CSCNN: Algorithm-hardware Co-design for CNN Accelerators using Centrosymmetric Filters

Jiajun Li*, Ahmed Louri*, Avinash Karanth†, Razvan Bunescu‡
*Department of Electrical and Computer Engineering, George Washington University, Washington, D.C.
†School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio
‡Department of Computer Science, University of North Carolina at Charlotte, Charlotte, North Carolina
Email: {lijiajun, louri}@gwu.edu, karanth@ohio.edu, rbunescu@uncc.edu

***Abstract*—Convolutional neural networks (CNNs) are at the core of many state-of-the-art deep learning models in computer vision, speech, and text processing. Training and deploying such CNN-based architectures usually require a significant amount of computational resources. Sparsity has emerged as an effective compression approach for reducing the amount of data and computation for CNNs. However, sparsity often results in computational irregularity, which prevents accelerators from fully taking advantage of its benefits for performance and energy improvement. In this paper, we propose CSCNN, an algorithm/hardware co-design framework for CNN compression and acceleration that mitigates the effects of computational irregularity and provides better performance and energy efficiency. On the algorithmic side, CSCNN uses centrosymmetric matrices as convolutional filters. In doing so, it reduces the number of required weights by nearly 50% and enables structured computational reuse without compromising regularity and accuracy. Additionally, complementary pruning techniques are leveraged to further reduce computation by a factor of 2.8-7.2× with a marginal accuracy loss. On the hardware side, we propose a CSCNN accelerator that effectively exploits the structured computational reuse enabled by centrosymmetric filters, and further eliminates zero computations for increased performance and energy efficiency. Compared against a dense accelerator, SCNN and SparTen, the proposed accelerator performs 3.7×, 1.6× and 1.3× better, and improves the EDP (Energy Delay Product) by 8.9×, 2.8× and 2.0×, respectively.**

***Index Terms*—Convolutional Neural Networks, Dataflow Accelerators**

I. Introduction

Convolutional neural networks (CNNs) have achieved unprecedented performance on many machine learning applications, ranging from image classification [1]–[3] to text classification [4], [5] and speech recognition [6]–[8]. Performance improvements are generally correlated with increases in both depth and the number of parameters. For example, Microsoft's ResNet [3] can reach up to 152 layers and over 20 million parameters, which improves performance, but also results in excess computation and memory accesses. Numerous customized accelerators for CNNs that can deliver high computational throughput have been proposed in the literature [9]–[29]. However, given the current trend towards deeper and larger models for higher accuracy, it remains a significant challenge to efficiently process large-scale CNNs.

Sparsity has emerged as an effective approach to reduce data and computation in CNNs [30]. Researchers have proposed many effective techniques to make CNNs sparse without compromising accuracy [30]–[35]. In recent work [30], weights that are below a small threshold are pruned to zero, followed by a retraining process to preserve the original accuracy. Such weights can be removed because their contribution to the final output is negligible, thereby significantly reducing the amount of data accesses and computation. However, the data and computation reduction does not necessarily translate into performance improvements for existing accelerators. Sparsity often results in computational irregularity, which prevents accelerators from fully realizing their potential in terms of performance and energy improvement [36]. Dense accelerators [37]–[39] cannot benefit from sparsity due to lack of dedicated support for irregular and sparse models. Sparse accelerators [40]–[43] cannot fully leverage the computation reduction for performance improvement, incurring load imbalance and indexing overhead [44]. As reported in prior work [36], [40]–[43], the reduction in execution time is much lower than the reduction in computation.

To avoid the drawbacks of computational irregularity resulting from sparsity, some approaches directly represent the network with structured matrices to reduce weight storage cost [44]–[46]. For example, CirCNN [44] represents neural networks using block-circulant matrices. However, CirCNN requires complicated FFT hardware and involves operations on complex numbers that incur much higher cost than operations on real numbers. PermDNN [46] partially addresses the drawbacks of CirCNN but is not applicable for convolutional layers, limiting its performance benefits since convolutional layers dominate the computations in CNNs [20].

In this paper, we propose CSCNN, an algorithm/hardware co-design framework for CNN compression and acceleration that mitigates the effects of computational irregularity and effectively exploits computational reuse and sparsity for increased performance and energy efficiency. On the algorithmic side, CSCNN replaces convolutional filters with centrosymmetric matrices to reduce model size. As shown in Figure 1, every weight (except the center point if the dimension is odd) shares the same value with the weight at its centrosymmetric position (the dual-weight), effectively decreasing the number of weights by nearly 50%. More importantly, centrosymmetric filters can translate the weight reduction to structured computation reduction because the multiplication between an input activation

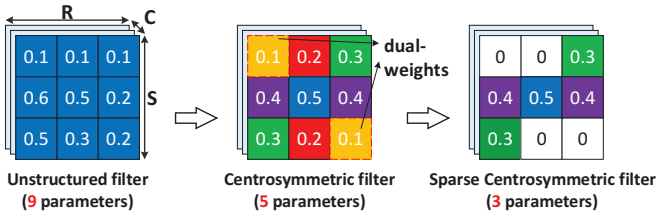


Fig. 1. Centrosymmetric matrix for convolutional filter representation.

and a weight can be regularly reused for the multiplication between the same activation and the dual-weight. As a result, CSCNN maintains locality and regularity in both computations and data accesses. Moreover, CSCNN employs a retraining process to preserve the original accuracy and can work cooperatively with pruning schemes to further reduce data and computation. Experimental evaluations show that combining centrosymmetric filters with weight pruning schemes leads to an overall reduction in computation by a factor of 2.8× to 7.2×, with only a marginal impact on accuracy. For example, the amount of computation in VGG16 is reduced by a factor of 4.3× (not considering zero activations), with less than 0.3% loss in both top-1 and top-5 accuracy.

On the hardware side, we propose a CSCNN accelerator, which exploits structured multiplication reuse and eliminates computations from both zero activations and weights to boost performance and energy efficiency. The multiplication reuse is efficiently exploited by augmenting the Processing Element (PE) in SCNN accelerator [43] with low hardware overhead. The CSCNN PE contains a multiplier array that accepts a vector of non-zero input activations and a vector of non-zero weights to perform Cartesian Product which naturally removes ineffective computations related to zero input activations and weights. Because the multiplication results also contribute to another group of output activations corresponding to the dual-weight, they are immediately reused by delivering them to an additional accumulator buffer. Given that multiply-and-accumulation (MAC) is the major arithmetic operation in CNNs and one multiplication consumes substantially more energy than one addition, the multiplication reuse significantly improves the performance and energy efficiency. The CSCNN accelerator employs a PE array organized into a 2D mesh topology to increase performance and capacity beyond a single PE. The mixed tiling strategy alleviates the impacts of both the inter-PE barrier and intra-PE fragmentation problems.

We evaluate the CSCNN accelerator on a set of representative CNN models. We design a cycle-level simulator and an RTL implementation of the CSCNN accelerator. Experimental evaluations show that the CSCNN accelerator achieves speedups of 3.7×, 1.6×, 1.3×, and energy savings by a factor of 2.4×, 1.7×, 1.5×, over a dense accelerator, SCNN, and SparTen, respectively. RTL synthesis results show that the CSCNN accelerator requires moderate area overhead (17.7%) when compared with the SCNN accelerator, demonstrating that data and computation reduction can be efficiently leveraged to speed up CNN processing.

In summary, this paper makes the following contributions:

- **Centrosymmetric Filters:** We propose a novel CNN compression method that replaces convolutional filters with centrosymmetric matrices, which significantly reduces data and computation while maintaining computational regularity and accuracy.
- **Hardware Accelerator:** We propose a novel accelerator that exploits the computational reuse enabled by centrosymmetric filters and eliminates zero computations, for increased throughput and energy efficiency.
- **Detailed Evaluation:** We thoroughly evaluate the proposed accelerator by an RTL prototype and a cycle-level simulator, demonstrating its superior performance and energy efficiency on a wide range of CNN models.

II. The CSCNN Model

This section presents the CSCNN model, including the training procedure, pruning, and the corresponding computational reuse, as well as compression results.

A. Centrosymmetric Filters

A convolutional layer applies K 3-dimensional ($R \times S \times C$) filters to 3-dimensional ($W \times H \times C$) input feature maps (IFMaps) to create output feature maps (OFMaps). We denote hereafter W/H as the width/height of IFMaps, R/S as width/height of filters, C/K as number of input/output channels, respectively. Table I lists the notation used in CNNs.

TABLE I
Notation for Convolutional Neural Networks.

Term	Meaning
$a_j^{(l)}$	j^{th} channel of l^{th} layer
$W_{ij}^{(l)}$	i^{th} channel of the j^{th} filter at l^{th} layer
$f(\cdot)$	element-wise non-linear operator
$\star$	convolution function
J	overall loss function
$\delta_j^l(u,v)$	$\frac{\partial J}{\partial z_j^{(l)}(u,v)}$, backpropagation error

The convolutional operation can be defined as follows:

$$z_j^{(l)} = \sum_i a_i^{(l-1)} \star W_{ij}^{(l-1)}, \quad a_j^{(l)} = f(z_j^{(l)}) \tag{1}$$

In CSCNN models, the filters are centrosymmetric across the $R \times S$ dimension. As shown in Figure 1, every weight shares the same value with the weight at its centrosymmetric position in each $R \times S$ filter slice. More precisely, every convolution kernel satisfies:

$$\begin{aligned} &W_{ij}^{(l)}(u,v) = W_{ij}^{(l)}(R-1-u, S-1-v) \\ &\forall\ 0 \le u \le R-1, 0 \le v \le S-1. \end{aligned} \tag{2}$$

We refer to the weights located in centrosymmetric positions as dual-weights. Because of the centrosymmetric structure, the filters can be easily compressed by about 2× as we only need

to record a single value for the dual-weights. Moreover, it does not impose indexing overhead.

Besides reducing the weight storage, centrosymmetric filters also enable a significant reduction in computations through computational reuse. Consider dual-weights $W_{ij}^{(l)}(u,v)$ and $W_{ij}^{(l)}(R-1-u,S-1-v)$ in an $R\times S$ kernel, which convolves an activation map of size $W\times H$ with unit stride. For each input activation $a_i^{(l)}(w,h)$, the computations related to the given dual-weights are as follows:

$$\begin{aligned} &z_j^{(l+1)}(w-u,h-v)+=a_i^{(l)}(w,h)\times W_{ij}^{(l)}(u,v)\\ &z_j^{(l+1)}(w+u-R+1,h+v-S+1)+=\\ &a_i^{(l)}(w,h)\times W_{ij}^{(l)}(R-1-u,S-1-v) \end{aligned} \tag{3}$$

Note that we use convolutions in full mode here because the results in other modes (valid or same) can be also obtained by cropping the results in full mode. When evaluated on hardware, the computation in Equation (3) entails reading activations and weights from memory, and performing a MAC operation on the activation-weight pair. It needs six memory reads (two input activations, two weights, two output activations), two multiplications and two additions. In conventional CNNs, the amount of memory reads can be reduced to five if data reuse is enabled, i.e. one memory read for the input activation can be saved. In CSCNN models, the amount of memory reads can be further reduced to four since the dual-weights share the same value. More importantly, the number of multiplications can be reduced to one because the two multiplications share the same input operands so that the result can be reused. Specifically, the computation in Equation (3) can be optimized as:

$$\begin{aligned} &tmp=a_i^{(l)}(w,h)\times W_{ij}^{(l)}(u,v)\\ &z_j^{(l+1)}(w-u,h-v)+=tmp\\ &z_j^{(l+1)}(w+u-R+1,h+v-S+1)+=tmp \end{aligned} \tag{4}$$

Given that MACs dominate the arithmetic operations in CNNs and one multiplication consumes significantly more energy than one addition, e.g. one 32bit-int-MULT consumes 31× more energy than one 32bit-int-ADD [47], the computational reuse can be leveraged to significantly improve the performance and energy efficiency of CNN accelerators. Meanwhile, the multiplication reduction ratio is identical to the weight reduction ratio, revealing that the sparsity created by centrosymmetric filters can be completely translated into performance and energy benefits. The details on how the proposed accelerator supports computational reuse will be presented in Section III-B. It should be noted that the computational reuse is not applicable for fully-connected layers since an individual weight in such layers is only multiplied by a single input activation. Besides, the computational reuse might not be applicable for convolutions with non-unit stride. For example, in the first layer of AlexNet (stride of 4, filter size of 11×11), an activation may not simultaneously multiply with a given pair of dual-weights because the non-unit stride skips one or both dual-weights. Therefore, centrosymmetric filters are not applied to fully-connected layers and convolutional layers with non-unit stride since it does not introduce computational benefits. Fortunately, CSCNN can be combined with prior pruning methods that work well with these layers, e.g., Deep Compression, which will be discussed in Section II-C.

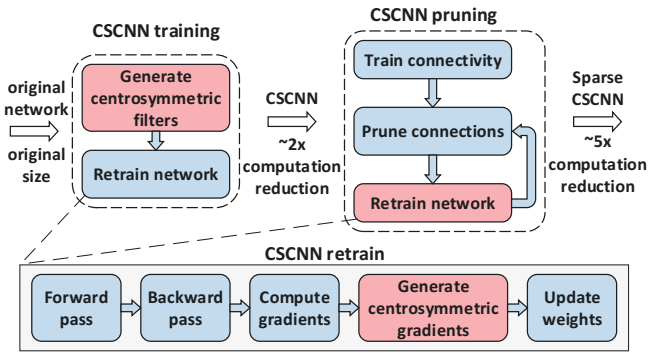


Fig. 2. CSCNN training and pruning.

B. CSCNN Training

We employ a two-step process to obtain a CSCNN model from a pre-trained conventional model, as illustrated in Figure 2, which begins by generating centrosymmetric filters based on original filters. We then employ a retraining process to retain the network accuracy.

In conventional CNNs, the original values of the dual-weights are usually not identical. There are several ways to initialize centrosymmetric filters, for example, retain the top-left weights and duplicate them to the bottom-right. We find it is better to initialize the dual-weights using the mean value of their original values before retraining. Specifically, the centrosymmetric filters are generated as follows:

$$\widetilde{W}_{ij}^{(l)}(u,v)=\widetilde{W}_{ij}^{(l)}(-u,-v)=\frac{W_{ij}^{(l)}(u,v)+W_{ij}^{(l)}(-u,-v)}{2} \tag{5}$$

where $W_{ij}^{(l)}(u,v)$ and $\widetilde{W}_{ij}^{(l)}(u,v)$ denote the weights in original and new kernels, respectively. We denote $W_{ij}^{(l)}(-u,-v)$ to represent $W_{ij}^{(l)}(R-1-u,S-1-v)$ hereafter for simplicity.

Unsurprisingly, the accuracy drops drastically after the weight initialization. For example, the accuracy of LeNet-5 [48] drops from 99.2% to 71.6%. Therefore, a retraining process is required to reattain the original accuracy.

The gradient of J with respect to weight $W_{ij}^{(l-1)}(u,v)$ is:

$$\frac{\partial J}{\partial W_{ij}^{(l-1)}(u,v)}=\sum_{u'}\sum_{v'}\frac{\partial J}{\partial z_j^{(l)}(u',v')}\cdot\frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(u,v)} \tag{6}$$

Given $\widetilde{W}_{ij}^{(l)}(u,v)=\widetilde{W}_{ij}^{(l)}(-u,-v)$ because of the centrosymmetric constraint , the gradient with respect to the combined weight is as follows:

$$\begin{aligned} &\frac{\partial J}{\partial \widetilde{W}_{ij}^{(l-1)}(u,v)}=\frac{\partial J}{\partial \widetilde{W}_{ij}^{(l-1)}(-u,-v)}\\ &=\sum_{u'}\sum_{v'}\frac{\partial J}{\partial z_j^{(l)}(u',v')}\cdot\frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(u,v)}+\sum_{u'}\sum_{v'}\frac{\partial J}{\partial z_j^{(l)}(u',v')}\cdot\frac{\partial z_j^{(l)}(u',v')}{\partial W_{ij}^{(l-1)}(-u,-v)} \end{aligned} \tag{7}$$

To implement training, we use the conventional CNN class in PyTorch where the dual-weights are still considered as separate weights, however before each gradient update during training their gradients are set to half the value derived from Equation (7). This gives the gradient a centrosymmetric structure and is theoretically equivalent to using a tied weight for the two dual-weights. This is because updating the tied weight gradient with the sum of the two gradients (as obtained using the chain rule of differentiation) is equivalent to updating it twice with the average sum in our implementation. In future work, we will implement a customized PyTorch class for CSCNN models where each pair of dual-weights are implemented as one tied weight, which will also reduce the amount of memory during training. We use a vectorized implementation (using "flip" function in PyTorch) for the centrosymmetric filters, so the training speed overhead compared with original training is negligible. We use the default training configuration (learning rate, momentum, etc.) in PyTorch for these models. We set the total number of epochs at 30, and the learning rate decays by a factor of 5 every 5 epochs.

C. CSCNN Pruning

Pruning techniques [30]–[33] are complementary to centrosymmetric filters and can be applied to further reduce data and computation. As a case study, we present the procedure of combining CSCNN with the weight pruning technique in Deep Compression [30]. As shown in Figure 2, the pruning method typically follows three steps: 1) train a normal network to learn the connectivity; 2) gradually prune the weights below a threshold to zeros; and 3) retrain the network to maintain accuracy. The first two steps are similar to conventional pruning, while the third step employs the CSCNN retraining procedure to maintain the centrosymmetric structure. Since the dual-weights will be pruned together or not given that they share the same value, the pruned network will maintain the centrosymmetric structure. The pruning method is applied to all layers including FC layers and convolutional with non-unit stride, which complements the CSCNN method as centrosymmetric filters are not applied in these layers. In our pruning experiments, we apply the same hyper-parameters and fine-tuning techniques as those in Deep compression [30].

D. Compression Results

We compare our method with prior arts on CNN compression, including unstructured pruning [30], structured pruning [49]–[53], [55], [58], and other pruning methods customized for hardware [56], [57]. Table II and Table III list the weight sparsity, multiplication reduction and accuracy of these techniques for Cifar-10 and Imagenet, respectively. For Cifar-10, we evaluate on ConvNet [59], VGG-16 [60] and WideResNet [61]. For ImageNet, we evaluate on ResNet-18/ResNet-50/ResNet/152 [3], VGG-16, AlexNet, SqueezeNet [62], ResNeXt101 [63], ShuffleNet-V2 [64], and EfficientNet-B7 [65]. The multiplication reduction listed in the tables only considers the effect of reduced weights, not taking the zero activations into account for a fair comparison.

TABLE II
COMPARISON OF ACCURACY AND THE COMPUTATION REDUCTION OF THE COMPRESSION METHODS FOR CIFAR-10.

Baseline	Models	Top-1 Accu. Baseline (%)	Top-1 Accu. (%)	Top-1 Accu. Drop (%)	Multiplication Reduction‡
ConvNet	Deep compression	75.8	75.7	0.1	3.8×
	CSCNN	75.8	75.8	0.0	1.7×
	CSCNN+Pruning	75.8	75.6	0.2	5.8×
VGG-16	Deep compression	92.8	92.8	0.0	5.3×
	CGNet	92.8	92.4	0.4	5.1×
	CSCNN	92.8	92.8	0.0	1.8×
	CSCNN+Pruning	92.8	92.5	0.3	7.2×
WideResNet	**CSCNN**	95.8	95.4	0.4	1.6×

‡ The multiplication reduction only considers the effect of reduced weights, not taking the zero activations into account to provide a fair comparison.

For ResNet18, CSCNN individually offers comparable multiplication savings with less accuracy drop compared to all the structured pruning techniques. CSCNN with pruning further achieves the highest multiplication reduction (2.8×) with less than 1% accuracy loss. CSCNN also offers considerable multiplication reduction for other CNN models with marginal accuracy losses.

Further empirical evidence for the effectiveness of CSCNN is provided by comparisons with other filter parameterization schemes. The first one is using smaller filters with the same number of parameters as centrosymmetric filters. For example, if the original filter size is 3×3, we replace it with 2×2 filters (4 effective parameters), and compare it with centrosymmetric filters in which the central entry is constrained to be zero (also 4 effective parameters). The results show that CSCNN provides better accuracy than the models using smaller filters. For example, we changed the filter size of VGG11/VGG13/VGG16 from 3x3 to 2x2, and observed an accuracy drop of over 4% for all the models. The reason is that a 2x2 filter has a smaller receptive field, which implies that it recognizes features that are constrained to be more local compared with a 3x3 centrosymmetric filter, which can recognize features over a larger size input. Another scheme is using upper/lower triangular matrices as filters, which reduces the same number of parameters as centrosymmetric filters. We found that CSCNN also shows better accuracy than this kind of filter designs.

Even though the experimental results already demonstrate that CSCNN is promising in network compression, we would also like to mention the theoretical foundation of CSCNN. In the theory of neural networks, the *universal approximation property* states that a neural network should be able to approximate any continuous or measurable function with arbitrary accuracy provided that an enough large number of parameters are available. We have proved that CSCNNs have this property. Detailed proof procedure is omitted because of space limitation.

III. CSCNN ACCELERATOR

In this section, we propose an accelerator architecture for CSCNN models. Our proposed architecture extends a

TABLE III
COMPARISON OF ACCURACY AND THE COMPUTATION REDUCTION OF THE COMPRESSION METHODS FOR IMAGENET.

Baseline	Compression Techniques	Top-1 Accu. Baseline (%)	Top-1 Accu. (%)	Top-1 Accu. Drop (%)	Top-5 Accu. Baseline (%)	Top-5 Accu. (%)	Top-5 Accu. Drop (%)	Multiplication Reduction†
ResNet-18	Deep compression [30]	69.2	69.0	0.2	88.8	88.5	0.3	2.0×
	Soft Filter Pruning [49]	70.3	67.1	3.2	89.6	87.8	1.8	1.7×
	Network Slimming [50]	69.0	67.2	1.8	88.7	87.4	1.3	1.4×
	Discrimination-aware Pruning [51]	69.6	67.3	2.3	89.0	87.6	1.4	1.9×
	Low-cost Collaborative Layers [52]	70.0	66.3	3.7	89.2	87.0	2.2	1.5×
	Feature Boosting& Suppression [53]	70.7	68.2	2.5	89.7	88.2	1.5	2.0×
	CGNet [31]	69.0	67.4	1.6	88.8	87.8	1.0	1.6×
	CSCNN	69.2	68.6	0.6	88.8	88.1	0.7	1.7×
	CSCNN+ Pruning	69.2	68.4	0.8	88.8	87.9	0.9	2.8×
VGG-16	Deep compression	68.5	68.8	-0.3	88.7	89.1	-0.4	3.0×
	Cambricon-S [54]	68.5	68.7	-0.2	88.7	-	-	2.8×
	Network Slimming [50]	63.3	63.3	0	-	-	-	1.4×
	Eigendamage [55]	68.5	65.6	2.9	88.7	85.5	3.2	2.9×
	Balanced Sparsity [56]	-	-	-	90.3	90.3	0.0	3.0×
	CSCNN	68.5	68.6	-0.1	88.7	88.7	0.0	1.8×
	CSCNN+ Pruning	68.5	68.4	0.1	88.7	88.4	0.3	4.3×
AlexNet	Deep compression	57.2	57.2	0.0	80.3	80.3	0.0	2.2×
	Cambricon-S [54]	57.2	57.3	-0.1	80.3	-	-	1.9×
	CGNet	57.2	42.9	14.3	80.3	80.0	0.3	2.6×
	CirCNN [44]	57.2	-	1∼2‡	-	-	-	-
	Viterbi-based pruning [57]	57.2	57.3	-0.1	80.3	80.2	0.1	2.2×
	CSCNN	57.2	57.2	0.0	80.3	80.1	0.2	1.5×
	CSCNN+ Pruning	57.2	57.0	0.2	80.3	79.9	0.4	2.9×
SqueezeNet	Deep compression	57.5	57.5	0.0	80.3	80.3	0.0	4.2×
	CSCNN	57.5	57.2	0.3	80.3	80.1	0.2	1.7×
	CSCNN+ Pruning	57.5	57.0	0.5	80.3	79.9	0.4	5.9×
ResNeXt-101	**CSCNN**	80.9	80.1	0.8	95.6	94.5	1.1	1.6×
ResNet-50	Deep compression	75.3	74.9	0.4	92.2	91.7	0.5	2.2×
	CSCNN	75.3	75.1	0.2	92.2	92.0	0.2	1.6×
	CSCNN+ Pruning	75.3	74.8	0.5	92.2	91.5	0.7	2.8×
ResNet-152	Deep compression	77.0	76.8	0.2	93.3	93.0	0.3	2.3×
	CSCNN	77.0	76.9	0.1	93.3	93.1	0.2	1.5×
	CSCNN+ Pruning	77.0	76.6	0.4	93.3	92.8	0.5	2.7×
ShuffleNet-V2	Deep compression	77.2	76.7	0.5	93.3	92.6	0.7	2.2×
	CSCNN	77.2	76.9	0.3	93.3	92.7	0.6	1.8×
	CSCNN+ Pruning	77.2	76.5	0.7	93.3	92.4	0.9	3.2×
EfficientNet-B7	Deep compression	84.3	84.0	0.3	97.0	96.8	0.2	3.1×
	CSCNN	84.3	84.1	0.2	97.0	96.8	0.2	1.7×
	CSCNN+ Pruning	84.3	83.8	0.5	97.0	96.6	0.4	4.3×

† The multiplication reduction only considers the effect of reduced weights, not taking the zero activations into account to provide a fair comparison.
‡ CirCNN does not provide specific accuracy values. The code of CirCNN is also unavailable.

Cartesian-product based architecture to handle the structured multiplication reuse introduced by CSCNN, and supports two-sided (both activations and weights) sparse execution and storage. We further employ a mixed spatial tiling strategy to spread the work across multiple PEs for increased performance, which alleviates the impacts of the inter-PE barrier and intra-PE fragmentation problems incurred by rigid tiling strategies. Then, we introduce the complete dataflow of the accelerator and discuss how it supports FC layers.

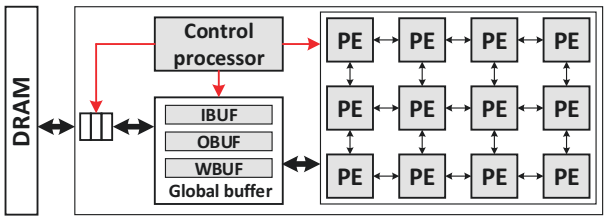


Fig. 3. The CSCNN accelerator.

A. Architecture Overview

Figure 3 shows the overall architecture of the CSCNN accelerator, which consists of the following main components: a PE array for computation, two buffers for input activations and output activations (IBUF and OBUF), a buffer for weights (WBUF), and a control processor (CP). The PE array consists of multiple PEs connected via simple interconnections. The CP controls the data and execution flow of all the modules. The accelerator is connected to off-chip DRAM that stores the input activations and weights of a network model. To accomplish the execution of a convolutional layer, the activations and weights are fetched from off-chip DRAM to on-chip buffers. Each PE receives weights and input activations from dedicated channels and performs convolutions with centrosymmetric filters, which is referred as *centrosymmetric convolution* hereafter for brevity. The result output activations are stored locally in the buffers

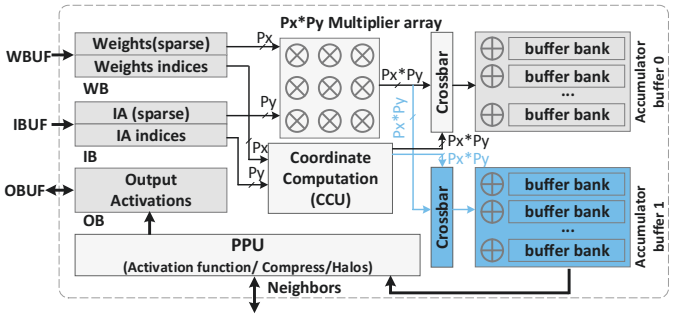


Fig. 4. CSCNN PE architecture. The blue components are additional to augment the SCNN PE for multiplication reuse.

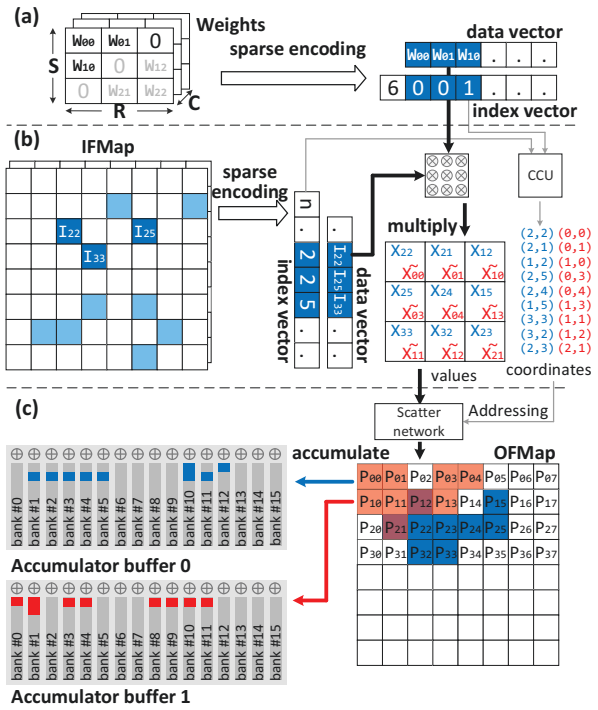


Fig. 5. CSCNN PE dataflow. (a) Sparse encoding of weights; (b) Duplication of multiplier outputs; (c) Accumulation of multiplier outputs.

of each PE for further accumulation, or transferred to neighbor PEs for halo value exchange [66]. When a layer contains multiple OFMaps and IFMaps, each PE continuously performs the computations of an OFMap, and will not move to the next OFMap until the current map has been constructed. When computations are completed, the output activations of intermediate layers are buffered on-chip if possible to minimize off-chip data transfers.

B. Processing Element (PE) Architecture

Our PE architecture is based on the SCNN PE [43] that exploits both activation and weight sparsity to improve performance and energy. Since SCNN PE cannot exploit the structured multiplication reuse, the computation reduction introduced by centrosymmetric filters cannot be translated into practical speedup. Therefore, we augment the SCNN PE to efficiently exploit the multiplication reuse in centrosymmetric convolutions.

1) Baseline PE Architecture: Figure 4 (non-blue modules) shows the CSCNN PE architecture, including a weight buffer (WB), input/output activation buffers (IB/OB), a multiplier array, an accumulator buffer (AB), a Coordinate Computation Unit (CCU) and a Post Processing Unit (PPU).

The workflow of the baseline PE is further illustrated by the example in Figure 5 (ignoring Accumulator buffer 1). Both weights and input activations are stored in compressed format that records the non-zero values and the number of zeros between adjacent non-zero values [43]. The multiplier array of size $P_x \times P_y$ accepts a vector of P_x weights (W_{00}, W_{01}, W_{20} in Fig. 5(a)) from WB and a vector of P_y input activations (I_{22}, I_{25}, I_{33}) from IB. Then the multiplier array performs a full Cartesian product of the two vectors and generate $P_x \times P_y$ multiplier outputs ($\mathcal{X}_{ij}$ in Fig. 5(b)). At the same time, the CCU computes the coordinates of the multiplier outputs. Then, the multiplier outputs are sent to the AB to update the corresponding partial sums at the matching coordinates (($\mathcal{P}_{ij}$, marked blue in OFMap)). Each multiplier output is accumulated with its corresponding partial sum with the same output coordinates. The number of banks in AB0 is $2 \times P_x \times P_y$ to reduce accumulator bank contention [43]. The non-linear activation and/or compression are performed in the PPU if necessary before writing the output activations to the OB. Since only non-zero weights and non-zero input activations are delivered to the multiplier array, the baseline PE can exploit both weight and activation sparsity.

2) CSCNN PE Architecture: The baseline PE cannot leverage computational reuse due to lack of dedicated support. We now describe how to augment it to leverage the performance and energy benefits of CSCNNs. The CSCNN PE retains the basic design and components of the baseline PE along with its dataflow (will be described in Section III-D), and only imposes an additional accumulator buffer as shown in Fig. 4.

Multiplication reuse. As described in Equation (4), the multiplications in CSCNNs can be reused to reduce the overall operations. Figure 5(b) and 5(c) presents an example of the multiplication reuse. When multiplying the weight $\mathcal{W}_{00}$ (value: 1, C-S-R index: 0-0-0) with the input activation I_{22}, the multiplier outputs contribute to the corresponding partial sums (P_{22}), which is the same in baseline PEs. Because of the centrosymmetric structure, it is the same as the multiplication of the input activation with the dual weight $\mathcal{W}_{22}$ (C-S-R index: 0-2-2), so the multiplier output also contribute to another partial sum P_{00} (marked red in Fig. 5(c)). Similarly, other multiplier outputs also contribute to another group of partial sums (marked red in Fig. 5(c)). Therefore, the CSCNN PE also sends the multiplier outputs (denoted as $\widetilde{\mathcal{X}_{ij}}$) to accumulator buffers for the additional accumulation. Notably, the CCU needs to compute the coordinates for the $\widetilde{\mathcal{X}_{ij}}$s. If the dimension of the filters is odd numbers, the dual-weight of the central weight is itself. In this case, the CCU will generate *nil* coordinates for the multiplier output generated by central weights to not

enable multiplication reuse.

If the multiplier outputs along with their duplicates are delivered to AB0 for accumulation of both partial sum groups (both *blue* and *red* elements in OFMap), it will cause buffer bank conflicts. In particular, because the number of partial sums waiting for accumulation equals the number of accumulator banks in AB0 (both are $2 \times P_x \times P_y$), the multiplier outputs and their duplicates may hash to the same accumulator bank with a high probability. One solution is to double the number of banks in AB0. However, since the multiplier outputs are routed to the accumulator banks using a crossbar switch, doubling the number of banks will significantly increase the complexity of the scatter network. Therefore, instead of using one single accumulator buffer, we employ an additional and independent accumulator buffer to relieve the bank contention. However, there is a data hazard between the two accumulator buffers because they are operating on the same partial sums. Specifically, the overlapped partial sums of the blue and red elements in OFMap should be accumulated with two multiplier outputs. For example, both $\mathcal{X}_{12}$ and $\widetilde{\mathcal{X}_{12}}$ should be accumulated to $\mathcal{P}_{12}$. If not eliminating the data hazard, both accumulator buffers will access $\mathcal{O}_{12}$ (original partial sum generated by other input channels) and perform accumulation. As a result, the original value of the partial sums is accumulated twice, causing wrong output results. To resolve data hazard, we delay the accumulation of $\mathcal{P}_{12}$ and $\widetilde{\mathcal{P}_{12}}$ to $\mathcal{O}_{12}$. Specifically, both accumulator buffers do not access the partial sums generated by other input channels. They accumulate the multiplier outputs to its local partial sums generated by input channels assigned to them. The results in AB0 and AB1 will be merged and accumulated with the partial sums generated by input channels in the PPU when they are flushed out from their accumulator buffers, respectively.

Computation order. We employ an input-stationary computation order in the multiplier array in which the input activations are held stationary as it is multiplied by all the non-zero weights in a single filter to make all of its contributions to the current OFMap. After finishing the computations in an IFMap related to the current OFMap, we hold the partial sums stationary in the accumulator buffers and move on to the next IFMap. This order minimizes the data movement of input activations inside a PE and minimized the data movement of output activations between PEs and global buffer.

C. Mixed Spatial Tiling

As described in Figure 3, the CSCNN accelerator consists of multiple PEs connected to a shared global buffer (GLB). These PEs run in parallel to increase performance and capacity. There are multiple strategies to spread the work across the PE array. For example, SCNN [43] employs planar tiling that partitions the activation plane ($W \times H$ dimension) into smaller planar tiles ($T_w \times T_h$) and distributes them to the PEs. However, the rigid planar tiling incurs two forms of inefficiency. One is intra-PE fragmentation when the layers do not have enough useful work to fully populate the multiplier array. Another is the inter-PE global barrier that leads to load imbalance among the PEs because PEs with denser workload would lag behind those with sparser workload. Specifically, using less but powerful PEs (each PE has a large multiplier array) can better alleviate the inter-PE global barrier but will exacerbate the intra-PE fragmentation problem, and vice versa. Besides, layers with small feature maps are more likely to suffer from intra-PE fragmentation problem, because the size of planar tiles would be very small in these layers, making it hard to fully populate the multiplier array.

In the CSCNN accelerator, we employ a mixed spatial tiling that combines local planar tiling and global output channel tiling to alleviate both forms of inefficiencies. The PE array is logically partitioned into smaller PE sub-arrays. For example, an 8×8 PE array can be partitioned into 4 PE sub-arrays, each containing 4×4 PEs. The output channel dimension is partitioned into K/T_k channel groups of size T_k that are distributed across the PE sub-arrays, resulting in a workload of size $T_k \times C \times W \times H \times R \times S$ for each PE sub-array to operate individually. Since all the input activations will be delivered to each PE sub-array, the load-balancing among PE sub-arrays depends solely on the density of the assigned filter groups (a $T_k \times C \times R \times S$ volume of weights) for each PE sub-array. Because filters do not change during inference, we sort offline a layer's filters by density so that the filter groups for each PE sub-array are similar in density. The T_k output channels for a PE sub-array is no longer assigned according to the channel id but the filter density. In doing so, the density of the workload $T_k \times C \times W \times H \times R \times S$ for each PE sub-array will be similar, removing the barrier among PE sub-arrays.

Furthermore, we employ planar tiling for the PEs inside a PE sub-array. The $W \times H$ activation plane is partitioned into smaller $T_w \times T_h$ planar tiles that are distributed across the PEs, resulting in an input activation volume of $T_k \times T_w \times T_h$ assigned to each PE. Because the PEs inside a PE sub-array share the same size of planar tiles and the same filter weights, they will finish their workload simultaneously, removing the barrier among PEs in a PE sub-array. Additionally, since the number of PEs in a PE sub-array is significantly reduced compared to the total number of PEs, the size of the planar tiles ($Tw \times T_h$) could be larger so that each PE has a greater opportunity to fully utilize the multiplier array. Therefore, the intra-PE fragmentation problem is also significantly alleviated. Note that the partitioning of the input activation plane introduces data halos between adjacent PEs [43]. The PE accommodates the output halos by exchanging incomplete partial sums with neighbors through the PPU. Similar to prior work [67], the tile size T_k, T_w, T_h may change layer to layer to fully populate PEs and multiplier array. The detailed tiling factor setting mechanism is omitted for brevity.

In summary, rigid tiling strategies incur inefficiency because of the variance of the layers. By combining local planar tiling and global output channel tiling and change the tile size layer to layer, our strategy is adaptive for different layers and significantly alleviates the inefficiency incurred by rigid tiling strategies.

```
    // PE array level
(A) parallel_for k3=[0:K/Tk){
      // PE sub-array level
(B)   parallel_for w2=[0:W/Tw){
        parallel_for h2=[0:H/Th){
          // PE level
          BUFFER wt_buf[C][Tk*R*S/Px][Px]
          BUFFER in_buf[C][Tw*Th/Py][Py]
          BUFFER acc0_buf[Tk][Tw+R-1][Th+S-1]
          BUFFER acc1_buf[Tk][Tw+R-1][Th+S-1]
          BUFFER out_buf[Tk][Tw+R-1][Th+S-1]
          for k1=[0:Tk){
            for c=[0:C){
              for a=[0:Tw*Th/Px){
                in[0:Px-1]=in_buf[c][a][0:Py-1]
                for w=[0:Tk*R*S/Px){
                  wt[0:Py-1]=wt_buff[c][w][0:Px-1]
(C)                 // Multiplier array level
                    parallel_for w0=[0:Px){
                      parallel_for a0=[0:Py){
                        k=Kcoord(w,Px);
                        x0=Xcoord0(a,a0,w,w0,R,S);
                        y0=Ycoord0(a,a0,w,w0,R,S);
                        x1=Xcoord1(a,a0,w,w0,R,S);
                        y1=Ycoord1(a,a0,w,w0,R,S);
(D)                     tmp=in[a0]*wt[w0];
                        acc0_buf[k][x0][y0]+=tmp;
                        acc1_buf[k][x1][y1]+=tmp;
                    } }
            } } }
(E)         out_buf[k1][0:Tw+R-1][0:Th+S-1]+=
              acc0_buf[k1][0:Tw+R-1][0:Th+S-1]
             +acc1_buf[k1][0:Tw+R-1][0:Th+S-1];
          }
    } } }
```

Fig. 6. The complete CSCNN dataflow.

D. CSCNN Dataflow

Figure 6 shows the pseudo-code for the complete CSCNN dataflow, which is summarized as follow:

1) Global output channel tiling: each PE sub-array produces T_k output channels (A).
2) Local planar tiling: each PE accommodates planar tiles ($T_w \times T_h$) of the input activation plane (B).
3) Removal of computations related to zero input activations and weights using Cartesian product based dataflow (C).
4) Enable the the multiplication reuse introduced by centrosymmetric filters (D).

The $Kcoord()$, $Xcoord0()$, $Xcoord1()$, $Ycoord0()$ and $Ycoord1()$ functions compute the k, x, and y coordinates of the uncompressed output activations using a de-linearization of the temporal loop indices a and w, the spatial loop indices P_x and P_y, and the known filter width and height [43]. The dataflow doesn't show the DRAM memory accesses assuming that all the data resides in the on-chip global buffer. When the data exceeds the on-chip storage, the input and output channel dimension can be temporarily tiled so that the PEs operate on a portion of activations at a time, like other accelerator architectures [43]. This temporal tiling may lead to frequent data transfer between on-chip and off-chip. Fortunately, researchers have extensively explored optimization techniques to reduce the off-chip memory accesses [12], [20], [68]. Since these techniques are orthogonal to our on-chip dataflow, we omit the discussion for brevity.

E. Support for Fully-connected Layers

In fully-connected (FC) layers, an individual weight is not reused across multiple input activations. Therefore, using Cartesian product based dataflow would lead to significant performance loss for these layers. Although it would make the proposed accelerator unattractive for networks that are dominated by FC layers such as BERT [69], it is not a significant limitation as these layers are memory-hungry [20], [43]. To achieve optimal efficiency for both CONV and FC layers, we believe designers should consider using both CSCNN and an architecture optimized for FC layers (such as EIE [42]).

IV. Experimental Methodology

We evaluate the CSCNN accelerator using a combination of a cycle-level simulator and RTL implementation.

Simulator. Our simulator is built based on the open-sourced TimeLoop simulator [70]. We made customization to the simulator to support CSCNN dataflow. The simulator is combined with DRAMSim2 [71] to evaluate the performance of the CSCNN accelerator. The simulator takes the weights and activations extracted from Pytorch as input and processes one layer at a time. It models the dataflow as well as the memory hierarchy and PE configurations, and collects the counts of arithmetic operations and memory accesses of different levels. The simulator estimates the compute time based on the number of arithmetic operations, while DRAMSim2 estimates the memory access latency. Then the results are combined to obtain overall execution time. Meanwhile, these statistical data are also used to build an energy model to estimate the energy consumption of the accelerator. For the energy model, energy numbers of arithmetic units and DRAM accesses are taken from [47], while SRAM energies are taken from CACTI 6.0 [72]. Additionally, the simulator can be configured to act as accelerators with other dataflows. For example, the ineffective computations with zero operands still consume computing cycles to mimic the execution flow of dense accelerators.

RTL implementation. We implement CSCNN PE in RTL and synthesize it with Synopsys Design Vision using the 45 *nm* technology FreePDK45 library, assuming an 800 MHz clock. We use 16-bit fixed-point arithmetic units as it has been proved to be effective in CNN computation. We also implement the RTL of the SCNN PE (described in Section III-B1) to evaluate the overhead of the CSCNN accelerator, and the RTL of the major components in SparTen to compare the area efficiency.

Baselines. We compare our design with a dense CNN accelerator (DCNN) and seven sparse CNN accelerators: Cnvlutin [40], Cambricon-X [41], SCNN [43], SparTen [73], CGNet [74], Cambricon-S [54], CirCNN [44]. The characteristics of the CNN accelerators are listed in Table IV. The DCNN accelerator adopts the PE architecture described in [11]. We mimic their dataflow in our simulator taking their design details as input, including partitioning, sparsity support, data reuse pattern, and load-balancing mechanisms. We profile their arithmetic operation and memory accesses and use these statistics to estimate their energy consumption. SparTen employs an offline software scheme called greedy balancing which groups filters by density to balance the workload among the PEs. Since this software technique doesn't require hardware modifications, we also apply this technique to other accelerators to provide

TABLE IV
COMPARISON OF THE CNN ACCELERATORS.

Accelerators	Compression	Sparsity	Inner spatial dataflow
DCNN	-	-	Matrix-scalar product
Cnvlutin	Deep compression	A	Vector-scalar product
Cambricon-X	Deep compression	W	Vector dot product
SCNN	Deep compression	A+W	Cartesian product
SparTen	Deep compression	A+W	Vector dot product
Cambricon-S	Coarse-grained pruning	A+W	Vector dot product
CGNet	Fine-grained channel gating	A	Vector dot product
CirCNN	Block-circulant matrices	W	FFT
CSCNN	Centrosymmetric filters	A+W	Cartesian product

a fair comparison. Since GEMM accelerators have shown promising results on accelerating deep learning workloads, we also compare the proposed accelerator against two GEMM accelerators: SIGMA [75] and SpArch [76]. They are also scaled to be equipped with the same number of multipliers. Because SIGMA/SpArch are specialized for GEMMs rather than CNNs, we remap the convolution operation into a GEMM via the Im2Col operation [77]. Note that the experimental results will not include CGNet and CirCNN because: 1) the layer-wise characteristic of CGNet is not available since it's not described in the original paper; 2) CirCNN transforms convolution to FFT computation and utilizes FFT-based multiplications whose computing block is completely different from that of other accelerators. Our simulator currently does not support cycle-level simulations for CirCNN.

Architectural configuration. The CSCNN accelerator is equipped with a 2×2 PE array, with each PE containing an 4×4 multiplier array. In each PE, the input and output buffer size is 40 KB in total. The weight buffer size is 10 KB for CSCNN PE, while 16 KB for SCNN PE. CSCNN PE uses smaller weight buffer because the centrosymmetric structure of filters can reduce the storage requirements of weights. The accumulator buffer in CSCNN PE is 12 KB, while in SCNN PE is 6 KB. The SCNN PE is equipped with a scatter crossbar of 16×32, while CSCNN PE employs two such scatter crossbars since it has an additional accumulator buffer. All the baseline accelerators are also equipped with the same number of multipliers so that we can compare the performance with almost identical computational resources. Additionally, the working frequency of the CSCNN accelerator and the baselines are kept the same at 800 MHz. Because of significant differences in buffer sizing/organization and implementation choices, our evaluated architectures may not precisely represent the prior proposals.

Benchmarks. We use the CNN models listed in Table II and Table III as the benchmarks to evaluate these accelerators, including ConvNet for Cifar10, AlexNet, VGG16, ResNet18/ResNet50/ResNet152, ShuffleNet-V2, and EfficientNet-B7 for ImageNet. We also evaluate on LeNet-5 for MNIST dataset. CSCNN accelerator runs the pruned

TABLE V
AREA ANALYSIS OF SCNN AND CSCNN PES.

	SCNN PE		CSCNN PE	
	Capacity	Area(mm^2)	Capacity	Area(mm^2)
Total	-	1.07 (100%)	-	1.26 (100%)
MulArray	16	0.05 (4.20%)	16	0.05 (3.57%)
IB+OB	40 KB	0.41 (38.66%)	40 KB	0.41 (32.86%)
WB	16 KB	0.22 (20.17%)	10 KB	0.14 (11.43%)
AB	6 KB	0.14 (12.61%)	12 KB	0.27 (21.43%)
Scatter	16×32	0.11 (10.08%)	16×32	0.22 (17.14%)
Network	crossbar		crossbar $\times 2$	
CCU	-	0.03 (2.52%)	-	0.05 (3.57%)
PPU	-	0.13 (11.76%)	-	0.13 (10.00%)

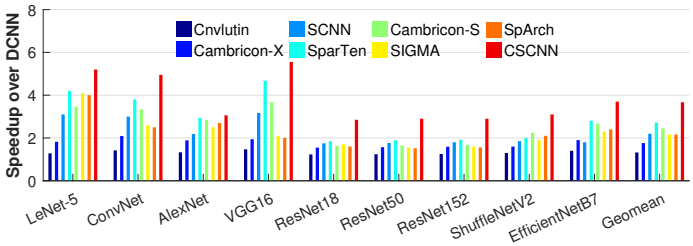


Fig. 7. Speedup over DCNN accelerator.

CSCNN models while the other accelerators except Cambricon-S run the model pruned by Deep compression. Since the CNN models for Cambricon-S is not open-sourced, we use the model characteristics described in their original papers to build the models with similar sparsity. The CSCNN models are extracted from Pytorch after applying our compression method. The CNN models for deep compression is obtained from Github [78].

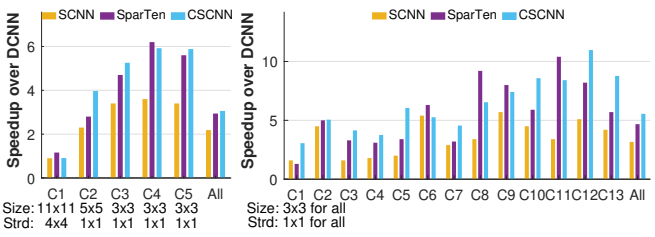
V. EXPERIMENTAL RESULTS

A. Hardware Characteristics

Table V presents the area of the major components in SCNN and CSCNN PE. Area numbers for the logic components, e.g., MulArray and PPU, are obtained from synthesis, while the area for the buffers is obtained using CACTI 6.0 [72]. Under the same computing resources, CSCNN PE increases the total area by 17.7% over SCNN, with 1.26 mm^2 vs. 1.07 mm^2. The main area overhead of CSCNN PE stems from the additional Accumulator buffer and the scatter network, which consumes an extra area of 0.13 mm^2 and 0.11 mm^2, respectively. In both PEs, the memories (IB, WB, OB, AB) contribute more than 65% of the PE area, while the multiplier array consumes no more than 5%. Although the size of AB is small, it consumes 21.43% of the CSCNN PE area because it's heavily banked for the parallel accumulation. In summary, CSCNN PE only incurs a moderate area overhead in exchange for a more efficient multiplication reuse.

B. Performance

We first compare the performance of the proposed accelerator with the dense and sparse accelerators. Figure 7 summarizes the speedups delivered by these accelerators over DCNN. Overall, the CSCNN accelerator consistently outperforms the baselines and achieves an average speedup of 3.7×, 2.8×,



(a) AlexNet (b) VGG16

Fig. 8. Layer-wise speedup over DCNN accelerator.

2.1×, 1.6×, 1.3×, 1.5×. 1.6×, 1.6× over DCNN, Cnvlutin, Cambricon-X, SCNN, SparTen, Cambricon-S, SIGMA and SpArch, respectively. In the following texts, "CSCNN" refers to the CSCNN accelerator for brevity, unless otherwise specified. The performance improvement of CSCNN varies across the neural networks. Specifically, CSCNN improves the performance by 2.9–5.5× over DCNN, 2.3–4.1× over Cnvlutin, 1.6–2.9× over Cambricon-X, 1.4–2.0× over SCNN, 1.0–1.6× over SparTen, 1.1–1.8× over Cambricon-S, 1.2–2.6× over SIGMA, and 1.1–2.7× over SpArch.

CSCNN outperforms them because 1) CSCNN supports two-sided sparsity; and 2) our compression technique removes more computations by enabling computational reuse. The performance of Cambricon-X and Cnvlutin are left behind because they only exploit one-sided sparsity, i.e., Cambricon-X for weight sparsity while Cnvlutin for activation sparsity. Although SCNN exploits two-sided sparsity, its performance benefit is hindered by its overheads, including both intra-PE fragmentation and inter-PE global barrier as described in Section III-C. Moreover, SCNN cannot support the multiplication reuse. Cambricon-S and SparTen are two state-of-the-art accelerators that address the irregularity problem in sparse CNNs. Cambricon-S employs a coarse-grained pruning technique to reduce irregularity and exploits both activation and weight sparsity to further reduce computations. SparTen employs auxiliary hardware modules for load balancing and sparse index computation, alleviating the problems incurred by SCNN. Because the remaining MACs of Deep compression is less than the pruning technique used in Cambricon-S, SparTen performs 1.17× better on average than Cambricon-S. Even so, CSCNN still outperforms SparTen due to the superior reduction on computation. Since SIGMA and SpArch are specialized for GEMMs rather than CNNs, they cannot efficiently exploit the parallelism and data locality that are available in CNNs. They have to transform convolutions to GEMMs via reordeing operations, which drastically increases the storage requirements and memory traffic thereby negatively affecting their efficiency of processing CNNs.

The performance results are better understood by looking at the layer-wise performance of SCNN, SparTen, and CSCNN on AlexNet and VGG16 in Figure 8. For C1 in AlexNet, because the input activations and weights are very dense (more than 80% density), SCNN and CSCNN are left behind because their Cartesian product based dataflow introduces unnecessary computations since the stride in C1 is larger than one. For C2, CSCNN significantly outperforms SCNN and SparTen in C2 because the density of activations and weights are moderate, whereas computational reuse contributes about 2× to the performance gains of CSCNN. For the top layers where activations and weights are both very sparse (rightmost layers), CSCNN performs comparably with SparTen but much better than SCNN. VGG16 also follows the same trend.

C. Energy Consumption

In Figure 9, we report the energy comparison of the architectures which has been normalized to the energy of DCNN. It should be noted that the energy consumption shown in Figure 9 does not include main memory accesses which usually dominates the total energy consumption. On average, CSCNN improves energy efficiency by 2.4×, 2.1×, 1.9×, 1.7×, 1.5×, 1.6×, 2.1× and 2.0× over the baseline accelerators, respectively. The improvement of energy efficiency varies widely across the models depending on the sparsity and the structure of the networks. We separate the energy into three parts: 1) *compute* includes energies from arithmetic operations; 2) *memory* includes the on-chip memory access energies; and 3) *others* contains the rest energies from control and auxiliary modules for each accelerator. All sparse accelerators significantly reduce *compute* energy since they eliminate computations related to zeros. Although they also save energy by eliminating the on-chip memory accesses of zeros, the benefit is weakened by the memory accesses of indices, which stores the locations of non-zeros. Because the dual-weights in CSCNN models don't need to be indexed, the energy consumption of CSCNN on index storage and accessing is less than other accelerators. SIGMA and SpArch consume 2.5× more energy on memory accesses than CSCNN because the transformation from convolutions to GEMMs increases the storage requirements and memory traffic.

We further show the energy breakdown by components of the SCNN and CSCNN accelerators. The energy consumption of the multiplier array in CSCNN reduces by a factor of 1.5× on average compared to SCNN. The IB+OB and WB in CSCNN consume 1.9× and 3.4× less energy respectively, which benefits from the reduction of weights. The energy benefits of AB in CSCNN is hindered by the additional accumulator buffer, achieving a reduction of 1.3× on average compared to SCNN.

D. Impact of Mixed Spatial Tiling

The PE tiling strategies affect intra-PE fragmentation and inter-PE barrier, two important factors for performance. We evaluate the impact of our mixed tiling strategy by comparing it with two rigid strategies: planar tiling only (used in SCNN) and output channel tiling. Figure 11(a) shows the performance of CSCNN using the three tiling strategies. The mixed tiling improves performance by 1.28× and 1.07× over planar tiling and output channel tiling. The output channel tiling performs as good as mixed tiling on AlexNet and VGG16, but incurs performance loss on LeNet-5 and ConvNet because the latter two don't have sufficient output channels to feed the PEs,

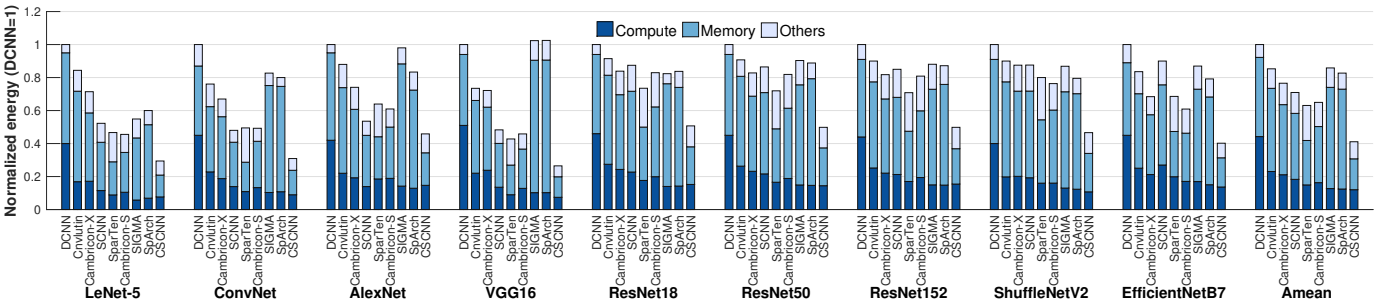


Fig. 9. Energy consumption of accelerators for various CNN architectures.

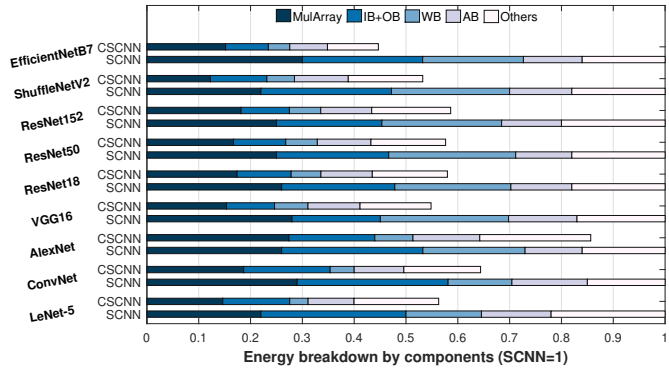


Fig. 10. Energy breakdown by components in the DCNN and CSCNN accelerators.

causing PE under-utilization. We also apply the mixed spatial tiling to SCNN and SparTen to investigate if they could benefit from such tiling optimization, as shown in Fig. 11(b) and (c). We observed that SparTen benefits only marginally from the tiling optimizations, as SparTen already employs a hardware/software balancing mechanism to balance the workload for the PEs, which attempts to solve the same problem as tiling optimizations. SCNN gains 1.2X speedup from the tiling optimizations over its original tiling strategy. CSCNN still performs 1.4x better than SCNN without tiling optimizations.

VI. Related Work

A. Pruning techniques

Weight pruning can be classified into unstructured and structured pruning. Unstructured pruning does not follow a specific geometry or constraint but prunes as more weights as possible [30]. However, unstructured pruning will inevitably cause irregular sparsity, which prevents accelerators from fully leveraging the performance and energy benefits. Structured pruning techniques [31], [32], [58], [79], [80] have been proposed to maintain the computational regularity and accelerate the decoding of sparse matrices, which can be categorized as channel-wise [58], filter-wise [32], and shape-wise pruning [79]. In filter-wise pruning, for example, all of the weights in a filter are pruned or not together. A recent work is CGNet [31], which provides structured sparsity by pruning contiguous input channels at a predetermined decision point. This kind of constraint enables accelerator to easily exploit computational savings. However, structured pruning exhibits relatively lower pruning rates compared to unstructured pruning [80].

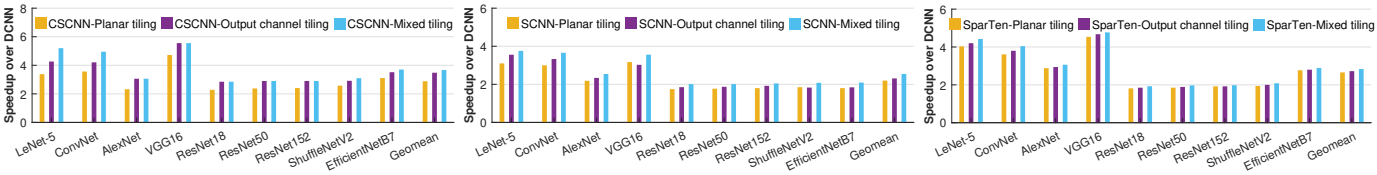
B. Structured Weight Matrices

Another approach for model compression is to represent networks with structured matrices. Cheng *et al.* [81] uses circulant matrices to represent the weights of fully-connected layers to save storage space and enable the use of FFT to speed up computation. CirCNN [44] extends this idea by using block-circulant matrices, and applies to convolutional layers for further computation reduction. Since they are based on FFT computations, they both involve high-cost arithmetic operations and require FFT hardware to reap the benefits of redundant weights. PermDNN [46] transforms sparse filters into permuted diagonal matrices but only for fully-connected layers.

C. Neural Network Accelerators

Although many neural network accelerators have been proposed to optimize computation [14]–[19], [82], memory [9], [11], [21]–[26], [83]–[88] and data reuse [12], [27], [28], [68], [89], [90], they cannot benefit from sparsity due to lack of dedicated support. To this end, sparse accelerators have been proposed to process sparsity efficiently [40]–[43], [91]–[94]. Cnvlutin [40] stores sparse activations in a compressed format and skips computation cycles for zero-valued activations to improve both performance and energy efficiency. Cambricon-X [41] exploits sparsity by compressing the pruned weights, skipping computation cycles for zero-valued weights. SCNN [43] leverages the sparsity in both weights and activations, exploiting an algorithm-based dataflow that eliminates ineffective computations from both zero-valued weights and activations simultaneously. EIE [42] performs inference on the compressed fully-connected layers and accelerates the resulting sparse matrix-vector multiplication. MASR [92] is a modular accelerator for sparse RNNs. However, irregularity caused by sparsity prevents accelerators from fully leveraging the computation and data reduction.

Regarding the irregularity problem, Mao *et al.* [95] shows that coarse-grained sparsity is more hardware-friendly and energy-efficient for sparse CNN accelerators. Scalpel [36] customizes DNN pruning for different hardware platforms based on their parallelism. Cambricon-S [54] employs coarse-grained



(a) CSCNN (b) SCNN (c) SparTen

Fig. 11. The impact of different tiling strategies.

pruning to reduce the irregularity of weights. SparTen [73] employs efficient inner-join and tackles load-imbalance by software/hardware hybrid approach. Stitch-X [96] stitches sparse weights and input activations together for parallel execution. However, due to the intrinsic irregularity, these approaches incur overhead for sparse matrix representation.

Exploiting computational reuse can also reduce CNN computation. Winograd style of convolution [97] factors out multiplies in convolution by taking advantage of the predictable filter slide. UCNN [67] exploits weight repetition to reuse CNN sub-computations and to reduce CNN model size. Riera *et al.* [98] reuses some results of the previous execution instead of computing the entire DNN to reduce computation. This line of research focuses on unstructured computational reuse, which is potentially complementary to our approach.

VII. Conclusions

This paper proposes CSCNN, an algorithm/hardware co-design framework for CNN compression and acceleration which explores the redundancy of parameters by replacing convolution filters with centrosymmetric matrices. The centrosymmetric structure substantially reduces the number of weight and computation with negligible accuracy loss while maintaining computational regularity. Additionally, pruning techniques are complementary to centrosymmetric filters and are leveraged to further reduce computation by a factor of up to 7.2× with a marginal accuracy loss. The CSCNN accelerator effectively exploits the structured computational reuse, and eliminates zero computations for increased performance and energy efficiency. Compared against a dense accelerator, SCNN and SparTen, the proposed accelerator performs 3.7×, 1.6× and 1.3× better, and improves the EDP by 8.9×, 2.8× and 2.0×, respectively.

Acknowledgment

This research was partially supported by NSF grants CCF-1702980, CCF-1812495, CCF-1901165, CCF-1703013 and CCF-1936794. We sincerely thank the anonymous reviewers for their excellent and constructive feedback.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *neural information processing systems*, pp. 1097–1105.

[2] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-First AAAI Conference on Artificial Intelligence*.

[3] 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*, pp. 770–778.

[4] A. Jacovi, O. S. Shalom, and Y. Goldberg, "Understanding convolutional neural networks for text classification," *arXiv preprint arXiv:1809.08037*, 2018.

[5] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, "A convolutional neural network for modelling sentences," *arXiv preprint arXiv:1404.2188*, 2014.

[6] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Transactions on audio, speech, and language processing*, vol. 22, no. 10, pp. 1533–1545, 2014.

[7] V. Pratap, A. Hannun, Q. Xu, J. Cai, J. Kahn, G. Synnaeve, V. Liptchinsky, and R. Collobert, "Wav2letter++: A fast open-source speech recognition system," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6460–6464, IEEE.

[8] Y. Zhang, W. Chan, and N. Jaitly, "Very deep convolutional networks for end-to-end speech recognition," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4845–4849, IEEE.

[9] 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," *architectural support for programming languages and operating systems*, vol. 49, no. 4, pp. 269–284, 2014.

[10] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 609–622.

[11] 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, pp. 92–104, ACM.

[12] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 367–379, IEEE.

[13] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, (3080246), pp. 1–12, ACM.

[15] K. Kwon, A. Amid, A. Gholami, B. Wu, K. Asanovic, and K. Keutzer, "Co-design of deep neural nets and neural net accelerators for embedded

vision applications," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE.

[16] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Scaling binarized neural networks on reconfigurable logic," in *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pp. 25–30, ACM.

[17] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 535–547, IEEE.

[18] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," 2019.

[19] A. Li, T. Geng, T. Wang, M. Herbordt, S. L. Song, and K. Barker, "BSTC: A novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets," 2019.

[20] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *field programmable gate arrays*, pp. 161–170.

[21] S. Koppula, L. Orosa, A. G. Yağlıkçı, R. Azizi, T. Shahroodi, K. Kanellopoulos, and O. Mutlu, "EDEN: Enabling energy-efficient, high-performance deep neural network inference using approximate DRAM," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 166–181, ACM.

[22] L. Pentecost, M. Donato, B. Reagen, U. Gupta, S. Ma, G.-Y. Wei, and D. Brooks, "MaxNVM: Maximizing DNN storage density and inference efficiency with sparse encoding and error mitigation," 2019.

[23] Q. Deng, L. Jiang, Y. Zhang, M. Zhang, and J. Yang, "DrAcc: a DRAM based accelerator for accurate CNN inference," 2018.

[24] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "LODESTAR: Creating locally-dense CNNs for efficient inference on systolic arrays," 2019.

[25] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," *arXiv preprint arXiv:1909.04548*, 2019.

[26] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit prudent in-cache acceleration of deep convolutional neural networks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 81–93, IEEE.

[27] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE.

[28] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 94–105.

[29] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (3173176), pp. 461–475, ACM.

[30] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *international conference on learning representations*.

[31] W. Hua, Y. Zhou, C. M. De Sa, Z. Zhang, and G. E. Suh, "Channel gating neural networks," in *Advances in Neural Information Processing Systems*, pp. 1884–1894.

[32] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.

[33] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," *arXiv preprint arXiv:1611.06440*, 2016.

[34] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images," *Nature*, vol. 381, no. 6583, pp. 607–609, 1996.

[35] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, "Efficient learning of sparse representations with an energy-based model," in *Proceedings of the 19th International Conference on Neural Information Processing Systems*, (2976599), pp. 1137–1144, MIT Press.

[36] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.

[37] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, and P. Raina, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 14–27, ACM.

[38] R. Venkatesan, Y. S. Shao, B. Zimmer, J. Clemons, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. S. Emer, C. T. Gray, S. W. Keckler, and B. Khailany, "A 0.11 PJ/OP, 0.32-128 Tops, scalable multi-chip-module-based deep neural network accelerator designed with a high-productivityVLSI methodology," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–24.

[39] J. Zhang and J. Li, "Unleashing the power of soft logic for convolutional neural network acceleration via product quantization," 2019.

[40] 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, pp. 1–13, IEEE Press.

[41] 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*, pp. 1–12, IEEE.

[42] 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 *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 243–254, IEEE Press.

[43] 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*, pp. 27–40, ACM.

[44] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," 2017.

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

[46] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 189–202.

[47] M. Horowitz, "Energy table for 45nm process," 2012.

[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[49] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," *arXiv preprint arXiv:1808.06866*, 2018.

[50] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2736–2744.

[51] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 875–886.

[52] X. Dong, J. Huang, Y. Yang, and S. Yan, "More is less: A more complicated network with less inference complexity," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5840–5848.

[53] X. Gao, Y. Zhao, Dudziak, R. Mullins, and C.-z. Xu, "Dynamic channel pruning: Feature boosting and suppression," *arXiv preprint arXiv:1810.05331*, 2018.

[54] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 15–28, IEEE.

[55] C. Wang, R. Grosse, S. Fidler, and G. Zhang, "Eigendamage: Structured pruning in the kronecker-factored eigenbasis," *arXiv preprint arXiv:1905.05934*, 2019.

[56] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient dnn inference on gpu," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 5676–5683.

[57] D. Lee, D. Ahn, T. Kim, P. I. Chuang, and J.-J. Kim, "Viterbi-based pruning for sparse matrix with fixed and high index compression ratio," in *International Conference on Learning Representations*.

[58] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1389–1397.

[59] A. Krizhevsky, "CUDA-Convnet: High-performance C++/CUDA implementation of convolutional neural networks," 2012.

[60] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *international conference on learning representations*.

[61] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[62] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[63] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500.

[64] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, pp. 116–131.

[65] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[66] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 751–764, ACM.

[67] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "UCNN: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 674–687.

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

[69] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[70] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315, IEEE.

[71] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE computer architecture letters*, vol. 10, no. 1, pp. 16–19, 2011.

[72] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," *HP Laboratories*, 2009.

[73] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," 2019.

[74] W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, "Boosting the performance of CNN accelerators with dynamic fine-grained channel gating," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 139–150, ACM.

[75] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 58–70, IEEE.

[76] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 261–274, IEEE.

[77] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe con troll: Shallow ideas to speed up deep learning," in *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, pp. 1–4.

[78] S. Han, "Deep-compression-alexnet," 2015.

[79] 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*, pp. 2074–2082.

[80] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[81] 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*, pp. 2857–2865.

[82] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 940–953.

[83] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "MnnFast: a fast and scalable system architecture for memory-augmented neural networks," 2019.

[84] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 541–552.

[85] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," 2019.

[86] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-memory for energy-efficient neural network training: A heterogeneous approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 655–668, IEEE.

[87] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 52–65, IEEE.

[88] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, and K. Roy, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 715–731, ACM.

[89] M. J. Marinella, S. Agarwal, A. Hsia, I. Richter, R. Jacobs-Gedrim, J. Niroula, S. J. Plimpton, E. Ipek, and C. D. James, "Multiscale co-design analysis of energy, latency, area, and accuracy of a reram analog neural training accelerator," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 1, pp. 86–101, 2018.

[90] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 662–673, IEEE.

[91] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern gpus," 2019.

[92] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G. Wei, and D. Brooks, "MASR: A modular accelerator for sparse RNNs," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 1–14.

[93] 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 *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 78–91, IEEE.

[94] J. Li, S. Jiang, S. Gong, J. Wu, J. Yan, G. Yan, and X. Li, "Squeezeflow: A sparse cnn accelerator exploiting concise convolution rules," *IEEE Transactions on Computers*, vol. 68, no. 11, pp. 1663–1677, 2019.

[95] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the granularity of sparsity in convolutional neural networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 1927–1934.

[96] 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," in *SysML Conference*.

[97] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021.

[98] M. Riera, J. Arnau, and A. Gonzalez, "Computation reuse in DNNs by exploiting input similarity," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 57–68.