AdaPrune: An Accelerator-Aware Pruning Technique for Sustainable CNN Accelerators

Jiajun Li and Ahmed Louri, *Fellow, IEEE*

Abstract—Convolutional neural network (CNN) accelerators have achieved great success from cloud to edge scenarios. However, given the trend towards even larger and deeper neural network models, it remains a challenging problem to efficiently process these CNNs especially on edge devices with limited energy budget. Accordingly, reducing the energy consumption is of paramount importance for sustainable CNN accelerators. In this paper, we propose AdaPrune, a novel pruning technique that reduces model size and computation to achieve performance improvement and energy savings for CNN accelerators. Unlike previous pruning techniques that sacrifice either computational regularity or accuracy, AdaPrune maintains both by customizing CNN pruning for the underlying accelerators to maximally leverage the sparsity benefits. AdaPrune consists of two techniques: input channel group pruning and output channel group pruning. By analyzing the weight fetching patterns of sparse CNN accelerators, AdaPrune adaptively switches between the two techniques to guarantee that the zeros are evenly distributed in each fetching group. In doing so, the pruned network structure preserves customized computational regularity for the underlying accelerators, thereby boosting the performance and energy efficiency. We evaluate AdaPrune on three sparse CNN accelerators with different spatial tiling strategies. The experimental results show that AdaPrune achieves up to 1.6× performance speedup, and 1.5× energy savings compared to unstructured pruning.

Index Terms—Convolutional neural networks, model compression, weight pruning

1 Introduction

Convolutional neural networks (CNNs) have achieved remarkable success in a wide range of applications, from image recognition [1], [2], [3] to speech processing [4], [5], [6]. The superior accuracy of CNNs is generally correlated with increases in both depth and computational complexity, which has inspired many customized accelerators [7], [8], [9], [10] for both cloud and edge scenarios. However, given the trend towards even deeper and larger models to further yield higher accuracy, it remains a challenging problem to efficiently process these CNNs on existing accelerators especially for edge scenarios. Therefore, it is of paramount importance to reduce the energy consumption of processing CNNs for sustainable CNN accelerators.

Weight pruning is an efficient approach to reduce model size and computation, which provides the opportunity to improve performance and energy efficiency for CNN accelerators. Several pruning techniques have shown promising results in CNN compression [11], [12], [13], [14]. In recent work [11], 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 they do not contribute to the final results, thereby significantly reducing the number of data accesses and computation.

Although these pruning techniques deliver a high compression ratio, we discovered that they face a tough choice between computational regularity and accuracy, i.e., they have to compromise either computational regularity or accuracy to acquire sparsity. Specifically, if the pruning has no geometric constraint (unstructured pruning [12]), it inevitably leads to irregular sparse weight matrices and computation patterns, thereby requiring extra storage and computation to encode and decode the sparse format. Moreover, the irregular computation patterns do not lend themselves to efficient processing on current accelerators which are customized for fine-grained thread or data parallelism. Although some sparse accelerators have been proposed to support sparsity [15], [16], [17], the performance gain is much lower than the reduction in computation. For example, SCNN reaches merely 2.6× performance speedup on AlexNet given 4.5× multiple-accumulate (MAC) reduction introduced by sparsity.

To this end, structured pruning is proposed to preserve computational regularity by placing non-zero weights at predefined locations [12]. This structured sparsity is very beneficial for parallel computation and can be efficiently processed by conventional accelerators. However, it often induces accuracy loss compared to unstructured pruning [14].

To address this challenge, we propose in this paper a new pruning approach, called *AdaPrune*, to reach an optimal balance between computational regularity and accuracy. AdaPrune customizes CNN pruning for sparse CNN accelerators to match the pruned networks to the accelerator dataflow, thereby maximizing the performance and energy efficiency of the accelerators. The pruned network preserves accuracy with the original network and also maintains customized computational regularity for the underlying accelerators. AdaPrune consists of two techniques: input channel group pruning and output channel group pruning. By analyzing the

• *The authors are with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052 USA. E-mail: {lijiajun, louri}@gwu.edu.*

Manuscript received 11 Nov. 2020; revised 28 Dec. 2020; accepted 29 Dec. 2020. Date of publication 19 Feb. 2021; date of current version 7 Mar. 2022. (Corresponding author: Jiajun Li.)
Recommended for acceptance by S. Guo.
Digital Object Identifier no. 10.1109/TSUSC.2021.3060690

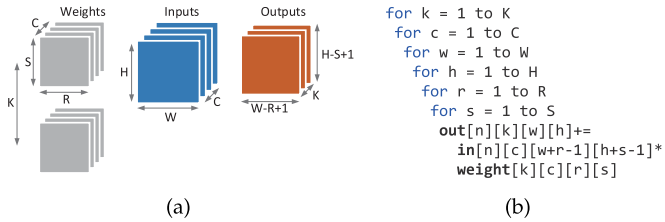


Fig. 1. Convolutional layer computation and parameters. (a) Layer parameters. (b) 6-D loop nest.

weight fetching patterns of sparse CNN accelerators, AdaPrune adaptively switches between the two techniques to guarantee that the zeros are evenly distributed in each fetching group. In doing so, the pruned network structure preserves customized computational regularity for the underlying accelerators, thereby significantly improving the performance and energy efficiency of the accelerators. Specifically, this paper makes the following contributions:

- We present a detailed analysis of the performance of CNN accelerators when running dense and sparse CNNs, and show that the workload imbalance among the Processing Elements (PEs) is the major cause of performance degradation.
- We propose AdaPrune that customizes pruned networks for the accelerator at hand. The pruned network preserves accuracy with the original network and also maintains customized computational regularity for the underlying accelerators, thereby eliminating performance loss of sparse accelerators and improving the energy efficiency.
- We compare the performance of AdaPrune with unstructured CNN pruning techniques using sparse CNN accelerators with different spatial tiling strategies. We replaced the unstructured pruning technique with the proposed AdaPrune, and found out that AdaPrune achieves up to 1.6× performance speedup and 1.5× energy savings compared to unstructured pruning.

The rest of this paper is organized as follows. Section 2 provides a brief background on CNN weight pruning and accelerators, and presents a detailed analysis of the performance of these accelerators when running the pruned networks. Section 3 presents our new pruning strategy. Section 4 describes the accelerator platforms to evaluate AdaPrune. Section 5 describes the experimental methodology. Section 6 details the evaluation of AdaPrune and the comparative studies with the state-of-the-art pruning techniques on three representative sparse CNN accelerators. Section 7 introduces related work and Section 8 concludes the paper.

2 Background and Motivation

2.1 CNN Basics

CNNs are constructed by stacking multiple computation layers where each layer contains heavily nested loops. The parallel loop structures and the regular computation patterns in CNNs make it easy to design dedicated CNN accelerators. The computation space for the convolutional layer (CVL) consists of a loop nest over six dimensions, as shown in Fig. 1. The core operation is a 2-dimensional sliding-window convolution convolves of an element filter ($R \times S$) by an element input activation plane ($W \times H$) and generates the corresponding element output activation plane ($(W - R + 1) \times (H - S + 1)$). There can be multiple input and output activations planes in a CVL. We refer hereafter the input and output activation planes as input/output channels. The parameters for a CVL is listed in Table 1.

TABLE 1
Hyper-Parameters in a Convolutional Layer

Parameters	Description
R/S	Height/width of the filter
W/H	Height/width of the activation plane
C/K	Number of input/output channels

2.2 Dense CNN Accelerators

Because multiply-add operations are associative, all permutations of the six loop variables are legal [18]. Fig. 1b demonstrates one permutation of the loop nest. Each MAC operation is formed by a single point in the 6-dimensional computation space. The loops can be arbitrarily ordered, partitioned, and parallelized to generate a CNN's dataflow [9], which is perhaps the biggest difference between many previous works on the CNN accelerators.

Fig. 2a depicts the processing of dense CNNs on an accelerator consisting of four processing elements (PEs). Note that in order to make it easy to understand, we use a 2-D computation space to describe the CNN computations, which is actually up to the aforementioned six dimensions. The CNN computation space can be easily partitioned using loop tiling techniques [18], as shown in Fig. 2a. Each computational block can then be allocated to the PEs and accelerated in parallel, hence delivering high computational throughput. However, such dense accelerators can benefit little from sparsity because they cannot efficiently handle sparse models. As shown in Fig. 2b, the zero weights are still delivered to the accelerators and perform computations which are actually unnecessary. Therefore, all the PEs are busy in the entire time slots. Some accelerators such as Eyeriss [9] exploit power gating techniques to reduce the energy of processing zeros, however, it cannot improve the performance. Hence, dense accelerators cannot utilize the sparsity for increased performance.

2.3 Sparse CNN Accelerators

To this end, it is urgent to process sparsity efficiently introduced by pruning. To exploit sparsity in CNNs, several recent CNN accelerator architectures have been proposed [15], [16], [19]. Cnvlutin [19] stores the sparse activations in a compressed format and eliminates the computations related to zero activations. Cambricon-X [15] skips computations related to zero weights by storing the sparse weights in compressed format. SCNN [16] eliminates the unnecessary computations related to both zero activations and zero weights simultaneously by leveraging *Cartesian Product* as its key operation. These methods can reap the benefits of sparsity by enabling the PEs to skip zero computations, thereby achieving improvements on both performance and energy efficiency.

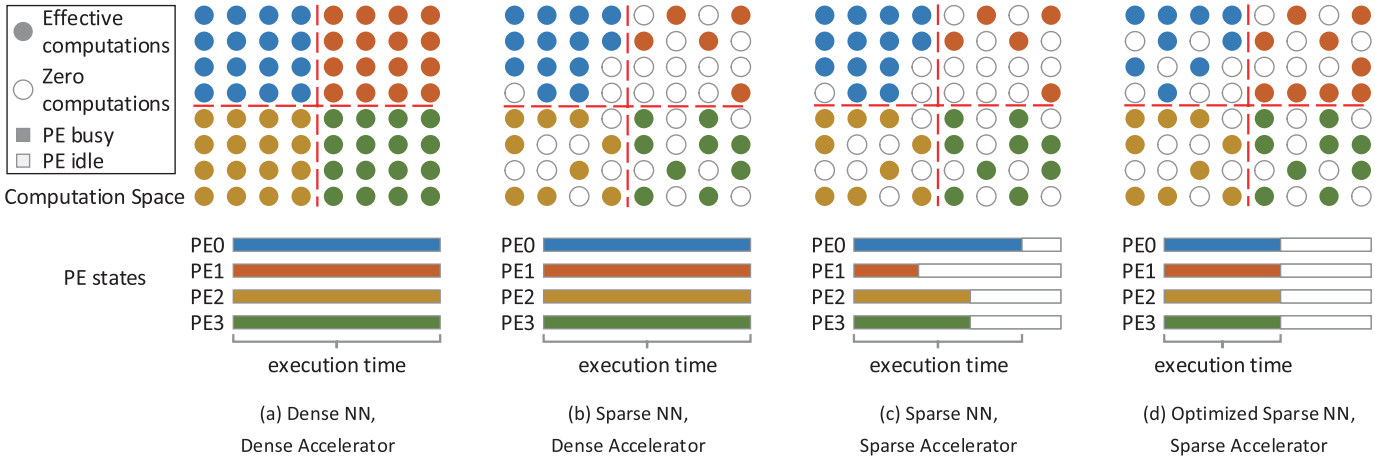


Fig. 2. Evolution of neural network accelerators.

However, we observed that they incur performance degradation due to the irregular and unbalanced sparsity distribution of the pruned networks.

These accelerators can be modeled by the example shown in Fig. 2c. Compared to Fig. 2b, the PEs are powered off when they are idle to save energy. However, since the workload for each PE is unbalanced, it causes under-utilization of computing resources and eventual performance degradation. Specifically, the four PEs carry out 13, 4, 9, 9 multiplications, respectively. However, since the results of these PEs have to be synchronized to proceed to the computation of the next stage, PE1, PE2, and PE3 are stalled until PE0 finishes its workload. This load imbalance among the PEs inevitably results in PE under-utilization and performance degradation. As confirmed by experimental results shown in Fig. 3, the attainable performance of these sparse CNN accelerators is significantly dwarfed compared to their nominal performance. For example, SCNN merely reaches a performance speedup of 2.2×, which is far below the computation reduction of 4.2×.

3 AdaPrune

3.1 Overview

To address the load-imbalance problem described above, we propose a new pruning technique called AdaPrune, which customizes CNN pruning for different accelerators to force workload balance among the PEs. Unlike unstructured pruning that generated randomly distributed zeros, AdaPrune guarantees that the number of zero weights are evenly distributed among the computational workload for each PE. As illustrated in Fig. 2d, the computational workloads allocated to the PEs share equal or comparable sparsity so that the PEs finish their workload almost simultaneously, thereby resolving the workload imbalance problem and consequently improving the performance. Although the key idea is straightforward, it faces three challenges: 1) how to preserve the accuracy of AdaPrune; 2) how to match the pruned network with accelerators since different accelerators may adopt different dataflow; and 3) how to evenly partition a computation space since it has up to six dimensions. We will address these challenges in this section.

The overview of AdaPrune is shown in Fig. 4. First, AdaPrune determines the spatial tiling strategy of the accelerators. We divide the strategies into three categories: input channel tiling, output channel tiling, and planar tiling. The tiling strategy determines the activation and weight fetching pattern for the PEs. For each tiling strategy, AdaPrune customizes the pruned network and forces the remaining non-zero weights to be evenly distributed among each fetching group. In doing so, the load imbalance problem in terms of zero weights is addressed.

3.2 Spatial Tiling Strategies

CNN accelerators usually use an array of PEs to fully leverage the data and computation parallelism in CNN computation. The spatial tiling strategy is employed to allocate the computational workload across the PEs. Specifically, it partitions the workload into different iterations and schedules the sub-workload to the PEs [16]. For example, parallelizing in the K dimension in Fig. 1b partitions the CVL by the K dimension and convert the *for* loop into multiple loops that

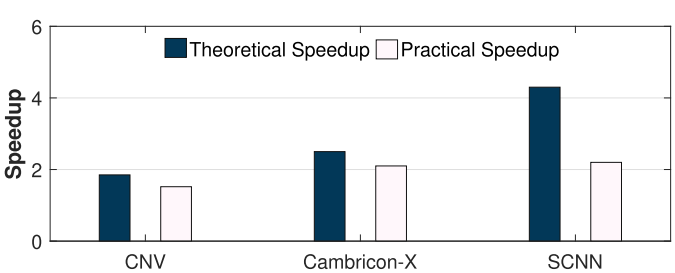


Fig. 3. Performance degradation of sparse accelerators (the speedup is normalized to a comparable provisioned dense accelerator).

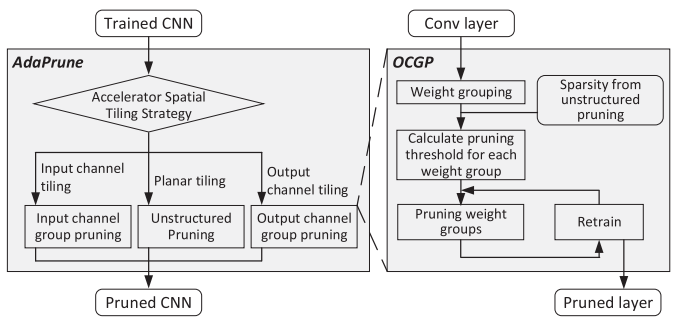


Fig. 4. Overview of AdaPrune and main steps of output channel group pruning (OCGP).

```
// PE level
parallel_for k1= [0:K/Tk)

// Computations inside a PE
for k0 = [0:Tk)
 for c = [0:C)
  for w = [0:W)
   for h = [0:H)
    for r = [0:R)
     for s = [0:S)
      out[n][k1*Tk+k0][w][h]+=
        in[n][c][w+r-1][h+s-1]*
        weight[k1*Tk+k0][c][r][s]
```

(a)

```
// PE level
parallel_for c1= [0:C/Tc)

// Computations inside a PE
for k = [0:K)
 for c0 = [0:Tc)
  for w = [0:W)
   for h = [0:H)
    for r = [0:R)
     for s = [0:S)
      out[n][k][w][h]+=
        in[n][c1*Tc+c0][w+r-1][h+s-1]*
        weight[k][c1*Tc+c0][r][s]
```

(b)

```
// PE level
parallel_for w1= [0:W/Tw)
 parallel_for h1= [0:H/Th)

  // Computations inside a PE
  for k = [0:K)
   for c = [0:C)
    for w0 = [0:Tw)
     for h0 = [0:Th)
      for r = [0:R)
       for s = [0:S)
        out[n][k][w1*Tw+w0][h1*Th+h0]+=
          in[n][c][w1*Tw+w0+r-1][w1*Tw+w0+s-1]*
          weight[k][c][r][s]
```

(c)

Fig. 5. Spatial tiling strategies. (a) input channel tiling; (b) output channel tiling; (c) planar tiling.

can be processed in parallel by the PEs. Similarly, other dimensions can also be tiled for parallel computation. Note that for each PE, the workload can be further spatially and temporally tiled to fit the storage size and MAC array. We primarily focus on the spatial tiling for different PEs, as it directly impacts the load balance for these PEs.

We divide CNN accelerators into three main categories based on their spatial tiling strategies. Fig. 5 shows the pseudo-code of the three categories: input channel tiling, output channel tiling, and planar tiling. In *input channel tiling*, the input channels (C) are partitioned into smaller T_c channel tiles that are distributed across the PEs. Each channel tile is extended fully into the element activation plane $W \times H$, generating an input-activation volume of $T_c \times W \times H$ allocated to each PE. Meanwhile, the weights are also partitioned accordingly and assigned to each PE, resulting in a weight volume of $T_c \times K \times R \times S$ for each PE. Afterward, each PE operates on its computation space formed by the input and output activations. The computation space for each PE is $K \times T_c \times W \times H \times R \times S$. Similarly, in *output channel tiling*, the output channels (K) are partitioned into smaller T_k channel tiles for the PEs. Each channel tile also extends fully across the element activation plane $W \times H$, generating an output-activation volume of $T_k \times W \times H$ allocated to each PE. The weights are also partitioned accordingly and assigned to each PE, resulting in a weight volume of $T_k \times C \times R \times S$ to each PE. Afterward, each PE operates on its computation space formed by the input and output activations. In *planar tiling*, we partition the $W \times H$ element activation plane into smaller element planar tiles of $T_w \times T_h$. Each planar tile is extended fully into the input- and output-channel dimension, generating an input-activation volume of $C \times T_w \times T_h$ and an output-activation volume of $K \times T_w \times T_h$ allocated to each PE. The weights are not partitioned but broadcast to the PEs, resulting in a weight volume of $K \times C \times R \times S$ to each PE. Table 2 summarizes the three categories of tiling strategies and the representative accelerators.

TABLE 2
Tiling Strategies and Representative Accelerators

Tiling strategies	Tiled dimension	Accelerators
Input channel tiling	C	Cnvlutin [19]
Output channel tiling	K	Cambricon-X [15]
Planar tiling	W, H	SCNN [16], SqueezeFlow [20]

3.3 Proposed Pruning Technique

No matter what spatial tiling strategy is used, the sparsity of the computational subset allocated to each PE varies widely because previous unstructured pruning results in randomly distributed zeros. To address this problem, we propose an accelerator-aware pruning technique, called AdaPrune, which customizes the pruned networks with a more balanced distribution of the zeros. It reaps the benefits of a high pruning rate of unstructured pruning while being free from the load-imbalance problem when performed on accelerators.

In AdaPrune, the weights are pruned within each subset allocated to the PEs to guarantee that all the subsets have the same number of non-zero weights. Fig. 6 presents an toy example of AdaPrune, where the convolutional layer characteristic is: $K = 3, C = 3, W = 8, H = 8, R = 3, S = 3$. Fig. 6a uses output channel tiling and uses three PEs ($T_k = 1$) for parallel computation. Previous unstructured pruning schemes cannot guarantee a balanced zero distribution among the subsets. The density of the three weight groups is 70.4 percent, 55.6 percent, and 40.7 percent respectively, resulting in workload imbalance for the three PEs. The early finishing PE2 has to be idle while waiting for PE0 because the work corresponding to the next layer cannot proceed until all the PEs finish the allocated output channels.

In contrast, AdaPrune prunes at a more fine-grained granularity and guarantees that each weight group has a comparable sparsity, as shown in Fig. 6b. We found the main reason for the sparsity variance is that previous pruning methods use a fixed threshold when performing pruning, i.e., the weights that are below a *fixed* threshold are pruned to zero. Therefore, some weight groups show a higher sparsity because most of the weights in the group are smaller than the threshold, while other weight groups may show a lower sparsity because fewer weights in the group are smaller than the threshold. Based on this observation, instead of using a fixed threshold for all weights, AdaPrune uses an adaptive threshold for each weight group to guarantee that they show comparable sparsity after pruning. The threshold for each filter group is set so that a fixed number of weights is pruned to zero. For example, if the target sparsity is 30 percent , the threshold for each weight group will be set as the 30th-percentile value of each weight group. Following this rule, the threshold for $weight\ group\ 0$ in Fig. 6b will be larger than

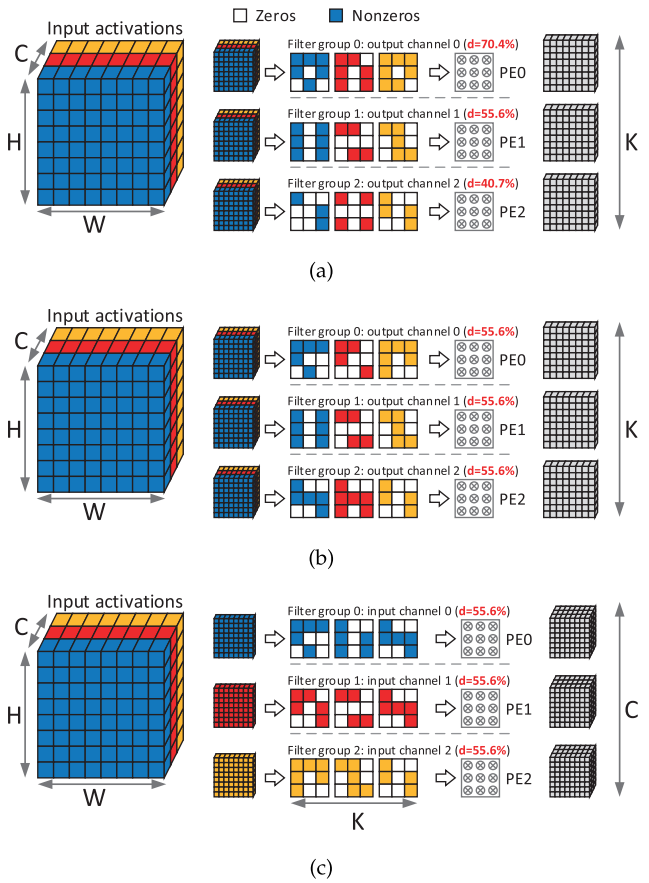


Fig. 6. Example of Output Channel Group Tiling (OCGP) and Input Channel Group Tiling (ICGP). (a) Unstructured pruning results in unbalanced sparsity between weight groups. (b) OCGP individually prunes weights grouped by output channels to force comparable sparsities in different weight groups. (c) ICGP individually prunes weights grouped by input channels to force comparable sparsities in different weight groups.

that of *weight group* 2 to prune more weights. Because the weights are grouped by output channels, we call it output channel group pruning (OCGP). OCGP works layer by layer. The main steps of OCGP for a given convolutional layer are shown in Fig. 4. The first step is weight grouping. For a given layer, we divide all the weights into multiple weight groups according to the tiling factor T_k. The second step is pruning each weight group so that the weight groups have the same sparsity. The sparsity level is set to be comparable with that in unstructured pruning [11] for each layer. The pruned network then will be retrained to recover the accuracy. We iteratively apply the pruning and retraining until the original accuracy is recovered.

Similarly, for accelerators using input channel tiling, the weights are grouped and pruned for each input channel, as shown in Fig. 6c. The rationale is the same and we omit the details for brevity. In planar tiling, since the weights are broadcast to the PEs, i.e., every PE is allocated all the weights of a layer, the density of weights are naturally kept the same for each PE. Therefore, we apply unstructured pruning in [11] for accelerators using planar tiling.

4 Accelerator Platform

We test AdaPrune on three different CNN accelerators: Cambricon-X, SCNN, and SqueezeFlow. The characteristics of these accelerators are summarized in Table 3. Since the RTL code of SCNN and Cambricon-X is not open-sourced, we implement a unified accelerator that supports the dataflow of the three accelerators. This section introduces the details of the accelerator platform.

TABLE 3
Accelerator Platforms Using Different Spatial Tiling Strategies

Architecture	SqueezeFlow	Cambricon-X	SCNN
Spatial tiling strategy	Planar tiling	Output channel tiling	Planar tiling
Inner spatial dataflow	Matrix scalar product	Dot product	Cartesian product
Avoid transfer of all zeros	No	No	Yes
Support weight sparsity	Yes	Yes	Yes
Support activation sparsity	No	No	Yes

4.1 Overview

Fig. 7 demonstrates the full accelerator architecture for evaluating AdaPrune. The accelerator consists of three major components: a Processing Unit (PU), a Global Buffer (GLB), and a Controller. The PU contains a PE array connected via a network-on-chip (NOC). Each PE can operate individually on its own workload so the accelerator can support parallel execution of the CVLs. Each PE can exchange data with neighbor PEs via the NoC. The accelerator provides a four-level memory hierarchy to maximize data reuse. Specifically, GLB is used to temporarily store a portion of data to hide DRAM access latency. The inter-PE connections exploit the data reuse among PEs and reduce the memory accesses to GLB. The four-level memory hierarchy significantly reduces the memory accesses thereby saving energy.

To process a CVL, the input activations and weights are fetched from the DRAM into the GLB. The Controller designates the data movement according to the dataflow and assigns the corresponding activations/weights to the PEs. Afterward, the PEs operate on their own workload to achieve high computing parallelism. After the PEs accomplish computing, the results are then written back to DRAM if needed.

Fig. 7 also presents the micro-architecture of the PEs. Each PE consists of a multiplier array, a PE controller, local buffers to store input activations and weights, a Data Dispatcher (DP), a Coordinate Computation Unit (CCU), and a Post-Processing Unit (PPU), a scatter network and accumulator buffers. The multiplier array performs the core multiply operations. The accumulation is performed in the accumulator buffers, while the activation function is performed in the PPU if necessary.

The PE accomplishes its assigned workload as follows. The assigned activation/weights from the GLB is stored in its local buffer (SB and NBin). The DP fetches the input activations and weights from the corresponding buffers to the multiplier array to perform the multiplication. The multiplication results are then delivered to the Accumulator Buffers

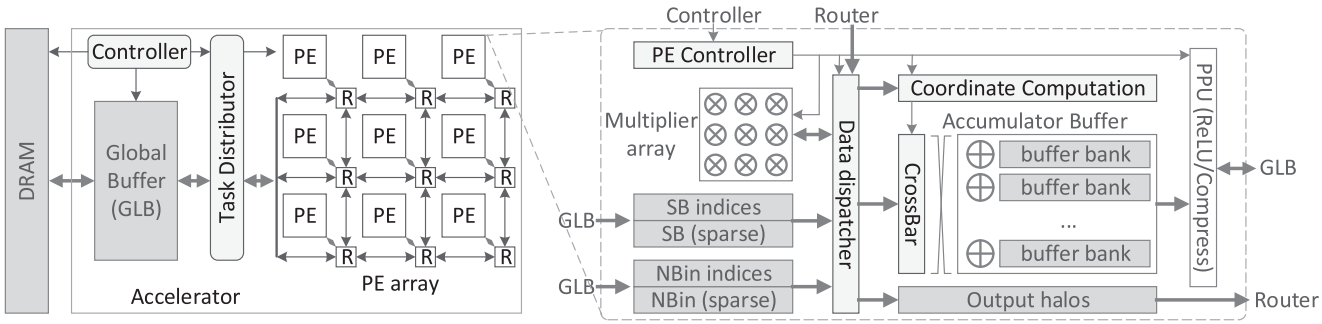


Fig. 7. Full accelerator architecture for evaluating AdaPrune.

for accumulation. The coordinates of the multiplication results are calculated in the CCU. The PE controller determines the processing order by controlling the order of data streaming from the local buffers to the multiplier array. In doing so, the PE can be reconfigured to match the dataflow of different accelerators.

4.2 Flexible Dataflow

This accelerator can act as Cambricon-X, SqueezeFlow, and SCNN by configuring the Control Unit and the PE controller. We take SqueezeFlow as an example to illustrate its working principle. SqueezeFlow is an accelerator that exploits PlanarTiling-OutputStationary-sparse dataflow that eliminates computations related to zero weights. Fig. 8 shows the pseudo-code SqueezeFlow's dataflow for a single PE. Specifically, step A and B are the loops over the K and C dimension assigned to the PE. Step C and D block in the W and H dimension to fit the multiplier array. Then, step E fetches a scalar of the weights and step F fetches the corresponding index from SB to compute the coordinates. Afterwards, the matrix scalar product is performed (step G) and the multiplication results are accumulated (step H).

In SqueezeFlow, the weights are stored in compressed format in SB. For a single PE, the volume of assigned weights is $Tc \times K \times R \times S$, which are encoded to a vector with V elements. SqueezeFlow cannot exploit activation sparsity so the input activations with a volume of $Tc \times W \times H$ are stored in NBin in dense format. In each computing cycle, the DP fetches a scalar from the SB and a matrix of size $Tw \times Th$ from NBin to the multiplier array to perform matrix scalar multiplication. The index of the weight is also fetched from SB and delivered to CCU to calculate the corresponding coordinates of the multiplication results using $Xcoord$ and $Ycoord$ functions. The multiplication results are then scattered to be accumulated in the Accumulator Buffers according to the coordinated derived from CCU. Since SqueezeFlow exploits output-stationary computation order in each PE, we also need to configure the PE controller to match this feature. When possible, the output activations are held in the Accumulator Buffers and are not replaced until finishing its related computations. When the PE completes its assigned workload, the output data in the Accumulator Buffers are delivered to PPU for the following processing. The details for SqueezeFlow can refer to the original paper [20]. Similarly, the accelerator can also be reconfigured as Cnvlutin, Cambricon-X or SCNN. We omit the details for brevity.

```
    BUFFER wt_buf[Tc][K][V];
    BUFFER wt_idx_buf[Tc][K][V];
    BUFFER in_buf[Tc][W][H];
    BUFFER out_buf[K][W+R-1][H+S-1];
(A) for k = 0 to K-1
    {
(B)   for c = 0 to Tc-1
      {
(C)     for w' = 0 to W/Tw-1
        {
(D)       for h' = 0 to H/Th-1
          {
            for v = 0 to V-1
            {
(E)           wt = wt_buf[c][k][v];
(F)           wt_idx = wt_idx_buf[c][k][v];
              // Multiplier array level
(G)           parallel_for (w = 0 to Tw-1) x (h = 0 to Th-1)
              {
                x = Xcoord(w', w, v, wt_idx);
                y = Ycoord(h', h, v, wt_idx);
(H)             out_buf[k][w'*Tw+w][h'*Th+h] += in[c][x][y]*wt;
    } } } } } }
```

Fig. 8. Dataflow of SqueezeFlow, single-PE loop nest.

4.3 Balancing Activation Sparsity

Besides weight sparsity, activation sparsity can also be leveraged for increasing performance. Two sources will cause activation sparsity: one is the rectified linear unit (ReLU) function [1] and the other is zero padding. Unlike weight sparsity that is fixed after pruning, the activation sparsity is generated on-the-fly and is related to the input data.

The random distribution of activation sparsity also results in workload imbalance for the PEs. Specifically, input channel tiling and planar tiling would incur workload imbalance from activation sparsity while output channel tiling would not. As shown in Fig. 6. the PEs share the input activations when using output channel tiling, so the activation sparsity imbalance would not cause workload imbalance for the PEs. However, when using input channel tiling, some PEs will be allocated input activations with a high sparsity so they will finish their workload earlier, causing workload imbalance. Accelerators using planar tiling will also incur the similar problem.

We adopt flexible tiling factors in our dataflow to address the workload imbalance from activation sparsity. We use planar tiling as an example to illustrate its working principle. In planar tiling, each PE is assigned with an input activation volume of $C \times T_w \times T_h$. As shown in Fig. 9, the sparsity of input activations is vastly different. To make the effective computations in each tile comparable, the size of the tiles does not necessarily keep the same. For example, the work size can be $T_{k1} \times T_{c1} \times T_{w1} \times T_{h1} \times T_{s1} \times T_{r1}$ for PE1, $T_{k2} \times T_{c2} \times T_{w2} \times T_{h2} \times T_{s2} \times T_{r2}$ for PE2, where

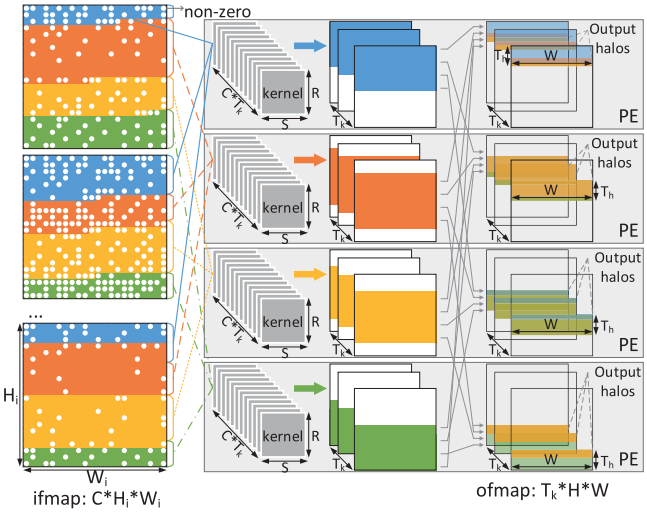


Fig. 9. Exploring dynamic tiling factors to avoid workload imbalance in terms of zero activations.

$T_{k1} \neq T_{k2}, T_{c1} \neq T_{c2}$, etc. It is hard to make all six dimensions variable because it will generate a huge exploration space. More variables indicate a larger exploration space. Alternatively, we can keep some tiling factors fixed while others are variable. We found that choosing T_h to be variable is enough to keep work balance. Then, PE_i will be allocated task size of $T_{hi} \times C \times W \times H \times S \times R$. If T_{hi} is carefully determined, the zero activations will be balanced. The Task Distributor is responsible for the flexible workload dispatch. It monitors the sparsity of input activations and determines T_{hi} to ensure that the PEs are allocated with a comparable workload.

5 Experimental Methodology

5.1 Architectural Configurations

We use the accelerator introduced in Section 4 to evaluate the proposed pruning technique. Table 4 lists the key design parameters of the accelerator, which employs a PU composed of a 4×4 array of PEs. The global buffer size is 4 MB. In each PE, NBin and SB SRAM size are both 84 KB. We use 16-bit fixed-point arithmetic units which are widely used in CNN accelerators [7]. The detailed configuration is summarized in Table 4.

5.2 Baselines

We compare AdaPrune with both unstructured (Deep compression) and structured pruning methods, including Soft Filter Pruning [21], Network Slimming [22], Discrimination-aware Pruning [23], Low-cost Collaborative Layers [24], Feature Boosting& Suppression [25], CGNet [26], and AAP [27]. Soft Filter Pruning [21] is a channel pruning technique that enables the pruned filters to be updated when training the model after pruning. Network Slimming [22] automatically identify the channels and pruned them during training. Discrimination-aware pruning [23] uses greedy algorithms to select the channels to be pruned. Low-cost Collaborative Layers [24] is used to equip the original convolutional layer to be more discriminative. Feature Boosting & Suppression is a dynamic channel pruning technique that predicatively amplifies salient convolutional channels and prunes them.

TABLE 4
Accelerator Configuration

PE Parameter	Value	Accelerator Parameters	Value
Data width	16 bits	#PEs	16
NBin	84 KB	#Multipliers	1,344 KB
SB	84 KB	NBin data	1,344 KB
Multiplier array	4×4	Global buffer	4 MB
Accumulator banks	16		
Accumulator bank entries	32		

CGNet [26] is a dynamic, fine-grained channel pruning scheme that identifies unimportant regions in the features and skip them for computational savings. AAP [27] is an accelerator-aware pruning scheme that considers the width of the activation buffer and the number of multipliers so that the same number of weights remain for each weight group.

Notably, some compression methods also leverage weight quantization [11], which reduces the number of bits to represent a weight to further reduce model size. Since quantization does not reduce the number of multiplications, we do not consider quantization in this paper. We implemented AdaPrune, Unstructured Pruning and AAP in the PyTorch framework to obtain the accuracy and weight density numbers. For other pruning methods, we use the numbers from the original papers. We run these models on the accelerator to evaluate how the pruning schemes improve performance and energy efficiency.

5.3 Benchmarks

We use the CNNs listed in Table 5 as the benchmarks to evaluate these accelerators, including LeNet5 for MNIST hand-written digit dataset, Cifar10 quick model [28], AlexNet, VGG16, GoogLeNet [29], and ResNet18 [3] for ImageNet dataset. We primarily focus on the CVLs of these networks. The CNN models are extracted from Pytorch after applying the pruning methods.

5.4 Implementation

Algorithm. The pruning and retraining were performed by Pytorch for the benchmark networks. We added weight grouping and group pruning stage to the deep compression algorithm to implement AdaPrune. The detailed implementation can refer to [11]. The batch size is 256, the learning rate schedule starts at 0.01 and decays by a factor of 5 every 5 epochs. The number of epochs is 30. Note that for ResNet18 the learning rate starts at 0.1.

Accelerator Simulator. Our simulator is built based on the open-sourced TimeLoop simulator [30]. The simulator is combined with DRAMSim2 [31] to evaluate the performance. The simulator takes the weights and activations extracted from Pytorch as input and processes one layer at a time. It simulates the dataflow, the memory hierarchy, and the 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

TABLE 5
Comparison of Accuracy and Weight Density of the Pruning Methods for ImageNet

Networks	Dataset	Pruning methods	Flop Reduction (%)	Top-1 error (%)	Top-1 Accuracy Change (%)†	Top-5 error (%)	Top-5 Accuracy Change (%)†
ResNet18	ImageNet	Deep compression	2.0×	31.03	-0.23	11.50	-0.30
		Soft Filter Pruning [21]	1.7×	32.90	-3.18	12.22	-1.85
		Network Slimming [22]	1.4×	32.79	-1.77	12.61	-1.29
		Discrimination-aware Pruning [23]	1.9×	32.65	-2.29	12.40	-1.38
		Low-cost Collaborative Layers [24]	1.5×	33.67	-3.65	13.06	-2.3
		Feature Boosting& Suppression [25]	2.0×	31.83	-2.54	11.78	-1.46
		CGNet [26]	1.6×	31.20	-0.40	12.20	-1.20
		AAP [27]	2.0×	31.57	-0.77	12.66	-1.46
		Input channel group pruning	2.0×	31.06	-0.26	11.57	-0.37
		Output channel group pruning	2.1×	30.93	-0.13	11.43	-0.23
VGG16	ImageNet	Deep compression	3.0×	31.17	+0.37	10.91	+0.41
		Network Slimming [22]	2.8×	36.7	0	-	-
		AAP [27]	3.0×	32.08	-0.54	12.15	-0.83
		Input channel group pruning	2.9×	31.40	+0.1	11.13	+0.19
		Output channel group pruning	3.0×	31.31	+0.19	10.95	+0.37
GoogLeNet	ImageNet	Deep compression	0.0×	31.32	-0.12	11.17	-0.17
		Input channel group pruning	0.0×	31.29	-0.09	11.15	-0.15
		Output channel group pruning	0.0×	31.28	-0.08	10.14	-0.14
AlexNet	ImageNet	Deep compression	3.4×	42.78	0	19.70	+0.03
		CGNet [26]	2.6×	57.10	-14.30	20.0	-0.27
		Input channel group pruning	3.4×	42.88	-0.10	19.83	-0.10
		Output channel group pruning	3.4×	42.83	-0.05	19.79	-0.06
ConvNet	Cifar10	Deep compression	3.8×	24.33	-0.13	-	-
		Input channel group pruning	3.8×	24.39	-0.19	-	-
		Output channel group pruning	3.9×	24.28	-0.08	-	-
LeNet5	MNIST	Deep compression	6.3×	0.74	+0.06	-	-
		Input channel group pruning	6.4×	0.77	+0.03	-	-
		Output channel group pruning	6.2×	0.74	+0.06	-	-

†*Since different work have different baseline accuracies, the accuracy change numbers are derived from the orignal papers.*

obtain overall execution time. Meanwhile, these statistical data are also used to build an energy model to estimate the energy consumption of the accelerator. We use the energy numbers in [32] to estimate the energy of arithmetic units and memory accesses, while SRAM energy is estimated by CACTI 6.0 [33]. We simulate three accelerators mentioned in Section 4: SqueezeFlow, Cambricon-X, and SCNN. As the original SCNN accelerator uses planar tiling, we also simulator the SCNN accelerator with different spatial tiling strategies, namely SCNN-I for input channel tiling, SCNN-O for output channel tiling, for a thorough evaluation.

CAD Tools. We implement the accelerator platform in Synopsys design flow on TSMC 65 nm technology: simulating with Synopsys Verilog Compile Simulator (VCS), synthesizing with Synopsys Design Compiler (DC), analyzing power with Synopsys PrimeTime (PT), and placing them with Synopsys IC Compiler (ICC).

6 Experimental Results

In this section, we first evaluate the accuracy and FLOP reduction of AdaPrune and the baseline pruning methods. We then evaluate the area, power, and speedup when we run the original dense networks, the pruned networks using unstructured pruning (Deep Compression) and AdaPrune on the sparse accelerators.

6.1 Pruning results

Table 5 lists the weight density and accuracy results of the pruning methods. In input/output channel group pruning, the channel group is set as $T_c = 4, T_k = 4$, respectively. Since AAP has the flexibility to tailor the number of weight to be pruned, we set the parameters so that AAP share comparable sparsity with our pruning method. As shown in the table, AdaPrune can basically preserve the original accuracy for all the networks. Moreover, AdaPrune achieves a comparable pruning ratio compared to deep compression, which demonstrates that AdaPrune is as effective as unstructured pruning on the algorithmic side. The structured pruning methods usually show higher accuracy loss compared with AdaPrune. For example, CGNet even incurs 14.3 percent top-1 accuracy loss for AlexNet.

In Fig. 10, we show the trade-off between top-5 accuracy and pruning rate on AlexNet. The accuracy of both methods begins to drop drastically when the pruning rate is below a certain number. The accuracy of ICGP and OCGP remains

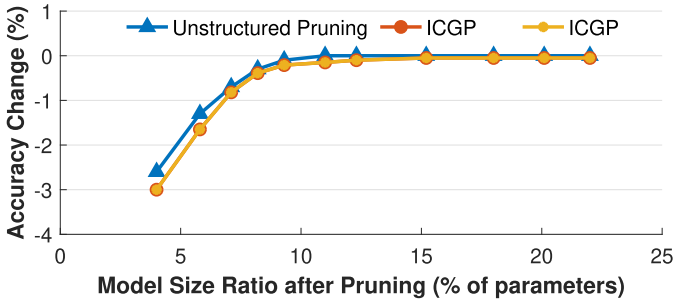


Fig. 10. Accuracy versus pruning rate of AlexNet under different pruning methods.

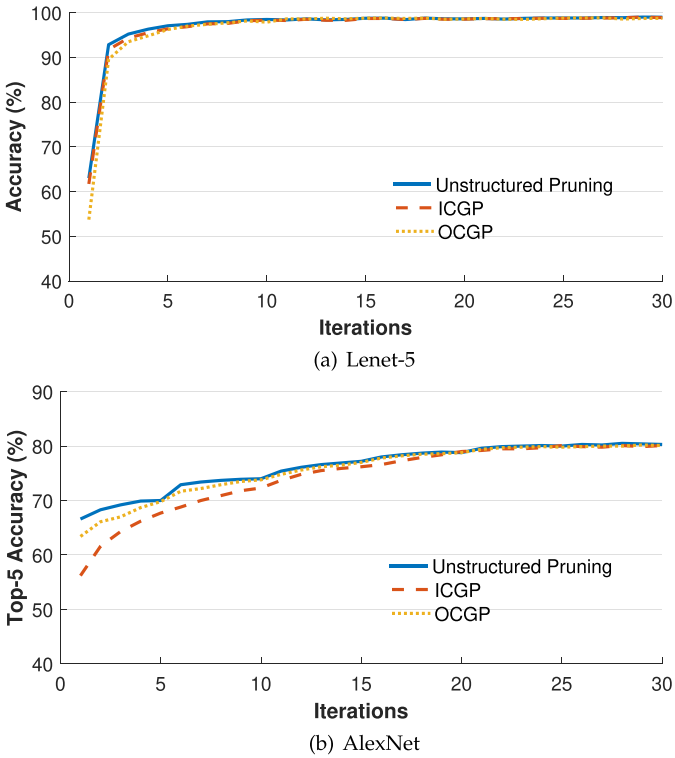


Fig. 11. Test accuracy under different pruning methods.

consistently comparable with unstructured pruning, which reveals that AdaPrune has a good capability of preserving accuracy.

In Fig. 11, we show the test performance for LeNet-5 and AlexNet using different pruning methods on every iteration. For both networks, unstructured pruning slightly edges out as the best performing. Specifically, unstructured pruning achieves a mean accuracy of $99.26 \pm 0.04\%$ for LeNet-5, and a mean Top-5 accuracy of $80.30 \pm 0.05\%$ for AlexNet. Although the convergence speed of ICGP and OCGP is slightly lower than unstructured pruning, the final accuracy of these two pruning methods is about the same with unstructured pruning. ICGP achieves a mean accuracy of $99.23 \pm 0.03\%$, $80.17 \pm 0.04\%$ for LetNet-5 and AlexNet, respectively, while OCGP achieves a mean accuracy of $99.26 \pm 0.02\%$, $80.22 \pm 0.03\%$ for LetNet-5 and AlexNet, respectively. The results again reveal that AdaPrune preserves accuracy while considering the accelerator constraints.

6.2 Layout Characteristics

Table 6 presents the layout characteristics of the accelerator platform. The PE consumes an area of $5.53mm^2$ and power of $396.33mW$, while the Task Distributor only consumes $0.21mm^2$ and $14.58mW$. As the Task Distributor is used to balance the activation sparsity, the results indicate that this function only introduces limited overhead.

6.3 Accelerators Using Output Channel Tiling

We first test AdaPrune on the SCNN-O and Cambricon-X accelerator that use output channel tiling. AdaPrune chooses OCGP as the pruning method for SCNN-O and Cambricon-X. Fig. 12a presents the performance speedups of the original and pruned networks. For all tested networks

TABLE 6
Area and Power of the Components in the Accelerator Platform

Components		Area(mm^2)	Power(mW)
PE		5.53	396.33
	Multiplier Array	0.46	146.54
	NBin	1.28	47.26
	SB	1.28	28.92
	Accumulator Buffer	0.98	46.17
	PE Controller	0.16	36.83
	Data Dispatcher	0.11	10.22
	CCU	0.24	19.48
	PPU	0.56	44.76
	Crossbar	0.46	16.15
Task Distributor		0.21	14.58

running on SCNN-O, OCGP consistently outperforms the other pruning techniques and achieves a speedup of $5.0\times$ and $1.6\times$ over original models and unstructured pruning, respectively. The performance improvement of OCGP exhibits a wide variation across different networks. Specifically, OCGP improves the performance by 4.4–$6.0\times$ over the original models, 1.3–$1.9\times$ over unstructured pruning, and 1.2–$1.8\times$ over ICGP. OCGP achieves the highest speedup over unstructured pruning on AlexNet. This is because AlexNet uses 11×11 kernel size in the first convolutional layer. In unstructured pruning, we observed that in this layer the nonzeros in the weight fetching group are easier to become unbalanced, which results in the severe performance degradation of SCNN.

In Fig. 12b, we report the energy consumption which has been normalized to the energy consumption of running the original models. In the energy evaluation, we do not consider the energy consumed by main memory accesses

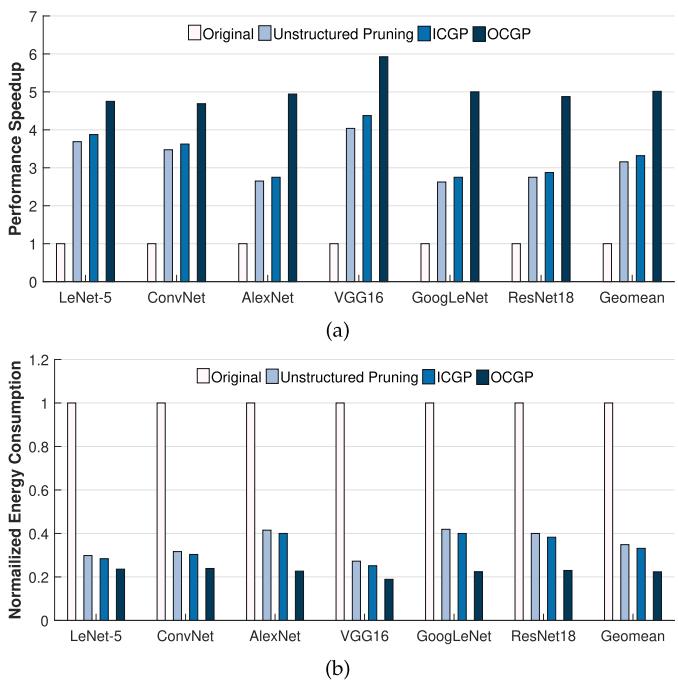


Fig. 12. Relative performance speedups (a) and energy savings (b) of the original models, unstructured pruning, ICGP, and OCGP on SCNN-O.

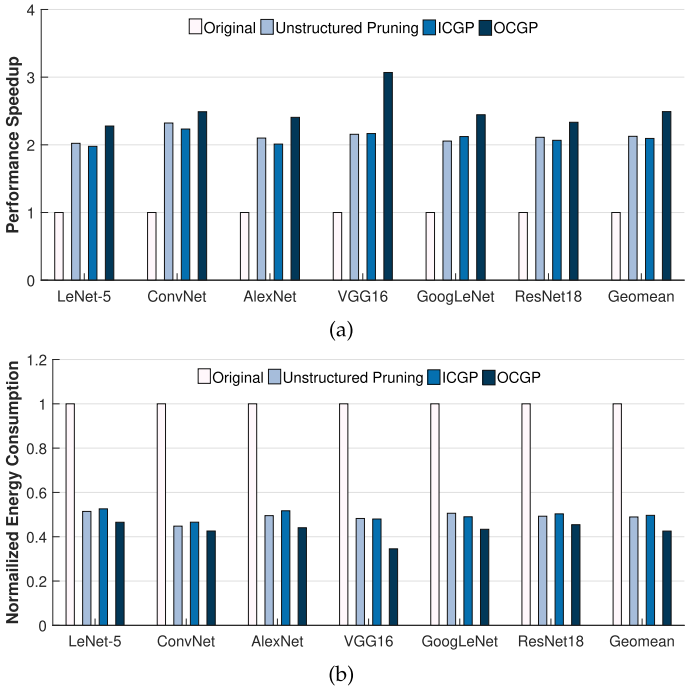


Fig. 13. Relative performance speedups (a) and energy savings (b) of the original models, unstructured pruning, ICGP, and OCGP on Cambricon-X.

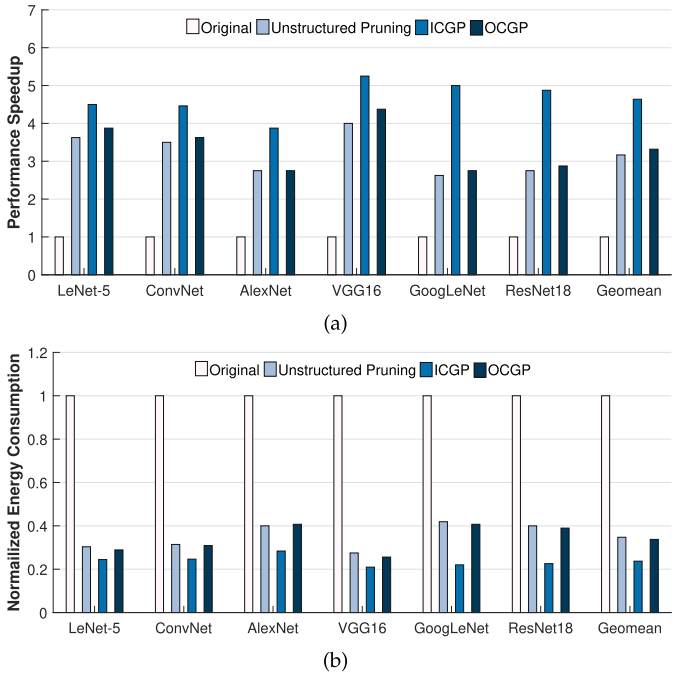


Fig. 14. Relative performance speedups (a) and energy savings (b) of the original models, unstructured pruning, ICGP, and OCGP on SCNN-I.

because it usually dominates the total energy consumption [7]. On average, AdaPrune improves the energy efficiency by 4.5×, 1.5× over the original models and the models pruned by unstructured pruning, respectively.

For Cambricon-X which uses output channel tiling but only removes zero-weight related computation, we also use OCGP to make the sparsity of each weight fetching group comparable. Fig. 13 presents the performance speedups and energy savings of the original and pruned networks. The average speedup of OCGP over the original models is 2.6×, which is dwarfed compared to SCNN-O because Cambricon-X still has to perform computations with zero activations. For all the networks, OCGP achieves speedups of 1.2×, 1.2×, and energy savings of 1.1×, 1.2× on average over traditional pruning and ICGP, respectively,

6.4 Accelerators Using Input Channel Tiling

For SCNN-I that uses input channel tiling, AdaPrune chooses ICGP as the pruning method. Fig. 12 presents the performance speedups and energy savings of the original and pruned networks. For all the networks, ICGP achieves speedups of 4.7×, 1.5×, 1.4×, and energy savings of 4.2×, 1.5×, 1.4× on average over original models, unstructured pruning and OCGP, respectively. The result again demonstrates that AdaPrune created a more efficient pruned network for the sparse accelerators.

6.5 Accelerators Using Planar Tiling

We further test AdaPrune on the SCNN and SqueezeFlow accelerator that use planar tiling. Figs. 15 and 16 presents the performance speedups and energy savings of the original and pruned networks on the two accelerators. Unstructured pruning, ICGP and OCGP achieve comparable speedups because planar tiling is free from the workload imbalance introduced by unbalanced weight sparsity distribution. In this case, AdaPrune chooses unstructured pruning as the pruning method. As mentioned in Section 4.3, we use flexible tiling factors to balance the activation sparsity. To evaluate the effect of this method, Fig. 15 also shows the

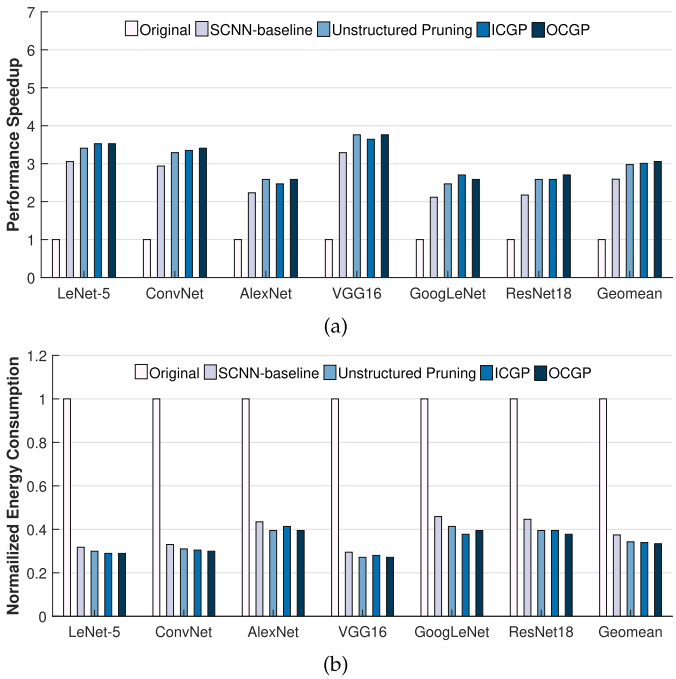


Fig. 15. Relative performance speedups (a) and energy savings (b) of the original models, unstructured pruning, ICGP and OCGP on SCNN. We also show the speedup of the orginal SCNN accelerator to evaluate the impact of activation sparsity.

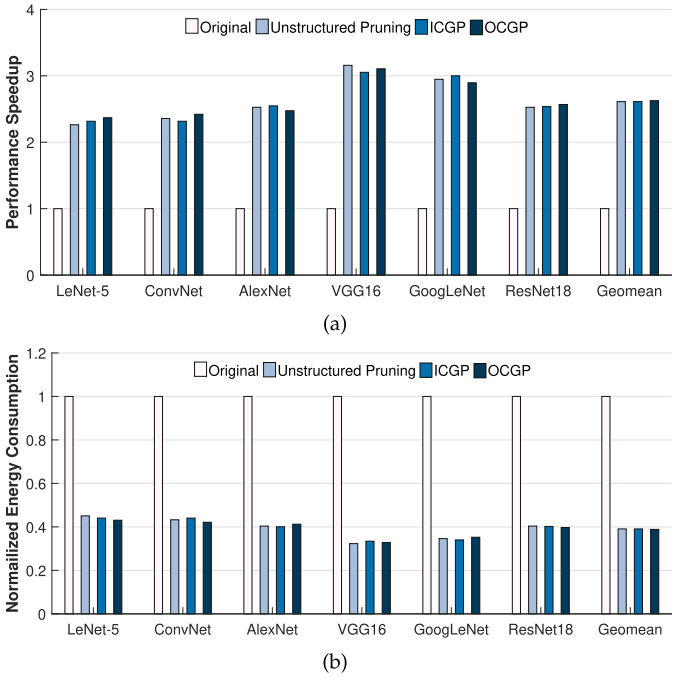


Fig. 16. Relative performance speedups (a) and energy savings (b) of the original models, unstructured pruning, ICGP and OCGP on SqueezeFlow.

performance and energy consumption of the original SCNN accelerator, denoted as SCNN-baseline. The results show that this technique improves performance by 1.18× and saves energy by a factor of 1.1× averaged on the six networks. Notably, planar tiling would cause intra-PE fragmentation problem [16] because each PE must process a larger working set to fully utilize the MAC units, so we found that the speedup of planar tiling is dwarfed compared to input/output channel tiling.

From the above evaluations, we observed a strong correlation between performance speedup and energy consumption. Specifically, higher performance speedup is usually associated with lower energy consumption. The reason is that the performance speedup is achieved by reducing the idle states of the computational units, which also reduces the energy consumption the computational units.

In summary, The networks pruned by AdaPrune achieve a better performance than unstructured pruning because AdaPrune customizes the pruned networks to match the underlying accelerators.

7 Related Work

Accelerating convolutional neural networks has been widely and extensively studied in these years. The most related work to ours is weight pruning, accelerator architectures supporting dense and/or sparse CNN models and weight quantization. We will review these works in this section.

7.1 CNN Weight Pruning

Although modern neural networks have been widely used in many applications, they are notorious for their intensive computation and memory accesses. Many efforts have been devoted to reducing model size and computations. One of the most effective approaches is weight pruning that removes the unimportant weights without affecting the accuracy. The pruning techniques can be classified into two categories: *unstructured pruning* and *structured pruning*.

Unstructured pruning has no geometric constraints but prunes as more weights as possible, which leads to more than 10× data reduction [11] with negligible accuracy loss. However, the irregular sparsity introduced by unstructured pruning destroys the regular computation patterns in CNNs, and is difficult to be efficiently processed accelerators designed for fine-grained parallelism. Although some sparse accelerators have been proposed to support sparsity [15], [16], [17], the performance gain is much lower than the reduction in computation.

To this end, pruning regularity has become an important metric because it's highly related to the practical speedup of the accelerators [12] have been proposed to maintain the computational regularity and accelerate the decoding of sparse matrices, which places non-zero weights at predefined locations. The structured pruning schemes can be divided into three categories, namely channel-wise, filter-wise, and shape-wise pruning. In filter-wise pruning, for example, all the weights in a filter are considered as a group and are pruned or not together. This geometric constraint preserves the computational regularity so that the pruned networks can be efficiently processed by the underlying hardware. However, it often induces accuracy loss compared to unstructured pruning [14].

7.2 Dense CNN Accelerators

There exist many accelerator architectures for dense neural networks, with implementations on either FPGAs or ASICs and optimizations for computation [10], [34], [35], [36], memory [7], [8], [37], [38], [39], [40], [41], [42], [43], [44], [45] and data reuse [9], [46], [47]. DianNao [7] focuses on the memory accesses optimization for large-scale layers. ShiDianNao [8] is designed for vision processing, which completely eliminates the DRAM accesses. Zhang *et al.* [18] exploits a roof-line model to optimize CNNs on FPGA accelerators. Suda *et al.* [48] presents a design space exploration considering the resource constraints to optimize FPGA accelerators. Eyeriss [9] employs a row stationary dataflow to achieve high throughput and energy efficiency for CNN accelerators. FlexFlow [49] utilize a flexible and reconfigurable PE array to realize different dataflow so that it can maximize the utilization for any given convolutional layer. DNA [50] also focuses on the inefficiency problem for different layers and supports reconfigurable computation patterns to match the given layers. MAERI [51] employs a modular and configurable design to support different CNN partitioning strategies to achieve optimal efficiency.

7.3 Sparse CNN Accelerators

Given that weight pruning significantly reduces model size and computation, many sparse accelerators have been proposed to take advantage of the sparsity benefits. Cambricon-X [15] and Cnvlutin [19] skip unnecessary computations related

to zero weights and activations, respectively. SCNN [16] eliminates the unnecessary computations related to both zero activations and weights by employing Cartesian Product as its inner dataflow. EIE [52] eliminates the zero computations in fully-connected layers by accelerating sparse matrix-vector multiplication. UCNN [53] observes the weight repetition in CNNs and leverages the computational reuse opportunity to boost accelerator performance. Although these solutions achieve a performance improvement, we found that they incur high hardware overhead and performance degradation because of the unbalance zero distribution.

Regarding the irregularity problem, Mao *et al.* [54] systematically evaluate the relationship between sparsity and accuracy and find that pruning at coarse-grained granularity is more hardware-friendly than at fine-grained granularity. Scalpel [55] customizes DNN pruning for underlying hardware platforms with different levels of parallelisms, but the technique is optimized for CPUs or GPUs other than accelerators. Kang [27] proposes an accelerator-aware pruning scheme that guarantees the load imbalance inside a PE, which is complementary to our proposed method. Cambricon-S [56] employs coarse-grained pruning combined with local quantization to reduce the irregularity of weights. SparTen [17] employs efficient inner-join and tackles load-imbalance by software/hardware hybrid approach. Stitch-X [57] stitches sparse weights and input activations together for parallel execution. However, due to the intrinsic irregularity, these approaches incur overhead for sparse matrix representation.

7.4 Weight Quantization

Weight quantization is a technique that uses fewer bits to represent weight elements to compress the model. Researchers have used binary values [58], ternary values [59] so that both storage and computation can be reduced. Weight quantization is often used cooperatively with weight pruning. ADMM-NN [60] is a joint framework of weight pruning and quantization that uses the Alternating Direction Method of Multipliers to solve non-convex optimization problems. As we primarily focus on weight pruning in this work, the combination with weight quantization will be left as future work.

8 Conclusion

Motivated by the observation that prior pruning techniques sacrifice either computational regularity or accuracy, we propose *AdaPrune* in this paper to customize CNN pruning for different sparse accelerators to improve their sustainability. AdaPrune consists of two techniques: *input channel group pruning* and *output channel group pruning*. By analyzing the weight fetching patterns of sparse CNN accelerators, AdaPrune adaptively switches between the two methods to guarantee that the zeros are evenly distributed in each fetching group. In doing so, the pruned network structure preserves customized computational regularity for the underlying accelerators, thereby significantly improving the performance and energy efficiency of the accelerators. We compare the performance of AdaPrune with unstructured CNN pruning techniques using sparse CNN accelerators with different spatial tiling strategies. The experimental results show that AdaPrune achieves up to 1.6× performance speedup and 1.5× energy savings compared to unstructured pruning.

Acknowledgments

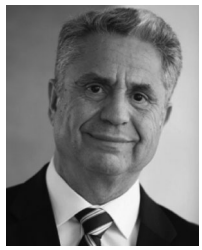
This work was supported by the US National Science Foundation under Grants CCF-1565273, CCF-1702980, and CCF-1901165. The authors would like to sincerely thank the anonymous reviewers for their excellent and constructive feedback.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Neural Inf. Process. Syst.*, 2017, 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 *Proc. 31st AAAI Conf. Artif. Intell.*, 2016.
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Visi. Pattern Recognit.*, 2015, pp. 770–778.
[4] O. Abdel-Hamid, A.-R. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu, "Convolutional neural networks for speech recognition," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 22, no. 10, pp. 1533–1545, Oct. 2014.
[5] V. Pratap *et al.*, "Wav2letter++: A fast open-source speech recognition system," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2018, pp. 6460–6464.
[6] Y. Zhang, W. Chan, and N. Jaitly, "Very deep convolutional networks for end-to-end speech recognition," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2017, pp. 4845–4849.
[7] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *Architect. Support Program. Lang. Operating Syst.*, vol. 49, no. 4, pp. 269–284, 2014.
[8] Z. Du *et al.*, "ShiDianNao: Shifting vision processing closer to the sensor," *ACM SIGARCH Comput. Architecture News*, vol. 43, pp. 92–104, 2015.
[9] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 367–379.
[10] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 1–12.
[11] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Representations*, 2016.
[12] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, 2017, Art. no. 32.
[13] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," 2016, *arXiv:1611.06440*.
[14] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2016, *arXiv:1608.08710*.
[15] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu IEEE/ACM Int. Symp Microarchitecture*, 2016, pp. 1–12.
[16] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 27–40.
[17] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 151–165.

[18] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2015, pp. 161–170.
[19] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 1–13.
[20] J. Li *et al.*, "Squeezeflow: A sparse CNN accelerator exploiting concise convolution rules," *IEEE Trans. Comput.*, vol. 68, no. 11, pp. 1663–1677, Nov. 2019.
[21] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 2234–2240.
[22] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 2736–2744.
[23] Z. Zhuang *et al.*, "Discrimination-aware channel pruning for deep neural networks," in *Proc. Advances Neural Inf. Process. Syst.*, 2018, pp. 875–886.
[24] X. Dong, J. Huang, Y. Yang, and S. Yan, "More is less: A more complicated network with less inference complexity," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 5840–5848.
[25] X. Gao, Y. Zhao, Ł Dudziak, R. Mullins, and C.-Z. Xu, "Dynamic channel pruning: Feature boosting and suppression," 2018, *arXiv: 1810.05331*.
[26] 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 *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 139–150.
[27] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 30, no. 7, pp. 2093–2103, Jul. 2020.
[28] A. Krizhevsky, "CUDA-Convnet: High-performance C++/CUDA implementation of convolutional neural networks," 2012.
[29] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comp. Vis. Pattern Recognit.*, 2014, pp. 1–9.
[30] A. Parashar *et al.*, "Timeloop: A systematic approach to DNN accelerator evaluation," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 304–315.
[31] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Comput. Architecture Lett.*, vol. 10, no. 1, pp. 16–19, Jan.–Jun. 2011.
[32] M. Horowitz, "Energy table for 45nm process," 2012.
[33] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to understand large caches," *HP Laboratories*, 2009.
[34] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comp. Architecture*, 2017, pp. 535–547.
[35] 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," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–30.
[36] J. Li *et al.*, "Synergyflow: An elastic accelerator architecture supporting batch processing of large-scale deep neural networks," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 24, no. 1, pp. 1–27, 2018.
[37] S. Koppula *et al.*, "EDEN: Enabling energy-efficient, high-performance deep neural network inference using approximate DRAM," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 166–181.
[38] L. Pentecost *et al.*, "MaxNVM: Maximizing DNN storage density and inference efficiency with sparse encoding and error mitigation," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 769–781.
[39] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit prudent in-cache acceleration of deep convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2019, pp. 81–93.
[40] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "MnnFast: A fast and scalable system architecture for memory-augmented neural networks," in *Proc. 46th Int. Symp. Comput. Architecture*, 2019, pp. 250–263.
[41] Y. Kwon, Y. Lee, and M. Rhu, "TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 740–753.
[42] 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 *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2018, pp. 655–668.
[43] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2018, pp. 52–65.
[44] A. Ankit *et al.*, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proc. 24th Int. Conf. Architectural Support Programm. Lang. Operating Syst.*, 2019, pp. 715–731.
[45] J. Li *et al.*, "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators," in *Proc. Des. Autom. Test Europe Conf. Exhib.*, 2018, pp. 343–348.
[46] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–12.
[47] A. Azizimazreah and L. Chen, "Shortcut mining: Exploiting cross-layer shortcut reuse in DCNN accelerators," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2019, pp. 94–105.
[48] N. Suda *et al.*, "Throughput-optimized openCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 16–25.
[49] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2017, pp. 553–564.
[50] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 8, pp. 2220–2233, Aug. 2017.
[51] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects," in *Proc. 23rd Int. Conf. Architectural Support Programm Lang. Operating Syst.*, 2018, pp. 461–475.
[52] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Int. Symp. Comput. Architecture*, 2016, pp. 243–254.
[53] 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 *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Architecture*, 2018, pp. 674–687.
[54] H. Mao *et al.*, "Exploring the granularity of sparsity in convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2017, pp. 1927–1934.
[55] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN pruning to the underlying hardware parallelism," *ACM SIGARCH Comput. Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.
[56] X. Zhou *et al.*, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture.*, 2018, pp. 15–28.
[57] C.-E. Lee *et al.*, "Stitch-X: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *Proc. Syst. Mach. Learn. Found. Conf.*, 2018.
[58] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 525–542.
[59] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," 2016, *arXiv:1605.04711*.
[60] A. Ren *et al.*, "ADMM-NN: An algorithm-hardware co-design framework of DNNs using alternating direction methods of multipliers," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 925–938.

Jiajun Li received the BE degree from the Department of Automation, Tsinghua University, Beijing, China, in 2013, and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2019. He is currently a postdoc researcher at the Department of Electrical and Computer Engineering, George Washington University. His current research interests include machine learning and heterogeneous computer architecture.



Ahmed Louri (Fellow, IEEE) received the PhD degree in computer engineering from the University of Southern California, Los Angeles, California, in 1988. He is currently the David and Marilyn Karlgaard Endowed chair professor of electrical and computer engineering at George Washington University, which he joined in August 2015. He is also the director of the High Performance Computing Architectures and Technologies Laboratory. From 1988 to 2015, he was a professor of electrical and computer engineering with the University of Arizona, and during that time, he served six years (2000 to 2006) as the chair of the Computer Engineering Program. From 2010 to 2013, He served as a program director with the National Science Foundation's (NSF) Directorate for Computer and Information Science and Engineering. He directed the core computer architecture program and was on the management team of several cross-cutting programs. He conducts research in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, optical interconnects for scalable parallel computing systems, reconfigurable computing systems, and power-efficient and reliable Network-on-Chips (NoCs) for multicore architectures. Recently he has been concentrating on energy-efficient, reliable, and high-performance many-core architectures, accelerator-rich reconfigurable heterogeneous architectures, machine learning techniques for efficient computing, memory, and interconnect systems, emerging interconnect technologies (photonic, wireless, RF, hybrid) for NoCs, future parallel computing models and architectures (including convolutional neural networks, deep neural networks, and approximate computing), and cloud-computing and data centers. He is the recipient of the 2020 IEEE Computer Society Edward J. McCluskey Technical Achievement Award for pioneering contributions to the solution of on-chip and off-chip communication problems for parallel computing and manycore architectures. He is currently the editor-in-chief of the *IEEE Transactions on Computers*. For more information, please visit https://hpcat.seas.gwu.edu/Director.html