Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization

H.T. Kung
Harvard University
kung@harvard.edu

Bradley McDanel
Harvard University
mcdanel@fas.harvard.edu

Sai Qian Zhang
Harvard University
zhangs@g.harvard.edu

Abstract

This paper describes a novel approach of packing sparse convolutional neural networks into a denser format for efficient implementations using systolic arrays. By combining multiple sparse columns of a convolutional filter matrix into a single dense column stored in the systolic array, the utilization efficiency of the systolic array can be substantially increased (*e.g.,* 8x) due to the increased density of nonzero weights in the resulting packed filter matrix. In combining columns, for each row, all filter weights but the one with the largest magnitude are pruned. The remaining weights are retrained to preserve high accuracy. We study the effectiveness of this joint optimization for both high utilization efficiency and classification accuracy with ASIC and FPGA designs based on efficient bit-serial implementations of multiplier-accumulators. We demonstrate that in mitigating data privacy concerns the retraining can be accomplished with only fractions of the original dataset (e.g., 10% for CIFAR-10). We present analysis and empirical evidence on the superior performance of our column combining approach against prior arts under metrics such as energy efficiency (3x) and inference latency (12x).

CCS Concepts • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Systolic arrays**; **Neural networks**; • **Hardware** → *Hardware-software codesign.*

Keywords systolic arrays; neural networks; sparsity; joint optimization; data flow architectures

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304028

ACM Reference Format:
H.T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304028

1 Introduction

Many recent hardware-based state-of-the-art deep learning accelerators use systolic arrays [28, 29] for efficient implementations of convolutional neural networks (CNNs). These systems, including the Google TPU [24] and numerous other efforts [8, 14, 38, 52, 56], leverage properties of systolic arrays such as parallel processing through a dataflow architecture, regular layout of processing elements, and efficient interprocessor communication, in order to achieve low power consumption and high throughput for CNN inference.

Concurrently, weight pruning techniques have been proposed to further reduce the computation cost of CNN inference by setting to zero (pruning) the majority of weights during training while minimally impacting classification accuracy. This reduces the amount of multiplier-accumulator (MAC) operations required for inference, as the multiplications with the zero (pruned) weight can be skipped. However, the remaining nonzero weights after pruning are distributed in an unstructured manner, making it challenging to efficiently utilize the regular structure of systolic arrays. Using a standard systolic array, the weights set to zero after pruning still occupy systolic cells in order to maintain synchronization across all cells in the array during matrix multiplication. Therefore, the weight reduction achieved through pruning does not necessarily lead to a reduced runtime and hardware resources required for CNN inference.

In this paper we propose a novel approach, called *column combining*, which can pack sparse convolutional networks for efficient implementations in systolic arrays. For each sparse filter matrix in a pruned CNN, all conflicting weights but the one with the largest magnitude are pruned. As we discuss in more detail in Section 3, by removing these conflicts, a sparse filter matrix fits compactly into our proposed regular systolic architecture. The classification accuracy of

the column combined CNN is then improved through additional retraining of the nonzero weights. Thus, our proposed column combining approach leverages a joint optimization opportunity present in CNNs. We optimize the topology of a CNN to fit the structure of the underlying computing hardware, such as systolic arrays, while preserving most of its classification accuracy via network retraining.

The main contributions of the paper are summarized as follows:

- **Column combining algorithm** (Section 3) for packing sparse CNNs with unstructured sparsity into a denser format for efficient systolic array implementations. To ease data routing, a row permuting scheme is described (Section 3.4) for a systolic array to output contiguous data items for those columns to be combined in the next layer. Additionally, the method can retrain remaining filter weights after column-combine pruning using only fractions of the original training dataset to mitigate data privacy concerns (Section 5.5).
- **Joint optimization methodology** (Algorithm 1 in Section 3) aiming at achieving two objectives simultaneously – high utilization efficiency of the systolic array and high classification accuracy of the CNN. The methodology leverages opportunities presented in CNNs in training for both utilization efficiency and accuracy simultaneously.
- **Bit-serial systolic arrays** (Section 4.2) to allow bit-level fine-grain control in implementing quantized computation and efficient multiplexing of multiple bit-line data streams into a single column in the systolic array to support column combining.
- **Cross-layer pipelining** (Section 3.5) for CNN inference over a series of systolic arrays, one for each layer. By eliminating storing and fetching intermediate results for each layer, the per sample (*e.g.,* an input image) inference latency is dramatically reduced.
- **ASIC and FPGA designs** to validate performance gains of our column combining approach (section 6) for energy efficiency, area efficiency, and latency.

Our PyTorch [41] implementation used to train CNNs with column combining is available at https://github.com/BradMcDanel/column-combine.

2 Background and Related Work

In this section, we first provide a brief review of the basic principles of using systolic arrays for the implementation of CNNs and introduce terminologies that we will use throughout. Then, we review related ASIC and FPGA accelerators for CNN inference, advances in CNN design, weight pruning, and input and weight quantization, all of which have led to large reductions in both model size and computation cost for training and inference.

2.1 Systolic Arrays for Convolutional Layers

The computation of a convolutional layer in a CNN can be viewed as a matrix-matrix multiplication between the data matrix (the output of the previous layer) and the learned filter matrix. Suppose that a convolutional layer has N filters operating on a data volume of depth M, as depicted in Figure 1a. Then, the result of the convolution computation is the matrix product of the filter matrix and the data matrix, as depicted in Figure 1b.

Figure 1c depicts a systolic array design for this matrix multiplication. It is a *weight-stationary* systolic array in the sense that filter weights stored in the array will not move during computation, whereas input data continuously move bottom-to-top and result data continuously accumulate left-to-right. For systolic array synchronization, items in the data and result matrices are properly skewed, as shown in the figure. We assume throughout the paper this weight-stationary systolic array design, while noting that use of other systolic designs are also possible such as one where the result data is stationary and the operand matrices move.

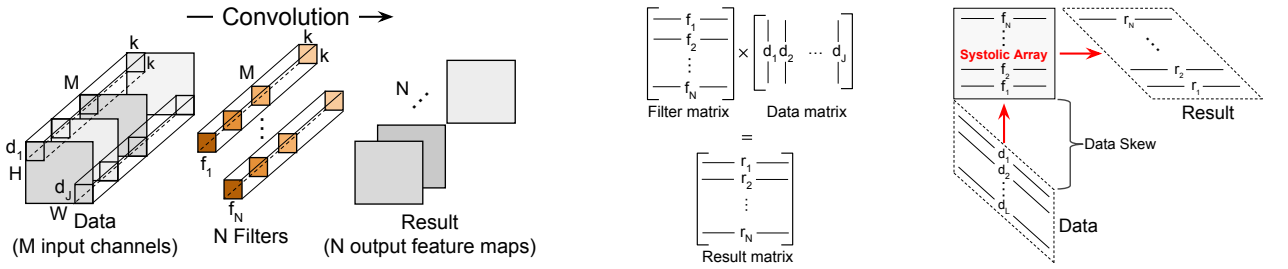
2.2 ASIC and FPGA Accelerators for CNNs

Over the past several years, there has been extensive work on constructing efficient ASIC and FPGA designs for CNNs which generally consider well studied networks such as LeNet-5 [31], AlexNet [27], and VGG [48] including [40, 43, 46, 47, 49, 58]. One of the main considerations for such systems is minimizing the number of off-chip DRAM accesses for storing and fetching the CNN weights, input samples, and intermediate layer results, as these incur significant energy consumption [19]. Therefore, a main focus of accelerator design is mapping CNN computations so that input and weights are fetched only once for all usages within a layer [7]. Another orthogonal direction is designing memory systems that are more suitable to the regular structure of CNN inference computation [44]. In Section 6.1, we show our design achieves state-of-the-art performance in terms of energy efficiency due to the reduced model size after column combining.

FPGAs allow for faster development time and therefore are often used to study system performance implications of various design and implementation alternatives for CNNs, such as low-precision and binary networks [10], novel training regimes [12], and model compression through weight pruning or novel CNN structures [18]. In Section 6, we validate the correctness and performance of our column combining algorithm with an FPGA implementation. Additionally, we compare our implementation to state-of-the-art FPGA results.

2.3 CNNs with Simplified Filter Structures

Figure 2 compares standard CNNs to two recent CNN variants, *separable convolution* [9, 22] and *shift convolution* [54].



(a) Computation of a convolutional layer, where J is HW

(b) Equivalent matrix-matrix multiplication

(c) Weight-stationary systolic array

Figure 1. (a) Computation of a convolutional layer, (b) viewed as a matrix multiplication, and (c) deployed in a weight-stationary systolic array, with skewed input data and output result.

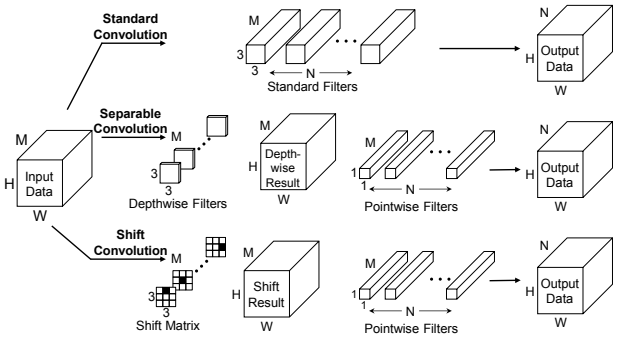


Figure 2. Standard, separable, and shift convolution.

Separable convolution decouples a standard convolution layer into two smaller convolution layers (depthwise convolution and pointwise convolution) in order to reduce the number of model parameters and amount of computation during inference. Each pointwise filter has only a single weight for each channel, and therefore does not utilize neighboring pixels in the spatial dimensions (width and height). Shift convolution replaces the depthwise convolution layer with shift operations that do not require any learned weights. In Section 5.3, we compare the performance of our column combining approach on CNNs with two types of filter structures, where each layer implements either standard or shift convolution, in order to show the impact of column combining under different types of convolution.

2.4 Weight Pruning During Training

Weight pruning methods aim to reduce the number of weights in a trained CNN by removing (pruning) unimportant weights. These pruning techniques have shown that many well studied networks such as AlexNet and VGG-16 have a large number of weights (up to 90%) that can be pruned without any impact on classification accuracy [19].

However, the remaining nonzero weights in each sparse filter matrix after pruning are distributed in an *unstructured* manner, making efficient implementations of CNN inference difficult. This has lead to recent work on structured pruning techniques, which add constraints so that the remaining filter matrix after pruning is still dense [16, 21, 23, 37, 39, 53]. This is generally achieved by removing entire rows (filters) and columns (channels) from the filter matrix, with some reduction to classification accuracy [53]. In Section 5.3, we show that the unstructured pruning in conjunction with column combining leads to higher accuracy compared to state of the art structured pruning techniques while maintaining a packed format that can be efficiently implemented in our proposed systolic array.

2.5 Input and Weight Quantization

Quantization is an important area of study in accelerating inference computations. In this work, we take a simple linear fixed-point quantization scheme [33]. We quantize both the inputs and weights to an 8-bit fixed-point representation from the 32-bit float-point representation [17, 34] used during training. This quantization has been shown to lead to minimal accuracy degradation even on challenging datasets [33]. Within a layer, the accumulation is done with 32-bit integers, which adds complexity to the bit-serial systolic array design and is discussed in Section 4.2. In the future, we could use low-precision quantization (*e.g.,* binary or power of two weights) in order to further improve the efficiency of CNN inference with systolic arrays.

3 Column Combining

As discussed in Section 2.4, training a CNN with unstructured weight pruning leads to small but highly sparse models with unstructured nonzero weights. This unstructured sparse weight matrix is not amenable to efficient implementation in systolic arrays traditionally designed for dense matrix-matrix multiplication. In this section, we propose a column combining algorithm, which jointly optimizes the CNN for both classification accuracy and utilization efficiency when deployed in the proposed systolic array described in Section 4.

3.1 Terminologies and definitions

Suppose that we are given the sparse filter matrix of weights associated with a convolutional layer of a CNN (see Figure 1a) after unstructured pruning has been performed. The columns of this filter matrix which have nonzero weights on a given row are said to be *conflicting* on the row, and the row is said to be a *conflicting row* for these columns. By *column combining*, we mean combining a selected group of columns into a single *combined* column. In a combined column, for the columns which conflict on a row, all nonzero weights on the row are pruned except for the one with the largest magnitude. We refer this pruning process as *column-combine pruning*.

Further, we say a group of columns has x conflicts if a total of x weights will be pruned when combining columns in the group. We say that a group of columns meets the *limited-conflict condition* for certain γ value, if the group has at most γ conflicts per row on average. The γ value can less than 1. For example, if $\gamma = 0.5$, then for every two rows at most one weight is pruned on average.

3.2 Column Combining Overview

Given a sparse filter matrix, we first partition it into *column groups* by grouping columns that have minimal conflicts. Then, for each column group, we combine the sparse columns in the group into a single combined column by applying column-combine pruning. We aim at achieving two objectives simultaneously. First, we pack the given sparse filter matrix into a dense matrix, called a *packed filter matrix*, with as a few combined columns as possible to allow efficient systolic array implementations with a small number of columns. Second, we minimize the impact of column-combine pruning on classification accuracy.

For high-density packing, we adopt a *dense-column-first* combining policy that favors selections of combining columns which result in high-density combined columns, where the density of a column is the percentage of nonzeros in the column. For high-classification accuracy, we then retrain the remaining weights after column-combine pruning.

The algorithm involves some parameters: α (the maximum number of sparse columns that can be combined into a single dense column), β (the pruning rate schedule for unstructured pruning) and γ (the average number of conflicts per row allowed for each group). Typically, α is small (*e.g.,* 2) for early smaller CNN layers and large (*e.g.,* 8) for later large CNNs layers which have more capacity to be pruned. For β, we follow a pruning rate schedule in a similar manner to that proposed in [59] and prune up to a target number of nonzeros after column combining for each layer. For γ, a value of 1.75 is sufficient to achieve good packing efficiency (*e.g.,* over 90% nonzeros) after column combining.

Figure 3 depicts a column combining example. In (a), a filter matrix **F**, associated with a sparse convolutional layer,

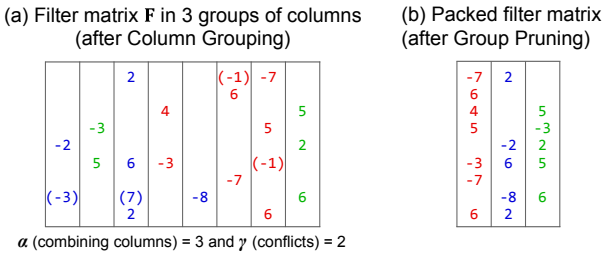


Figure 3. Example of combining columns.

is divided along columns into three groups (blue, green, and red). The zero-valued weights in **F** due to previous pruning steps are omitted for illustration clarity. The objective of column grouping is to select sparse columns that, when combined, result in a single combined column with high packing efficiency (*i.e.,* components are mostly nonzero). As we show in Section 5, high packing efficiency translates to a high utilization efficiency of the systolic array, as more MACs will perform useful computation by storing nonzero weights. A small number of conflicting elements γ are allowed between the columns in a group. For instance, in the blue group, (-3) in column 1 conflicts with (7) in column 3 and -8 in columns 5. The conflicting (-3) and (7) weights are pruned and -8 is kept as it has the largest magnitude. In (b), each group is combined into a single column in order to be loaded into a column in the proposed systolic array (as discussed in Section 4).

3.3 Column Combining Algorithm

The column combining scheme, outlined in Section 3.2, combines columns in a sparse filter matrix that do not have significant conflicts. Algorithm 1, which calls Algorithm 2 and Algorithm 3, is the top level algorithm used to train sparse CNNs that can be implemented with systolic arrays of high utilization efficiency. The training process works in an iterative fashion, where at each pruning epoch the model is pruned based on the pruning rate schedule set by β, followed by column combining. Generally, pruning is performed in a gradual manner, with up to 10 pruning epochs over the course of training. Training continues without pruning for the final 50% of epochs. In Section 5, we provide analysis on the effect that each parameter of Algorithm 1 has on both classification accuracy and utilization efficiency.

The limited-conflict condition assures that in column combining for each group (Algorithm 2) at most γ weights are pruned per row on average (Algorithm 3). This helps minimize the impact of column-combine pruning on classification accuracy. The fact that each group can have at most α columns (e.g., $\alpha = 8$) limits the degree of multiplexing that systolic cells (described in Section 4.2) need to support, while allowing as many as α columns to be combined in order to

Algorithm 1: CNN Training with Column Combining

Input: C is a CNN with L convolution layers
A is the maximum number of combined columns per column group for each layer
β is the pruning schedule (amount pruned per prune epoch)
γ is the number of conflicts (*i.e.,* pruned weights) allowed on average per row for each column group (fixed across layers)
E is the number of epochs for training
Output: $\hat{\mathbf{C}}$ is a pruned version of C with combined columns
G are the column groups for each of the L layers in $\hat{\mathbf{C}}$

$\hat{\mathbf{C}} \leftarrow \mathbf{C}$;
for $e \leftarrow 1$ *to* E **do**
  if *e is a prune epoch* **then**
    for $l \leftarrow 1$ *to* L **do**
      ▷ **Step 1** Perform unstructured pruning by removing the smallest magnitude weights up to β_e
      $\hat{\mathbf{C}}_l \leftarrow \texttt{prune}(\hat{\mathbf{C}}_l, \beta_e)$;
      ▷ **Step 2** Form column groups (Algorithm 2)
      $\mathbf{G}_l \leftarrow \texttt{group-columns}(\hat{\mathbf{C}}_l, A_l, \gamma)$;
      ▷ **Step 3** Prune conflicts in groups (Algorithm 3)
      $\hat{\mathbf{C}}_l \leftarrow \texttt{group-prune}(\hat{\mathbf{C}}_l, \mathbf{G}_l)$;
  $\hat{\mathbf{C}} \leftarrow \texttt{train}(\hat{\mathbf{C}})$; ▷ Network Training

achieve high packing density. Finally, we note that the dense-column-first combining policy is analogous to that of some popular bin-packing algorithms which pack large items first.

Algorithm 2: Column Grouping (`group-columns`)

Input: $\mathbf{F} \in \mathbb{R}^{N \times MHW}$ a filter matrix with N rows and MHW columns (see Figure 1a)
α is the maximum number of combined columns per group
γ is the number of conflicts (*i.e.,* pruned weights) allowed on average per row for each column group
Output: $\mathbf{g}$ are the P groups of columns in $\mathbf{F}$

$\mathbf{g} \leftarrow [\{\}]$;
$\mathbf{u} \leftarrow \{1, 2, \ldots, MHW\}$;
Loop
  ▷ exit if every column is in a group
  if $\mathbf{u} = \emptyset$ **then break**;
  $c \leftarrow \texttt{pop}(\mathbf{u})$; ▷ select ungrouped column c
  ▷ compute densities $\mathbf{d}$ between $\mathbf{g}$ and c
  $\mathbf{d} \leftarrow \texttt{pairwise-density}(\mathbf{F}, \mathbf{g}, c)$;
  ▷ compute number of conflicting weights between $\mathbf{g}$ and c
  $\mathbf{o} \leftarrow \texttt{pairwise-overlap}(\mathbf{F}, \mathbf{g}, c)$;
  ▷ select the group with the highest density while satisfying both the group size α and the limited-conflict γ conditions
  $g_p \leftarrow \texttt{densest-group}(\mathbf{g}, \mathbf{d}, \mathbf{o}, \alpha, \gamma)$;
  $g_p \leftarrow g_p \cup c$; ▷ add c to the group g_p

3.4 Row Permutation for Contiguous Column Groups

We can permute rows of a filter matrix of the current layer to ensure that the columns from the same group for the next layer are output next to each other. In Figure 4, systolic arrays for various layers are denoted as rectangles with a thick black boundary. In (a), a systolic array of eight columns

Algorithm 3: Column-Combine Pruning (`group-prune`)

Input: $\mathbf{F} \in \mathbb{R}^{N \times MHW}$ a filter matrix of N rows by MHW columns
$\mathbf{g}$ are the P groups of columns in $\mathbf{F}$
Output: $\hat{\mathbf{F}}$ is $\mathbf{F}$ with conflicting entries within each group pruned

$\hat{\mathbf{F}} \leftarrow \mathbf{F}$;
▷ For each p group, prune all but one entry per row
for $p \leftarrow 1$ *to* P **do**
  $\hat{\mathbf{F}}_p \leftarrow \hat{\mathbf{F}}[:, \mathbf{g}_p]$; ▷ Submatrix of $\hat{\mathbf{F}}$ containing columns in $\mathbf{g}_p$
  for $n \leftarrow 1$ *to* N **do**
    $w \leftarrow \max(|\hat{\mathbf{F}}_p[n]|)$; ▷ Find largest magnitude weight w
    `found` $\leftarrow$ false; ▷ Breaks ties for largest weights
    for $k \leftarrow 1$ *to* $\texttt{size}(\mathbf{g}_p)$ **do**
      if `found` *or* $|\hat{\mathbf{F}}_p[n][k]| < w$ **then**
        $\hat{\mathbf{F}}_p[n][k] \leftarrow 0$; ▷ Prune (set to 0)
      else
        `found` $\leftarrow$ true;

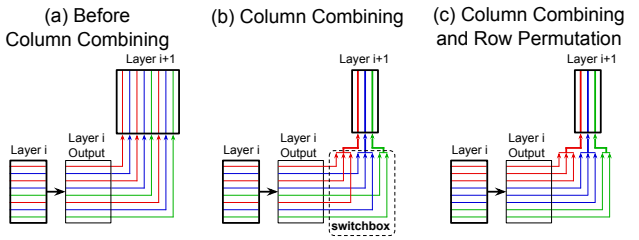


Figure 4. Applying Column Combining and Row Permutation.

for layer i+1 is for an original sparse filter matrix of this layer consisting of three column groups, indicated in three colors, for column combining. In (b), column combining is performed on the three column groups, which results in a reduced systolic array of three columns for layer i+1. This reduced systolic array is for a packed filter matrix consisting of three combined columns. A relatively expensive switchbox function is needed for routing output of layer i to input of the reduced systolic array for layer i+1. In (c), by permuting the rows of the layer i filter matrix according to the column groups in layer i+1, we avoid the expensive switchbox.

Note that such row permutations are valid, as the column combining operation on a filter matrix are not affected by row permutations on the previous filter matrix. Thus, row permutations for layer i can be determined by the column groups of a row permuted filter matrix for layer i+1. This makes the columns within each group contiguous and removes the need to reorder the output using a switchbox at inference runtime.

3.5 Cross-layer Pipelining of CNN Inference under Column Combining and Row Permutation

In many realtime application scenarios, single sample latency is a more important metric than throughput, as an input sample (*e.g.,* an image) must be processed as soon as it is

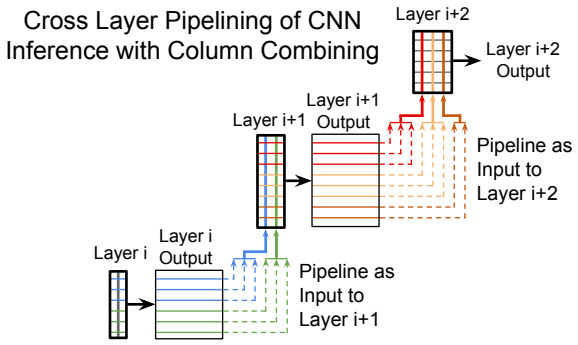


Figure 5. Pipelining CNN inference across three layers with column combining and row permutation applied to each layer.

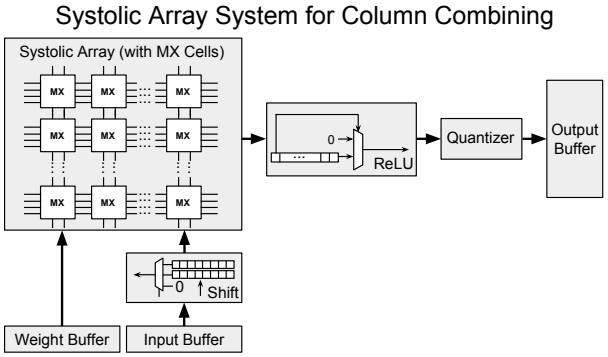


Figure 6. Systolic array system.

received by the system, and therefore cannot wait to be processed in large batches.

To address this concern, we propose cross-layer pipelining which pipes the output data elements from the previous layer immediately as input into the next layer as soon as it exits from the systolic array. Figure 5 shows this pipelining approach for three sparse CNN layers (Layer i, Layer i+1, and Layer i+2), each deployed in a separate systolic array after column combining and row permutation have been applied to each layer. The dashed lines emitted from each layer output denote that each data element is immediately pipelined into the next layer. In Section 6.4, we show that this approach reduces the inference latency for our ASIC implementation of LeNet-5 by 3.5×. Having the effect of narrowing systolic arrays for convolutional layers of a CNN, column combining can reduce data skew (see Figure 1c), which further reduces the latency.

For more challenging datasets which require much larger CNNs, such as ImageNet[11], there are not enough systolic cells to implement the entire network in systolic arrays at once. For these networks we take a more conventional weight tiling approach as depicted in Figure 16a.

4 Systolic Array System Description for Column Combining

In this section, we describe the systolic array system and its components in support of the proposed column combining approach presented in Section 3.

4.1 Systolic Array System Overview

The systolic array system which implements packed filter matrices after column combining is shown in Figure 6. The weights of the packed filter matrices corresponding to each convolutional layer of a CNN are stored in the weight buffer. These weights are loaded into the MX cells of the systolic array (discussed in Section 4.2) before matrix multiplication is performed with the input data. The input data is loaded from the input buffer and passed through the shift block (discussed in Section 4.3). The shift block performs shift operations, as depicted in Figure 2, and passes the output to the systolic array in a bit-serial fashion, which then performs matrix multiplication with the weights stored in the systolic cells. The output of each row in the systolic array is passed to the ReLU block (discussed in Section 4.4), which performs the ReLU activation function. Finally, the result from the ReLU block is passed to the quantization block and stored in the output buffer.

4.2 Bit-serial Systolic Arrays

In this section, we describe our bit-serial implementation of a systolic array for matrix multiplication. Figure 7 show our bit-serial MAC design which is used across all systolic array implementations for 8-bit input Xi and 8-bit filter weight W. The white logic elements shown implement the bit-serial multiplication between the input Xi and compute the absolute value of the filter weight. The blue logic elements negate the product based on the sign of the filter weight. The pink full adder performs bit-serial addition between the multiplication result and the input accumulation Yi.

In Figure 9a we illustrate a 3×3 bit-serial systolic array for multiplying a 3 × 3 filter matrix and a 3×M data matrix. We pre-store in the systolic cell at position (i, j) the corresponding filter weight $W_{i,j}$ and its sign in the filter matrix. Data arrives from the bottom of the array. Matrix multiplication results come out from the right side of the array, as depicted earlier in Figure 1c.

First, consider a simple scenario where each systolic cell has *balanced* I/O and computation time. This is the case when input data, filter weights and accumulation values all have the same number of bits. Suppose that they are all 8-bit. In this case, under the bit-serial MAC implementation of Figure 7, we will have a systolic cell as depicted in Figure 8a or a BL (balanced) cell in Figure 10a. In the corresponding systolic array, as depicted in Figure 9a, for data synchronization purposes, neighboring input and accumulation data streams are skewed by one clock to accommodate the communication

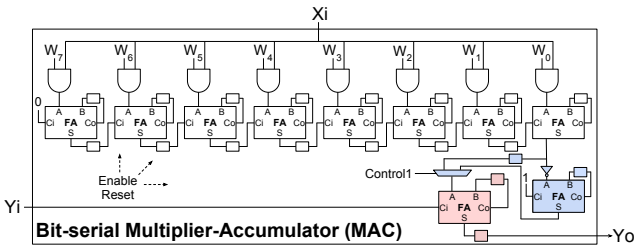


Figure 7. Bit-serial multiplier-accumulator (MAC).

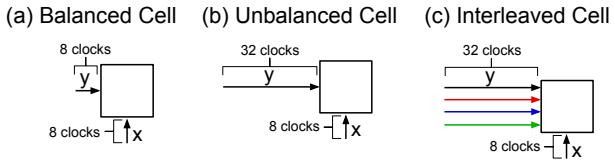


Figure 8. Systolic cells under different computation settings.

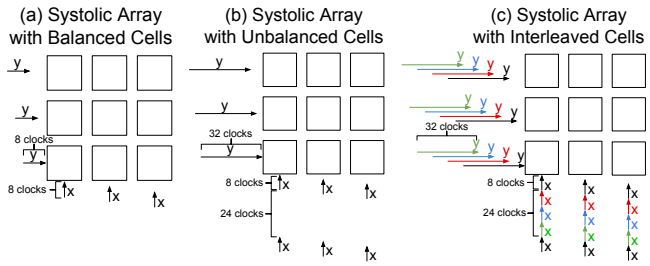


Figure 9. Systolic arrays under mixed precision settings.

delay between the cells. However, this simple scenario is not applicable to high-precision accumulation that is necessary for holding the partial result of matrix multiplication [33].

To accommodate high-precision accumulation, bit-serial systolic cells will have *longer* computation time than I/O. Suppose that input data and filter weights are 8-bit and accumulation values are 32-bit. In this case, under a bit-serial MAC implementation of Figure 7, we have the unbalanced systolic cell as depicted in Figure 8b. In the corresponding systolic array, as depicted in Figure 9b where input of y takes 32 clocks and input of x takes 8 cycles, there is a 24-clock gap between the input data streams. This gap allows for the additional computation time required by the 32-bit accumulation for y.

We can fill in these gaps for each cell by processing *four* independent input data streams simultaneously in an interleaved manner, while expanding the processing power and accumulation data path by 4×, as depicted in Figure 8c and the IL (interleaved) cell in Figure 10b. The corresponding systolic array is depicted in Figure 9c with more details in Figure 11b.

Given the input channel groups determined by the column combining algorithm, we now describe an efficient systolic

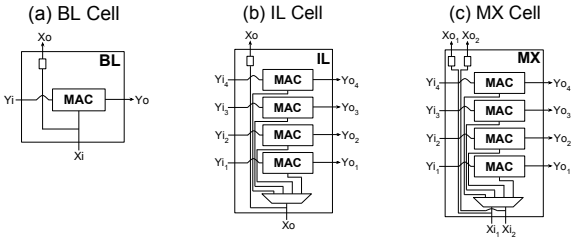


Figure 10. Systolic cell types used for the corresponding systolic array in Figure 11.

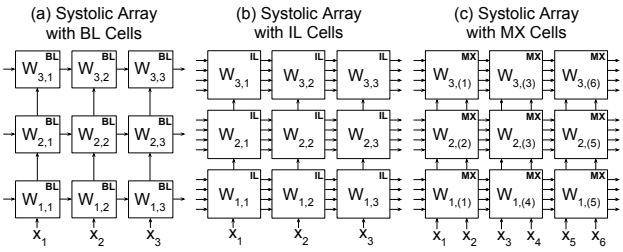


Figure 11. Three types of systolic arrays based on the three cell designs in Figure 10. In (c), $W_{i,(j)}$ denotes a weight to be used by channel x_j. For example, the weight $W_{2,(3)}$ stored in the middle cell is used by channel x_3.

array implementation which can utilize the combined input channel groups. In Figure 10c, the multiplexed input (MX) cell, takes in two x inputs, from two input channels, utilizes one of them inside each MAC, and forwards both inputs to the cell above. Note that while for illustration simplicity this figure shows only two instances of input x_i, in our ASIC and FPGA designs we pack up to 8 channels (requiring 8 instances of input x_i) into a single cell. This highlights the importance of the bit-serial design, as in the case of 8 inputs, each cell takes in only 8 bits per cycle, as opposed to a bit-parallel design where each cell would require 64 inputs per cycle in the case of 8-bit input.

Figure 11c shows how a systolic array connects the MX cells. In this example, for the first column, the first and third rows (filters) use input channel 1, denoted by the $W_{1,(1)}$ and $W_{3,(1)}$ weights stored within the cells, and the second row uses input channel 2, denoted by the $W_{2,(2)}$ weight stored in the cell. As shown, these channels indexes are after row permutation (Section 3.4), and are therefore guaranteed to be contiguous.

4.3 Shift Block

Figure 12 shows the design for the shift operation. Based on the direction of the spatial translation specified by the shift control signal, the memory controller fetches the corresponding 8 bits input maps from the input buffer to the register array, which sends the input to the systolic arrays in a bit-serial fashion. We use double buffering to prefetch

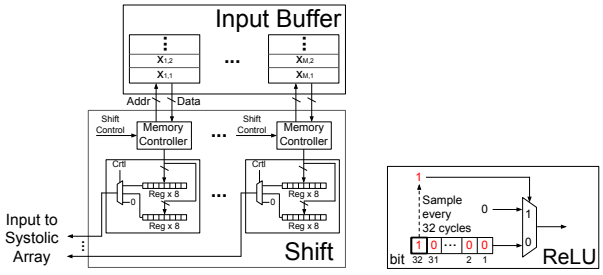


Figure 12. Shift and ReLU blocks.

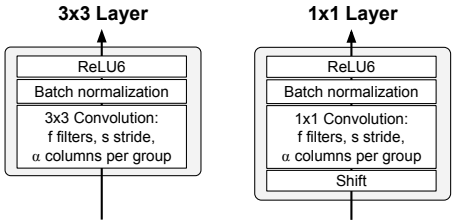


Figure 13. The 1×1 and 3×3 layers for networks of Figure 14.

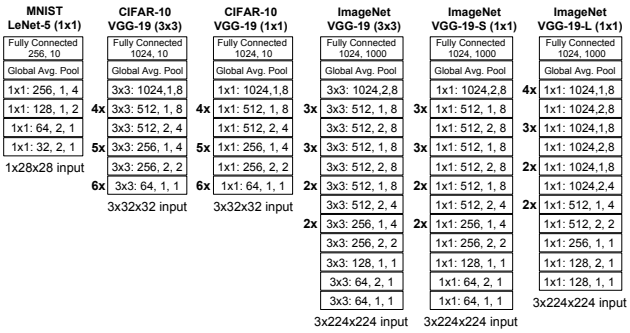


Figure 14. Network structures used for evaluation. Details of the 1×1 and 3×3 layers are shown in Figure 13.

the next data tile so that the output time can overlap with the data transfer overhead from the input buffer to register arrays.

4.4 ReLU and Quantization

Figure 12 shows the design for ReLU operation. The 32-bit input stream comes in a bit-serial fashion and is stalled in a register array until the last bit arrives. The sign of the 32-bit input stream is determined by the most significant bit (32nd bit). If the 32nd bit is 1, then the multiplexer outputs a 32-bit stream of 0; otherwise the multiplexer simply outputs the input stream. The output from the ReLU block is then re-quantized and saved in the output buffer. This output can then be transferred to the input buffer to be used as input for the next layer.

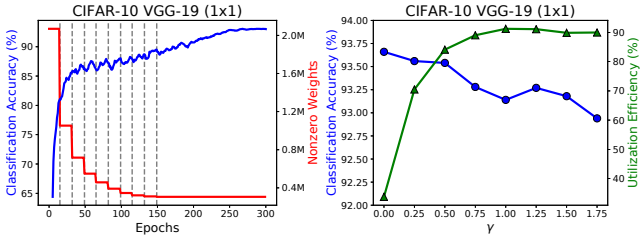
5 Performance Analysis for the Column Combining Algorithm

We analyze our column combining approach described in Section 3 on three datasets MNIST [30] (28×28 greyscale images of handwritten digits), CIFAR-10 [26] (32×32 RGB images of 10 object classes), and ImageNet [11] (224×224 RGB images of 1000 classes). We evaluate column combining on modified versions of well-studied networks: Lenet-5 for MNIST, and VGG-19 for CIFAR-10 and ImageNet.

We use two different layer structures, which are shown in Figure 13, to determine how column combining performs on different filter sizes. The 3×3 layer corresponds to standard convolution with 3×3 filters, as shown at the top of Figure 2. This is the filter size used by the majority of layers in the original VGG-19. The 1×1 layer corresponds to shift convolution (bottom of Figure 2) where only 1×1 filters are used. The shift directions are randomly initialized before training and are not learnable as in [54]. The α parameter corresponds to the maximum size of each column group during column combining. This is used to facilitate the amount of unstructured pruning performed during training for column combining (less aggressive pruning in small layers and more aggressive pruning in large layers).

The network structures used for each dataset are shown in Figure 14. For the VGG-19 networks, we removed all but the final fully connected layer and added a global average pooling operation which is common in modern networks (*e.g.,* ResNet [20]). In Section 5.3, we compare the effectiveness of column combining against structured pruning methods. We provide results for both 1×1 and 3×3 filter networks in order to provide a more clear comparison against previous state of the art which only uses 3×3 filters. We evaluate both a small and large network for ImageNet with 1×1 filters, denoted ImageNet VGG-19-S (1×1) and ImageNet VGG-19-L (1×1), respectively. In Section 6, all ImageNet evaluations use the small 1×1 network unless otherwise stated due to the smaller size of our FPGA.

All networks are trained using Stochastic Gradient Descent (SGD) with an initial learning rate η of 0.05 for Lenet-5 and 0.1 for VGG-19. A Nesterov momentum of 0.9 [45] is used for all networks. A cosine shape learning rate schedule [36] is used to decay the learning rate to 0 over training. All MNIST, CIFAR-10, and ImageNet models are trained for 150, 300, and 120 epochs, respectively. Unstructured pruning, specified by the β schedule, is performed after each epoch up to the 50% mark of training. At this point, the target sparsity has been reached and column combining is performed to pack the remaining nonzero weights. After column combining, no more pruning is performed for the final 50% of training. An l_1 penalty $\lambda = 10^{-7}$ is added to the loss function to encourage unstructured sparsity in the weights during the first 50% of epochs where pruning occurs as shown in Equation 1



(a) Training with Column Combining

(b) Impact of γ

Figure 15. (a) Classification accuracy for validation set and number of nonzero weights over training epochs (grey vertical lines denote epochs where pruning occurs). (b) Increasing γ greatly improves utilization efficiency while minimally impacting classification accuracy.

$$E(\mathbf{W}) = E_D(\mathbf{W}) + \lambda \cdot ||\mathbf{W}||_1 \quad (1)$$

where $\mathbf{W}$ are the CNN weights across all layers, $E_D(\mathbf{W})$ is the objective loss function (*e.g.,* cross-entropy loss), and $||\mathbf{W}||_1$ is the ℓ_1 penalty with weight λ[1].

Before deployment on ASIC or FPGA platforms discussed in Section 6, the batch normalization parameters are folded into the convolution layer and 8-bit linear quantization is applied to the weights.

5.1 Iterative Training with Column Combining

Training a network with column combining occurs over a series of pruning iterations (Algorithm 1), where, at each pruning stage, unstructured pruning and column combining are performed to decrease the model size. Figure 15a shows the classification accuracy and number of nonzeros weights for the VGG-19 (1×1) model on the CIFAR-10 dataset over each training epoch. Each dashed vertical line denotes a pruning epoch. At each epoch, the number of weights in the model is shown by the red line. The pruning schedule is set to be more aggressive in the early epochs of training, where the learning rate is higher and the pruned model can adjust the remaining weights, as proposed in [59].

5.2 Impact of the Limited-Conflict Condition

The limited-conflict condition, as described in Section 3.1, allows for γ conflicting entries per row on average between columns within a group. All but the largest magnitude weight among conflicting weights are pruned during column-combine pruning (Algorithm 3). Figure 15b shows how classification accuracy and utilization efficiency vary as a function of γ for 8 VGG-19 (1×1) networks trained on the CIFAR-10 dataset. Larger values of γ allow for more conflicts between the columns in a group and therefore prune more weights, possibly with relatively large magnitudes, in order to achieve higher utilization efficiency across all layers in the CNN. This dramatically increases the utilization efficiency from 69% ($\gamma = 0.25$) to 90% (for $\gamma = 1.0$ and larger). As discussed in the previous subsection, column-combine pruning has a small impact on classification accuracy (around 1%) since additional training is performed after pruning in order to allow the remaining weights to adjust to the loss of the pruned weights.

[1]Our notation follows [53].

Dataset	Pruning Method	Network	Accuracy (Top-1)	Parameters	Pruned (%)
CIFAR-10	Network Slimming [35]	VGG-19	93.8%	2.3M	88.5%
	Channel Pruning [21]	ResNet-56	91.8%	12.8M	50.0%
	Column Combining (ours)	VGG-19 (3×3)	94.7%	3.0M	84.0%
	Column Combining (ours)	VGG-19 (1×1)	92.9%	0.3M	84.7%
ImageNet	Structured Pruning [53]	AlexNet	58%	46.5M	25%
	Network Slimming [35]	VGG-19	63.34%	23.2M	82.5%
	Channel Pruning [21]	VGG-19	69%	26.6M	80.0%
	Column Combining (ours)	VGG-19 (3×3)	71.81%	4.1M	87.4%
	Column Combining (ours)	VGG-19-S (1×1)	55.99%	1.5M	66.6%
	Column Combining (ours)	VGG-19-L (1×1)	65.19%	2.8M	79.9%

Table 1. Comparison of structured pruning methods to column combining on CIFAR-10 and ImageNet.

5.3 Comparison with Structured Pruning

We compare the effectiveness of column combining, which converts filter matrices with unstructured sparsity into a mostly dense packed format, to structured pruning methods which perform pruning so that the result matrix is always dense. Generally, structured pruning has difficulty achieving the same classification accuracy as unstructured pruning given the same number of parameters as the pruning criteria is significantly constrained. Table 1 compares column combining to state of the art structured pruning techniques on the CIFAR-10 and ImageNet datasets. We see that column combining achieves higher classification accuracy than the other methods while using fewer nonzero parameters after pruning[2]. The pruned (%) is the percentage of weights pruned from the number of weights at the start of training.

5.4 Dramatic Tiling Reduction in Partitioned Matrix Multiplication with Column Combining

When a systolic array is smaller than the weights of a convolutional layer, matrix multiplication can be performed in multiple passes, where each pass executes matrix multiplication between a submatrix (tile) of the layer weights and the corresponding input data. Figure 16a shows how this partitioning process is performed on a sparse filter matrix of (96 rows by 94 columns), which is larger than the systolic array (32 rows by 32 columns). The filter matrix is partitioned into 9 tiles, each with a maximum size of 32 by 32, and the input data is tiled in a similar manner along the columns, but not along the rows (batch size × image height × image width).

The full matrix multiplication is performed by alternating between weight loads and matrix multiplications for each of

[2]Our accuracy is reported before quantization for a fair comparison to the other works which do not perform quantization. Results in Section 6 report accuracy after quantization.

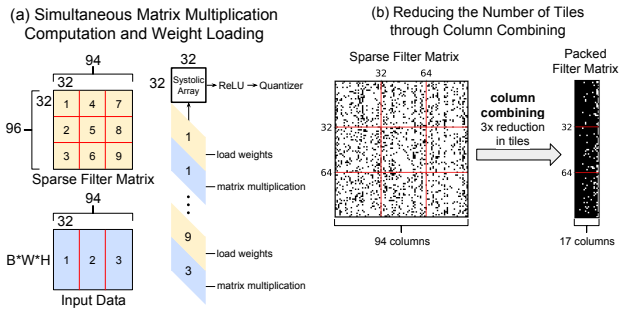
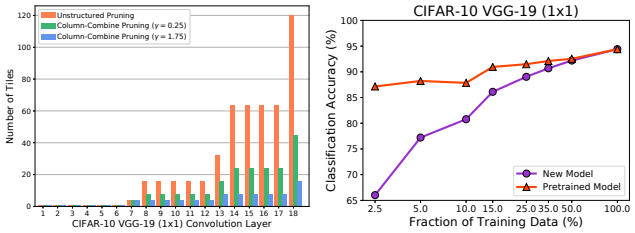


Figure 16. (a) Partitioned matrix multiplication of 9 tiles on systolic array alternating between weight loads and matrix multiplication. (b) Column Combining reduces the number of tiles.



(a) Number of Tiles in Systolic Array

(b) Pruning with Limited Data

Figure 17. (a) Number of tiles for CIFAR-10 VGG-19 (1×1) using a 64×64 systolic array. (b) Comparing training with column combining on a new model versus a pretrained model with a limited datasets.

the submatrices (tiles). The filter matrix and input data enter the systolic array as depicted in a skewed fashion in order to maintain data synchronization within the systolic array. Note that in the steady state every systolic cell is busy all the time, either doing the matrix multiplication computation or loading the weights for the next tile. ReLU and quantization are performed on the output of the systolic array after the final tile for a set of rows in the filter matrix. (Note that in Section 6, we evaluate settings where a CNN layer must be partitioned in tiles, as shown in Figure 16a and also settings where the each layer can fit entirely into a systolic array which does not require partitioning.)

Figure 16b shows a sparse filter matrix and a corresponding packed filter matrix after column combining, which is stored in the systolic array with MX cells (described in Section 4). As in Figure 16a, the sparse filter matrix has 96 rows and 94 columns, with only 16% of the weights being nonzeros. For a 32×32 systolic array, this sparse filter matrix is partitioned into 9 tiles (denoted by the red lines) in order to perform the full matrix multiplication. The packed filter matrix is the output after column combining, which has arranged the 94 columns of the sparse filter matrix into 17 groups. Each group is a single column in the 32×17 packed filter matrix, which can then be loaded into the systolic array of 544 MX cells. This packed format has 89% nonzeros and requires only 3 tiles to perform matrix multiplication (a 3× reduction in tiles).

Figure 17a shows the number of tiles required to perform matrix multiplication with a 64×64 systolic array for each layer in VGG-19 (1×1) models for the CIFAR-10 dataset trained using Algorithm 1 under three different settings of γ. The unstructured pruning setting trains the CNN without column-combine pruning ($\gamma = 0$). The two column-combine models use different settings for γ (0.25 and 1.75), which lead to a large difference in packing efficiency. The $\gamma = 0.25$ setting reduces the number of tiles over the unstructured pruning setting by approximately 50% across all layers. By comparison, the $\gamma = 1.75$ setting further reduces the number of tiles by a substantial margin across all layers and achieves an 8× reduction in the number of tiles for the final 6 layers (13 through 19). Generally, this shows that it is difficult to effectively combine sparse columns when a small γ value is used, as a few conflicts in any row for a potential group will make the combination invalid. By adding a modest amount of column-combine pruning (*e.g.*, $\gamma = 1.75$) the combining algorithm is able to substantially improve the utilization efficiency and decrease the number of tiles.

5.5 Column Combining with Limited Datasets

In many real world scenarios, customers may provide pretrained models to vendors for deployment on their devices (*e.g.*, mobile devices). In these settings, a customer may not wish to divulge entire datasets used to train the model to the vendor for a number of reasons, such as the dataset containing sensitive private information or being a competitive advantage. In this scenario, model pruning is difficult, as pruning weights without retraining leads to a significant degradation in classification accuracy.

We propose that these data privacy concerns can be partially mitigated by providing only a subset of the original dataset to perform training with column combining. Figure 17b compares the effects of column combining on a pretrained dense VGG-19 (1×1) model, trained on the full CIFAR-10 train dataset, to a new network, over different fractions of training data. All models use $\gamma = 1.75$ for column combining. The largest difference in performance between the two approaches is when only 2.5% of the full training data is used (a difference of 20% in classification accuracy), as the weights in the pretrained model are already initialized to reasonable values. At 15% of the full training data, the pretrained model can achieve over 90% classification accuracy. This shows that a small amount of training data can be sufficient to perform the retraining in column combining while still maintaining a relatively high classification accuracy. By comparison, training a new model requires 50%

of the training dataset to achieve an over 90% classification accuracy.

6 Hardware Implementation Experiments and Performance Evaluation

In this section, we evaluate the performance of our systolic array system for column combining described in Section 4 for both ASIC and FPGA implementations. Throughout, we compare designs in terms of accuracy, throughput, area efficiency, and energy efficiency. Additionally, we pay attention to performance for single or a small number of input samples (*e.g.,* the end-to-end latency) and energy required to process a single input sample (*e.g.,* an input image). As stated earlier in Section 3.5, in realtime scenarios, single sample latency is a more important metric than throughput, as an input sample must be processed immediately to meet a set latency budget. For all ImageNet evaluations, only the VGG-19-S (1×1) model is used due to the smaller size of our FPGA.

6.1 ASIC Implementation and Evaluation

We synthesize our ASIC design using the Synopsys Design Compiler [2] with 45nm NanGate Open Cell Library [3] and CACTI 7.0 [1]. We estimate the hardware performance of static random-access-memory (SRAM) with CACTI 7.0 and synthesize the remaining components of the design including Systolic Arrays with MX cells (Section 4.2), Shift (Section 4.3), ReLU and Quantization (Section 4.4) using the Synopsys Design Compiler.

6.1.1 Comparison Against Prior Designs for MNIST

We compare our ASIC implementation of LeNet-5, trained on MNIST, to prior state-of-the-art CNN accelerator designs. Due to the small size of the LeNet-5 model, each layer can fit entirely into a systolic array and therefore no tiling is required. Additionally, we use 16-bit accumulations for the systolic array for this experiment, as the filter matrix for each layer is small and therefore does not require 32-bit accumulations. With 16-bit accumulations, a single MAC operation will take half amount of cycles compared with 32-bit accumulations. All other designs use LeNet-5 (except for SpiNNaker [25] which uses a Deep Belief Network and TrueNorth [6] which uses a Spiking Neural Network). Table 2 compares the designs in terms of accuracy, area efficiency, and energy efficiency. Generally, our design has both the highest area efficiency and energy efficiency across all the designs. Compared to next best design (SC-DCNN), our design achieves a 2.2× improvement in area efficiency and a 3× improvement in energy efficiency, for a slightly reduced classification accuracy.

6.2 FGPA Implementation and Evaluation

For our FPGA implementation, we use the Xilinx VC707 evaluation board. We synthesize our design using the Xilinx Vivado Design Suite [5]. We use 32-bit accumulation for the systolic array implementation. The resources of the VC707 allow for a systolic array with column combining of size 64 rows by 64 columns to be implemented.

Table 3 compares our VGG-19 (1×1) implementation to other FPGA implementations for CIFAR-10 in terms of classification accuracy and energy efficiency. Our design achieves top-1 accuracy of 93.0%, which is around 5-6% higher than other models. Moreover, our design achieves a 3× improvement on energy efficiency over the next best design. While it is possible for the other designs to increase the accuracy by using more hardware, it is hard for them to attain a low energy efficiency as our design.

Table 4 compares our ImageNet VGG-19-S (1×1) model to other FPGA-based accelerators for ImageNet. In term of energy efficiency, our design outperforms the most recent work [51, 57], by 3.34× and 1.93×, respectively. Our approach allows for substantially smaller models trained with column combining to achieve reasonable accuracy while being efficiently implemented in a high-utilization systolic array. This smaller model translates to a reduced number of MAC operations required by the model which leads to higher energy efficiency.

Platform	Network	Platform	Accuracy	Area Eff.	Energy Eff.
Ours	CNN	ASIC	97.62%	46603	658053
SC-DCNN	CNN	ASIC	98.26%	21439	221287
2x Xeon W5580	CNN	CPU	98.46%	2.5	4.2
Tesla C2075	CNN	GPU	98.46%	4.5	3.2
SpiNNaker	DBN	ARM	95.00%	N/A	166.7
TrueNorth	SNN	ASIC	99.42%	2.3	9259

Table 2. Comparison of our ASIC implementations of LeNet-5 to other CNN accelerators for MNIST.

	[50]	[58]	[13]	Ours
Xilinx FPGA chip	N/A	XC7Z020	Cyclone V 5CEA9	VC707
Frequency (MHz)	N/A	143	100	150
Accuracy (Top-1)	N/A	87.73%	88.3%	93.0%
Efficiency (img./S/W)	6109	1320	36	12112

Table 3. Comparison of our VGG-19 (1×1) model to state-of-the-art FPGA implementations for CIFAR-10.

Table 4. Comparison with FPGA-based CNN accelerators for the ImageNet dataset.

	[57]	[42]	[55]	[32]	[47]	[51]	Ours
Xilinx FPGA Chip	VC706	ZC706	ZC706	VC709	Virtex-7	ZC706	VC707
Frequency (MHz)	200	150	100	150	100	200	170
Accuracy (Top-1)	53.30%	64.64%	53.40%[+]	53.40%[+]	55.70%	52.60%	51.28%
Efficiency (img./S/W)	23.6	0.46	6.13	12.93	8.39	40.7	**78.7**

[+] These papers did not directly report classification accuracy, but used 8-bit or 16-bit quantized versions of the CNN. These classification accuracy numbers are our estimate based on reported numbers in [4].

6.3 Optimality in Energy Efficiency

We provide an analysis showing that our systolic array design can achieve a nearly optimal energy efficiency for the given design architecture. The total energy consumption of processing an input sample is:

$$E_{total} = E_{comp} + E_{mem} = E_{mac} \times N_{mac} + E_{mem} = E_{mac} \times cN_{mac}^{opt} + E_{mem}$$

where E_{comp} and E_{mem} are the energy consumption for all MAC computations and SRAM, respectively, E_{mac} is the energy consumption for a single MAC operation, N_{mac} is the number of MAC operations in the CNN after column combining, and N_{mac}^{opt} is the optimal number of MAC operations where no multiplications with zeros are performed. Let c denote the ratio between N_{mac} and N_{mac}^{opt}. Suppose that all designs have the same E_{mac} and E_{mem}. Then, the energy efficiency of a design is:

$$\text{Energy Eff.} = \frac{1}{E_{total}} = \frac{1}{E_{comp} + E_{mem}} = \frac{1}{E_{mac} \times cN_{mac}^{opt} + E_{mem}}$$

and the optimal energy efficiency for this design is:

$$\text{Optimal Energy Eff.} = \frac{1}{E_{mac} \times N_{mac}^{opt} + E_{mem}}$$

We have observed from synthesized results that when the input size is relatively small, $r = \frac{E_{mem}}{E_{comp}}$ tends to be small. For example, $r = 0.06$ and $r = 0.1$ for MNIST and CIFAR-10, respectively. In this case, we have

$$\frac{\text{Energy Eff.}}{\text{Optimal Energy Eff.}} = \frac{E_{mac} \times N_{mac}^{opt} + E_{mem}}{E_{mac} \times cN_{mac}^{opt} + E_{mem}} = \frac{\frac{1}{c} + r}{1 + r} \approx \frac{1}{c}$$

Note that $1/c$ is the packing efficiency achievable by column combining. Thus when r is small, the ratio between Energy Eff. and Optimal Energy Eff. is dominated by the packaging efficiency. Consider, for example, the scenario depicted in Figure 15b, for $\gamma = 1.75$. Column combining can achieve a packing efficiency over 90% with a modest degradation of classification accuracy of about 0.7% in absolute percentage. Thus in this case the energy efficiency of our design is about 90% of the optimal energy efficiency, for small values of r.

6.4 Reduction in End-to-end Inference Latency with Cross-layer Pipelining

In this section, we evaluate the FPGA performance of cross-layer pipelining, described in Section 3.5, in terms of end-to-end inference latency for a single sample for CIFAR-10. Cross-layer pipelining reduces the latency significantly by 9.3× compared to without pipelining for CIFAR-10. Table 5 compares our column combined VGG-19 (1×1) model with cross-layer pipelining to other hardware implementations including GPU, CPU, and FPGA accelerators for CIFAR-10 in terms of accuracy and single frame latency. The latency 652μs of [15] shown in Table 5 only includes the latency for all convolutional layers (thus the total is greater than 652).

	CPU[58]	GPU[58]	[58]	[15]	Ours
Accuracy (Top-1)	88.42%	88.42%	88.42%	85.88%	93.0%
Latency (microseconds/frame)	14800	730	5940	>652	55.68

Table 5. Comparison of our VGG-19 (1×1) model with cross-layer pipelining to state-of-the-art CNN accelerators for CIFAR-10.

Our design achieves an end-to-end latency over 12× smaller than next best implementation, while also obtaining a higher classification accuracy.

7 Conclusion

In this paper, for CNN inference, we have presented a solution to a long-standing parallel processing challenge about how one can make efficient use of regular parallel processing arrays, such as systolic arrays, for *sparse* computations. Specifically, for a given sparse CNN, we have proposed a novel approach of using column combining to pack the filter matrix associated with each convolutional layer for its efficient systolic array implementation. In combining columns, we prune all weights on conflicting rows but the one with the largest magnitude. We then continue to improve classification accuracy of the pruned network via retraining. We iterate on this column-combining and network-retraining step to improve both utilization efficiency of the systolic array and the classification accuracy of the network.

Being able to transform sparse computations to fit highly efficient regular processor arrays is powerful. As demonstrated in the paper, our proposed column combining approach can increase the utilization efficiency of a systolic array by approximately 8×, with a slight increase in the complexity of systolic cells for providing multiplexing (MX) support. This has led to superior performance of our proposed method against prior arts under metrics such as energy efficiency (3×) and inference latency (12×).

Acknowledgments

This work is supported in part by the Air Force Research Laboratory under agreement number FA8750-18-1-0112, a gift from MediaTek USA and a Joint Development Project with TSMC.

References

[1] [n. d.]. CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model. ([n. d.]). https://github.com/HewlettPackard/cacti.

[2] [n. d.]. Design Compiler: RTL Synthesis. ([n. d.]). https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html.

[3] [n. d.]. NanGate FreePDK45 Open Cell Library. ([n. d.]). http://www.nangate.com/?page_id=2325.

[4] [n. d.]. Quantized ImageNet Models Implemented by github.com/aaron-xichen. ([n. d.]). https://github.com/aaron-xichen/pytorch-playground.

[5] [n. d.]. Vivado Design Suite - HLx Editions Productivity. Multiplied. ([n. d.]). https://www.xilinx.com/products/design-tools/vivado.html.

[6] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente Kuang, Rajit Manohar, William Risk, Bryan Jackson, and Dharmendra Modha. 2015. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 10 (2015), 1537–1557.

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices* 49, 4 (2014), 269–284.

[8] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2017. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138.

[9] François Chollet. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. *arXiv preprint arXiv:1610.02357* (2016).

[10] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.

[12] Roberto DiCecco, Lin Sun, and Paul Chow. 2017. FPGA-based training of convolutional neural networks with a reduced precision floating-point library. In *Field Programmable Technology (ICFPT), 2017 International Conference on*. IEEE, 239–242.

[13] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 395–408.

[14] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.

[15] Xing T. Zhao R. Zhang Z. Srivastava M. B. Tu Z. Gupta R. K. ELin, J. H. 2017. Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration. In *CVPR Workshops*. 344–352.

[16] Scott Gray, Alec Radford, and Diederik Kingma. 2017. GPU Kernels for Block-Sparse Weights. https://s3-us-west-2.amazonaws.com/openai-assets/blocksparse/blocksparsepaper.pdf. (2017). [Online; accessed 12-January-2018].

[17] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. 2016. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168* (2016).

[18] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William J. Dally. 2017. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 75–84.

[19] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[21] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, Vol. 2.

[22] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[23] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. CondenseNet: An Efficient DenseNet using Learned Group Convolutions. *arXiv preprint arXiv:1711.09224* (2017).

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[25] Muhammad Mukaram Khan, David R Lester, Luis A Plana, A Rast, Xin Jin, Eustace Painkras, and Stephen B Furber. 2008. SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor. In *Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on*. Ieee, 2849–2856.

[26] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. (2014).

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[28] H. T. Kung. 1982. Why systolic architectures? *IEEE Computer* 15 (1982), 37–46. Issue 1.

[29] H. T. Kung and C. E. Leiserson. 1979. Systolic Arrays (for VLSI). In *Sparse Matrix Proceedings 1978*. Society for Industrial and Applied Mathematics, 256–282.

[30] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[31] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[32] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 1–9.

[33] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*. 2849–2858.

[34] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009* (2015).

[35] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning efficient convolutional networks

through network slimming. In *Computer Vision (ICCV), 2017 IEEE International Conference on*. IEEE, 2755–2763.

[36] Ilya Loshchilov and Frank Hutter. 2016. SGDR: stochastic gradient descent with restarts. *Learning* 10 (2016), 3.

[37] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342* (2017).

[38] Rick Merritt. 2018. ARM at Risk on AI Chip Marketc. *EE Times India* (April 2018).

[39] Sharan Narang, Eric Undersander, and Gregory F. Diamos. 2017. Block-Sparse Recurrent Neural Networks. *CoRR* abs/1711.02782 (2017). arXiv:1711.02782 http://arxiv.org/abs/1711.02782

[40] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. 2017. Scale-out acceleration for machine learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 367–381.

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[42] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.

[43] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 267–278.

[44] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–13.

[45] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[46] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, and Hans Peter Graf. 2009. A massively parallel coprocessor for convolutional neural networks. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*. IEEE, 53–60.

[47] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 535–547.

[48] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[49] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A pipelined ReRAM-based accelerator for deep learning. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 541–552.

[50] John V. Arthur Andrew S. Cassidy Rathinakumar Appuswamy Alexander Andreopoulos David J. Berg Jeffrey L. McKinstry Timothy Melano Davis R. Barch Carmelo di Nolfo Pallab Datta Arnon Amir Brian Taba Myron D. Flickner Steven K. Esser, Paul A. Merolla and Dharmendra S. Modha. 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. *National Academy of Sciences* (2016).

[51] Junsong Wang, Qiuwen Lou, Xiaofan Zhang, Chao Zhu, Yonghua Lin, and Deming Chen. 2018. Design Flow of Accelerating Hybrid Extremely Low Bit-width Neural Network in Embedded FPGA. *arXiv preprint arXiv:1808.04311* (2018).

[52] Shihao Wang, Dajiang Zhou, Xushen Han, and Takeshi Yoshimura. 2017. Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1032–1037.

[53] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. 2074–2082.

[54] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2017. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. *arXiv preprint arXiv:1711.08141* (2017).

[55] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 62.

[56] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 20.

[57] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 56.

[58] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. 2017. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 15–24.

[59] Michael Zhu and Suyog Gupta. 2017. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878* (2017).