Table V show the task performance and compression ratio of different PERMDNN models on different types of datasets. Notice that for one FC/CONV layer with block size p for its permuted diagonal (PD) weight matrix/tensor, the compression ratio for that layer is p. The details of the experimental setting are described as follows:

- **AlexNet [28]:** The block sizes (p) for the permuted diagonal weight matrices of three FC layers (FC6, FC7 and FC8) are set as different values (10, 10 and 4) for different FC layers.
- **Stanford Neural Machine Translation (NMT) [29]:** This is a stacked LSTM model containing 4 LSTMs with 8 FC weight matrices for each LSTM. In the experiment the value of p for all the FC layers is set as 8.
- **ResNet-20 [11]:** In this experiment for the group of CONV layers without 1x1 filter kernel, the value of p is set as 2. For the group of CONV layers with 1x1 filter kernel, p is set as 1.
- **Wide ResNet-48 [12]:** The widening parameter of this model is 8. For the group of CONV layers without 1x1 filter kernel, the value of p is set as 4. For the group of CONV layers with 1x1 filter kernel, p is set as 1.
- **Selection of Permutation value (k_l):** k_l can be selected via either natural indexing or random indexing. Our simulation results show no difference between task performance for these two setting methods. Table II - Table V are based on natural indexing. For instance, for a 4-by-16 block-permuted diagonal weight matrix with $p = 4$, $k_0 \sim k_3$ is set as $0 \sim 3$.

Table II - Table V show that imposing permuted diagonal structure to DNN models enables significant reduction in the weight storage requirement for FC or CONV layers. Meanwhile, the corresponding task performance, in terms of test accuracy (for computer vision) or BLEU scores [30] (for language translation), are still retained as the same or only exhibits negligible degradation. In short, PERMDNN models can achieve high compression ratios in network size and strong spatial network structure, and simultaneously, preserve high task performance. It makes the corresponding hardware-friendly architecture (Section IV) very attractive.

TABLE II: AlexNet on ImageNet [4]. PD: Permuted Diagonal.

AlexNet	Block size (p) for PD weight matrix of FC6-FC7-FC8	Top-5 Acc.	Compression for overall FC layers
Original 32-bit float	1-1-1	80.20%	234.5MB(1×)
32-bit float with PD	10-10-4	80.00%	25.9MB(9.0×)
16-bit fixed with PD	10-10-4	79.90%	12.9MB(18.1×)

TABLE III: Stanford NMT (32-FC layer LSTMs) on IWSLT15 for English-Vietnamese Translation.

Stanford NMT (32-FC layer LSTMs)[5]	Block size (p) for PD weight matrix of ALL FC Layers	BLEU Points	Compression for overall FC layers
Original 32-bit float	1	23.3	419.4MB(1×)
32-bit float with PD	8	23.3	52.4MB(8×)
16-bit fixed with PD	8	23.2	26.2MB(16×)

TABLE IV: ResNet-20 on CIFAR-10 [31].

ResNet-20	Block size (p) for PD weight tensor of CONV layers	Acc.	Compression for overall CONV Layers
Original 32-bit float	1	91.25%	1.09MB(1×)
32-bit float with PD	2 for most layers	90.85%	0.70MB(1.55×)
16-bit fixed with PD	2 for most layers	90.6%	0.35MB(3.10×)

TABLE V: Wide ResNet-48 on CIFAR-10.

Wide ResNet-48	Block size (p) for PD weight tensor of CONV layers	Acc.	Compression for overall CONV layers
Original 32-bit float	1	95.14%	190.2MB(1×)
32-bit float with PD	4 for most layers	94.92%	61.9MB(3.07×)
16-bit fixed with PD	4 for most layers	94.76%	30.9MB(6.14×)

E. Outline of Theoretical Proof on Universal Approximation

In CIRCNN the *universal approximation property* of block-circulant matrix-based neural network was given to theoretically prove the effectiveness of using circulant matrices. In this work we discover that the PERMDNN also exhibits the universal approximation property, thereby making the rigorous foundation for our proposed permuted diagonal structure-imposing method. The details of the proof will be provided in an individual technical report and this subsection gives a brief outline of the proof as follows. First, we prove that the "connectedness" of PERMDNN, – that means, thanks to the unique permuted diagonal structure of each block, when k_l is not identical for all permuted diagonal matrices, the sparse connections between adjacent block-permuted diagonal layers do not block away information from any neuron in the previous layer. Based on this interesting property, we further prove that the function space achieved by the block-permuted

[5]Here one FC in LSTM means one component weight matrix.

diagonal networks is dense. Finally, with the Hahn-Banach Theorem we prove that there always exists a block-permuted diagonal neural network that can closely approximate any target continuous function defined on a compact region with any small approximation error, thereby showing the universal approximation property of block-permuted diagonal networks. Besides, we also derive that the error bound of this approximation error is in the order of $O(1/n)$, where n is the number of model parameters. Consequently, the existence of universal approximation property of PERMDNN theoretically guarantees its effectiveness on different DNN types and applications.

F. Applicability on the Pre-trained Model

Besides training from scratch, the permuted diagonal matrix-based network model can also be obtained from a pre-trained dense model. Fig. 3 illustrates the corresponding procedure, which consists of two steps: permuted diagonal approximation and re-training/fine-tuning. First, the original dense weight matrices/tensors need to be converted to permuted diagonal matrices/tensors via *permuted diagonal approximation*. The mechanism of permuted diagonal approximation is to convert a non-permuted diagonal matrix/tensor to a permuted diagonal format by only keeping the entries in the desired permuted diagonal positions. **Mathematically, such approximation is the optimal approximation in term of l_2 norm measurement on the approximation error**. After that, the converted model already exhibits permuted diagonal structure and then can be further re-trained/fine-tuned by using Eqns. (2)-(6) to finally obtain a high-accuracy permuted diagonal network model. Such two-step generating approach is applied to all types of pre-trained models and can lead to high accuracy. For instance, for pre-trained dense LeNet-5 model on MNIST dataset, with $p = 4$ for CONV layer and $p = 100$ for FC layer, the finally converted permuted-diagonal network after re-training achieves 99.06% test accuracy and overall $40\times$ compression ratio without using quantization.

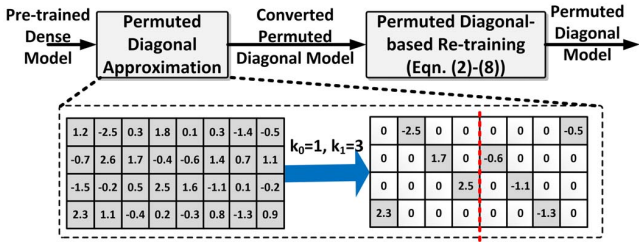


Fig. 3: Train a PERMDNN from a pre-trained dense model.

G. PERMDNN *vs Unstructured Sparse DNN*

Compared to the existing network sparsification approaches, the proposed PERMDNN enjoys several attractive advantages:

First, PERMDNN *is a hardware-friendly model*. As illustrated in Fig. 4, due to the inherent regular structure of permuted diagonal matrix, the position of each non-zero entry can now be calculated using very simple modulo operation[6], thereby completely eliminating the needs of storing the indices of entries. From the perspective of hardware design, this elimination means that the permuted diagonal matrix-based PERMDNN completely avoids the huge space/computation overhead incurred by the complicated weight indexing/addressing in the state-of-the-art sparse DNN accelerator (e.g. EIE), and hence achieves significant reduction in the space requirement and computational cost for DNN implementations.

Second, PERMDNN *provides controllable and adjustable compression and acceleration schemes.* The reductions in the model sizes and number of arithmetic operations are no longer heuristic based and unpredictable. Instead it can now be precisely and deterministically controlled by adjusting the value of p. This further provides great benefits to explore the design space exploring the tradeoff between hardware performance and test accuracy.

Finally, PERMDNN *enables direct end-to-end training while preserving high accuracy*. Since the structure of sparse weight matrices of PERMDNN can now be pre-determined by the model designers at the initialization stage of training, with our training algorithm that preserves this fixed structure, the entire structured sparse network can be trained from scratch, completely avoiding the increasing complexity incurred by the extra iterative pruning and/or re-training process in the conventional unstructured sparse DNN training schemes.

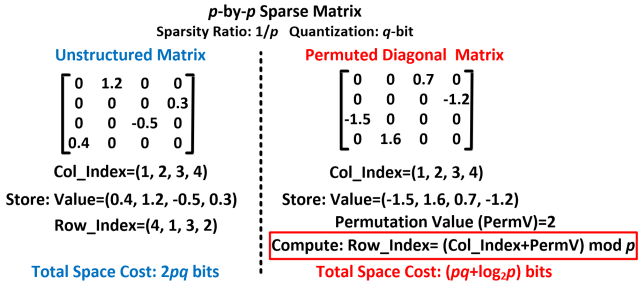


Fig. 4: Storage requirement comparison.

H. PERMDNN *vs* CIRDNN

While PERMDNN and CIRDNN are both based on the structured matrices, PERMDNN does not suffer from the drawbacks of CIRDNN. Table VI compares the two methods.

Most importantly, PERMDNN uses simpler arithmetic computation. Different from complex number computation-based CIRDNN, the computation of PERMDNN is purely based on real numbers, thereby requiring significantly lower arithmetic cost than CIRDNN with the same workload and same compression ratio.

Second, PERMDNN allows compression ratio to be flexibly adjusted. Different from CIRDNN, the computation in PERMDNN does not have any restrictions on the size of component permuted diagonal matrix. Therefore, the hardware architecture of PERMDNN can support different compression ratios based on the needs. This makes PERMDNN more attractive for a wide range of applications.

Finally, the fundamental distinction is that PERMDNN can fully utilize input sparsity. Since the computation of PERMDNN is in the time domain, the important input sparsity can

[6]As shown in Fig. 9, modulo circuit is simple by using LSB bits.

still be leveraged by PERMDNN to further reduce computational cost and power consumption. Because sparsity of input vectors widely exists in numerous applications, this benefit greatly expands the advantage of PERMDNN over CIRDNN.

TABLE VI: Advantages of PERMDNN over CIRCNN.

	CIRCNN	**PERMDNN**
Arithmetic Operation	Complex number-based	Real number-based
Flexible Compression	No	Yes
Utilize Input Sparsity	No	Yes

IV. PERMDNN: ARCHITECTURE

In this section, we develop the PERMDNN architecture based on the proposed structure and algorithms. Similar to most existing DNN accelerators [19] [20] [14], the PERMDNN architecture is designed for inference tasks.

A. Data Mapping and Processing Scheme

In general, the proposed computing engine is a scalable array of processing elements (PEs). Each PE performs the forward propagation on part of the weight matrix. Different PEs perform independent and parallel computations to maximize the processing throughput. Fig. 5 illustrates this partition scheme on an 8-by-8 block-permuted diagonal weight matrix with $p = 4$. We can see that the 4-by-4 permuted diagonal sub-weight matrices belonging to the same block row are processed by the same PE[7]. To support such arrangement, the non-zero entries of these sub-matrices are stored in each PE's associated SRAM. Therefore, the entire weight matrix of the FC layer is stored in a distributed manner across multiple SRAM banks to enable parallel processing of multiple PEs.

Besides, in order to fully utilize the potential dynamic sparsity in the input vector $\mathbf{x}$ (also the activation vector output $\mathbf{y}$ from previous layer), the entire PERMDNN adopts *column-wise* processing style. Specifically, as illustrated in Fig. 5, in each clock cycle all PEs perform the multiplications between x_i, the non-zero entry of $\mathbf{x}$, and $\mathbf{w_i}$, the corresponding column vector of weight matrix $\mathbf{W}$. The products are then accumulated for updating the intermediate values of the corresponding entries in the output vector $\mathbf{a}$. After all the non-zero entries of current $\mathbf{w_i}$ have been multiplied with x_i, the PEs will then move on to perform the multiplications between x_j and $\mathbf{w_j}$, where x_j is the next non-zero entry that follows x_i in the $\mathbf{x}$. After all the x_j and $\mathbf{w_j}$ have been processed, the final calculated values of all the entries of output vector $\mathbf{a}$ are available simultaneously. Obviously, by adopting this column-wise processing scheme, the computations that are involved with zero entries of $\mathbf{x}$ can be completely skipped, thereby leading to significant saving in energy consumption when there exists non-negligible sparsity in $\mathbf{x}$.

[7]In general, each PE can be in charge of multiple permuted diagonal matrices along column direction.

B. Overall Architecture

Based on the data mapping and processing scheme, the overall architecture of PERMDNN computing engine is shown in Fig. 6. The entire design consists of an array of N_{PE} PEs that perform the kernel $M \times V$ operations and the non-linear activation operations. After all the y_i's, the entries of output activation vector $\mathbf{y}$ for the current FC layer, have been calculated and stored in the PEs, they are then written to the activation SRAM. The writing operation is performed in a group-writing manner: the entire activation SRAM is partitioned into N_{ACTMB} banks, where each SRAM bank is in charge of the y_i's from N_{ACC}/N_{PE} PEs. In each clock cycle, among all the PEs that belong to the same SRAM bank, one of them outputs W_{ACTM}/q activation values y_i's to its corresponding SRAM bank, where W_{ACTM} and q are the width of activation SRAM and the bit-width of y_i, respectively. Consider there are in total $N_{ACTMB}W_{ACTM}/q$ y_i's that are simultaneously written to the activation SRAM in one cycle. An activation routing network is designed to ensure each y_i is correctly written to the target position in the activation SRAM.

In the reading phase of activation SRAM, as described in Section IV-A, each time only one non-zero activation value x_i is fetched and broadcasted to all the PEs. To achieve that, with the help of control signals from main controller, an activation selector is designed to select the correct x_i from multiple activation SRAM banks. After the examination from a zero-detector, the non-zero x_i is then sent to an activation FIFO for its broadcast to PE arrays. The purpose of using this activation FIFO is to build up a backlog for the non-zero x_i's, thereby ensuring that the PEs can always receive their required x_i for the current computation in time.

C. Processing Element

Fig. 7 shows the inner architecture of the PE. Here, each PE is equipped with its own SRAM that stores the non-zero entries of part of block-permuted diagonal weight matrices. Similar to EIE, a weight lookup table (LUT) is located in the PE to support weight sharing strategy [15] if needed. In that case, the weight SRAM of the PE stores the virtual weight tag (or so-called clustered class) that each actual weight has. After being decoded by the weight LUT, N_{MUL} actual weights are multiplied with the input x_i and the products are sent to a group of accumulators in each clock cycle. Because each PE is equipped with N_{ACC} accumulators, which corresponds to N_{ACC} rows of weight matrix that one PE is in charge of, N_{MUL} accumulation selectors are designed to ensure those products are routed to the target accumulators. Specifically, as it will be elaborated in detail later, the permutation parameters of the corresponding permuted diagonal weight sub-matrices are used to enable very low-cost selection. For each selector, it is associated with its own accumulator bank, and each accumulator bank contains $g = N_{ACC}/N_{MUL}$ accumulators that always contain a target accumulator for the input product. After all PEs finish their processing for the current column of weight matrix, the calculated a_i's in the accumulators are sent to the activation units (ActU) to produce y_i's. Here, each

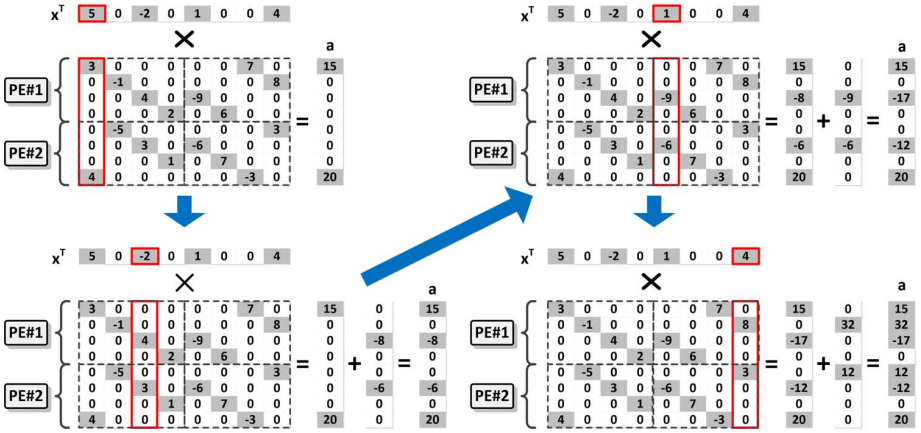


Fig. 5: Example column-wise processing procedure with input zero-skipping scheme.

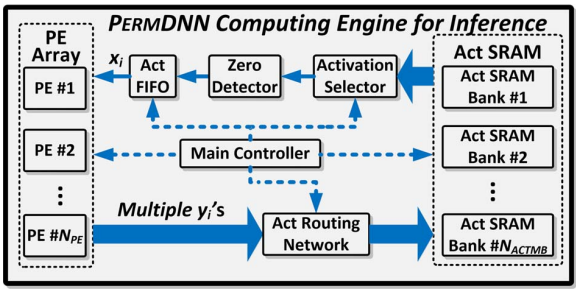


Fig. 6: Overall architecture of PERMDNN hardware.

activation unit can be reconfigured to act as either Rectified Linear Unit (ReLU) or hypertangent function ($tanh(\cdot)$) unit to support the needs of different applications.

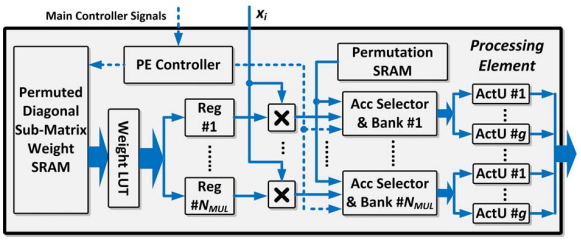


Fig. 7: Inner architecture of PE.

Weight SRAM. To reduce the read/write energy consumption for the weight SRAM, we further break the weight SRAM bank of each PE into multiple weight SRAM sub-banks. By adopting this strategy, in each cycle only one weight SRAM sub-bank is selected while the others are disabled to save energy. Accordingly, in order to achieve the full utilization of all the multipliers in most cases, the width of all the weight SRAM sub-banks is selected to ensure that each row of every SRAM sub-bank at least contains N_{MUL} weight entries.

Besides, to accommodate the column-wise processing scheme and the block-permuted diagonal structure of weight matrix, the non-zero weight is specially arranged in the weight SRAM of each PE. Fig. 8 illustrates the data allocation in one weight SRAM sub-bank for one block-permuted diagonal weight matrix. Here, each row of the weight SRAM sub-bank stores the non-zero entries of the same column of the weight matrix. By adopting this transpose-like data layout, each row access to the weight SRAM sub-bank can fetch the required data for column-wise data processing. Because each column of one permuted diagonal sub-matrix only has one non-zero entry, each PE always receives and processes the same number of non-zero weights, thereby completely eliminating the risk of load imbalance that prior work [14] may suffer from.

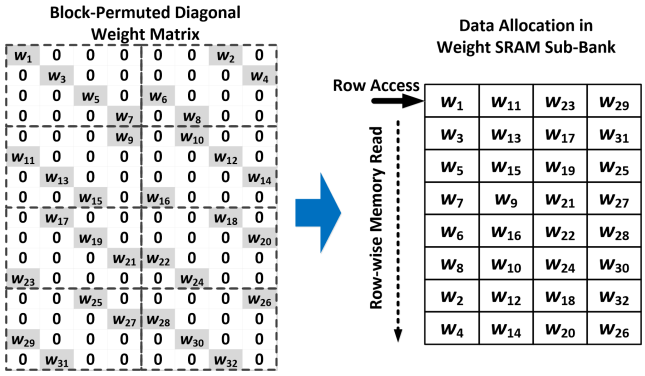


Fig. 8: Data allocation in weight SRAM

Accumulation Selector & Bank. Since the number of accumulators (N_{ACC}) is typically much larger than that of multipliers (N_{MUL}) in each PE, accumulation selectors are required to route the outputs of multipliers to the target accumulators. As shown in Fig. 9, an accumulation selector consists of two parts, an index calculator and an array of reconfigurable comparators and multiplexers. The index calculator is used to calculate the correct row index of the non-zero entry in the permuted diagonal sub-matrix that the current PE is in charge of. Specifically, due to the unique structure of permuted diagonal matrix, such calculation is essentially the modulo operation between the sum of permutation value and column index and the size of permuted diagonal matrix p; and hence it can be easily implemented using a circuit consisting of b-width

adder, subtractor and comparator, where $b = ceil(\log_2 p)$ and $ceil(\cdot)$ is the ceiling function. After the row index is calculated, among the bank of accumulators, the target accumulator is first identified by the array of comparators and multiplexers, and then it is updated using the output of the multiplier.

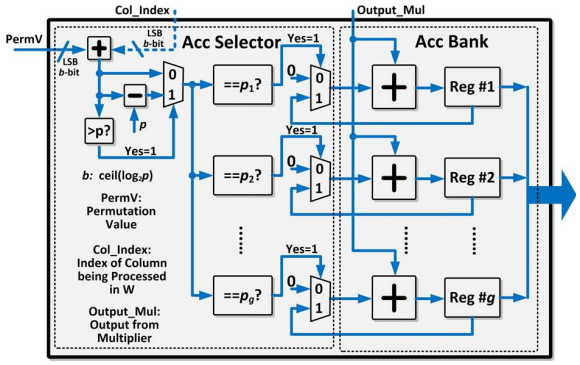


Fig. 9: Inner architecture of accumulation selector and bank.

Permutation SRAM. The permutation values (PermVs) sent to accumulation selectors are provided by permutation SRAM. Different from weight SRAM and activation SRAM, in each PE the permutation SRAM consists of a single bank instead of multiple partitions. In addition, each row of permutation SRAM contains multiple permutation values to support the computation in different accumulation selectors. Given that the bit-width of permutation value is typically small (as $\log_2 p$), the width of permutation SRAM is not very large.

D. Design for Flexibility

Because the value of p and network size vary with models, the PE of PERMDNN is designed to provide flexibility for different needs. In general, designing a flexible PE involves the considerations of several factors: N_{ACC}, the number of accumulators in each PE; N_{MUL}, the number of multipliers in each PE; $N_{ROWPE} = m/N_{PE}$, the number of rows that each PE is in charge of; and p. In the following, we describe the computation scheme of PE in three cases.

Case 1. $N_{ROWPE} \geq pN_{MUL}$ and $N_{ACC} \geq N_{ROWPE}$. In this case, since $N_{ACC} \geq N_{ROWPE}$, each PE has sufficient registers to store all the calculated y_i's which it is in charge of. Meanwhile, consider in each cycle one PE processes N_{MUL} size-p permuted diagonal sub-matrices and identifies N_{MUL} non-zero entries among pN_{MUL} rows. Therefore, only pN_{MUL} comparators of one PE are activated to identify the required row indices of non-zero entries. This means that in each accumulation selector of Fig. 9 only p out of N_{ACC}/N_{MUL} comparators and the corresponding multiplexers are used in each cycle. Therefore, if we denote $N_{ROWPE} = k(pN_{MUL})+d$, where $d \equiv N_{ROWPE} \mod k(pN_{MUL})$, then at the i-th cycle of every k cycles, only $(p(i-1)+1)$-th to (pi)-th comparators and multiplexers in Fig. 9 are activated to identify the target register that should be updated. In other words, different p copies of comparators, selectors and accumulators of Fig. 9 are activated sequentially. Notice that because $N_{ROWPE} = k(pN_{MUL}) + d$ and $N_{ACC} \geq N_{ROWPE}$, PEs always have enough hardware resource to continuously process the current column of weight matrix, and it takes k (for $d = 0$) or $(k+1)$ (for $d > 0$) cycles to fully process one column. For instance, assume we have 2 PEs with $N_{MUL} = 1$ and $N_{ACC} = 4$ to process an 8-by-8 weight matrix with $p = 2$. Then as shown in Fig. 10(a), it takes two cycles to process one column and such column-wise processing is continuous.

Case 2. $N_{ROWPE} \geq pN_{MUL}$ and $N_{ACC} < N_{ROWPE}$. In this case, since $N_{ROWPE} = k(pN_{MUL}) + d$, there must exist an integer $f \leq k$ that satisfies $f(pN_{MUL}) \leq N_{ACC} < (f+1)(pN_{MUL})$. Therefore, after f cycles of sequential activation described in Case 1, at the $(f+1)$-th cycle the inactivated accumulation selectors and accumulators in Fig. 9 are not sufficient for future processing. Accordingly, as illustrated in Fig. 10(b) with $p = 3$, the overall computation scheme needs to be changed for this case: all the other columns need to be first partially processed to calculate part of y_i's. Then the accumulators that were once allocated for the previously calculated y_i's are released and used to perform the rest unfinished computation. Such procedure may need to be repeated for many times if f is much smaller than k.

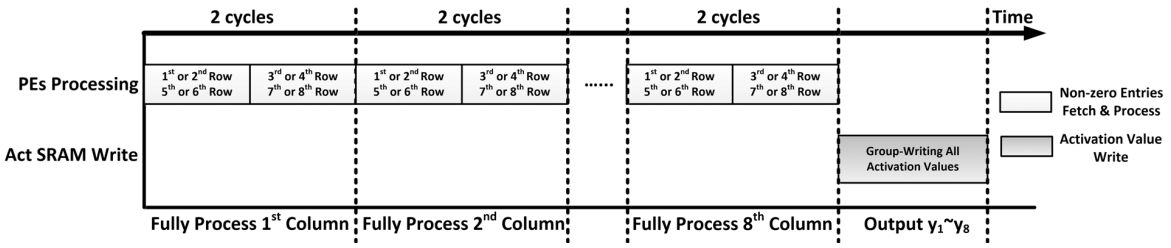
Case 3. $N_{ROWPE} < pN_{MUL}$. This is a very rare case since the number of PEs should be properly determined to ensure each multiplier is in charge of at least one permuted diagonal sub-matrix. When such case occurs, it means that p is very large (a very sparse model) and some of the PEs become redundant. To solve this problem, we need to change the original single column-processing scheme: different PEs can now process multiple columns simultaneously. Consequently, this strategy leads to two benefits: 1) increase the overall throughput; and 2) equivalently improve N_{ROWPE} for each column processing, and thereby transforming this case to the aforementioned Case 1 or 2.

V. EVALUATION

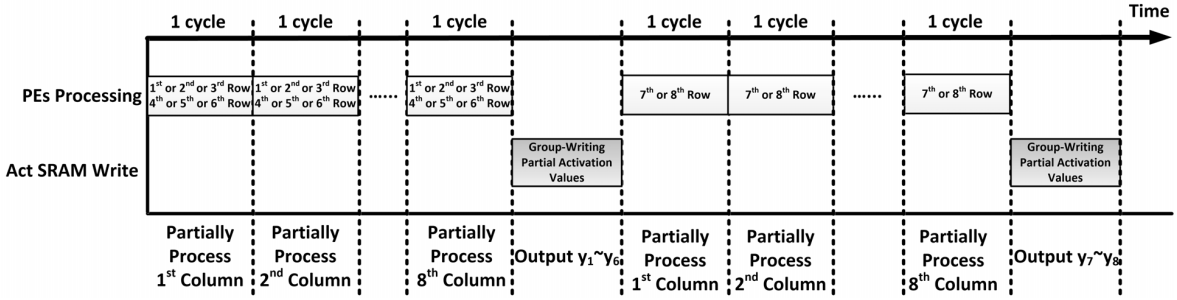
A. Experimental Methodology

Simulation and CAD Tools. We developed a cycle-accurate bit-accurate simulator to model the functional behavior of PERMDNN architecture. This simulator also serves as the golden reference for the correctness of Verilog implementation. After verifying the RTL model via validating its outputs against the outputs of the simulator, we synthesized our design using Synopsis Design Compiler with CMOS 28nm library. Here the switching activity was extracted from simulation and we annotated the toggle rate to the gate-level netlist. For place and route, we used Synopsis IC compiler to generate layout (see Fig. 11). The power consumption was estimated using Prime-Time PX. Notice that the area and power consumption of SRAM part, including weight SRAM, activation SRAM and permutation SRAM, were estimated and reported by Cacti.

Benchmarks. Our evaluation uses a set of PERMDNN models with the FC layer described in Section III-D. Specifically, similar to EIE, we evaluate the FC layers of these compressed sparse models individually, and the layers with



(a) Computation scheme with $p = 2$.



(b) Computation scheme with $p = 3$.

Fig. 10: Example computation schemes for a 2-PE PERMDNN with $N_{MUL} = 1$ and $N_{ACC} = 4$ for an 8-by-8 weight matrix.

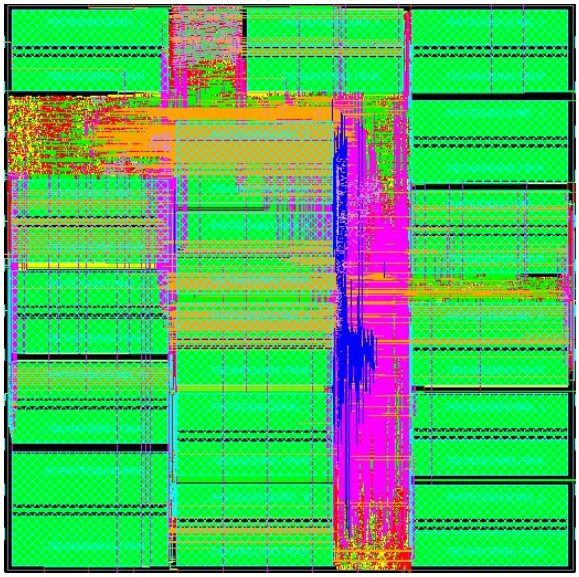


Fig. 11: Layout of one PE using CMOS 28nm technology.

different sizes are viewed as different workloads. Accordingly, the information of six benchmark layers, including the sizes, constant weight sparsity ratio and statistically calculated activation sparsity ratio[8], are listed in Table VII.

B. Design Configuration & Hardware Performance

Design Configuration. Table VIII lists the design configuration parameters for PE and the overall PERMDNN computing engine. Here one PE is equipped with eight 16-bit multipliers and 128 24-bit accumulators. In addition, each PE is also designed to contain 16×32 bit$\times 2048 = 128$KB weight SRAM and 48bit$\times 2048 = 12$KB permutation SRAM. It should be noted that the sizes of SRAM are selected by using the over-design strategy to ensure the architecture can support a wide range of applications. For instance, by using 4-bit weight sharing strategy a 32-PE PERMDNN computing engine can store a compressed layer with 8M parameters, which has the double size of the famous large-size compressed VGG FC6 layer [32]. Similarly, the activation SRAM is also designed with large capacity to facilitate the execution on very large models. As illustrated in Table VIII, the activation SRAM in this design has 8×64bit$\times 2048 = 128$KB, which corresponds to a 16-bit 64K-length vector. Such large size is usually sufficient for the FC layers in most practical models.

TABLE VII: Information of evaluated FC layers.

Layer	Size	Weight	Activation	Description
Alex-FC6	4096, 9216	10% (p=10)	35.8%	CNN model for image classification
Alex-FC7	4096, 4096	10% (p=10)	20.6%	
Alex-FC8	1000, 4096	25% (p=4)	44.4%	
NMT-1	2048,1024	12.5% (p=8)	100%	RNN model for language translation
NMT-2	2048, 1536	12.5% (p=8)	100%	
NMT-3[9]	2048, 2048	12.5% (p=8)	100%	

Hardware Performance. Table IX shows the power and area breakdowns for one PE and overall PERMDNN computing engine. It is seen that each PE occupies 0.271mm^2 and consumes 21.874mW. For the overall 32-PE computing engine, it occupies 8.85mm^2 and consumes 703.4mW, where the PE array is the most resource-consuming part (99.5% power and 97.9% area).

Regarding the throughput, being equipped with 32 PEs

[8]Lower sparsity ratio means more sparsity.

[9]The weight matrices of NMT model have three types of shapes.

TABLE VIII: Design configuration parameters.

PE Parameter		Value
Multiplier	Amount (N_{MUL})	8
	Width	16 bits
Accumulator	Amount (N_{ACC})	128
	Width	24 bits
Weigth SRAM sub-Banks	Amount	16
	Width	32 bits
	Depth	2048
Permutation SRAM	Width	48 bits
	Depth	2048
PERMDNN Computing Engine Parameter		**Value**
Amount of PEs (N_{PE})		32
Quantization scheme		16 bits
Weight sharing strategy		4 bits
Number of pipeline stages		5
Activation SRAM Bank	Amount (N_{ACTMB})	8
	Width (W_{ACTM})	64 bits
	Depth	2048
Activation FIFO	Width	32 bits
	Depth	32

running at 1.2GHz, the PERMDNN computing engine can achieve 614.4GOPS for a compressed DNN model. For the equivalent processing power on dense model, since it varies with different compression ratios, we adopt a pessimistic conversion scheme of assuming $8\times$ weight sparsity and $3\times$ activation sparsity. Consequently, the equivalent application throughput of PERMDNN achieves 14.74TOPS. Notice that our selected conversion scheme is quite conservative. For instance, EIE adopts conversion scheme of assuming $10\times$ weight sparsity and $3\times$ input sparsity, which is more optimistic than ours.

TABLE IX: Power and area breakdowns.

Component	Power (mW)	Area (mm^2)
PE Breakdown		
Memory	3.575 (16.35%)	0.178 (65.68%)
Register	4.755 (21.74%)	0.01 (3.69%)
Combinational	10.48 (47.91%)	0.015 (5.53%)
Clock Network	3.064 (14.00%)	0.0005 (0.18%)
Filler Cell		0.0678 (25.02%)
Total	21.874	0.271
	Power (mW)	**Area (mm^2)**
PERMDNN Computing Engine Breakdown		
32 PEs	700	8.67
Others	3.4	0.18
Total	703.4	8.85

C. Comparison with EIE and CIRCNN

In this subsection we compare PERMDNN with two most relevant compressed DNN-oriented architectures: EIE and CIRCNN. The reasons for selecting these two works as reference are: 1) similar to PERMDNN, EIE is a type of FC layer-targeted DNN architecture and it is the state-of-the-art design in this category; 2) similar to PERMDNN, CIRCNN also imposes structure to the weight matrix of DNN to gain improved hardware performance.

Comparison with EIE. Table X summarizes the hardware parameters and performance of EIE and PERMDNN. For fair comparison, our design adopts the weight sharing strategy and quantization scheme that are the same to EIE's configuration. Also, because EIE uses a different technology node from ours (45nm vs 28nm), we adopt the scaling strategy used in EIE to project EIE to the same 28nm technology.

TABLE X: Comparison of EIE and PERMDNN.

Design	EIE		PERMDNN
Number of PEs	64		32
CMOS Tech.	45nm (reported)	28nm (projected)[10]	28 nm
Clk. Freq. (MHZ)	800	1285	1200
Memory	SRAM		SRAM
Weight Sharing	4 bits		4 bits[11]
Quantization	16 bits		16 bits
Area (mm^2)	40.8	15.7	8.85
Power (W)	0.59	0.59	0.70

Fig. 12 compares the hardware performance of EIE and PERMDNN on executing three FC layers of AlexNet[12] in terms of speedup, area efficiency and energy efficiency. It can be seen that compared with the projected EIE using 28nm, PERMDNN achieves $3.3\times \sim 4.8\times$ higher throughout, $5.9\times \sim 8.5\times$ better area efficiency and $2.8\times \sim 4.0\times$ better energy efficiency on different benchmark layers. As analyzed in Section III-G, such improvement is mainly due to the elimination of unnecessary storage for weight index, inconvenient address calculation and load imbalance.

Comparison with CIRCNN. Table XI summarizes the hardware performance of CIRCNN and PERMDNN. Again, for fair comparison we project CIRCNN to the same 28nm technology. Notice that because CIRCNN reported synthesis results, here in Table XI the listed performance characteristics of PERMDNN are also from synthesis reports. Meanwhile, because CIRCNN only provides power and throughput information, we compare our PERMDNN and CIRCNN on overall throughput (TOPS) and energy efficiency (TOPS/W). Table XI shows that PERMDNN achieves $11.51\times$ higher throughput and $3.89\times$ better energy efficiency.

As discussed in Section III-H, compared with CIRCNN the PERMDNN enjoys the benefits of utilizing input sparsity and real-number arithmetic computation. Because the architecture of PERMDNN and CIRCNN are quite different, and the two implementations have their individual design configurations (e.g., compression ratios, size of SRAM, number of multipliers etc.), the contributions of these two advantages to the overall performance improvement are analyzed roughly as follows: 1) Because the utilization of dynamic input sparsity can bring linear reduction in computational cost, if all the other factors that affect throughput and energy efficiency, including compression ratio, clock frequency, number of multipliers and power, were the same for PERMDNN and CIRCNN designs, $3\times$ input sparsity of PERMDNN would enable about $3\times$ increase of throughput and energy efficiency; 2) For the simplicity of real number arithmetic computation, recall that a

[10]The projection follows rule in [14]: linear scaling for frequency, quadratic scaling for area and constant power scaling.

[11]Our experiments show 4-bit weight sharing does not cause accuracy drop.

[12]They are the benchmark layers that both EIE and PERMDNN evaluate.

complex multiplication consists of 4 real multiplications and 2 real additions. Also, notice that performing inference on the same-size compressed weight matrix (e.g., p-by-p), the permuted diagonal matrix-based approach needs p real multiplications, while circulant matrix-based method requires p complex multiplications in element-wise multiplication phase plus $plogp$ constant complex multiplications in FFT/IFFT phases, Therefore, when the compression ratio is the same, the use of real-number arithmetic computation can roughly bring about $4\times$ reduction in computational cost, which can translate to a significant improvement in hardware performance.

D. Scalability

As the size of weight matrix of FC layer grows, it is easy to scale up PERMDNN by adding more PEs. In general, thanks to block-permuted diagonal matrix's unique characteristic of even distribution of non-zero entries along row and column directions, PERMDNN enjoys strong scalability since the load imbalance problem [14] that challenges other sparse DNN accelerators does not exist for PERMDNN at all. Fig. 13 shows the speedup of PERMDNN architecture using different numbers of PEs. We see that our design achieves very good scalability on all benchmarks.

TABLE XI: Comparison of CIRCNN and PERMDNN. Both results are from synthesis reports.

Design	CIRCNN		PERMDNN
Number of PEs	N/A		32
CMOS Tech.	45 nm (reported)	28 nm (projected)	28 nm
Clk. Freq. (MHZ)	200	320	1200
Quantization	16 bits		16 bits
Area (mm^2)	N/A	N/A	6.64
Power (W)	0.08	0.08	0.236
Throughput (Equivalent TOPS)	0.8	1.28	**14.74 (11.51×)**
Energy Efficiency (Equivalent TOPS/W)	10.0	16.0	**62.28 (3.89×)**

VI. RELATED WORK

Besides EIE and CIRCNN, various works have been reported for exploring high-performance DNN design. Diannao family [33]–[38] and Google's TPU [25] propose a series of DNN processors and the corresponding instruction sets. In [19]–[22], [39]–[41], several sparse DNN accelerators are investigated. Considering the importance of meomory in high-performance DNN hardware, [42]–[46] extensively study better utilization of memory in DNN.

Among the research effort on data flow in DNN, [47]–[49] propose optimized data flow to provide high flexibility or reduce memory access. Also, [50]–[52] investigate high-performance DNN hardware using bit-serial inputs. Besides, various FPGA-based DNN designs have been implemented and reported, including [53]–[59].

Beyond the research works on digital accelerators for inference tasks, DNN hardware using analog or mixed-signal circuits [60]–[65] are also the potential high-efficiency solutions for deploying deep learning. Considering training is a very time-consuming process, several architecture-level approaches are proposed in [66]–[71] to improve the efficiency of training.

DNN model compression is also an active research topic in machine learning community. Various approaches, including pruning [15], low rank decomposition [17], quantization [72] [73], structured matrix [74] and clustering [75], have been proposed in prior works. [76] presents a survey of different compression techniques.

VII. CONCLUSION

This paper proposes PERMDNN, a novel approach to generate and execute hardware-friendly structured sparse DNN models using permuted diagonal matrices. Compared with prior model compression approaches, PERMDNN enjoys the benefits of regularity, flexibility and simple arithmetic computation. We then propose PERMDNN architecture, a scalable FC layer-targeted computing engine. Implementation results show PERMDNN achieves $3.3\times \sim 4.8\times$ higher throughout, $5.9\times \sim 8.5\times$ better area efficiency and $2.8\times \sim 4.0\times$ better energy efficiency than EIE. It also achieves $11.51\times$ higher throughput and $3.89\times$ better energy efficiency than CIRCNN.

VIII. ACKNOWLEDGEMENT

The authors would like to appreciate anonymous reviewers' valuable comments and suggestions. This work is funded by the National Science Foundation Awards CCF-1815699, CCF-1814759, CNS-1717984 and CCF-1750656.

REFERENCES

[1] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, 2006.
[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.
[3] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: An overview," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013.
[4] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and F.-F. Li, "Imagenet: A large-scale hierarchical image database," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 2009.
[5] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning*. ACM, 2008.
[6] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013.
[7] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, 2015.
[8] Q. V. Le, "Building high-level features using large scale unsupervised learning," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013.
[9] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. Sohl-Dickstein, "On the expressive power of deep neural networks," *arXiv preprint arXiv:1606.05336*, 2016.
[10] S. Liang and R. Srikant, "Why deep neural networks for function approximation?" *arXiv preprint arXiv:1610.04161*, 2016.
[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
[12] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.
[13] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, vol. 1, no. 2, 2017.

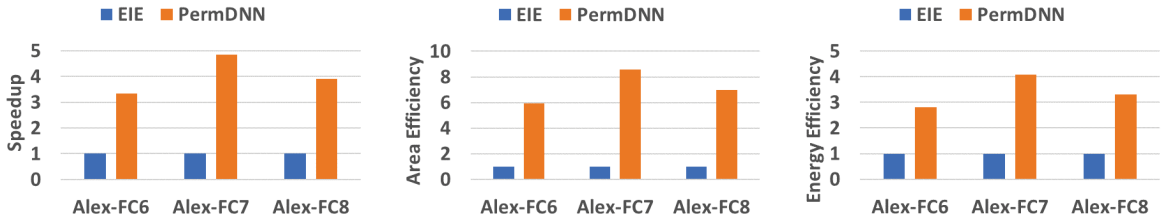


Fig. 12: Comparison between hardware performance of EIE and PERMDNN on different benchmark FC layers.

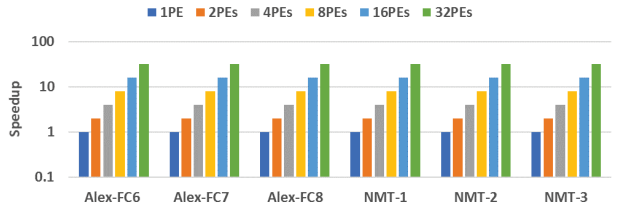


Fig. 13: Scalability of PERMDNN on different benchmarks.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016.

[15] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015.

[16] B. Liu, M. Wang, H. Foroosh, M. Tappen, and M. Pensky, "Sparse convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[17] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[18] 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*, 2016.

[19] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016.

[20] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[21] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016.

[22] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[23] V. Y. Pan, *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media, 2012.

[24] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017.

[25] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[26] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient dnns," in *Advances In Neural Information Processing Systems*, 2016.

[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, 1986.

[28] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.

[29] M.-T. Luong and C. D. Manning, "Stanford neural machine translation systems for spoken language domains," in *Proceedings of the International Workshop on Spoken Language Translation*, 2015.

[30] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002.

[31] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[32] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[33] 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 Sigplan Notices*, vol. 49, 2014.

[34] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015.

[35] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "Pudiannao: A polyvalent machine learning accelerator," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015.

[36] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014.

[37] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An instruction set architecture for neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016.

[38] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[39] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017.

[40] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016.

[41] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks."

[42] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016.

[43] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016.

[44] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017.

[45] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[46] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[47] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017.

[48] A. S. Hyoukjun Kwon and T. Krishna, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[49] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[50] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[51] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2017.

[52] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: highly-scalable deep convolutional neural network using stochastic computing," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017.

[53] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016.

[54] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[55] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016.

[56] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial cnn accelerators," in *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 2016.

[57] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable fpgas," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017.

[58] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A customizable matrix multiplication framework for the intel harpv2 xeon+ fpga platform: A deep learning case study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018.

[59] D. J. Moss, E. Nurvitadhi, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "High performance binary neural networks on the xeon+ fpga platform," in *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 2017.

[60] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," *ACM SIGARCH Computer Architecture News*, vol. 44, 2016.

[61] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016.

[62] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016.

[63] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017.

[64] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann, "An always-on 3.8 μj/86% cifar-10 mixed-signal binary cnn processor with all memory on chip in 28nm cmos," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*. IEEE, 2018.

[65] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[66] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[67] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and optimizing asynchronous low-precision stochastic gradient descent," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.

[68] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[69] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[70] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, "In-situ ai: Towards autonomous and incremental deep learning for iot systems," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 2018.

[71] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim, and H. Esmaeilzadeh, "Tabla: A unified template-based framework for accelerating statistical machine learning," in *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 2016.

[72] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016.

[73] Y. Choi, M. El-Khamy, and J. Lee, "Universal deep neural network compression," *arXiv preprint arXiv:1802.02271*, 2018.

[74] Y. Cheng, F. X. Yu, R. S. Feris, S. Kumar, A. Choudhary, and S.-F. Chang, "An exploration of parameter redundancy in deep networks with circulant projections," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015.

[75] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[76] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," *arXiv preprint arXiv:1710.09282*, 2017.