OMNI: A Framework for Integrating Hardware and Software Optimizations for Sparse CNNs

Yun Liang, *Senior Member, IEEE*, Liqiang Lu, and Jiaming Xie

***Abstract*—Convolution neural networks (CNNs) as one of today's main flavor of deep learning techniques dominate in various image recognition tasks. As the model size of modern CNNs continues to grow, neural network compression techniques have been proposed to prune the redundant neurons and synapses. However, prior techniques disconnect the software neural networks compression and hardware acceleration, which fail to balance multiple design parameters, including sparsity, performance, hardware area cost, and efficiency. More concretely, prior unstructured pruning techniques achieve high sparsity at the expense of extra performance overhead, while prior structured pruning techniques relying on strict sparse patterns lead to low sparsity and extra hardware cost. In this article, we propose OMNI, a framework for accelerating sparse CNNs on hardware accelerators. The innovation of OMNI stems from that it uses hardware amenable on-chip memory partition patterns to seamlessly engage the software CNN model compression and hardware CNN acceleration. To accelerate the compute-intensive convolution kernel, a promising hardware optimization approach is memory partition, which divides the original weight kernels into several groups so that the different hardware processing elements can simultaneously access the weight. We exploit the memory partition patterns including block, cyclic, or hybrid as a means of CNN compression patterns. Our software CNN model compression balances the sparsity across different groups and our hardware accelerator employs hardware parallelization coordinately with the sparse patterns, leading to a desirable compromise between sparsity and performance. We further develop performance models to help the designers to quickly identify the pattern factors subject to an area constraint. Last, we evaluate our design on application specific integrated circuit (ASIC) and field-programmable gate array (FPGA) platform. Experiments demonstrate that OMNI achieves 3.4×–6.2× speedup for the modern CNNs, over a comparably ideal dense CNN accelerator. OMNI shows 114.7× energy efficiency improvement compared with GPU platform. OMNI is also evaluated on Xilinx ZC706 and ZCU102 FPGA platforms, achieving 41.5 GOP/s and 125.3 GOP/s, respectively.**

***Index Terms*—Accelerator, CNN, FPGA, NAS, sparse.**

Manuscript received April 12, 2020; accepted August 31, 2020. Date of publication September 14, 2020; date of current version July 19, 2021. This work was supported in part by the National Key Research and Development Program of China under Grant 2019AAA0105200; in part by the Beijing Academy of Artificial Intelligence; in part by the Beijing Natural Science Foundation under Grant JQ19014 and Grant L172004; and in part by the Key-Area Research and Development Program of Guangdong Province under Grant 2019 B010155002. This article was recommended by Associate Editor P. A. Beerel. *(Corresponding author: Yun Liang.)*

Yun Liang is with the CECA, Department of Computer Science and Technology, School of EECS, Peking University, Beijing 100871 China, and also with the Department of Artificial Intelligence, Peng Cheng Laboratory, Shenzhen 518066, China (e-mail: ericlyun@pku.edu.cn).

Liqiang Lu and Jiaming Xie are with the CECA, Department of Computer Science and Technology, School of EECS, Peking University, Beijing 100871, China (e-mail: liqianglu@pku.edu.cn; jmxie@pku.edu.cn).

Digital Object Identifier 10.1109/TCAD.2020.3023903

I. Introduction

Advances in deep neural networks (DNNs) are leading to a number of emerging applications, such as image recognition, semantic segmentation, and speech recognition [1]–[3]. The evolution of DNNs has already piqued interest in hardware acceleration as both DNNs training and inference demand a tremendous amount of computation. As a result, hardware accelerators, such as GPUs, field-programmable gate arrays (FPGAs), and customized application specific integrated circuits (ASICs) have been employed to accelerate DNNs [4]–[25]. However, DNN designers are still hampered by the growing complexity of DNN models. The trend of convolution neural networks (CNNs) is toward deeper and more complex topological structure [26]. The enormous weights in CNNs bring high computation, memory and storage requirements, and limit the deployment of CNNs.

This has subsequently triggered a wide-spreading research endeavor on DNN model compression that compresses the DNNs by pruning the unnecessary synapses and neurons. Early progress was focused on the unstructured pruning, which arbitrarily prune the weights at fine granularity of pixels [14], [27]–[30]. The unstructured pruning techniques can achieve very high sparsity (90% on average) [27], which helps to significantly reduce the on-chip storage requirement for hardware accelerators. However, the high sparsity does not necessarily lead to high performance speedup due to extra encoding and indexing overhead, workload imbalance, poor data locality, etc. It has been shown that it can even hurt the performance when the distribution of sparsity is highly skewed after pruning [14].

Recently, important advances have occurred in structured pruning [4], [31]–[34], which aim at pruning the networks following a certain sparsity pattern. These techniques follow the software-driven design principle. More concretely, it starts with a strict sparsity pattern obeying certain mathematical properties and then designs hardware to support the required mathematical transformation. Therefore, it relinquishes the existing hardware functions and optimization. Although these techniques can achieve high regularity and computation efficiency, they can only achieve limited sparsity (typically 50%) owing to the strict sparsity pattern. Moreover, they require extra hardware units to support the mathematical transformation. For example, CirCNN [35] prunes the weights into block-circulant format. It only achieves 60.5% sparsity for

[1]OMNI is a knowledgeable and brave hero in a game. In this article, we use OMNI to represent powerful and efficient design.

Alexnet. The computation of CirCNN relies on fast Fourier transformation, which is an expensive arithmetic function and requires an extra circuit area. Another observation is that these software-driven structured pruning techniques will likely fall short for modern CNNs with complex neural network topology due to the complexity of training. For example, CirCNN [35] only succeed in small-scale networks, such as LeNet, AlexNet, and PERMDNN [13] is only applicable to FC layer, without validation on modern CNNs such as Resnet.

In this article, we focus on CNNs, which is one of today's main flavor DNNs. The typical computation of convolution is a computation-intensive kernel that performs addition and multiplication by walking through multidimensional tensors (feature map, weight). To increase the hardware parallelism, the original convolution structure is often reorganized into tiles, where each tile corresponds to a hardware processing element (PE). Multiple PEs will access the tensors simultaneously during computation, causing memory bottlenecks. An effective optimization to avoid excessive memory access conflicts is memory partitioning, which divides the original on-chip memories into several banks so that different PEs can access simultaneously. Memory partition has been supported by today's synthesis tools for both FPGAs and ASICs [36]–[38]. Programmers can easily partition the memory into different patterns, such as block or cyclic patterns using directives or pragmas [39], [40].

Inspired by the memory partition patterns, we propose OMNI, a framework for accelerating sparse CNNs on hardware accelerators. OMNI first exploits the memory partition patterns including block, cyclic, and hybrid as a means of compressing DNNs. Then, OMNI prunes the weight by maintaining the same sparsity across different memory partition groups to avoid the PE workload unbalanced issue. After pruning, OMNI efficiently represents the weights in sparse format where the weights from different groups are stored continuously. For the hardware design, OMNI designs an architecture which performs computation directly on the pruned weights. The PE features element-matrix multiplication dataflow, which can maximize the weight reuse and reduce the decoding overhead. To take advantages of the CNN compression, the hardware accelerator must comply with the parallelism where the weights are partitioned so that the PE can be efficiently pipelined with low data access latency. Finally, OMNI optimizes the memory layout and develops resource models for design space exploration.

The advantage of OMNI is that it forgoes the strict sparsity pattern employed by prior structural pruning techniques [4], [13], [35]. Instead, ONMI exploits patterns from existing hardware amenable memory partition techniques. Thanks to the cooperation between hardware memory partition techniques and sparsity patterns, hardware accelerators are able to perform computations concurrently with high data access throughput. The relaxed pruning method of OMNI leads to the easy training process and high sparsity. OMNI can work with virtually any combination of memory partition techniques that the existing hardware already support.

This work makes the following contributions.

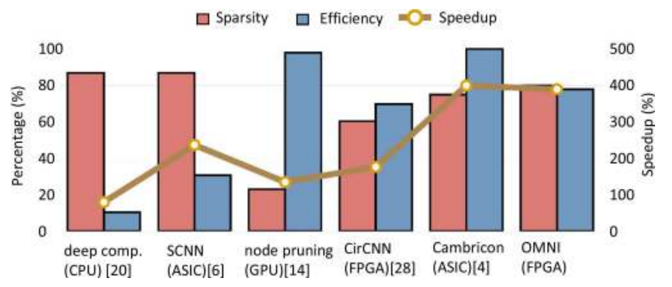


Fig. 1. Sparsity, speedup, and efficiency with different pruning methods for AlexNet.

1) We propose a framework OMNI which uses hardware amenable memory partition patterns to seamlessly combine software CNN model compression and hardware CNN acceleration.
2) We leverage several memory partition patterns as means of CNN compression pattern, which achieves balanced and high sparsity.
3) We design an efficient sparse hardware architecture for CNNs. The architecture employs a novel element-matrix multiplication dataflow. The on-chip memory is partitioned to cooperate with the sparsity pattern, resulting in high data access throughput.

To demonstrate the performance and energy efficiency, we evaluate OMNI architecture in ASIC and FPGA platform. Experiments demonstrate that OMNI achieves 5.8 ×, 6.2×, 3.4×, and 4.2× speedup for the convolutional layers for Alexnet, VGG16, Resnet, and you only look once (YOLO), over a comparably ideal dense CNN accelerator. OMNI achieves more than 114.7× energy efficiency compared with GPUs. OMNI achieves 43.7 GOP/s and 127.1 GOP/s for VGG, 41.5 GOP/s, and 125.3 GOP/s for Tiny-YOLO on Xilinx ZC706 platform and ZCU102 platform, respectively.

II. Background and Motivation

A. Background

A typical CNN is a stack of different layers, such as convolution layers, fully connected layers, and pooling layers. The output of one layer becomes the input of the next layer. Due to their intensive computation, convolutional layers are always the computation bottleneck in CNNs [41]. In this article, we consider the forward procedure in a typical convolutional layer, which receives M channels of $H \times W$ input feature maps $Z_{M\times H\times W}$ and outputs N channels of $R \times C$ feature maps $Y_{N\times R\times C}$ as shown in (1). To generate N channels of output feature map, M channels of input feature maps are convolved with $N \times M$ kernels in size of $kx \times ky$. In the following discussion, we assume the dense weight is a 4-D tensor F with the size of $N \times M \times kx \times ky$:

$$Y_{k,i,j} = \sum_{t=1}^{M}\sum_{p=1}^{kx}\sum_{q=1}^{ky} F_{k,t,p,q} \times Z_{t,i*S+p,j*S+q} \tag{1}$$

where S is the stride of the kernel.

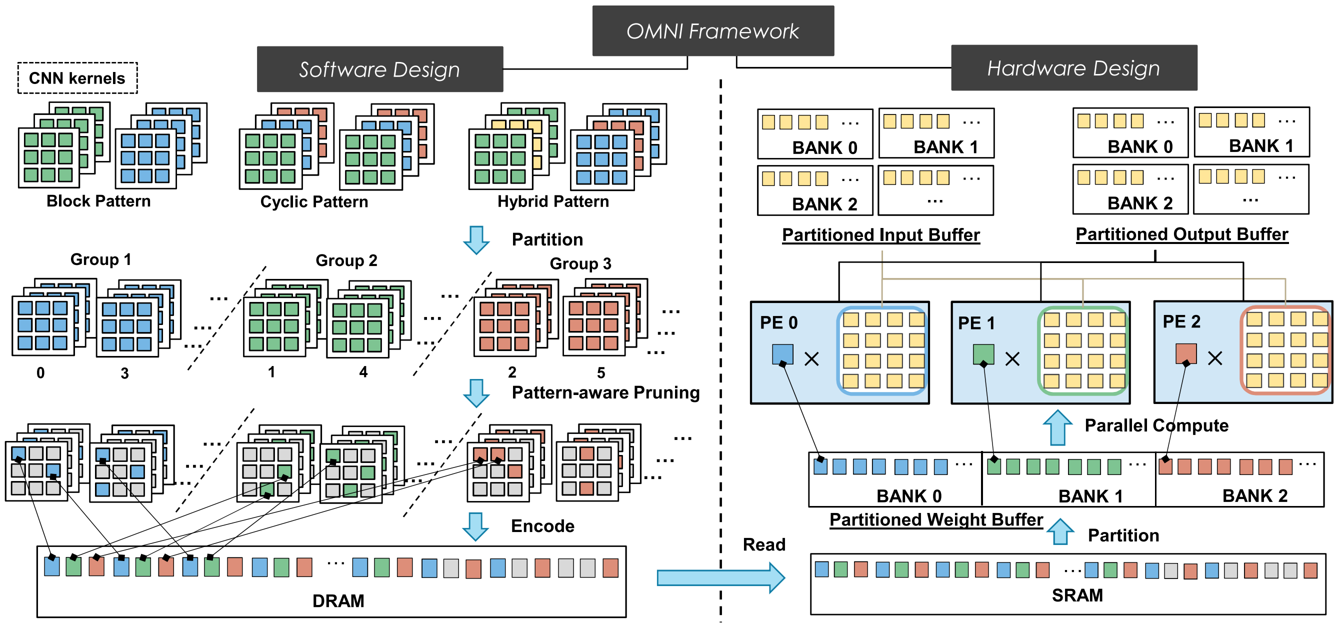


Fig. 2. OMNI framework overview.

B. Motivation

The enormous weights in CNNs is challenging for the memory system in terms of both storage and memory transfer. Therefore, a great number of studies have made efforts in CNNs compression. We classify the existing works into two categories: 1) unstructured pruning and 2) structured pruning. In Fig. 1, we analyze the prior techniques from the sparsity (S), actual hardware speedup (*spe.*), and efficiency (*eff.*) perspectives using AlexNet as an example [42]. The efficiency is calculated as

$$\text{eff.} = (1 - S) \times \text{spe.} \tag{2}$$

More clearly, the theoretical speedup resulting from pruning is $(1/[1 - S])$ compared with handling a dense model. The actual hardware speedup is always less than the theoretical speedup. The hardware efficiency indicates how efficiently the hardware can utilize the pruned sparsity.

Unstructured Pruning: We first evaluate Deep Compression [27], [43] on CPUs using MKL library. It achieves high sparsity (87%) but hurts the performance (spe. = 0.8). SCNN [6] is an ASIC hardware accelerator design for sparse CNNs using the Deep Compression technique. It achieves 2.37× speedup, while the implementation efficiency is less than 30%. The gap between high sparsity and low efficiency mainly comes from two reasons. First, unstructured pruning requires extra memory access and computation to locate the data. For example, in Deep Compression [27], it needs to access the column index in CSR format first to locate the input vector. SCNN [6] applies the Cartesian product dataflow, introducing extra cost for computing the coordinates. Second, the skewed weight distribution can lead to load imbalance issue among parallel threads or hardware units [27].

Structured Pruning: CirCNN [35] uses circulant matrix to represent sparse weights. Circulant matrix is a square circulant matrix where each row vector is strictly required to be the circulant reformat of the other row vector. As shown in Fig. 1, CirCNN can achieve 70% implementation efficiency. However, the final speedup is only 1.9× due to the low sparsity (60% for Alexnet). This phenomenon also happens in Cambricon-S [4]. Cambricon-S prunes the an entire window where the number of large weights is under a threshold. The speedup of Cambricon-S is low as it is also bounded by the sparsity (75% sparsity for convolutional layers of Alexnet and VGG). These techniques all adopt a software-driven methodology. They enforce the weights to exhibit strict sparsity patterns but ignore the hardware features, leading to a limited sparsity. Moreover, utilizing these strict patterns requires extra hardware area. It requires FFT computation to enable the block circulant computation in CirCNN [35]. Similarly, Cambricon-S requires a neuron selector module (NSM) to process the window-based sparsity pattern with shared indexes.

In contrast, OMNI is a hardware-driven structured pruning solution. It uses hardware amenable memory patterns to guide software model compression. Scalple [14] bears some similarity to us. It proposes a node pruning technique specifically for GPU architecture. However, It only yields very low sparsity (23% on average). More importantly, since ASIC and FPGA accelerator architecture are quite distinct from GPUs the node pruning technique amenable to GPUs may lead to a suboptimal performance on accelerators, such as FPGAs and ASICs.

III. OMNI Overview

Fig. 2 gives an overview of our proposed framework OMNI, which consists of a software CNN model compression and an efficient sparse CNN hardware accelerator.

Memory partition is an effective hardware optimization technique to avoid excessive memory access conflicts, which divides the original on-chip memories into several banks to increase data access throughput. We devise the weight sparsity pattern relying on the memory partition patterns including block, cyclic, and hybrid as shown in Fig. 2. Given the original weight tensor, OMNI first partitions it into groups which are the basic units of pruning. In each group, each element is free to be pruned away. The advantages of utilizing these memory partition patterns are twofold. For one thing, element-wise pruning performs within each partition group. This fine-granularity approach enables high weight sparsity. For another, elements in the same group will be stored together in the same memory bank and each group has its independent memory port, which omits data access conflicts when concurrently accessing different groups. As a result, OMNI leads to a desirable compromise between sparsity and performance. In Section IV, we present the details of how memory partition patterns are applied to weights along with the pruning process.

From the hardware's aspect, OMNI proposes an architecture which directly computes results with the weights stored a sparse format. In our architecture, each PE is responsible for all the computation belonging to the same partition group. PE continuously conducts element-matrix multiplication, where the element (nonzero) is from weight tensor and the matrix is a subinput tile. This computation dataflow can maximize weight reuse and reduce the weight decoding overhead. We propose a sparse format to encode the weights on the hardware. In this format, we stagger weight elements from different groups. When the weights are loaded on-chip, elements of the same group will be reassembled with no extra effort. Furthermore, we optimize the on-chip memory using memory partition technique to cooperate with sparsity pattern. In Section V, we introduce the architecture components and memory optimization. We also employ a design space exploration to explore various parameters.

IV. Sparsity Pattern Pruning

In this section, we first propose three sparse patterns inspired by hardware memory partition, which divide the weights into several *groups*. Then, we present our pruning method that prunes every group to the same sparsity.

A. Sparsity Patterns

Memory partition is a common memory optimization technique, where data elements are partitioned into different groups. Memory partition allocates the array elements into multiple banks to reduce the access conflicts in parallel computation [36]–[38]. In our pruning process, we form the weight sparsity pattern using three memory partition patterns: 1) block; 2) cyclic; and 3) hybrid patterns. For block and cyclic patterns, consecutive elements, and interleaved elements are partitioned into the same group, respectively. We can also obtain a hybrid pattern by combining block and cyclic patterns for different weight tensor dimensions.

We assume that the weights are initially stored as a 4-D tensor F with size $N \times M \times kx \times ky$. Then we use (d_1, d_2, d_3, d_4) to represent the indices of 4-D tensor. A partition on F with partition factor P is to divide F into P groups $G_i, 0 < i < P$

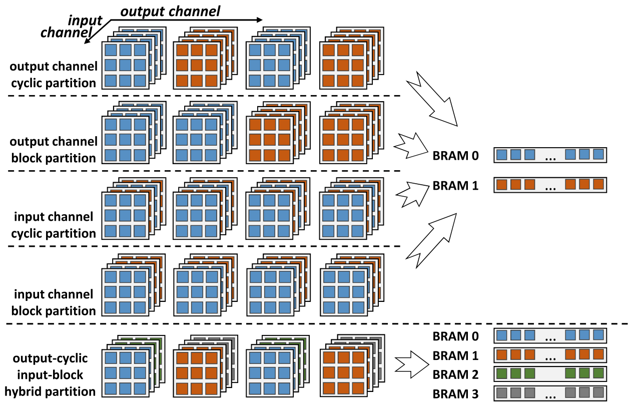

input channel
output channel
output channel cyclic partition
output channel block partition
input channel cyclic partition
input channel block partition
output-cyclic input-block hybrid partition
BRAM 0
BRAM 1
BRAM 2
BRAM 3
...

Fig. 3. Weight partition along output/input channel with $P_N = P_M = 2$.

$$F = \bigcup_{i=1}^{p} G_i \tag{3}$$

$$G_i \cap G_j = \emptyset \quad \forall i \neq j \tag{4}$$

1) *Block Pattern:* It partitions F along the xth dimension with partition factor P

$$G^i_{blk} = \{F_{d_1,\ldots,d_x,\ldots,d_4} | \lfloor d_x \ / \ (D_x \ / \ P) \rfloor = i\}. \tag{5}$$

2) *Cyclic Pattern:* It partitions F along the xth dimension with partition factor P

$$G^i_{\text{cyc}} = \{F_{d_1,\ldots,d_x,\ldots,d_4} | \ d_x \bmod P = i\}. \tag{6}$$

3) *Hybrid Pattern:* It partitions F using block pattern along the xth dimension with factor P_1 and using cyclic pattern along the yth dimension with factor P_2

$$G^{i,j}_{hbd} = \{F_{d_1,\ldots,d_x,\ldots,d_y,\ldots,d_4} | \lfloor d_x \ / \ (D_x \ / \ P_1) \rfloor = i \quad d_y \bmod P_2 = j\}. \tag{7}$$

Based on the partition definition above, the first step of our pruning method is to divide the weights into multiple *groups*. We can partition the 4-D weight tensor along any dimension (input channel, output channel, height, and width). However, in practice, as the size of a weight kernel is small (from 1×1 to 7×7, typically 3×3), there is little partition space within the dimension of the weight kernels' height and width. So we only consider the partition along weight's output channel and input channel dimensions. Hybrid partition is also supported in our pruning where both output channel and input channel are partitioned. In this case, the group index is a pair, where each index can be computed according to the dimension's partition method.

Fig. 3 gives a partition example on a weight tensor with $4 \times 4 \times 3 \times 3$ size and partition factor $P_M = P_N = 2$, where P_N and P_M are the partition factors along output and input channel, respectively. The weights with the same color are partitioned into the same group. For example, when we conduct block partition along output channel, weights in output channels 0 and 1 are partitioned into group 0. And weights in

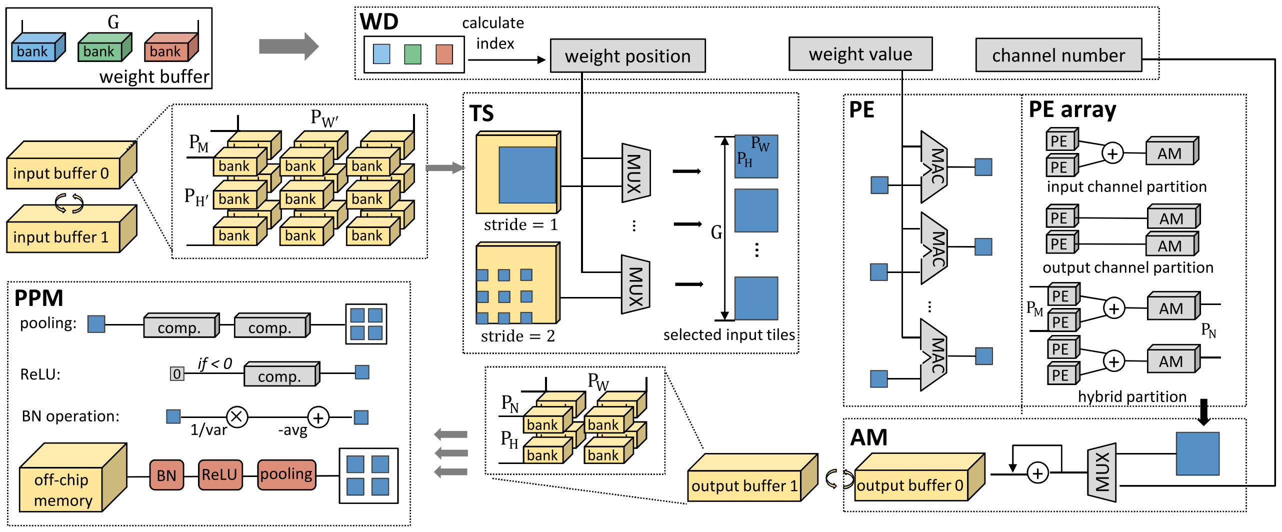


Fig. 4. Architecture overview.

output channels 2 and 3 are partitioned into group 1. For cyclic partition, output channels 0 and 2 are partitioned into group 0. While 1 and 3 are in group 1. We can also form hybrid partition which combines cyclic partition along output channel and block partition along the input channel. As we conduct partition along two dimensions simultaneously with $P_N = P_M = 2$ in each dimension, we would get $P = P_N \times P_M = 4$ groups. Weights of output channels 0, 2 and input channels 0, 1 are in group 0 while weights of output channels 1, 3 and input channels 2, 3 are in group 3.

B. Unified Sparsity Pruning

After partitioning the weights into groups, different groups will be issued to different hardware PEs. This will cause load imbalance problem if different groups have a different number of nonzeros. Hence, to obtain a balanced workload among the groups, we propose *unified sparsity pruning*.

When pruning the weight tensors, we need to set a threshold, where the elements below the threshold will be set to 0. Instead of sharing the same threshold across all the groups, we share the same sparsity target in all groups. Thus, different groups may set their own thresholds to obtain the target sparsity. We set the target sparsity of all groups to r. If the pruning sparsity increases too rapidly, the network would hardly adapt to the new pruning sparsity, and fails to recover the accuracy. However, if the pruning sparsity increases too slowly, it will take more iterations to achieve high sparsity and retrain the network, which is time-consuming. To compromise the trade-off, we employ a multistep pruning method which dynamically adjusts the pruning sparsity as well as its adjustment speed as follows.

Step 1: Set the initial pruning sparsity r and pruning sparsity adjustment speed v, minimum adjustment speed $v_{\min}$ and stage iteration iter_s.

Step 2: Prune the groups with pruning sparsity r, then retrain the network. Adjust the pruning sparsity $r = r + v$.

Step 3: Add the iteration counter by 1. If the counter is a multiple of iter_s, then we switch to the next stage by decaying the pruning sparsity adjustment speed $v = v/2$. If $v < v_{\min}$, then $v = v_{\min}$.

Step 4: Goto step 2.

V. Accelerator Architecture

In this section, we introduce the sparse CNN accelerator architecture. First, we present the architecture overview. Then, we introduce a sparse format that can efficiently store the pruned weights meanwhile enable high data parallelism. Then we show the computation dataflow, PE design, and other hardware modules. Finally, we present the memory optimization which cooperates with the sparsity patterns.

A. Architecture Overview

Fig. 4 depicts the architecture overview of OMNI. The dataflow of the architecture employs multiple element-matrix multiplication operations using PE array, where the element refers to the sparse weight and the matrix refers to an input tile from the input feature maps. The architecture is composed of five modules: 1) weight decoder (WD); 2) tile selector (TS); 3) PE; 4) accumulator module (AM); and 5) post-processing module (PPM).

The entire computation dataflow consists of five steps.

Step 1: The compressed weights from weight buffer are decoded by the WD and then sent to PE array. The WD continuously reads multiple compressed weights in the sparse format and computes the indices which will be sent to other modules as shown in Fig. 4.

Step 2: The TS is used to select tiles for the computation. Specifically, given a region of input pixels, the TS selects the necessary input pixels according to the weight indices from the WD.

Step 3: Each PE performs element-matrix multiplication, all these multiplications are performed in parallel. And there

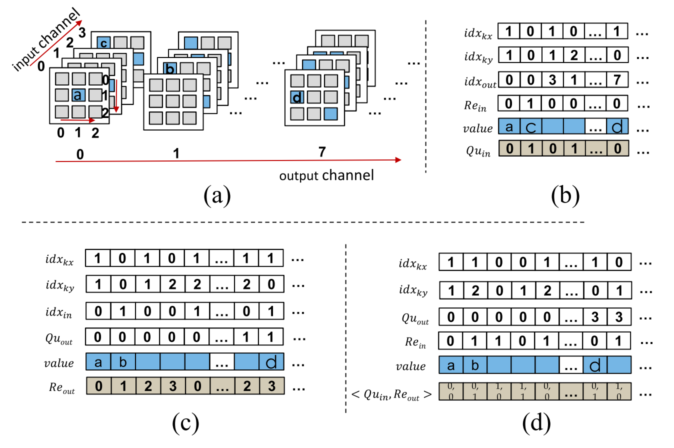


Fig. 5. Proposed sparse format. (a) Sparse 4-D weight tensor. (b) Block partition with factor = 2 in input channel dimension. (c) Cyclic partition with factor = 4 in output channel dimension. (d) Hybrid partition with factor = 2 in both channel dimensions.

TABLE I

SPARSE FORMAT AND DATA TYPE: (a) IS AN EXAMPLE OF BLOCK SPARSITY PATTERN; (b) IS AN EXAMPLE OF CYCLIC SPARSITY PATTERN; AND (c) IS AN EXAMPLE OF HYBRID SPARSITY PATTERN

(a) block pattern	tuples	idx_{kx}	idx_{ky}	idx_{out}	Re_{in}	value
	bit width	4	4	10	10	16
(b) cyclic pattern	tuples	idx_{kx}	idx_{ky}	idx_{in}	Qu_{out}	value
	bit width	4	4	10	10	16
(c) hybrid pattern	tuples	idx_{kx}	idx_{ky}	Re_{in}	Qu_{out}	value
	bit width	4	4	10	10	16

are multiple PEs processing the sparse weights from different channels concurrently.

Step 4: The AM receives the output channel index from the WD, then accumulates the results from PEs to the associated output buffer.

Step 5: The PPM is responsible for other common operations in CNNs, such as rectified linear unit (ReLU) and pooling functions.

B. Sparse Format

OMNI prunes the weights in block, cyclic, and hybrid patterns. We use (idx_{kx}, idx_{ky}, idx_{in}, idx_{out}) to represent the index of the 4-D weight. Specifically, idx_{kx} and idx_{ky} are the spatial indices. idx_{in} and idx_{out} are the indices in the input channel dimension and output channel dimension, respectively. We propose a sparse format to help the hardware to compute the results of the weights from different groups in parallel. Table I shows the format for different sparse patterns. First, we use a pair (Qu, Re) to represent the quotient and remainder of dividing the index idx by the partition factor P, where

$$idx = Qu \times P + Re. \tag{8}$$

For example, the output channel is partitioned with the factor 4, then the output channel index 7 can be represented by the pair $(Qu_{\text{out}}, Re_{\text{out}}) = (1, 3)$. In our format, we store the weights from different groups continuously. We use the 5-tuple format to encode each associated sparse format as shown in Table I. To be more specific, Table I(a) shows the sparse format of the block pattern with respect to the dimension of the

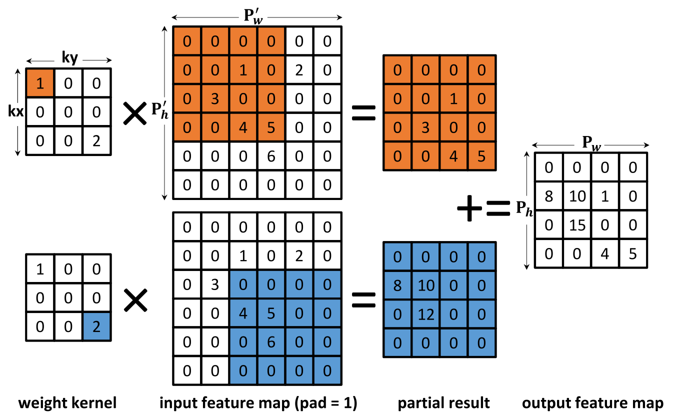


Fig. 6. OMNI computation dataflow.

input channel. The weights are stored in ascending order of Qu as shown in Fig. 5(b). As for the cyclic pattern, we store the weights in ascending order of Re as shown in Fig. 5(c). Furthermore, our sparse format also supports a hybrid pattern. Table I(c) is an example of a hybrid pattern where input and output channels are in block and cyclic patterns, respectively. In this case, the weights are sorted by the pair (Re;Qu) or (Qu;Re). As shown in Fig. 5(d), the weights are sorted first by output channel dimension (Re_{out}) then by input channel dimension (Qu_{in}).

Fig. 5(b) shows an example of block sparse pattern in input channel dimension with partition factor 2. The element c is encoded as $<0, 0, 0, 1, c>$ whose original input channel index is 3. Fig. 5(c) shows an example of cyclic pattern where the partition factor is 4. The element d is encoded as $<1, 0, 1, 1, d>$. Similarly, the output channel index is 7. Fig. 5(d) shows an example of hybrid pattern, where the input channel is block pattern and output channel is cyclic pattern.

C. Dataflow and PE design

Computation Dataflow: The PE operations in prior sparse CNN implementation [6], [13], [16] are based on vector-matrix and vector–vector matrix multiplications. However, these operations need to regather the sparse weights into a new vector or matrix, resulting in the overhead of matching the index between the vector and the matrix. OMNI solves this problem by featuring an element-matrix multiplication dataflow for the PE operation. The advantage of this operation is that it can maximize the weight reuse so that the decoding overhead can be effectively reduced. Besides, OMNI parallelizes the operation in channel dimension, which can eliminate the data dependency problem. In this dataflow, the element refers to the sparse weight, and the matrix refers to an input tile from the TS. To generate a $P_H \times P_W$ output tile, each weight will multiply with a $P_H \times P_W$ region in the input tile from the input feature map. The selection of this region is determined by the position of the weight. For example, the red weight in Fig. 6 locates in the position (2, 2), which means the weight corresponds to the bottom-right tile. Finally, the results of the same output channel need to be accumulated together.

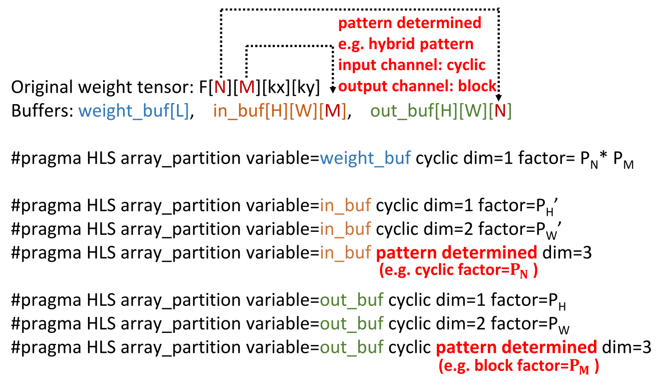
PE Design: Weight reuse is a crucial factor in the sparse CNN implementation because each weight needs to be decoded before being sent to the PE. Increasing weight reuse can effectively reduce the decoding overhead. To simplify the discussion, we assume $S = 1$ in 1. The total number of operations in one convolutional layer is $H \times W \times N \times M \times kx \times ky$ and the total number of weights is $N \times M \times kx \times ky$, which means the weight reuse arises from the parallelization in spatial dimension of output feature maps. Therefore, we parallelize the computation in both row dimension and column dimension of feature maps. That is, computing $P_H \times P_W$ output pixels in parallel. Besides, to avoid redundant computation, we constrain that the output tile size is within the minimum feature map size in the network.

In the convolution operation, the data dependencies between weights and pixels are quite complicated with respect to the spatial dimension. A complex data dependency will lower the data access efficiency when the PE is pipelined. However, the channel dimensions do not exhibit any dependency. Therefore, we initiate the PE array with the weight in each PE from different channels. Fig. 4 (PE array) shows different PE architectures which is determined by the weight sparsity pattern. There are $P_N \times P_M$ homogeneous PEs working in parallel. Different partition patterns lead to different topology of the interconnection between PEs. For the groups that are partitioned from the input channel dimension, the results from PEs are accumulated together via an adder tree. If the groups are partitioned from the output channel dimension, the results are stored independently which shows a lower PE latency compared with partitioning in the input channel dimension. However, partitioning in the output channel leads to more fanouts of the PE which can increase the requirement for multiplexers. Moreover, the partition factor determines the pipeline length of the PE. A large partition factor can result in a short pipeline length which may fail to overlap the latency of different modules.

D. Other Hardware Modules

Weight Decoder: The WD is a dedicated component for decoding weights stored in our sparse format. Each time, the WD reads continuous weights from weight buffer and then calculates the input channel and output channel indices according to (8). Once all indices are obtained, the WD sends all information to other modules as Fig. 4 shows. One destination is the TS which uses positions of each weight element to determine the input tile region. The other destination is the AM which uses the output channel index to locate where the results are accumulated to.

Tile Selector: The TS module first receives the decoded indices of weights from the WD. Then the TS selects P_M input tiles with size $P_H \times P_W$ from input buffer according to the indices. Clearly, the TS module is a static look-up table. Given a kernel size $kx \times ky$ and an input tile $P_{H'} \times P_{W'}$, the TS will generate all possible $P_H \times P_W$ regions and store them separately in the look-up table. Therefore, we can directly get the necessary input pixels in the look-up table which has been prefetched as long as the input tile is determined. The advantage of the static look-up table is to replace runtime index matching with a simple array indexing operation resulting in a low decoding latency. This also helps to save the



```
Original weight tensor: F[N][M][kx][ky]
Buffers: weight_buf[L],   in_buf[H][W][M],   out_buf[H][W][N]

#pragma HLS array_partition variable=weight_buf cyclic dim=1 factor= P_N* P_M

#pragma HLS array_partition variable=in_buf cyclic dim=1 factor=P_H'
#pragma HLS array_partition variable=in_buf cyclic dim=2 factor=P_W'
#pragma HLS array_partition variable=in_buf pattern determined dim=3
                                            (e.g. cyclic factor=P_N )

#pragma HLS array_partition variable=out_buf cyclic dim=1 factor=P_H
#pragma HLS array_partition variable=out_buf cyclic dim=2 factor=P_W
#pragma HLS array_partition variable=out_buf cyclic pattern determined dim=3
                                                   (e.g. block factor=P_M )
```

Fig. 7. Pseudo code of memory partition on buffers.

logic resources significantly since the runtime index matching requires much more multiplexers. Besides, the TS can handle different kernel size and kernel sliding stride as shown in Fig. 4 (TS). The input tiles are broadcast to the PE array to perform element-matrix multiplication.

Accumulator Module: The AM is used to accumulate the partial results generated by PEs to the final output. Each time, the PE array generates P_N output tiles with size $P_H \times P_W$. These tiles belong to distinct output channels. AM receives the output channel index to locate the positions of the output tiles in the output buffer. As shown in Fig. 4 (AM), a multiplexer is inserted for each output buffer bank. According to the output channel index, the multiplexer will select the corresponding address that the result needs to be accumulated to.

Post-Processing Module: After all weights are computed, AM stops updating the output buffer. Then PPM reads output pixels from the output buffer and performs post-processing. There are three types of operations, as illustrated in Fig. 4 (PPM): 1) pooling operation which outputs the maximum values in subregions of input feature maps; 2) ReLU which sets any input value less than zero to zero; and 3) batch normalization which normalizes the output results to improve the stability of artificial neural networks. The pooling module and the ReLU module are implemented by introducing comparison operators to the output buffers. The batch normalization operation normalizes the pixels in the output feature maps.

E. Memory Optimization

In OMNI architecture, we store the input and output buffers as a 3-D array, as shown in Fig. 4, which include dimension row, column, and channels. The weights are stored as a vector in the sparse format. Fig. 7 gives a pseudo code example, where input feature map buffer (*in_buf*), output feature map buffer (*out_buf*), and weight buffer (*weight_buf*) use different partition strategies.

OMNI prunes the weights with partition patterns in channel dimension. To leverage on the sparse pattern, the input buffer, and output buffer need to be partitioned with the same pattern. Specifically, the original 4-D weight tensor shares the same dimension with the output buffer which is output channel dimension. Each time the PE array will send the results of different output channels to the AM. In order to accumulate the results to the output buffer simultaneously, the output buffer needs to be partitioned with the same pattern as the weights. Similarly, the input buffer also needs to be partitioned in the channel dimension if the weights show a partition pattern in input channel dimension. These partition patterns are easily implemented using modern High-Level Synthesis tools [39], [40]. Fig. 7 shows an example of memory partition technique in Xilinx HLS tool [39], where we only need a partition *pragma* after the buffer statement.

In addition, we also partition the on-chip buffers in the spatial dimension to increase the data access throughput between hardware modules. As mentioned in Section V-B, the weights are compressed into a vector where the weights from different groups are continuous. Therefore, we apply cyclic partition to the vector so that the weights from different groups can be assigned to the PE array simultaneously. Besides, to lower the latency of the TS module, we apply cyclic partition to the input buffer with the factor $P_{H'}$ horizontally and $P_{W'}$ vertically as shown in Fig. 4. We also partition the output buffer with the factor P_H horizontally and P_W vertically as shown in Fig. 4.

F. Design Space Exploration

In this section, we develop an analytic model that can predict resource utilization. The purpose of our model is to generate the optimal partition factors to maximize the throughput under a specific area constraint. Then the software part of OMNI will prune the weights with these partition factors into target pattern. Here, we focus on three resources that are multipliers, multiplexers, and banks. There are $P_N \times P_M$ PEs with each PE performs $P_H \times P_W$ multiplications in parallel, therefore the number of multipliers in the PE is $P_H \times P_W \times P_N \times P_M$. Besides, each weight requires a multiplier to compute the address in the WD module (WDM). In total, the number of multipliers can be computed as follows:

$$\text{\# of MUL} = P_H \times P_W \times P_N \times P_M + P_N \times P_M. \tag{9}$$

In OMNI accelerator, the on-chip memory is partitioned into multiple dual-port banks. The bank for input and output buffer is a 64×16 bit SRAM. And the weight buffer is a 512×64 bit SRAM. According to our memory design in Section V-E, the weight buffer is partitioned into $P_N \times P_M$ banks. And the input buffer is partitioned into $P_{H'} \times P_{W'} \times P_M$ banks, and the output buffer is partitioned into $P_H \times P_W \times P_N$ banks. Considering the double buffer design, the total number of banks is

$$\begin{aligned}\text{\# of BANK} = {} & P_M \times P_N \\ & + 2 \times P_{H'} \times P_{W'} \times P_M \\ & + 2 \times P_H \times P_W \times P_N.\end{aligned} \tag{10}$$

In the AM and TS module, multiplexers are required to locate the address of input or output buffers according to the index from the WD. And the scale of multiplexer depends on the number of input signals (P_M, P_N). We apply 2-to-1 multiplexer as the basic unit to construct the multiplexer with more input wires. We assume the resource of the multiplexer with P_M and P_N input signals is $\beta(P_M)$ and $\beta(P_N)$, respectively. In total, the resource for multiplexers is

$$\begin{aligned}\text{\# of MUX} = {} & 2 \times P_{H'} \times P_{W'} \times \beta(P_M) \\ & + 2 \times P_H \times P_W \times \beta(P_N)\end{aligned} \tag{11}$$

VI. Experiment

A. Experimental Setup

Benchmarks: We evaluate OMNI with four state-of-the-art networks, including Alexnet, VGG16, Resnet152, and YOLO. The main component of Resnet152 is residual model [26] which consists of five convolutional layers, five batch normalization layers, and two ReLU layers. YOLO is a CNN model for real-time object detection system [46]. We use Tiny-YOLO version to evaluate our design. Tiny-YOLO consists of nine convolutional layers and six max pooling layers. We implement our pruning technique using Caffe framework [47].

Platforms: For ASIC accelerator, we first use the Catapult HLS tool [48] to generate RTL code from C/C++-based CNN designs. In the following, the Synopsys Design Compiler [49] is used to perform the placement and routing with Synopsys IC compiler under the SMIC 65-nm technology. The synthesized frequency is 400 MHz. In addition to ASIC, we also prototype OMNI on two FPGA platforms which are Xilinx ZC706 and ZCU102. The Xilinx ZC706 platform consists of a Kintex-7 FPGA and dual ARM Cortex-A9 processors. The external memory is 1 GB DDR3. Our FPGA implementation is operated at 166 MHz frequency on this platform. Xilinx ZCU102 consists of an UltraScale FPGA, quad ARM Cortex-A53 processors, 500 MB DDR3. Our FPGA implementations is operated at 200 MHz frequency on this platform. To measure the runtime power, we plugged a power meter in the FPGA platform. We use Xilinx Vivado HLS (v2017.4) and Xilinx SDSoC (v2017.4) for implementation. We also compare the performance and energy efficiency of our design with Titan GPU platform.

In the following, we first examine the accuracy loss under different partition strategies. We then present the performance comparison in Section VI-C. In Section VI-D, we discuss the detailed energy, area characteristics, and scalability. We further test the sensitivity to different sparsity in Section VI-E. We also implement OMNI on a FPGA platform in Section VI-F and compare with GPU platform in Section VI-G.

B. Pruning Result

We compare our pruning method with an unstructured pruning technique Deep Compression [27] and a state-of-the-art structured pruning technique Cambricon-S [4]. In summary, OMNI is able to prune over 90%, 88%, 75%, and 80% convolution parameters for AlexNet, VGG16, Resnet152, and Tiny-YOLO, respectively. We conduct pruning using various partition patterns and factors as shown in Table II.

As Deep Compression and Cambricon-S focus on pruning the FC layers, they cannot achieve high sparsity for convolutional layers. In AlexNet and VGG16, Deep Compression prunes about 70% convolution parameters, and Cambricon-S is only able to prune less than 65% parameters. Our pattern-aware pruning method prunes more than 90% convolution parameters in AlexNet and 88% in VGG16. In Resnet152, the achieved sparsity is significantly reduced compared with AlexNet and VGG16. Deep Compression and Cambricon-S only prunes about 45% convolution parameters out. But our pruning method achieves higher sparsity around 75% since our

TABLE II

PRUNING RESULT. E DENOTES THE TOP-1 ERROR RATE (IN PERCENTAGE) FOR ALEXNET, VGG16, AND RESNET152, WHILE FOR TINY-YOLO, E DENOTES THE VALUE OF $1 - mAP$. S DENOTES THE SPARSITY OF ALL CONVOLUTION LAYERS IN THE CORRESPONDING NETWORK. $\text{block}_{\text{in\_ch}}4$ DENOTES THE BLOCK PATTERN PRUNING ALONG INPUT CHANNEL WITH FOUR GROUPS ($P_M = 4$), AND $\text{cyclic}_{\text{out\_ch}}16$ DENOTES THE CYCLIC PATTERN PRUNING ALONG INPUT CHANNEL WITH 16 GROUPS. hybrid 4×4 DENOTES HYBRID PATTERN PRUNING (BLOCK ALONG INPUT CHANNEL, CYCLIC ALONG OUTPUT CHANNEL) WITH 16 GROUPS ($P_N = P_M = 4$). ALL THE ALEXNET, VGG16, AND RESNET152 ARE TRAINED ON THE IMAGENET DATASET [44]. TINY-YOLO IS TRAINED ON PASCAL VOC 2007 AND 2012 DATASET [45]

model	Ref E(%)	Deep Cmp.[27]		Cambricon-S[4]		Ours									
						$block_{in\_ch}$ 4		$block_{out\_ch}$ 16		$cyclic_{in\_ch}$ 4		$cyclic_{out\_ch}$ 16		$hybrid$ 4×4	
		S(%)	E(%)	S(%)	E(%)	S(%)	E(%)	S(%)	E(%)	S(%)	E(%)	S(%)	E(%)	S(%)	E(%)
Alexnet	42.78	70.12	42.78	64.75	42.72	90.03	42.52	90.11	42.80	90.77	42.70	90.21	42.77	90.17	42.75
VGG16	31.50	67.24	31.17	64.83	31.33	88.70	31.10	88.75	31.36	89.53	31.22	88.28	31.70	88.90	31.72
Resnet152	25.00	45.00	24.40	45.69	25.05	75.97	24.80	75.63	24.70	76.11	25.03	75.10	25.10	75.00	24.88
Tiny-YOLO	47.3	-	-	-	-	80.30	47.5	80.74	47.3	80.54	47.0	80.92	47.3	80.50	47.2

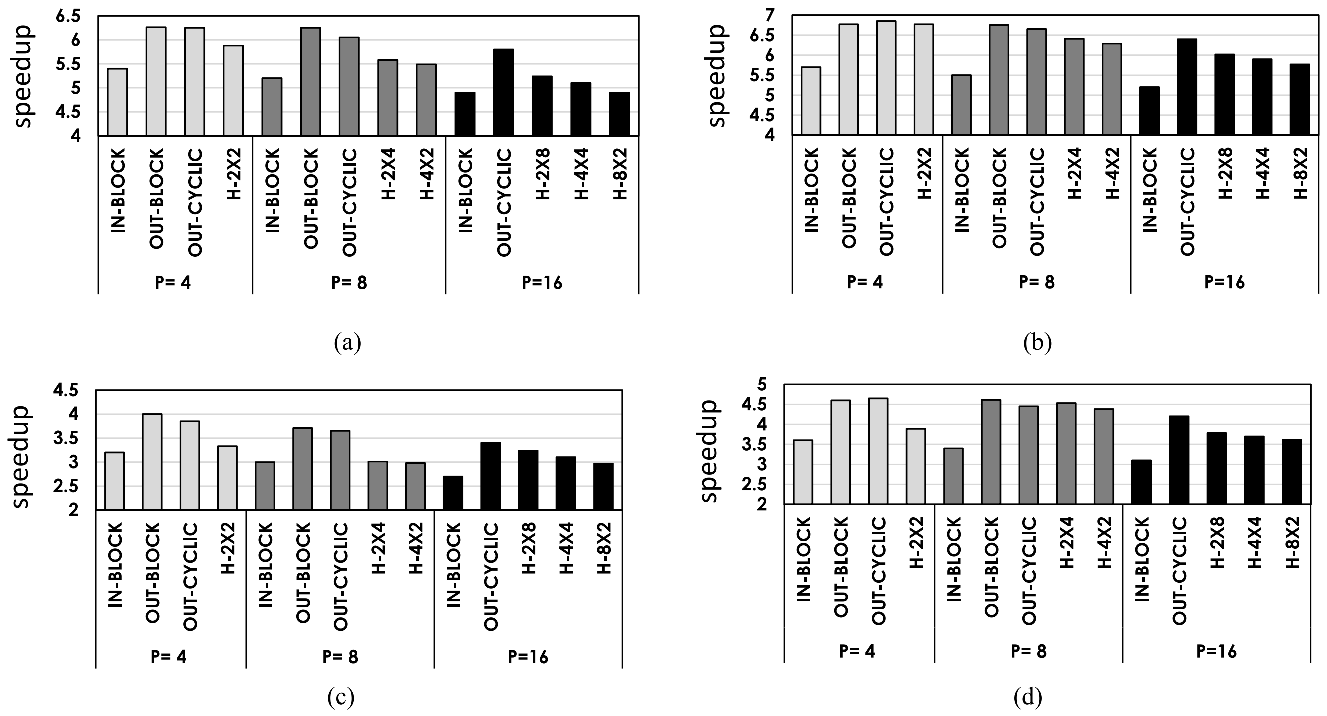


Fig. 8. Speedup with different partitions on modern CNNs. (a) Speedup with different partitions on AlexNet. (b) Speedup with different partitions on VGG. (c) Speedup with different partitions on Resnet152. (d) Speedup with different partitions on Tiny-YOLO.

sparsity pattern is less strict in mathematical formalization. This phenomenon is due to the fact that Resnet has batch normalization layer and residual connection, which greatly reduces over-fitting [4], [28]. Besides, Resnet152 has more complex network typologies and deeper structure. We also apply our pruning method to Tiny-YOLO and achieve sparsity above 80%.

We also compare the accuracy results of our pruning method with Deep Compression and Cambricon-S. As shown in Table II, compared with original dense networks our pattern-aware pruning achieves negligible accuracy loss (e.g., less than 0.1% for AlexNet with block pattern in output channel dimension, 0.22% for VGG16 with hybrid pattern) or even slight accuracy gain (e.g., 0.4% for VGG16 with block pattern in input channel dimension, 0.3% for Resnet152 in hybrid pattern) as shown in Table II. The achieved accuracies of Deep Compression, Cambricon-S, and OMNI are quite close as well (e.g., no more than 0.1% difference in AlexNet). Pattern-aware pruning on Tiny-YOLO also achieves no accuracy loss in all the three settings. From Table II, we observe that different sparse patterns and partition factors achieve consistent high sparsity and accuracy.

C. Performance Comparison

In this section, we show the speedup of OMNI by comparing to the ideal dense CNN accelerator which shares the same resources as OMNI. We first present the effect of different weight sparsity patterns and partition factors in Fig. 8. We vary the partition factor from 4 to 16. In Fig. 8, H-$a \times b$ means the input channel is partitioned with factor a and output channel is partitioned with b. We find that OMNI shows a stable speedup as the parallelism increases from 4 to 16, this benefit comes from two aspects: 1) the sparsity of different partitions is almost the same and 2) our pruning technique balances the workload for the hardware. In Fig. 8, we find that the pruning method applying output channel partition always offers the

TABLE III
OMNI AREA AND POWER BREAKDOWN

OMNI Component	Area(mm^2)	Power (mW)
Combinational logic	0.432	38.56
Buf/inverter	0.036	20.33
Memory	0.059	423.14
other	0.068	5.04
Total	0.595	487.05

highest speedup. Because these networks have a small number of input channels in the first few layers, which is less than parallelism degree in the PE array. Besides, parallelizing input channel dimension requires an adder in the PE which increases the PE process latency.

On average, OMNI outperforms the baseline across convolutional layers of Alexnet, VGG16, Resnet152, and YOLO, achieving up to 5.8×, 6.2×, 3.4×, and 4.2× speedup, respectively. In our design, we effectively eliminate the useless multiplications in CNN models meanwhile keep a balanced workload for each PE. Though Alexnet achieves higher sparsity, the speedup on VGG shows a better result than on Alexnet. This difference is caused by the layer diversity in the Alexnet, since Alexnet involves different kernel size (e.g., 11 × 11, 5 × 5, 3 × 3). In addition, the sparsity on Alexnet is high, which can increase the proportion of the initial time on the circuit. VGG16 network and tiny-YOLO show similar sparsity because their network typologies are close to each other and they share the same kernel size. The speedup on Resnet is relatively small, which mainly comes from two aspects. First, the feature map size in the CNN layers might not divide P_H and P_W evenly, leading to redundant computations in the boundary of feature maps. Second, in our architecture, we apply pipeline technique in PEs. When the workload is small after pruning, the latency of PEs can be bounded by the length of pipeline. For example, Resnets consist of many convolutional layers with 1 × 1 kernels of which workloads are relatively small.

D. Energy and Area Characteristics

Energy and Area Breakdown: We first select the partition factors as $P_H = P_W = 4$ and $P_N = 4$, resulting in a relatively small scale OMNI. Table III shows the OMNI area and power breakdown with the configuration. The total area is 0.595 mm^2 of which the hardware modules occupies 72% area. The total power is 487.05 mW and the memory access consumes 86.8% energy. This is because, in sparse CNNs, the computation decreases dramatically, requiring a high communication-to-computation ratio.

Hardware Overhead: In Table IV, we compare the extra hardware resource utilization for processing structured sparsity pattern in weights between OMNI and previous accelerators [4], [13], [35]. To make a fair comparison, we estimate the proportion of the extra hardware area in a single PE. The inference process of CirCNN [35] consists of three steps which are input transformation of the frequency domain, complex multiplication, and inverse transformation to the time domain. The transformation operation is an extra operation to support the circulant weight pattern. We implement the FFT-based convolution operation with the same technology as OMNI and find that the area of transformation module accounts for 68% area of the whole PE. In PERMDNN [13], the weights are pruned with block-permuted diagonal matrix. To locate a weight, it needs to get the block position and the permutation order. We also implement the key operation of PERMDNN which is a matrix-vector multiplication operation and find that this operation accounts for 27% area. In Cambricon-S [4], the weights are gathered into dense blocks. We calculate the proportion of NSM and WDM in the PE circuit.[1] For OMNI accelerator, the extra operation is (8) which is to compute channel coordinate. The area of WDM is 0.074 mm^2 which only accounts for 17% of the whole combinational logic.

TABLE IV
SOFTWARE-ORIENTED PRUNING METHODS REQUIRE ADDITIONAL HARDWARE SUPPORT. THE TARGET CNN MODEL IS VGG WHICH IS SUPPORTED BY ALL FOUR ACCELERATORS. THE IMAGE RESOLUTION IS 224 × 224 × 3

Pruning technique	Weight pattern	Extra operation	Area overhead
CIRCNN[35]	circulant matrix	FFT transformation	68 %
PERMDNN[13]	permuted diagonal matrix	matrix permutation	27%
Cambricon-S[4]	gathered clusters	Block coordinate computation	41%
OMNI	cyclic/block based partition	Group ID computation	17%

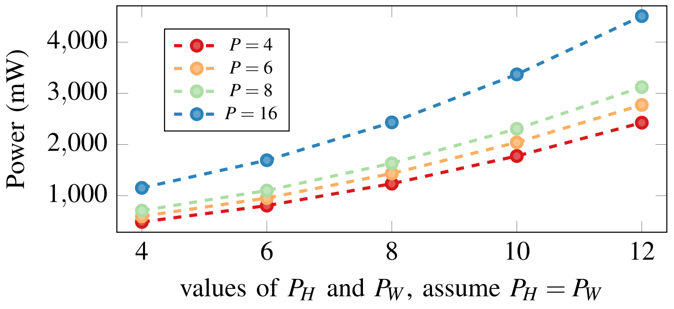


Fig. 9. Power characteristics with different configuration parameters.

Scalability: OMNI accelerator can be scaled across a wide range with parameter $\{P_H, P_W, P_M, P_N\}$. P_H and P_W determine the number of multipliers in the PE and the local buffer size of the TS module. $P_M \times P_N$ is the PE number. P_M and P_N also represent the number of input signals of the multiplexer in the TS module and the AM module, respectively. In our evaluation, we increase $P_H \times P_W$ from 4 × 4 to 12 × 12, because the feature map size is small as the network goes deeper. And we set the upper bound of $P = P_M \times P_N$ to 16 so that the network can be pruned without accuracy loss.

Power: Fig. 9 compares the power consumption of different configurations. In this figure, the power usage is nearly proportional to $P_W \times P_H$ when the P is fixed. This is because the power usage of combinational logic mainly comes from the on-chip multipliers, and the memory access power mainly depends on the input/output tile buffer size and the number of ports on input/output RAMs which are equal to $P_H \times P_W$.

[1]The data has been reported in this article [4].

TABLE V
PERFORMANCE COMPARISON ON VARIOUS PLATFORMS. -D MEANS DENSE CNN MODEL. -S MEANS SPARSE CNN MODEL

	FPGA(classification)				**FPGA**(detection)			**GPU**		**ASIC**
Implementation	[50]	[51]	OMNI	OMNI	DNNDK[52]	OMNI	OMNI	cuDNN	cuSparse	OMNI
CNN	Dirac-DeltaNet	Alexnet-s	VGG-s	VGG-s	Yolo-v3	Yolo-s	Yolo-s	VGG-d	VGG-s	VGG-s
Precision	1-4bit	16bit	16bit	16bit	8bit	16bit	16bit	FP32	FP32	16bit
Top-1 Acc	68.5%	68.5%	68.6%	68.6%	50.9%	52.7%	52.7%	68.5%	68.5%	68.5%
Device	Zynq ZU3EG	Zynq ZC706	Zynq ZC706	Zynq ZCU102	Zynq ZCU102	Zynq ZC706	Zynq ZCU102	TitanX	TitanX	-
Technology	16nm	28nm	28nm	16nm	16nm	16nm	28nm	28nm	28nm	65nm
Frequency(MHz)	250	-	166	200	333	200	166	1075	1075	400
Performance (GOP/s)	47.1	71.2	43.7	127.1	822.2	41.5	125.3	2270	126	368
Power(W)	5.5	9.6	9.6	23.6	23.6	23.6	9.6	134	136	1.9
FPS	96.5	19.1	12.6	36.7	43.3	33.2	11.0	73.71	36.4	106.2

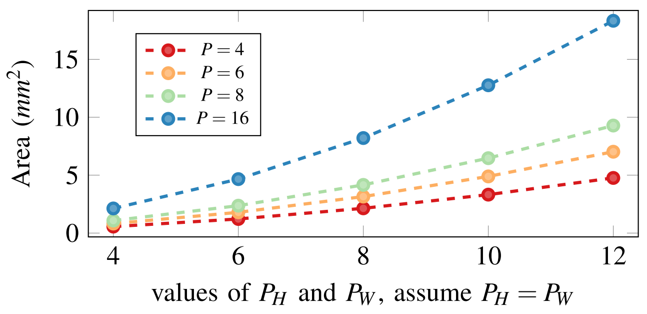


Fig. 10. Area characteristics with different configuration parameters.

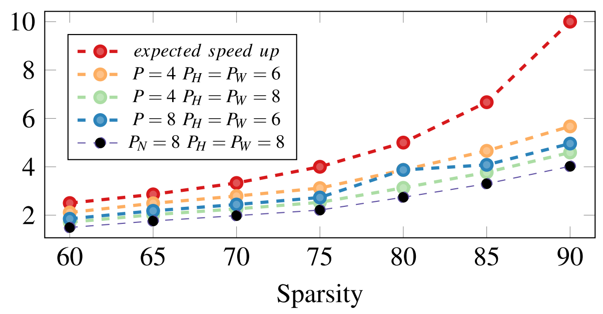


Fig. 11. Sensitivity of OMNI to the sparsity.

When the $P_W \times P_H$ is fixed, the power is nearly proportional to P. Fig. 10 shows the area comparison which has a similar trend like power characteristics.

E. Sensitivity to Sparsity

In this section, we set the sparsity ranging from 60% to 90% at the step of 5% to explore the sensitivity of OMNI to the weight sparsity. The sparsity is generated randomly with 3×3 kernel which is the most common kernel size, we only expect that each weight group has similar nonzero number. Fig. 11 depicts the sensitivity to the sparsity of OMNI. In the figure, we also draw the line of theoretical speed up. We find that when the sparsity is lower than 80%, OMNI can achieve almost ideal speedup. When the sparsity continues to increase, the speedup grows slowly. The reason is that the short pipeline length leads to the inefficiency of the PE array. Besides, the off-chip memory bandwidth can also limit the speedup when the sparsity is high.

F. Results on FPGAs

In this section, we evaluate OMNI on the Xilinx ZCU102 platform and ZC706 platform using VGG16 and Tiny-YOLO networks. In OMNI FPGA implementation, we set ($P_H = P_W = 8, P_N = 8$) for ZCU102 platform and ($P_H = P_W = 8, P_N = 4$) for ZC706 platform. According to Table V, our implementations achieve 43.7 GOP/s and 127.1 GOP/s for VGG16 network on ZC706 and ZCU102. The performance of Tiny-YOLO implementation is 41.5 GOP/s and 125.3 GOP/s on ZC706 and ZCU102, respectively.

We compare our FPGA implementation with three state-of-the-art design [50]–[52]. For classification task, the design in [50] target dense CNN model. Compared with [51], we achieve 2.7× GOP/s speedup. The design in [50] shows a higher frame-per-second (FPS). This is because DiracDeltaNet is much smaller than the compressed VGG which only uses 1×1 convolution kernel. And it applies 1–4 fixed point data type which further reduces the computation size. For object detection task, we compare our design with DNNDK 3.0 [52], which is a highly optimized framework for deploying CNN models on FPGA. DNNDK shows a higher GOP/s resulting from targeting dense models, lower precision, and higher frequency.

G. Comparison With GPU

In this section, we conduct a comparison with the GPU platform using VGG16. To make a fair comparison, we test the performance of TitanX with the latest cuDNN [53]. Power on GPU is obtained using NVIDIA profiling tools. Table V shows the comparison results. We set ($P_H = P_W = 6, P_N = 16$) for OMNI-ASIC design and ($P_H = P_W = 8, P_N = 8$) for OMNI-FPGA design. On GPUs, due to the indexing overhead and memory uncoalesing of the sparse version, the dense version of cuDNN [53] achieves better performance than the sparse version. As shown, TitanX gives better performance, but our implementation on FPGA and ASIC achieves 3.14× and 114.7× energy efficiency improvement.

VII. RELATED WORK

Unstructured Pruning Techniques and Accelerators: Recently, there are a few works that exploit sparsity in

CNN accelerator on hardware. The multiplications can be eliminated when the input feature maps or the weights is zero. Eyeriss [15] gated computation cycles for zeros in the input feature maps to save energy and the data is stored in compress format in DRAM. However, this gating operation cannot effectively reduce the latency. Besides, Eyeriss did not consider the sparsity in weights which contains more zeros. Han *et al.* [16] proposed EIE CNN accelerator which operated directly on compressed networks and enables the large neural network models to fit in on-chip SRAM. EIE exploits sparsity both in input feature maps and weights but only focused on the fully connected layer. In EIE design, multiple rows are computed together. EIE performs the sparse matrix and sparse vector multiplication operation by scanning input vector to find its next nonzero value and broadcasting nonzeros along with its index to all PEs. Parashar *et al.* [6] proposed SCNN accelerator with a novel dataflow named PT-IS-CP (planar-tiled input-stationary Cartesian-product). The core part of SCNN accelerator is the Cartesian-product operation in which all nonzeros have to be multiplied with one another. However, using Cartesian product can lead to additional operations to compute the corresponding positions in the feature maps. In SCNN, the input feature maps are partitioned into several tiles then distributed multiple PEs. Since different tiles can have different nonzeros, the workload in PEs can be different. Han *et al.* proposed ESE accelerator that enables sparse LSTM on hardware. In ESE accelerator, the sparse weights is pruned with each row sharing the similar number of nonzeros, so that the workload of each PE can be balanced. Zhang *et al.* [54] presented Cambricon-X accelerator which apply step indexing techniques. In Cambricon-X design, the nonzeros in the same row is divided into multiple segments with the same size in subsequent addresses. And the row that contains nonzeros less than the size will be aligned to the size. Albericio *et al.* [5] proposed zero-free neuron array Format to skip the multiplication for zeros in the input feature map.

Structured Pruning Techniques and Accelerators: Dense CNNs are typically over parameterized and exist redundant connections. Many pruning methods have been proposed to compress the weights with structured sparsity pattern. In [31]–[33], the weights are pruned in the granularity of channels, and an entire channel is removed together to keep the dense structure of weights. Yu *et al.* [14] proposed Scalpel pruning method which consists of SMID-aware weight pruning for CPUs and node pruning for GPUs. Deng *et al.* [13] proposed PERMDNN where the sparse weights are compressed into multiple permuted diagonal matrices, and PERMDNN only focused on fully connected layers. Zhou *et al.* [4] observed that local weights tended to gather into clusters. Based on this observation, they proposed a pruning method to reduce the irregularity of sparse weights where a window of weights would be set to zeros if it meets specific criteria. And they also developed an accelerator call Cambricon-S which exploit both input pixel sparsity and weight sparsity. Cambricon features a NSM to process the static sparsity with shared indexes and synapse selector module to process the input pixel sparsity. Ding *et al.* [35] proposed CirCNN which represented sparsity using circulant matrix which can be efficiently implemented in hardware.

VIII. Conclusion

In this work, we propose a framework OMNI which integrates the pruning technique with memory partition patterns for accelerating sparse CNN models. We start with the weight representation that is inherently supported on hardware which can be seamlessly integrated with software pruning. In the pruning process, we partition the weights into several groups in cyclic or block pattern and enforce that each group has the same number of nonzeros. As a result, our pruning method achieves sparsity of 90% in Alexnet, 88% in VGG16, 74% in Resnet152, and 80% in Tiny-YOLO. In hardware design, we present an efficient architecture where the memory partition strategy is matched with the partition patterns in sparse weights. We further develop a resource model to find the optimal memory partition factors, and the factors will determine how the weights are partitioned in the pruning. Experiments show that OMNI achieves 3.4×–6.2× speedup for the modern CNNs, over a comparably ideal dense CNN accelerator. OMNI shows 114.7× energy efficiency improvement compared with GPU platform. OMNI is also evaluated on Xilinx ZC706 and ZCU102 FPGA platforms, achieving 41.5 GOP/s and 125.3 GOP/s, respectively.

References

[1] 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. (ICASSP)*, 2017, pp. 4845–4849.

[2] J. Liu, G. Wang, P. Hu, L.-Y. Duan, and A. C. Kot, “Global context-aware attention LSTM networks for 3D action recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 7, 2017, p. 43.

[3] Z. Zheng, L. Zheng, and Y. Yang, “Unlabeled samples generated by GAN improve the person re-identification baseline *in vitro*,” in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 3774–3782.

[4] 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. Microarchit. (MICRO)*, 2018, pp. 15–28.

[5] 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. Architect. (ISCA)*, 2016, pp. 1–13.

[6] A. Parashar *et al.*, “SCNN: An accelerator for compressed-sparse convolutional neural networks,” in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Architect. (ISCA)*, 2017, pp. 27–40.

[7] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2017, pp. 45–54.

[8] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, “Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs,” in *Proc. 54th Annu. Design Autom. Conf. (DAC)*, 2017, p. 62.

[9] L. Lu, Y. Liang, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on FPGAs,” in *Proc. 25th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2017, pp. 101–108.

[10] L. Lu and Y. Liang, “SpWA: An efficient sparse winograd convolutional neural networks accelerator on FPGAs,” in *Proc. 55th Annu. Design Autom. Conf. (DAC)*, 2018, pp. 1–6.

[11] X. Wei *et al.*, “Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs,” in *Proc. 54th Annu. Design Autom. Conf. (DAC)*, 2017, pp. 1–6.

[12] S. Liu *et al.*, "CambriCON: An instruction set architecture for neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architect. (ISCA)*, 2016, pp. 393–405.

[13] C. Deng *et al.*, "PERMDNN: Efficient compressed DNN architecture with permuted diagonal matrices," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, 2018, pp. 189–202.

[14] J. Yu *et al.*, "SCALPEL: Customizing DNN pruning to the underlying hardware parallelism," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 548–560, 2017.

[15] Y. 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. Architect. (ISCA)*, 2016, pp. 367–379.

[16] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Architect. (ISCA)*, 2016, pp. 243–254.

[17] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2016, pp. 26–35.

[18] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," 2018. [Online]. Available: arXiv:1802.06367.

[19] Y. Liang, L. Lu, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 4, pp. 857–870, Feb. 2019.

[20] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on FPGAs," in *Proc. IEEE 27th Annu. Int. Symp. Field Program. Custom Comput. Mach. (FCCM)*, 2019, pp. 17–25.

[21] Q. Xiao and Y. Liang, "ZAC: Towards automatic optimization and deployment of quantized deep neural networks on embedded devices," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2019, pp. 1–6.

[22] Q. Xiao and Y. Liang, "Fune: An FPGA tuning framework for CNN acceleration," *IEEE Design Test*, vol. 37, no. 1, pp. 46–55, Apr. 2019.

[23] Q. Xiao, L. Lu, J. Xie, and Y. Liang, "FCNNLib: An efficient and flexible convolution algorithm library on FPGAs," in *Proc. 57th Annu. Design Autom. Conf. (DAC)*, 2020, p. 9.

[24] X. Wei, Y. Liang, P. Zhang, C. H. Yu, and J. Cong, "Overcoming data transfer bottlenecks in dnn accelerators via layer-conscious memory managment," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, 2019, pp. 120–120.

[25] X. Wei *et al.*, "TGPA: Tile-grained pipeline architecture for low latency cnn inference," in *Proc. Int. Conf. Comput.-Aided Design*, 2018, pp. 1–8.

[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 770–778.

[27] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015. [Online]. Available: arXiv:1510.00149.

[28] T. Zhang *et al.*, "A systematic DNN weight pruning framework using alternating direction method of multipliers," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 184–199.

[29] M. Dmitry, A. Arsenii, and V. Dmitry, "Variational dropout sparsifies deep neural networks," 2017. [Online]. Available: arXiv:1701.05369.

[30] S. Li, J. Park, and P. T. P. Tang, "Enabling sparse winograd convolution by native pruning," 2017. [Online]. Available: arXiv:1702.08597.

[31] J.-H. Luo, J. Wu, and W. Lin, "THINET: A filter level pruning method for deep neural network compression," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 5058–5066.

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

[33] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang, "Accelerating convolutional networks via global & dynamic filter pruning," in *Proc. IJCAI*, Jul. 2018, pp. 2425–2432.

[34] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2074–2082.

[35] C. Ding *et al.*, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2017, pp. 395–408.

[36] J. Cong, W. Jiang, B. Liu, and Y. Zou, "Automatic memory partitioning and scheduling for throughput and power optimization," *ACM Trans. Design Autom. Electron. Syst.*, vol. 16, no. 2, pp. 1–25, 2011.

[37] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong, "Memory partitioning and scheduling co-optimization in behavioral synthesis," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, 2012, pp. 488–495.

[38] Y. B. Asher and N. Rotem, "Automatic memory partitioning: Increasing memory parallelism via data structure partitioning," in *Proc. IEEE/ACM/IFIP Int. Conf. Hardw. Softw. Codesign Syst. Synth.*, 2010, pp. 155–162.

[39] Xilinx. (2018). *UG902—VIVADO Design Suite User Guide: High-Level Synthesis (Ver 2018.2)*. [Online]. Available: http://www.xilinx.com/

[40] Intel. (2018). *Intel—High Level Synthesis Compiler User Guide (Ver 2018.9)*. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls.pdf

[41] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," 2017. [Online]. Available: arXiv:1703.09039.

[42] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, p. 5.

[43] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 1135–1143.

[44] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2009, pp. 248–255.

[45] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, 2010.

[46] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2016, pp. 779–788.

[47] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.

[48] Mentor. (2018). *Catapult High-Level Synthesis*. [Online]. Available: https://www.mentor.com/hls-lp/

[49] Synopsys. (2018). *Synopsys DC Compiler*. [Online]. Available: https://www.synopsys.com/

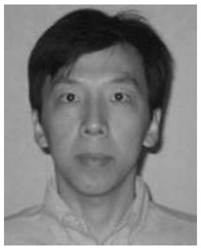
[50] Y. Yang *et al.*, "Synetgy: Algorithm-hardware co-design for Convnet accelerators on embedded FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2019, pp. 23–32.

[51] S. Li *et al.*, "An FPGA design framework for CNN sparsification and acceleration," in *Proc. FCCM*, 2017, p. 9.

[52] Xilinx. (2019). *DNNDK*. [Online]. Available: https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html#dnndk

[53] *NVIDIA cuDNN*. Accessed: Aug. 2020. [Online]. Available: https://developer.nvidia.com/cudnn

[54] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitect. (MICRO)*, 2016, pp. 1–12.



Yun Liang (Senior Member, IEEE) received the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2010.

He worked as a Research Scientist with the University of Illinois at Urbana–Champaign, Urbana, IL, USA. He is an Associate Professor (with tenure) in CECA with the Department of Computer Science and Technology, EECS, Peking University, Beijing, China. He has authored over 70 scientific publications in premier international journals and conferences in related domains. His research focuses on heterogeneous computing (GPUs, FPGAs, ASICs) for emerging applications, such as AI and big data, computer architecture, compilation techniques, programming model and program analysis, and embedded system design.

Dr. Liang research has been recognized by best paper award at FCCM 2011 and ICCAD 2017 and the best paper nominations at DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008. He serves as an Associate Editor for *ACM Transactions in Embedded Computing Systems* and serves in the program committees in the premier conferences in the related domain, including HPCA, PACT, CGO, ICCAD, ICS, CC, DATE, CASES, ASPDAC, and ICCD.

Liqiang Lu received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2017, where he is currently pursuing the Ph.D. degree with the School of EECS.

His research focuses on algorithm-level and architecture-level optimizations of FPGA for machine learning applications.

Jiaming Xie received the B.S. degree from the Institute of Microelectronics, Peking University, Beijing, China, in 2018, where he is currently pursuing the Ph.D. degree with the School of EECS.

His research focuses on GPU programming and system-level optimization for FPGA cluster.