An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-matrix Multiplication

MOHAMMADREZA SOLTANIYEH, RICHARD P. MARTIN, and SANTOSH NAGARAKATTE, Rutgers University, USA

This article proposes a novel hardware accelerator for the inference task with sparse convolutional neural networks (CNNs) by building a hardware unit to perform Image to Column (Im2Col) transformation of the input feature map coupled with a systolic-array-based general matrix-matrix multiplication (GEMM) unit. Our design carefully overlaps the Im2Col transformation with the GEMM computation to maximize parallelism. We propose a novel design for the Im2Col unit that uses a set of distributed local memories connected by a ring network, which improves energy efficiency and latency by streaming the input feature map only once. The systolic-array-based GEMM unit in the accelerator can be dynamically configured as multiple GEMM units with square-shaped systolic arrays or as a single GEMM unit with a tall systolic array. This dynamic reconfigurability enables effective pipelining of Im2Col and GEMM operations and attains high processing element utilization for a wide range of CNNs. Further, our accelerator is sparsity aware, improving performance and energy efficiency by effectively mapping the sparse feature maps and weights to the processing elements, skipping ineffectual operations and unnecessary data movements involving zeros. Our prototype, SPOTS, is on average 2.16×, 1.74×, and 1.63× faster than Gemmini, Eyeriss, and Sparse-PE, which are prior hardware accelerators for dense and sparse CNNs, respectively. SPOTS is also 78× and 12× more energy-efficient when compared to CPU and GPU implementations, respectively.

CCS Concepts: • Computer systems organization → Neural networks;

Additional Key Words and Phrases: Convolutional neural networks, hardware accelerators, systolic arrays, GEMM, sparse computation

ACM Reference format:
Mohammadreza Soltaniyeh, Richard P. Martin, and Santosh Nagarakatte. 2022. An Accelerator for Sparse Convolutional Neural Networks Leveraging Systolic General Matrix-matrix Multiplication. ACM Trans. Archit. Code Optim. 19, 3, Article 42 (May 2022), 26 pages.
https://doi.org/10.1145/3532863

1 INTRODUCTION

Inference tasks on edge devices. Neural networks are widely used in numerous domains such as video processing [22], speech recognition [8], and natural language processing [18, 37]. They have attained either near-human or better accuracy with many such tasks. To attain such accuracy, the training phase involves large datasets with several weight-update iterations, which can take...
several hours or days to complete. Hence, the training phase is typically performed in the cloud or on a large cluster of machines. In contrast to training, the inference task is performed both in the cloud and at the edge devices (e.g., mobile devices or in the context of the Internet of Things (IoT)). It is often desirable to compute on the edge devices, especially when network connectivity is either unavailable or limited. The edge devices typically have limited memory and compute resources with strict requirements on energy usage. Hence, this article focuses on designing an efficient hardware accelerator for convolutional neural networks’ (CNNs’) inference task targeting edge devices.

**Accelerating convolutional neural networks.** Among various neural networks, CNNs are widely used in many applications, such as image processing. CNNs can have multiple types of layers, including convolution layers, fully connected layers, and pooling layers, with the majority of the computation being performed in the convolution layers. A convolution operation involves sliding a smaller filter window over the input array with a stride size, producing patches (Figure 2). Each CNN layer has multiple features: the number of filters, kernel size, stride size, and channel size. Hence, designing an accelerator that performs well for all types of layers in a CNN is challenging given the diverse set of features. Further, supporting sparse inputs introduces additional complexity to the design.

**Some drawbacks of prior CNN accelerators.** Given the importance of CNNs in various applications, numerous CNN accelerators have been explored by the community [2, 5, 7, 9, 12, 14–16, 19, 26, 31, 33–35, 44]. Many designs are tailored to a particular CNN architecture [16, 43]. They suffer from low resource utilization for certain layer shapes and sizes. With respect to sparsity awareness, many prior approaches handle sparsity in either the weights [23, 44] or the input feature map [1, 2]. Many recent designs support sparsity in both the feature map and weights. As an example, SparTen [15] uses a costly prefix sum unit to locate the non-zero pairs that match. Their sparsity awareness method for finding the non-zero pairs contributes to 42% and 62% of the total area and energy, respectively. Sparse-PE [33] avoids the expensive hardware units for finding the non-zero pairs by decompressing the sparse vectors into a dense format before locating the non-zero pairs. However, this solution introduces an additional decompression step (i.e., zero insertion) and requires large buffer sizes inside each core to store the vectors in a dense format. Unlike the other two methods, SCNN [31] uses a Cartesian product method to avoid the index matching phase altogether. The main drawback of such an approach is that it introduces irrelevant partial products. Finally, while prior sparsity-aware accelerators, including SparTen and Sparse-PE, successfully prevent the ineffectual multiplications (multiplications involving zeros), they fail to avoid many unnecessary data transfers. The index matching phase requires the sparse vectors to be fetched by each processing element (core) even if they do not contribute to any output result (e.g., when they are not matched with any non-zero).

**Convolution as matrix multiplication.** One approach to implement CNNs is to realize a convolutional layer as a large, single General Matrix-Matrix Multiplication (GEMM) using a data reorganization transformation called Image-to-Column (Im2Col). Unsurprisingly, many mainstream frameworks use this approach since highly optimized GEMM primitives are available (e.g., BLAS [4] or CuBLAS [30]). One method to accelerate the convolution computation is to offload the GEMM operation to a hardware accelerator. However, the Im2Col operation accounts for a sizable fraction of the execution time (29% of the total time). Further, Im2Col performs many redundant memory accesses, which contributes to the overall energy consumption. Further, offloading only the GEMM operation to a hardware accelerator and doing the Im2Col operation in software prevents fine-grained pipelining of the Im2Col transformation and the matrix multiplication operation. Thus, performing the Im2Col operation in hardware avoids significant data transfer between the CPU and the hardware accelerator.
This article. We make a case for building a hardware accelerator that implements the convolution layer as a single large GEMM operation using Im2Col. Our accelerator for sparse convolutional networks, which we call SPOTS, performs Im2Col in hardware along with the GEMM operation. It effectively pipelines the Im2Col operation with the GEMM operation and eliminates redundant memory accesses. In addition, our design supports sparse weights and feature maps tailored for our GEMM and Im2Col pipeline. Finally, we achieve generality by supporting various CNN layers, such as fully connected and pooling layers, while maintaining high processing element (PE) utilization for various CNN layers.

A dedicated Im2Col unit in SPOTS. We propose a dedicated hardware Im2Col unit that operates in parallel with the hardware GEMM unit. This specialized Im2Col unit enables us to avoid redundant accesses with data reuse, significantly accelerating inference and improving energy consumption. A novel aspect of the Im2Col unit in SPOTS is that it has a collection of patch units (PUs) that streams the input only once, performs data reorganization, creates multiple patches in parallel, and eliminates redundant accesses. To eliminate redundant accesses, each patch unit in the Im2Col unit has three local buffers that identify overlapped elements between patches and avoid costly DRAM accesses. These patches are subsequently fed into a systolic-array-based GEMM unit.

A dynamically reconfigurable GEMM unit in SPOTS. The GEMM unit in SPOTS is efficiently pipelined with the Im2Col unit. The GEMM unit in SPOTS can be configured as multiple GEMM units with square-shaped systolic arrays with PEs or a single tall-thin unit. The tall-thin shape better balances the memory bandwidth requirement of the GEMM unit and the throughput of the Im2Col unit, which allows efficient pipelining of operations between the PEs performing the matrix multiplication and the PUs executing the Im2Col reorganization. This dynamic reconfigurability of the GEMM units enables SPOTS to achieve high PE utilization with various kinds of convolutional layers that differ in the number of filters, kernel size, stride values, and feature map dimensions. In addition to the convolution and fully connected layers, SPOTS supports pooling layers with a minor enhancement to the Im2Col unit.

SPOTS is sparsity aware. SPOTS efficiently handles zeros in both inputs: weights and the input feature map. Sparsity in weights results from the pruning step in CNNs. Pruning reduces the computation and memory footprint by eliminating weights after training without substantively changing network accuracy. SPOTS uses sparsity to skip data transfer and computation for sparse regions. Our new sparse format, tailored for our group-wise pruning, substantially reduces the storage requirement for the weights in comparison to random pruning techniques [16] while providing high bandwidth access to the weights necessary to keep all the PEs busy. Finally, SPOTS tags and skips blocks of zeros in the result of the Im2Col unit and weights before entering the systolic array, saving computation cycles and memory transfers. Further, this approach helps SPOTS avoid the potential load imbalance caused by an uneven distribution of the zeros in the inputs since the zero blocks are skipped for all PEs.

The three key innovations in our accelerator are (1) a novel Im2Col unit that allows it to pipeline GEMM and Im2Col computations to improve performance, (2) a dynamically reconfigurable GEMM unit with the capability to adapt to different CNN layers and shapes, and (3) sparsity awareness that allows the design to support sparsity in both the feature map and filters. These techniques combine to improve CNN performance and energy efficiency over prior accelerators. We evaluate our design for four popular CNNs, AlexNet, VGGNet, ResNet, and GoogleNet, which feature a diverse set of convolution layers with different memory and computation requirements. We compare the performance and energy efficiency of SPOTS with other state-of-the-art hardware accelerators for CNNs. Our results show that SPOTS is on average 2.16×, 1.74×, and 1.63× faster than Gemmini [14], Eyeriss [7], and Sparse-PE [33], respectively. SPOTS is also 78× and 12× more...
energy efficient when compared to CPU and GPU systems, respectively. In addition, we demonstrate that SPOTS can achieve high PE utilization under different CNN shapes. Finally, we show that our novel Im2Col unit improves the energy efficiency by 60% compared to an Im2Col unit that does not reuse the data.

2 BACKGROUND AND MOTIVATION

We provide background on CNNs, structuring the convolution operation as general matrix-matrix multiplication with the help of the Im2Col transformation, and leveraging sparsity in the inputs to improve performance and energy efficiency.

2.1 Convolution Neural Networks

A CNN consists of a series of layers. Each layer in a CNN extracts a high-level feature of the input data called a feature map (fmap). CNNs often have different layers, including convolution, activation (e.g., non-linear operator), pooling, and fully connected layers. The convolutional layers are the main layers in a CNN. They perform the bulk of the computation. Each convolution layer has several filters. The values of these filters (i.e., weights) are learned during the training phase. In the inference phase, the network classifies new inputs presented to the network.

Figure 1(a) visualizes the computation in the convolution layer. The input feature map is structured as a 3-D tensor with W, H, and C as its width, height, and the number of channels, respectively. Similarly, the filters are structured as 3-D tensors with width (R), height (S), and C channels. The filters and the input feature maps have the same number of channels. There are K filters in this example. Typically, a collection of N input feature maps are convolved with K filters (i.e., a batch size of N). For inference tasks, it is common to use a batch size of 1. For some convolution layers, a 1-D scalar bias is also added to the result, which is not shown in Figure 1(a).

One method to build a hardware accelerator for CNNs. The sliding-window nature of the convolution operation introduces overlaps between the patches. It makes the job of designing the hardware accelerator challenging because mapping the computation of a convolution operation to a set of PEs is more complex. One commonly used method is to design a fetch unit within each PE that fetches the input patch, communicates the patches with other PEs, and manages the partial results. A specialized interconnect is typically used to facilitate the communication between the PEs based on the specific dataflow. Prior works such as SCNN [31] and Eyeriss [7] adopt this approach. The main weakness of this approach is that the interconnection network and
dataflow are heavily customized for the convolution operation. Hence, both SCNN and Eyeriss can be inefficient for other layers, such as fully connected layers. For example, SCNN can achieve 25% of the peak throughput when used for the fully connected layers. Similarly, Eyeriss fails to achieve high PE utilization for small batch sizes.

2.2 Transforming Convolution to General Matrix-Matrix Multiplication

The convolution operation can be transformed into GEMM using the Im2Col transformation. To structure the convolution operation as matrix multiplication, we need to create two matrices from the two inputs of a convolution layer: input feature map and the K filters. Figure 2 illustrates how the two matrices are built. The product of these two matrices will be equivalent to the result of the convolution operation. For building the weight matrix, each filter is mapped to one row of the weight matrix. When there are K filters, there will be K rows in the weight matrix (Figure 2(a)). The number of columns in the weight matrix is \( R \times S \times C \). In contrast to the weight matrix, a more complex transformation is required to build a 2-D matrix from the original 3-D input feature map. This transformation is called Image to Column (Im2Col). The Im2Col result depends on the kernel size and the stride size, which are the two parameters of the convolution operation. In convolution, each filter slides across different positions in the input feature map. We call all elements in the input feature map covered by the filter as a patch or a tile. Patches are often overlapped with each other when the stride size is less than the filter size. This overlap results in the repetition of the same element of the input feature map in multiple patches. Figure 2(b) and Figure 2(c) illustrates the Im2Col transformation with an example filter of size \((3 \times 3 \times C)\) and a stride of 1. Each column of the matrix produced by the Im2Col transformation corresponds to one patch where the filter is applied for all \( C \) channels, and it has \( R \times S \times C \) rows. Figure 2 shows the patches for one channel. Finally, the product of the two matrices (Figure 2(a) and 2(c)) generates the output of the convolution operation.

**Benefits and challenges of convolution with Im2Col.** By using a separate Im2Col transformation, the task of building input patches and the eventual computation on them can be decoupled. The Im2Col transformation can identify data overlap among different patches as each filter slides across different positions in the input feature map. Further, a separate Im2Col transformation can enable one to use highly optimized primitives or even available hardware accelerators for GEMM. However, doing the Im2Col transformation in software may not provide the best possible performance for the following reasons. First, a naive Im2Col transformation can result in numerous redundant memory accesses. Sliding the filters over the input feature map creates numerous
repetitions in the Im2Col patches. Depending on the filter size and the stride size, the number of memory accesses can be 9× higher on average than the number of elements, which indicates that many elements are redundantly accessed multiple times. The Im2Col transformation can account for as much as 60% of the total execution time of the convolution operation. On average, the decoupled Im2Col transformation spends 29% of the overall execution time for various layers in AlexNet, VGGNet, and GoogleNet for a CPU system.

Section 3 describes our accelerator that performs Im2Col on the fly, extracts significant parallelism between various patches, and uses the hardware Im2Col unit to simplify the hardware accelerator for GEMM without the need for complex interconnection networks.

2.3 Sparsity Awareness in CNNs

Given a layer in the CNN, a significant fraction of the values in the weights and the feature map values can be zeros. A pruning step is often applied to remove unimportant and redundant weights during the training phase, resulting in zeros in the final trained weights. Unlike the zeros in the weights that are known after the training phase, the input feature map can also have zeros that are not known until the inference task.

The pruned weights can be compressed using a sparse format. In addition to reducing the model size, different hardware accelerators use sparsity to improve performance and energy efficiency of the design. The performance improvement comes from eliding multiplications and minimizing data movement when it involves zeros.

**Drawbacks of prior sparsity-aware designs.** Next, we review some drawbacks of the prior sparsity-aware designs.

**Redundant multiplications.** One challenge of a sparsity-aware design is to find non-zero pairs to multiply depending on their indices. SCNN [31] employs a Cartesian product method to skip the index matching step entirely. The Cartesian product of two vectors produces an output vector that includes the product of each element from the first vector with all the elements from the second vector. This all-to-all nature of a Cartesian product removes the need for an additional step to match the non-zeros in the two vectors. The major weakness of this approach is that it generates some unnecessary partial products during the multiplication phase that does not contribute to any final output.

**Expensive hardware to find the non-zero pairs to multiply.** To avoid redundant multiplications in SCNN, other proposals such as SparTen [15] and GoSPA [9] use an index matching step often termed as an intersection operation. This approach has two major drawbacks. First, the intersection step is often accomplished by using expensive hardware (e.g., prefix sum in SparTen). As a result, the intersection unit introduces significant area and energy costs to a sparsity-aware design. For example, in SparTen, the sparsity handling contributes to 42% and 62% of the total area and energy of the design, respectively. Second, the intersection step is in the execution’s critical path. Thus, multiplication units may experience frequent idle cycles waiting for the result of the intersection step.

**Extra decompression steps.** Some methods like Sparse-PE decompress the sparse data into a dense format before finding the non-zero pairs. Using a dense format helps them simplify the hardware unit for index matching. However, it introduces an additional decompression step (e.g., zero insertion step) and requires large buffer sizes to store the vectors in a dense format, which reduces the benefits of using a compressed format.

**Unnecessary data traffic.** Methods used in SparTen and Sparse-PE are successful in avoiding redundant multiplications that involve zeros as a result of the index matching step. However, both designs still produce unnecessary data traffic. The index matching phase requires the sparse
vectors to be fetched by each processing element (core) even when they do not contribute to the final result.

**Custom routing needed for partial products.** Many sparsity-aware accelerators such as SparTen, Sparse-PE, and SCNN have two separate units, one for multiplication (i.e., generating partial products) and one for accumulation (i.e., adding the partial product to generate the final output). Hence, this approach introduces extra complexity to the design because the partial products need to be routed from the multiplication units to the accumulation units.

In contrast to existing work on sparsity-aware designs, our goal with SPOTS is to eliminate redundant multiplications, minimize expensive hardware units, reduce metadata and the memory footprint, and reduce the data traffic resulting in both performance and energy improvements.

**Techniques for pruning filters.** Pruning is typically employed to increase the sparsity of the weights in the filters. There are two strategies for pruning: random pruning and structured pruning. The *random pruning* strategy sets a weight to zero if it is below a threshold value \[17\]. Typically after the pruning step, non-zero weights need to be stored in a compressed sparse format. However, using a sparse format involves indirect accesses and requires extra steps for extracting the non-zero elements and matching indices. In contrast, the *structured pruning* strategy addresses irregular accesses due to random pruning \[20\], \[23\], \[41\]. Structured pruning removes redundant weights only at well-defined locations or with specific block sizes.

Figure 1(b) shows pruning at different levels with various pruning methods. The dark points represent pruned weights in the filter. When we convert a 3-D filter to a 2-D representation using the strategy shown in Figure 2(a), the resulting zeros in the 2-D matrix are shown in the second row of Figure 1(b). The random pruning strategy results in an irregular pattern of zeros. A coarse-grained structure (e.g., channel-wise) for pruning can result in a group of zero columns in the 2-D matrix, which is more hardware friendly. However, it can sacrifice network accuracy. A fine-grained structure (e.g., shape-wise or group-wise) gets closer to the accuracy of random pruning while having a regular structure with zeros. We will describe the details of our group-wise pruning in Section 4.

### 3 SPOTS ARCHITECTURE

We design a hardware accelerator, SPOTS, for the inference phase that provides significant performance and energy benefits for CNNs with different layer characteristics using a GEMM-based formulation of a convolution operation. Our design goals are fourfold: (1) significant performance and energy benefits, (2) support multiple CNN layers and filters of varying sizes, (3) efficiency even with sparsity in the weights and the filters, and (4) fine-grained pipelining of the Im2Col operation with the GEMM computation.

We propose a hardware unit for the Im2Col transformation that is synergistic and pipelined with the hardware unit for GEMM. The Im2Col unit reads the input feature map, a 3-D array, and creates a set of linearized patches. The Im2Col unit consists of PUs where each PU is responsible for constructing a linear patch. As values are streamed in, the PU constructing the patch will forward overlapped elements to neighboring PUs. Once the PU collects all the values in a patch, it forwards in-order partial patches to the GEMM unit. This approach allows the Im2Col unit to read values from the input feature map only once and reuses the values avoiding redundant memory accesses.

We design a dynamically reconfigurable GEMM unit with a systolic-array-based design. It can be configured as a tall array to balance the work between Im2Col and GEMM computation. To maintain a high PE utilization with CNN layers with varying shapes, the GEMM units can be configured as small GEMM units (Section 3.4). This dynamic reconfigurability enables our hardware to adapt to CNN layers with varying dimensions and shapes. Further, it also helps with sparsity
awareness by enabling our design to detect and skip zeros in the input feature map (Section 3.3). Figure 3(a) shows the overall architecture of our accelerator. The two main components are the unit for the Im2Col transformation and the GEMM unit. They are connected by two buffers that allow effective pipelining of the operations between the Im2Col unit and the GEMM unit. The compress unit detects and skips the zero blocks in the feature map and weights before they are sent to the GEMM unit. Next, we describe the details of each component.

### 3.1 The Im2Col Unit

The Im2Col transformation creates a 2-D matrix from the 3-D input feature map, which reduces convolution to matrix multiplication (Section 2.2). The Im2Col transformation is challenging because it inherits a part of the complexity of convolution, has complex memory access patterns, and results in redundant accesses. We propose a distributed hardware structure consisting of a series of PUs to both accelerate Im2Col and minimize the number of accesses to the elements of the input feature map. The key insight in our Im2Col unit is to exploit the localities resulting from the overlap between the patches as we slide the filters across the input feature map both vertically and horizontally. Each PU is responsible for building one patch at a time. One of our design goals is to read the input feature map only once from SRAM. To accomplish this goal, each patch unit has small local buffers that store some values that will be useful for building future patches. The PUs are also connected using a ring network, which allows the PUs to communicate elements locally and avoid redundant accesses to the input feature map in SRAM. Figure 3(b) shows the overall architecture of our Im2Col unit that consists of three main components: input controller, PUs, and output controller.

The input controller reads the input feature map from SRAM and forwards them to the appropriate PUs. Apart from sending values from the input feature map to the respective PUs, the input controller maintains extra metadata for every scheduled patch. This metadata carries information about the position of the current patch. For some convolution layers, the stride size is the same as the kernel size. In those cases, there is no overlap between the patches. For those scenarios, the input control forwards its output directly to the output controller by skipping the PUs.

Our Im2Col unit has multiple PUs within it. The PUs are the main components of the Im2Col unit for generating patches. Figure 3(b) shows the internals of the PU. Each PU has three buffers: the new buffer, the neighbor buffer, and the reserved buffer. The new buffer (N) maintains the newly generated feature map, the neighbor buffer (L) stores the values from the neighboring patches, and the reserved buffer (R) holds the values that are not yet used for building the current patch. The PUs are connected in a ring network, which allows them to communicate elements locally and avoid redundant accesses.

Our Im2Col unit significantly reduces the number of accesses to the input feature map and minimizes redundant operations, thereby improving the overall efficiency and performance of the accelerator.
fetched element received from the input controller. The neighbor buffer (G) stores the elements received from the neighboring PU. The reserved buffer (R) stores some of the elements previously received at that PU in the previous rounds. We store the row and column indices (i.e., coordinates) along with the value for each element. The control unit within each PU manages the buffer and generates patches. It decides whether an element needs to be forwarded to the neighboring PU and whether it should be maintained in the reserve buffer for future use.

A unique identifier identifies each patch (i.e., row and column index of top-left element). The control unit in a PU uses the patch identifier, the filter size, and the stride size to determine which elements need to be (1) fetched from the input feature map, (2) forwarded to the neighboring PUs, and (3) stored in the reserve buffer for future rounds. For example, all elements need to be fetched from the input feature map when a PU processes the first patch in the first round.

All elements that are necessary for adjacent patches in a given round are provided by the neighboring PUs. A PU typically receives $K^2 - K \times S$ elements from the neighboring patches as long as it is not the first patch in a given round, where $K$ is the size of the kernel and $S$ is the stride size. We assign all patches that belong to the same column (i.e., column index of the top-left element) in different rounds to the same PU. Hence, the PUs also store some elements that may be useful to build patches in subsequent rounds in the reserved buffer. This procedure is repeated for all C channels in the feature map.

The total number of elements that are overlapped between the vertical patches for a given filter size is $C \times W \times (K - S)$, where $W$ is the width of the input feature map. This is the maximum data reuse that can be attained with the reserve buffer. Further, the width and the channel size are inversely proportional to each other. For example, the first few layers of a CNN often have a small number of channels that are wider. In contrast, the later layers of the CNN have larger channels of smaller width. Thus, a small reserve buffer can provide significant data reuse even for larger layers. When the number of overlapped elements between the vertical patches is larger than the size of the reserved buffer, the input controller skips the reserved buffer and fetches the element again from SRAM. In such cases, data reuse is restricted to horizontally adjacent patches. Finally, the output controller organizes patches formed by each PU and manages communications with the GEMM unit. It coordinates double buffering that enables the overlapped execution of the Im2Col unit and the GEMM unit.

Figure 4 illustrates the process of generating the patches using the PUs in our Im2Col unit. For example, PU1 receives four elements (A1, A6, A2, A7) from the input controller and stores them in the new buffer in step 1. Similarly, PU2 receives two new elements (A3, A8). PU2 will receive other elements from the PU1 in subsequent steps (i.e., step 2).

In summary, our hardware Im2Col unit provides two benefits: energy efficiency and performance. Accessing the smaller SRAM and performing integer operations (for computing on row and column indices) consumes significantly less energy than accessing DRAM and large SRAMs. Hence, our design provides significant energy benefits. Further, our distributed collection of PUs unlocks extra parallelism beyond parallelism among the channels, allowing multiple patches to be built simultaneously by different PUs in the Im2Col unit that boosts performance.

### 3.2 The GEMM Unit

Our hardware unit for accelerating GEMM is a systolic-array-based design. Unlike many prior proposals that use systolic arrays for GEMM [7, 23, 24], we add dynamic reconfigurability to the GEMM unit. The GEMM unit in SPOTS can be configured either as a tall-shaped systolic array (the height is considerably larger than the width) to maximize data reuse or as multiple GEMM units with square-shaped systolic arrays. Figure 5(b) shows our systolic-array-based design for GEMM with a tall array.
Fig. 4. Illustration of patch generation using the PUs in the Im2Col unit. We show an Im2Col unit with two PUs for exposition. (a) The input feature map with one channel. We show the sliding windows used to generate patches with a stride of 1. (b) The two PUs are interconnected by a ring network. (c) There are two rounds. Round 1 corresponds to patches belonging to the first row of sliding windows over the input feature map. Similarly, round 2 corresponds to patches belonging to the second row of sliding windows.

Fig. 5. Illustration of our GEMM unit. (a) Inputs to the GEMM unit. (b) A tall array for the GEMM unit. (c) Illustration of GEMM computation at various steps. We show the current inputs and the partial results computed till a step for each PE. We demonstrate the output-stationary attribute of our design.

There are two main benefits in using a tall systolic-array-based architecture for GEMM. First, one of the inputs of the GEMM unit comes from the Im2Col unit. Using a tall-shaped array reduces the memory bandwidth requirement for the input arriving from the Im2Col unit. Thus, we can attain high PE utilization in the GEMM unit with less throughput from the Im2Col unit. This helps us to build an Im2Col unit with fewer resources and memory bandwidth requirements. Second, the tall array helps our design to exploit sparsity in the output of the Im2Col unit to skip zeros and increase performance. As the width of the tall array is smaller than its height, fewer columns from the Im2Col transformation enter the systolic array at any instant of time, which increases the opportunity for detecting and skipping entire rows of inputs with zeros before entering the

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 3, Article 42. Publication date: May 2022.
systolic array. Thus, using a tall-shape array helps to simplify our mechanism to skip redundant computations involving zeros in the input feature map. We describe our techniques for handling sparsity in Section 3.3.

Our GEMM unit uses an output-stationary dataflow, where a given PE computes the final result by accumulating the partial products for a particular output element. This output-stationary dataflow ensures maximum reuse of the output data. Besides, with a tall array, SPOTS can attain high data reuse for the result of the Im2Col transformation (i.e., feature map input). More importantly, with output-stationary dataflow, there is no need for separate multiplication and accumulation units. This eliminates multiple levels of multiplication and addition and the routing logics between the two units (Section 2.3). Figure 5(a) shows the weight matrix from the filter and the output of the Im2Col transformation that forms the input to the GEMM unit. The values of the filter matrix enter the GEMM unit’s systolic array from left to right, while the result of the Im2Col unit enters the systolic array from top to bottom. Figure 5(c) shows the various steps and partial results computed in the GEMM unit. Our design is parameterizable with $M$ rows and $N$ columns in the systolic array. In our design, each row handles multiple rows of the filter matrix. Our specific prototype used 128 rows of PEs and 4 columns. These numbers are chosen based on the characteristic of common CNN layers. Further, each row of the systolic array can be assigned multiple rows of the filter matrix depending on the scheduling mode. The majority of layers in state-of-the-art CNNs have fewer than 512 rows of the filter matrix in each convolution layer.

Each PE has a single multiply-accumulate (MAC) unit that uses two 16-bit fixed-point inputs and accumulates the result in a 24-bit register. To handle multiple rows of the filter matrix, each PE has $K$ registers to compute the final result (e.g., in our design, we use $K = 4$). Each PE has three FIFOs. Two FIFOs are for each arriving input. The other FIFO works as the work queue for the MAC unit. In GEMM, the coordinates of the elements of the two input matrices should match before multiplying the inputs. In the fetch unit, we ensure that the inputs are sent to the PEs in the proper order; thus, we do not need additional logic to perform index matching inside a PE. Additionally, our output-stationary dataflow ensures all the partial products produced in a PE belong to the same output element. Next, we describe how to support sparsities in both inputs without requiring any index matching units inside the PEs.

### 3.3 Handling Sparsity in CNNs

Most CNNs have sparsity in both filters and the input feature map. Figure 9 quantifies the amount of sparsity (percentage of zeros when compared to the total number of elements) for the commonly used CNNs. We use structured sparsity learning (SSL) [41] as our pruning method that is further enhanced with optimizations to suit our hardware design better (Section 4). We propose a new sparse format to store the pruned weights. Our sparse format delivers high bandwidth access to the filters necessary to keep the PEs in a tall systolic array active. Our sparsity-aware design identifies and skips the zeros on the fly and in block granularity. Skipping the zeros as blocks instead of individual elements helps to simplify our sparsity-aware design. SPOTS detects and skips the zero blocks in the input controller and outside the PEs. Thus, we avoid using costly hardware units for index matching or introducing any redundant multiplications (Section 2.3).

**Our sparse format for filters.** Once the weights for the filters are learned during the training phase, we divide the weights into blocks. The block size is equal to the group size used for pruning, which is a design parameter. Logically, the filter matrix will be a 2-D matrix of blocks when viewed in the dense representation. To minimize the memory footprint for storing filters during inference, we convert them into a sparse representation that is aware of the number of SRAM banks in the design. Our sparse format uses three arrays to store the pruned weights compactly. Figure 7(a) shows our custom sparse format. We store all non-zero blocks separately in one array (Array A)
Fig. 6. Comparing our custom sparse format with other state-of-the-art sparse formats. We used a matrix with 1,632 rows and 36,548 columns. We assume the values are 2 bytes. The last bar shows our sparse format. 

Our sparse format is independent of the density of the zeros in the matrix, and the size of metadata is less than 1 MB across all the density ratios. We compare our sparse format with CSR, RLC-4, Bitmap, and DCSR in the following order. The size of the sparse metadata is shown at the top of each bar in megabytes.

that is distributed in multiple banks based on the row index of the block (i.e., vertical position in the filter matrix). We use two bitmap arrays M1 and M2 to store the metadata. The bitmap array M1 encodes whether a column has any non-zeros in the filter matrix. A zero in the bitmap array M1 indicates an empty column. The bitmap array M2 maintains whether a block in a non-zero column is non-zero. A zero in M2 indicates the corresponding block is zero (i.e., as a block is a collection of values, it implies that all values in the block are zeros). These three arrays of our sparse format (i.e., A, M1, and M2) are distributed across the various banks of the SRAM so that the input controller for the GEMM unit can access them in parallel.

Figure 6 compares the memory footprint of our format with some of the most commonly used sparse formats in prior work. Unlike sparse formats like run-length encoding (RLC), CSR, and DCSR, our format does not require additional storage to keep the count of the non-zeros (e.g., RLC) or data pointers (e.g., row pointer in CSR). Thus, the metadata size for our proposed sparse format is independent of the sparsity of filters, and it only depends on the total number of blocks in the weight matrix. Another important benefit of our sparse format over index-based sparse formats such as CSR is that it allows the non-zeros of a column to be stored in multiple banks. Each bank can be processed independently and in parallel. Index-based sparse formats like CSC, CSR, and DCSR do not have this feature since they only maintain the beginning of a column (CSC) or a row (CSR and DCSR). According to Figure 6, our sparse format outperforms other sparse formats for various density ratios. Compared to other bit-mask sparse formats like the one used in SparTen or Sparse-PE, our sparse format needs nearly $8 \times$ less metadata by using the mask bits in a more coarse-grained fashion (i.e., block level).

In summary, using a structured pruning method together with a proper sparse format enables SPOTS to gain a meaningful advantage over other sparse formats in storing the pruned weights. This can directly translate into energy consumption savings since the memory accesses (including both SRAM and DRAM accesses) are the main contributors to the overall energy consumption, as previous studies show [7].

Skipping zeros in the feature map and weights. The compress component before the GEMM unit in our accelerator (Figure 3(a)) identifies a block of zeros in the result of the Im2Col transformation. It creates a bitmap for every block coming out of the Im2Col unit. If all elements in a block in the output of the Im2Col unit are zeros, the bit is set to zero for that block; otherwise, the bit is set to one. Subsequently, the input controller of the GEMM unit uses this bitmap and M1 level bitmaps for the weights (Figure 7(a)) to skip blocks of the input feature map and weights on the fly when they are all zeros.

One unique feature of our approach is that we skip MAC operations involving zeros outside the PEs and in the input controller. These have two advantages. First, we avoid the unnecessary
An Accelerator for Sparse CNNs Leveraging Systolic General Matrix-Matrix Multiplication

Fig. 7. (a) Our custom sparse format to store filters. (b) Illustration of how our design skips rows and columns with all zeros. (1) Weight matrix with the metadata about columns with all zeros. (2) The Im2Col result with the metadata about rows with all zeros. (3) If a row or a column is all zeros, all such rows and columns can be skipped (i.e., and operation of the row and column metadata). (4) GEMM computation when rows and columns are skipped. For example, the first element of column C4 will be fetched by the first PE in cycle 2 (skipping columns C2 and C3).

The PEs will then ignore the blocks with all zeros, and the MAC units are gated to reduce energy consumption.

Finally, we highlight the role of the tall systolic array in our on-the-fly detection of the non-zero blocks in the feature map. A tall systolic array limits the number of elements entering the systolic array to blocks with a small number of elements (e.g., blocks consist of four elements in our prototype). Small block sizes increase the possibility of detecting blocks that include only zero elements and are easier to skip. The ⋆ marker in Figure 9 indicates the percentage of zeros in the output of the Im2Col transformation that is skipped on the fly with this technique.

3.4 Handling Various CNN Layers/Shapes

CNNs have multiple layers that can be of different shapes and sizes. With a fixed configuration of hardware PEs, they can be underutilized for some layers, shapes, and/or sizes. Each filter forms a row of the weight matrix that is assigned to a distinct row of the systolic array. When the GEMM unit is configured as a tall systolic array and the number of filters is relatively smaller than the systolic array’s height (e.g., 128), some PEs will remain unused.

Dynamic reconfigurability of the GEMM unit enables us to support CNN layers with different attributes (Figure 8). Specifically, the PEs in the GEMM unit can be configured either as one tall array or multiple small arrays. Each such configuration has the same number of columns. This

ACM Transactions on Architecture and Code Optimization, Vol. 19, No. 3, Article 42. Publication date: May 2022.
enhancement allows our design to be more adaptive to different layer shapes and thus maintains high PE utilization under different conditions. Figure 8(a) demonstrates how a tall array can be used as two smaller arrays using the multiplexers. Hence, the PEs now either can receive the input from the PEs above (i.e., it forms a tall array) or can get the input from a different Im2Col unit. These multiplexers can be configured based on the mode register dynamically depending on the structure of a layer. The weight matrix is broadcast to all small systolic arrays when the GEMM unit is configured as smaller systolic arrays. Each small GEMM unit receives the feature map input from their assigned Im2Col units. The two GEMM units compute two independent groups of columns of the final result matrix (i.e., GEMM 1 computes result columns from 0 to N/2, GEMM computes the columns from N/2+1 to N). In our prototype, we have four Im2Col units. There is one main Im2Col and three smaller Im2Col units to support the two configurations. The main Im2Col unit is used for the tall array configuration. For the other configuration, all four Im2Col units are being used. This dynamic reorganization of the GEMM unit’s systolic array coupled with the multiple Im2Col units enables our hardware to maintain high PE utilization for various CNN layers with different shapes.

Supporting fully connected layers. Most CNNs have one or more fully connected layers at the end of the network. The inputs to the fully connected layers are the matrix weights learned during the inference and the output feature map resulting from the final pooling or convolutional layer that is flattened to a vector. With a batch size of 1, the computation for a fully connected layer is equivalent to matrix-vector multiplication. By increasing the batch size, we can structure it as a matrix-matrix multiplication operation. As we use a tall array, the batch sizes need not be large to utilize the whole array of PEs fully (e.g., can be as small as 4).

Supporting pooling layers. The pooling layers help to summarize the features generated by a convolution layer. There are two common types of pooling layers: max pooling and average pooling. Among them, max pooling, which picks the maximum element from a feature covered by the filter, is more common. Similar to convolution layers, the pooling layer has two parameters, filter size and stride size. We support the pooling layer by adding the pooling operation (e.g., MAX) to the output of the PUs in the Im2Col unit.

3.5 Strategies to Improve Load Balance in SPOTS

Load imbalance happens in sparse CNNs due to the uneven distribution of the non-zeros in weight and feature map inputs. The choice of the dataflow and the data reuse strategies determine the
source of the load imbalance in an accelerator. Generally, accelerators adopt either an input-stationary or an output-stationary dataflow. Subsequently, an input-stationary dataflow can be weight stationary or feature map stationary. In input-stationary dataflow, one of the inputs is held stationary in the PEs while the other input is broadcast to each PE to ensure data reuse. When there is an uneven distribution of non-zeros in the inputs, some PEs may receive fewer inputs, forcing them to remain idle until the other PEs process their inputs before they all can receive new inputs.

SPOTS adopts an output-stationary dataflow with a tall systolic array (Section 3.2). In a tall systolic array, the feature map values are passed through as many PEs as possible to ensure maximum data reuse. As described in Section 3.3, we skip the zeros in the feature map input inside the input controller before entering the systolic array. Thus, the non-zeros are skipped for all PEs (not just for an individual PE) in the systolic array. SPOTS’s early zero detection approach avoids the potential load imbalance caused by the uneven distribution of non-zeros in the feature map. Similarly, SPOTS detects and skips the zeros in the weights outside the PE when the zeros span the whole filters (i.e., an entire column of the weight matrix).

For partially zero columns in the weight matrix (i.e., some blocks are zeros, some non-zeros), some PEs may receive a zero block while others receive a non-zero block. This can introduce a work imbalance between the PEs. The load imbalance among the PEs can be quantified using the metric proposed by [11] where the load imbalance is quantified as follows:

\[
\text{imbalance\_percentage} = \frac{\text{maximum\_work} - \text{average\_work}}{\text{maximum\_work}} \times \frac{n}{n - 1}.
\]

(1)

The imbalance percentage corresponds to the percentage of time the PEs with less work are not engaged in useful work and are waiting for the PE with the maximum work. A perfectly balanced work distribution results in zero imbalance percentage. Thus, lower imbalance percentage implies fewer idle cycles for the PEs.

One way to improve the load balance in the PEs is to rearrange (shuffled) the non-zero blocks in the weights offline to make the distribution of the non-zero blocks more balanced. However, this reshuffling can change the position of the output channels and thus requires an additional step to reorder the output before the next layer uses them [15, 23]. In Section 5 we present the average imbalance percentage for all four CNN architectures with SPOTS. Since the PEs in SPOTS did not suffer from load imbalance, we did not use any additional load balancing steps to avoid introducing extra complexity to the design.

4 EXPERIMENTAL METHODOLOGY

We built a prototype of our design in Verilog and synthesized it using the Synopsys Design Compiler with the FreePDK 45nm technology [39]. Our design achieves a maximum of 500 MHz frequency. FreePDK 45 does not include SRAM cells. Thus, we separately model the area and power of SRAM/DRAM using Cacti 7.0 [3]. Table 1 provides the parameters of the SPOTS prototype and the area breakdown for different components. We perform cycle-accurate simulation of the RTL model of SPOTS in Verilog using Verilator. We used the traces from the RTL simulation and estimated the power consumption of our design with Synopsys’s PowerPrime tool. During our simulation, we executed each layer at a time. The pruned weights are preprocessed and are provided in our proposed sparse format. For the input feature map, we extracted each layer’s data from the models in Caffe. We also developed additional infrastructure to perform fast design space exploration and to collect statistics.

**CPUs, GPUs, and other ASICs used for our evaluation.** We compare our prototypes with CPUs, GPUs, and other ASICs. We use Caffe to evaluate various CNN architectures on a modern
Table 1. System Configurations for SPOTS, CPU, and GPU

(a) SPOTS design parameters and area

| Unit              | Size | Area (mm²) |
|-------------------|------|------------|
| GEMM              |      | 2.048      |
| #PE units         | 512  |            |
| Multiplier width  | 16 bits |          |
| Accumulator width | 24 bits |          |
| Systolic array configurations | one (128 × 4) | |
|                   | four (32 × 4) | |
| PE’s local buffers| 2 KB  |            |
| Im2Col            |      | 1.137      |
| #PU units         | 4    |            |
| Reserved buffers  | 32 KB |          |
| Other SRAM buffers| 2 MB  |            |
| On-chip memory    |      | 5.428      |
| Filter SRAM       | 1 MB  |            |
| Fmap SRAM         | 512 KB |           |
| SPOTS total       |      | 8.611      |

(b) CPU and GPU configurations

| Platform            | Number of cores | Frequency | Main memory | Technology | Cache |
|---------------------|-----------------|-----------|-------------|------------|-------|
| Intel Xeon E5-V3    | 4               | 3 GHz     | 32 GB DDR4 (2666 Mhz) | 22 nm | 10 MB Smart Cache |
| Titan X Pascal      | 3,584           | 1.53 GHz  | 24 GB of GDDR5 | 16 nm | -     |

CPU and GPU. The details of the CPU and GPU that we use for the evaluation are shown in Table 1. The CPU and GPU we used in our experiments are manufactured with 22 nm and 16 nm cell technology, compared to 45 nm technology used for SPOTS. The Caffe framework uses IntelMKL for CPU computation and Nvidia’s CUDA library, cuSparse, for GPU computation. Similar to our design, Caffe adopts a Im2Col + GEMM approach for doing convolution layers. We measured the energy consumption of the XEON CPU using the Processor Counter Monitor (PCM). For GPU, we measured the power consumption with the NVIDIA System Management Interface (Nvidia-smi) that queries the power using the built-in sensors. According to NVIDIA, the reported data is accurate (i.e., within ± 5 Watt).

Eyeriss. We use Eyeriss [7], which is an ASIC designed for accelerating sparse CNNs, to compare against our design. We measure the performance of Eyeriss using the publicly available simulator [13]. Eyeriss chip is fabricated at 65 nm CMOS and operates at 200 MHz clock frequency. Since we used a different cell technology (i.e., 45 nm) for SPOTS, we assume that the frequency of Eyeriss to be exactly equal to the frequency of SPOTS when we report the execution time. We also configured Eyeriss to use the same number of MAC units and on-chip memory as SPOTS.

Gemmini [14] is a recent open-source full-stack DNN accelerator generator. Similar to SPOTS, the core unit in Gemmini is a systolic array composed of PEs. Each PE performs dot products and accumulations. The PEs read the data from a local, explicitly managed scratch-pad of banked SRAMs. We failed to build a design with an exact total number of PEs as SPOTS. Thus, we used tiles with 32 × 32 PEs for Gemmini, which translates to a total of 1,024 MAC units, which is 2× more MAC units than our prototype. We also set the on-chip memory for Gemmini to match the on-chip memory used for SPOTS.

Sparse-PE is a recent hardware accelerator that supports sparse input for both feature maps and weights. Sparse-PE consists of a set of cores. Each core reads the inputs in a compressed form and performs three operations (i.e., selection, computation, and accumulation) to generate the final result. We model their accelerator using the cycle counts and sparsity ratios reported in their paper. Their design natively supports CNN layers with a kernel size of 3. For other kernel sizes, they perform kernel factorization. Besides, they only report the sparsity of the layers in AlexNet and VGGNet. Thus, we only compare our performance with Sparse-PE for those two networks. Finally, we assumed SPOTS and Sparse-PE have the same number of multiply units. This provides some advantage to Sparse-PE since their design requires additional units for the accumulation part.
Table 2. Network Characteristics, the Top1 and Top5 Result Accuracy, and the Overall Sparsity for the Original (with No Pruning), Random Pruning, and Our Structured Pruning Using the Imagenet Dataset

| Model     | #conv | Max. Layer | Baseline | Random Pruning | Our Structured Pruning |
|-----------|-------|------------|----------|----------------|------------------------|
|           | Layer | Weight     | Top 1 (%)| Top 5 (%)      | Top 1 (%)             | Top 5 (%)             | Sparsity (%) |
| AlexNet   | 5     | 2.5 MB     | 56.81    | 79.95          | 56.75                 | 79.28                 | 63.1         |
| VGGNet    | 13    | 4.5 MB     | 68.27    | 88.36          | 68.21                 | 88.25                 | 62.8         |
| GoogleNet | 57    | 1.3 MB     | 68.92    | 89.14          | 68.42                 | 88.85                 | 62.5         |
| ResNet    | 53    | 4.5 MB     | 72.71    | 90.66          | 72.4                  | 90.58                 | 60.51        |

Weights and activations assume a data-type size of 2 bytes.

Fig. 9. Sparsity in the filters and input feature maps for AlexNet, VGGNet, ResNet, and GoogleNet. The ★ marker indicates the percentage of zeros in the output of the Im2Col transformation that is skipped on the fly by our design.

CNN architectures and pruning. We used four widely used CNN architectures: AlexNet [22], VGGNet-16 [37], GoogleNet [40], and ResNet-50 [18] to evaluate our prototype. Throughout the article, we refer to VGGNet-16 and ResNet-50 as VGGNet and ResNet, respectively. These four CNN architectures vary in the number of layers, layer types, and sizes, as shown in Table 2. We used a batch size of 1 for all of our experiments, which is the standard usage mode for an inference task. We used the input images from Imagenet [10], a widely used dataset for image classification tasks, to train the networks. We pruned all four networks using the pruning algorithm based on SSL [41]. SSL is generic and can be applied on different levels, including filters, channels, and shapes. We applied SSL at the shape level. As our hardware exploits sparsity at a much finer granularity than a shape, we optimize SSL by pruning in a more fine-grained fashion. Specifically, we zeroed the weights that are below the threshold in some but not all elements of a shape. This generates zero blocks of a certain size (i.e., the number of filters in the group). Figures 1(b) and (d) show our group-wise pruning. Figure 9 reports the sparsity in the weights and input feature map after pruning for the layers of various CNN architectures. It shows that sparsity varies across both layers and networks. Finally, we retrained the pruned network to regain its accuracy, which is the norm with pruning. Table 2 summarizes the accuracy and overall sparsity percentage for baseline (with no pruning), random pruning [17], and our structured pruning method. For accuracy results, we report the top 1 (i.e., the first prediction is the correct result) and top 5 (i.e., the correct result is in the first 5 predicted values). Our pruned networks are within 1% to 2% accuracy of the original model without pruning. A structured pruning method achieves the same accuracy as a randomly
pruned network while having about $2 \times$ less sparsity than a randomly pruned network. However, as was shown in Section 3, our structured pruning method simplifies the sparsity awareness design significantly. Besides, our sparse format outperforms the sparse format used by designs with random pruning (see Figure 6).

5 EXPERIMENTAL EVALUATION

We demonstrate the performance and energy efficiency of SPOTS in comparison to other ASIC designs for CNNs, including Eyeriss, Gemmini, Sparse-PE, and general-purpose CPU and GPU implementations.

Performance of SPOTS compared to other ASIC designs. Figure 10 reports the speedup of SPOTS, Eyeriss, and Sparse-PE relative to Gemmini for all four CNN architectures. The layers are sorted for each CNN architecture based on where they appear in the network (top, middle, or bottom). Figure 10(a) reports the speedup for all layers in AlexNet. On average, SPOTS is almost $2 \times$ faster than Eyeriss and Gemmini. SPOTS is nearly $4 \times$ faster than Eyeriss and Gemmini for the layers in the middle, where the sparsity ratio in the two inputs (e.g., weights and feature map) is higher. In addition to the sparsity awareness that gives SPOTS an advantage over Eyeriss and Gemmini, the layers in the middle and bottom have more filters that favor a tall systolic array. For Sparse-PE, we only measure the layers with a kernel size of 3 (see Section 4). SPOTS is almost $1.8 \times$ faster than Sparse-PE for the measured layers.

Figure 10(b) reports the speedup for VGGNet. On average, SPOTS is $1.85 \times$ and $1.86 \times$ faster than Eyeriss and Gemmini, respectively. Similar to AlexNet, SPOTS achieves higher speedup with layers with more sparsity. SPOTS performs slightly worse than Eyeriss for the first two layers, where the number of filters is relatively small, and the inputs are dense (Figure 9). Later in this section, we will demonstrate the connection between the number of filters in a layer and the PE utilization. Compared to Sparse-PE, SPOTS is on average $1.6 \times$ faster. SPOTS has better performance than Sparse-PE because of its data-reuse strategies and efficient methods to skip zero elements without the need for zero insertion and selection operations.
Table 3. Comparing the Performance and Efficiency of SPOTS with Different ASIC Designs for Sparse CNNs

| Accelerator | SCNN [31] | NullHop [1] | SparTen [19] | SPOTS |
|-------------|-----------|-------------|--------------|-------|
| Pruning Method | Random | N/A | Random | Structured |
| Bitwidth | 16 | 16 | 16 | 16 |
| Number of Multipliers | 1,024 | 1,024 | 1,024 | 312 |
| Clock Frequency (MHz) | 800 | 400 | 800 | 500 |
| Core Area (mm²) | 22.21 | 10.12 | 24.51 | 8.61 |
| Throughput (Inference/s) | | | | |
| VGGNet | 37.55 | 10.96 | 60.09 | 15.21 |
| AlexNet | 479.92 | N/A | 767.88 | 249.79 |
| Normalized Throughput (Inference/s) | | | | |
| VGGNet | 2.93 | 6.85 | 3.75 | 15.21 |
| AlexNet | 149.97 | N/A | 239.96 | 249.79 |
| Power Efficiency (GOPS/Watt) | | | | |
| VGGNet | N/A | 1.357 | 440.84 | 469.33 |
| AlexNet | N/A | N/A | 326.62 | 446.91 |
| Area Efficiency (Inference/s/mm²) | | | | |
| VGGNet | 1.69 | 1.08 | 2.45 | 1.76 |
| AlexNet | 21.61 | N/A | 31.32 | 29.01 |

All designs are scaled to 45 nm technology.

Figure 10(c) shows the speedup of SPOTS over Eyeriss and Gemmini for ResNet. SPOTS is 1.77× and 2.66× faster than Eyeriss and Gemmini on average for ResNet. SPOTS is up to 8× and 13× faster than Eyeriss and Gemmini for the layers where the weight and feature map sparsity are high. Similar to VGGNet, SPOTS performs slightly worse or similar to Eyeriss for the first eight layers in ResNet because the first few layers in ResNet have a few filters per each layer. Hence, PEs are underutilized compared to layers in the middle or at the end of the network. Figure 10(d) shows for GoogleNet that SPOTS is 1.38× and 1.91× faster than Eyeriss and Gemmini, respectively. In contrast to other CNN architectures, GoogleNet has a few convolutional layers at the beginning with a small number of filters that do not favor our tall array. Thus, overall, SPOTS enjoys less speedup for GoogleNet compared to the three other networks.

Table 3 compares the throughput, area efficiency, and power efficiency of SPOTS with some of the prior sparse CNN accelerators. All the accelerators are scaled to 45 nm technology. We reported the exact number of multipliers as reported in each paper. The number of multipliers in all three designs is twice the number of multipliers used in the SPOTS prototype. The clock frequencies for SCNN and NullHop are scaled to 45 nm as in prior work [21]. The theoretical throughput varies for each design depending on the clock frequency and the number of their MAC units. We use the number of inference tasks completed per second to compare the throughput. Table 3 shows the achieved throughput for each design as well as the normalized throughput. When the throughput is normalized (to have the same theoretical throughput as SPOTS), SPOTS outperforms all the other three accelerators. We used the achieved Giga operations per second (GOPS) per watt to compare power efficiency. SCNN does not report the power consumption. SPOTS outperforms SparTen in power efficiency for both AlexNet and VGGNet. However, NullHop achieves the highest power efficiency despite delivering less overall throughput than SPOTS. Finally, in area efficiency, SPOTS is better than SCNN and NullHop and comparable to SparTen.

**Performance comparison with CPUs and GPUs.** We evaluate the performance and energy efficiency of SPOTS in comparison to execution with CPUs and GPUs. Figure 11(a) reports the speedup of SPOTS for the convolution layers over the CPU implementation. SPOTS has 5×, 20×, 6×, and 8× speedup over the CPU implementations using Intel MKL for AlexNet, VGGNet, GoogleNet, and ResNet, respectively. SPOTS attains this speedup while operating at a frequency almost 6× less than the CPU. Figure 11(a) also shows the speedup of GPUs for the convolution layers over the CPU implementation. Compared to GPU, SPOTS is about 2× slower than GPU for AlexNet and VGGNet, while it performs slightly better or similar to GPU for GoogleNet and ResNet. VGGNet and AlexNet layers are relatively larger than the other two networks, resulting in larger matrices that favor the GPU with abundant MAC units compared to SPOTS. The second bar (Figure 11(a))
Fig. 11. (a) Speedup with SPOTS, GPU, and GPU implementations with the normalized number of MAC units over the CPU implementation as the baseline. (b) The energy efficiency of SPOTS and GPU implementations compared to a CPU baseline.

Fig. 12. (a) MAC utilization (i.e., active cycles) for different filter sizes. (b) Comparing PE utilization of SPOTS and tall systolic array (i.e., active PEs) for different filter sizes.

shows the GPU performance when its number of MAC units is normalized to the number of MAC units in SPOTS. For the normalized number of MAC units, SPOTS outperforms the GPU on average by 6×. Finally, some prior work observed that the performance degrades for CPUs and GPUs when the sparse features are used when the networks are pruned randomly. However, we observed that using our structured pruning helps the CPU and GPU implementations to attain higher overall performance using sparse linear algebra kernels.

Energy efficiency compared to CPUs and GPUs. Figure 11(b) demonstrates the energy efficiency of SPOTS and GPU implementations when compared to a CPU baseline for four CNNs. We did not include Gemmini energy results since their tool does not report the power consumption. The energy results include the off-chip memory accesses in this data. Our accelerator consumes 78×, 12×, and 1.4× less energy than a CPU, a GPU, and Eyeriss, respectively.

Sensitivity to shapes of various layers. Widely used CNN networks vary in the depth and the number of filters used in each layer. Even within a CNN, the layer shape and filter sizes can vary significantly. The dynamic reconfigurability in SPOTS provides flexibility to use the GEMM unit as a tall systolic array or as multiple small systolic arrays, which allows it to adapt to various shapes and filter sizes. When the filters are small (e.g., less than 128), the GEMM unit is configured as multiple small systolic arrays, which use different Im2Col units. All the PEs in the systolic array are active 100% of the time for all filter sizes other than 16 (see Figure 12(b)). In contrast, a tall systolic array without the enhancement we proposed in Section 3.4 fails to achieve full PE utilization for smaller filter sizes, as Figure 12(b) shows. Figure 12(a) shows the utilization of the multiply-accumulate units in the PEs of the systolic array (i.e., active cycles) when the layer has a specified number of filters (i.e., x-axis reports the size of the filter). When the filter size increases, we assign more rows to a PE, which can fetch up to four elements per read operation. Hence, there
An Accelerator for Sparse CNNs Leveraging Systolic General Matrix-Matrix Multiplication

Fig. 13. (a) The reduction in energy consumption for the Im2Col unit of SPOTS over the baseline design. (b) Speedup with SPOTS over the software-based Im2Col as the baseline Im2Col design with no data reuse. (c) Fraction of the work performed by the Im2Col unit when compared to GEMM (i.e., GEMM bar is 100%). We report the average and the median for the Im2Col’s work. When the mean exceeds the median, there will be instances where the Im2Col does more work compared to GEMM for some layers. (d) The load imbalance percentage in the pruned weights for AlexNet, VGGNet, ResNet, and GoogleNet are based on the metric defined in Equation (1).

There are more opportunities to keep the multiply-accumulate units in the PE active (i.e., almost 80% active cycles).

**Amount of work performed by Im2Col and GEMM units in SPOTS.** As the Im2Col and the GEMM units are pipelined in SPOTS, ideally, the work done by the Im2Col unit and the GEMM unit should be balanced. Figure 13(c) shows the relative percentage of cycles where the Im2Col and GEMM units are active relative to the GEMM unit for the four CNN architectures. As we report the active cycles relative to the GEMM unit, the bar for the GEMM unit is 100%. The average work performed by the Im2Col unit and the GEMM unit are almost similar for AlexNet and ResNet (i.e., the work is balanced). In contrast, GEMM dominates the total work in VGGNet. This data suggests that adding more PEs to the GEMM unit may improve the overall execution time for VGGNet. As the Im2Col unit is inactive due to low bandwidth with AlexNet and ResNet, adding more PEs without increasing the bandwidth will not improve performance.

**Energy efficiency from data reuse in the Im2Col unit.** One of the key ideas in the Im2Col’s patch unit is to read the input feature map only once from the SRAM and reuse the data with the help of local buffers. Figure 13(a) reports the percentage decrease in energy consumed by using local buffers to reuse the data in the patch units compared to a naive version of Im2Col that accesses SRAMs multiple times without any data reuse. On average, the mechanisms that we added to reuse the input feature map in the patch units result in the Im2Col unit consuming 60% less energy when compared to the Im2Col unit without such reuse.

**Comparing SPOTS with software Im2Col.** SPOTS has a hardware Im2Col unit that performs the Im2Col transformation on the fly. Figure 13(b) compares the speedup of using a hardware Im2Col unit compared to a software-based Im2Col as the baseline. For the baseline system, the
hardware only performs GEMM, while the CPU executes the Im2Col. The figure also shows the ideal situation for a software-based Im2Col design where the software Im2Col and the hardware GEMM computations are overlapped. Even when we provide an ideal scenario for the software Im2Col, SPOTS outperforms the ideal software-based Im2Col. On average, SPOTS outperforms the baseline software Im2Col by 2.3×, which shows the benefits of our hardware Im2Col unit.

Load imbalance in SPOTS. We evaluate the load imbalance in SPOTS introduced by the non-uniform distribution of zero blocks in the pruned weights. We used the metric defined in Equation (1) to quantify the load imbalance percentage among the PEs for the four studied sparse CNNs. For our analysis, we discarded all the layers whose sparsity ratio is below 5%. A lower imbalance percentage indicates less idle time for the PEs due to the uneven distribution of the non-zero blocks. All CNNs experience a very low load imbalance (less than 20%). The load imbalance for VGGNet and GoogleNet is as low as 4%, which highlights that the load balancing strategies in SPOTS have been useful.

6 RELATED WORK

There is a large body of literature on using custom hardware accelerators to improve the performance and energy efficiency of neural networks [2, 5, 7, 9, 12, 14, 16, 31, 33, 34, 38, 42, 44]. Table 4 qualitatively compares SPOTS with more closely related work. The table shows that SPOTS supports various operations in CNNs, is adaptive to layers of different shapes with high PE utilization, and efficiently supports sparsity in both the feature map and the weights.

Support for sparse inputs. Prior work has improved energy efficiency by supporting sparse inputs during inference. Cnvlutin [2] exploits sparsity in the input feature map to skip multiplication operations and to avoid data movement with zero elements. CambriconX [44] supports sparsity in the weights. Similar to our work, SCNN [31] and CambriconS [45] support sparsity in both the feature map and the weights to improve energy efficiency and performance. Prior work also uses data gating techniques to reduce the power consumption when the operands are zeros [7, 34]. While this technique is effective in reducing power consumption, it does not reduce the number of effective operations. Similar to SPOTS, prior hardware designs have developed techniques to skip zeros and to minimize data transfer [16, 29, 31].

Support for various layers in CNNs. Often, hardware designs are customized for one type of computation and do not support all types of layers in CNNs, such as pooling layers [23]. EIE [16] is intended for the fully connected layers in CNNs. It stores the input feature map and filters in a compressed format and passes only non-zero operands to the multipliers. In contrast, SCNN [31] and Eyeriss [6, 7] primarily focus on the convolution layers. Hence, they can underperform for the fully connected layers. SCNN can achieve 25% of peak throughput when performing the fully connected CNN layers. Similarly, Eyeriss provides significant energy gains only when batch sizes are larger than 16. In contrast, SPOTS supports all the common layers that exist in CNNs.
**Systolic array designs for CNNs.** Recent work [23] uses a preprocessing step (i.e., column combining) to pack a sparse CNN into a denser form before passing the inputs to a systolic array for GEMM. It is unclear how to prepare input feature maps for matrix multiplication. It will not provide benefits when there is abundant sparsity in the input feature map. Our group-wise pruning provides higher accuracy than the column combining method. **Simultaneous multithreaded systolic array (SMT-SA)** [36] addresses the underutilization and load imbalance introduced by random pruning of the weights in a CNN. Similar to SPOTS, recent work [27] utilizes a structured pruning accompanied by a novel data format called **density-bound block (DBB)** to better map the sparse inputs to the systolic architecture. Similar to SPOTS, Gemmini [14] uses a GEMM to accelerate CNNs. The authors explored both software and hardware Im2Col units. Similar to our work, their results demonstrate that using a hardware Im2Col can significantly improve performance. Unlike SPOTS, Gemmini is not sparse aware. In addition, the organization of the PEs in their design is rigid, resulting in the underutilization of PEs for certain layer shapes. Like SPOTS, recent work [46] proposes a memory-efficient hardware for the Im2Col. The main insight is to lay out the feature map elements in the SRAM in a **Channel-First** fashion. In this way, the feature map inputs are sent to the GEMM unit one column at a time. Unlike their method, we use multiple PUs connected with a ring network, which allows our design to generate multiple columns of inputs in parallel. Their design stores the feature map elements in one large SRAM. Instead, we use multiple smaller SRAMs to store the feature maps to reduce energy consumption.

**Flexible interconnects.** Flexible interconnects between PEs are useful in supporting various filter sizes [25, 32]. Maeri [25] enables a flexible dataflow mapping over DNN accelerators using a tree-based reconfigurable interconnects network. A downside of MAERI is that it does not handle input feature map sparsity. Similarly, FlexFlow [28] develops a flexible dataflow architecture that exploits different types of parallelism along with different CNN workloads. In contrast to them, SPOTS uses a regular interconnect network between the PEs. SIGMA [32] is another recent work that proposes a flexible non-blocking interconnect to achieve high compute utilization across layers of varying shapes. SIGMA is primarily optimized for high-precision inputs during the training phase. Besides, they solely focus on the GEMM and do not study the Im2Col transformation and support other types of layers in a CNN. Recent work [43] designs a **configurable multi-directional systolic array (CMSA)** that improves the PE utilization for small-scale convolution or depthwise convolution. However, their design solely focuses on improving PE utilization and thus does not address other aspects such as sparse inputs and Im2Col design.

**Load balance in sparse CNN accelerators.** Many hardware accelerators for sparse CNNs do not support load balancing [29, 31, 33]. GoSPA [9] uses a passive strategy to deal with the load imbalance problem in their design. It uses a two-stage buffering technique to maintain the multiplier utilization high in the presence of load imbalance. However, using large buffers can negatively impact their design’s area and power consumption.

In contrast, SparTen [15] and Column Combining [23] adopt a systematic approach to address the load imbalance in their design. SparTen proposes a greedy balancing method with two variants: a pure software approach and a software-hardware hybrid.

Further, SparTen load-balances at two different granularities (e.g., at the whole filter or at the chunk level). Doing the load balancing at a finer granularity (i.e., chunk level) necessitates a multi-stage permutation network in hardware to **unshuffle** the partial sum of each chunk to the appropriate output sum. Column Combining [23] suggests an entirely new training method to pack the sparse CNNs into a denser format for efficient executions using systolic arrays. They combine multiple sparse columns of a convolutional filter matrix into a single dense matrix. Like SparTen, they introduce extra hardware to permute the rows.
7 CONCLUSION

This article proposes SPOTS, a hardware accelerator for sparse CNNs with a matrix multiplication formulation of convolution using the Im2Col transformation. The hardware Im2Col unit reads the input feature map only once, reuses the data, and executes in parallel with a tall systolic array for the GEMM unit. We add flexibility to the systolic array that allows it to achieve high PE utilization for CNN layers of varying sizes and shapes. SPOTS supports sparsity in both the input feature map and the filters. SPOTS is faster and more energy efficient than state-of-the-art systolic-array-based ASICs, CPU, and GPU implementations for sparse CNNs.

REFERENCES

[1] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B. Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, and Tobi Delbruck. 2019. NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. IEEE Transactions on Neural Networks and Learning Systems 30, 3 (2019), 644–656. https://doi.org/10.1109/TNNLS.2018.2852335

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvltunin: Ineffectual-neuron-free deep neural network computing. SIGARCH Comput. Archit. News 44, 3 (June 2016), 13 pages. https://doi.org/10.1145/3007787.3001138

[3] Rajeek Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shaftee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. ACM Trans. Archit. Code Optim. 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

[4] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. 2002. An Updated Set of Basic Linear Algebra Subprograms (BLAS). ACM Trans. Math. Softw. 28, 2 (Jun 2002), 135–151. https://doi.org/10.1145/567806.567807

[5] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A machine-learning supercomputer. In 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture. 609–622. https://doi.org/10.1109/MICRO.2014.58

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyriess: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA’16). 367–379. https://doi.org/10.1109/ISCA.2016.40

[7] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyriess: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE Journal of Solid-State Circuits 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.261357

[8] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. J. Mach. Learn. Res. 12 (Nov. 2011), 2493–2537.

[9] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GoSPA: An energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA’21). 1110–1123. https://doi.org/10.1109/ISCA52012.2021.00090

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In 2009 IEEE Conference on Computer Vision and Pattern Recognition. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[11] Luiz DeRose, Bill Homer, and Dean Johnson. 2007. Detecting application load imbalance on high end massively parallel systems. In Proceedings of the 13th International Euro-Par Conference on Parallel Processing (Euro-Par’07). Springer-Verlag, Berlin, 150–159.

[12] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. Shifting vision processing closer to the sensor. In 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA’15). 92–104. https://doi.org/10.1145/2749469.2750389

[13] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Provence, RI, USA) (ASPLOS’19). Association for Computing Machinery, New York, NY, USA, 807–820. https://doi.org/10.1145/3297858.3304014

[14] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. arXiv:1911.09925 [cs.DC]
An Accelerator for Sparse CNNs Leveraging Systolic General Matrix-Matrix Multiplication 42:25

[15] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’52). Association for Computing Machinery, New York, NY, 151–165. https://doi.org/10.1145/3352460.3358291

[16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient inference engine on compressed deep neural network. In 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA’16), 245–254. https://doi.org/10.1109/ISCA.2016.30

[17] Song Han, Huizi Mao, and William J. Dally. 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In International Conference on Learning Representations (ICLR’16).

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR’16), 770–778. https://doi.org/10.1109/CVPR.2016.90

[19] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS’19). Association for Computing Machinery, New York, NY, 821–834. https://doi.org/10.1145/3279858.3304028

[20] Hyoeyong-Ju Kang. 2020. Accelerator-aware pruning for convolutional neural networks. IEEE Transactions on Circuits and Systems for Video Technology 30, 7 (2020), 2093–2103. https://doi.org/10.1109/TCSVT.2020.2911674

[21] Dae Hyun Kim and Sung Kyu Lim. 2015. Impact of TSV and Device Scaling on the Quality of 3D ICs. Springer New York, New York, NY, 1–22. https://doi.org/10.1007/978-1-4939-2163-8_1

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[23] 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 Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLoS’19). Association for Computing Machinery, New York, NY, 821–834. https://doi.org/10.1145/3279858.3304028

[24] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. SIGPLAN Not. 53, 2 (March 2018), 461–475. https://doi.org/10.1145/3296957.3173176

[25] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference. IEEE Computer Architecture Letters 19, 1 (2020), 34–37. https://doi.org/10.1109/LCA.2020.2979965

[26] Zhi-Gang Liu, Paul N. Whatmough, and Matthew Mattina. 2020. Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference. IEEE Computer Architecture Letters 19, 1 (2020), 34–37. https://doi.org/10.1109/LCA.2020.2979965

[27] Wenyan Lu, Guihui Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. 2017. FlexFlow: A flexible dataflow accelerator for convolutional neural networks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA’17), 553–564. https://doi.org/10.1109/HPCA.2017.29

[28] Huiyu Mo, Leibo Liu, Wenjing Hu, Wenping Zhu, Qiang Li, Ang Li, Shouyi Yin, Jian Chen, Xiaowei Jiang, and Shaojun Wei. 2017. FlexFlow: A flexible dataflow accelerator for convolutional neural networks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA’17), 553–564. https://doi.org/10.1109/HPCA.2017.29

[29] NVIDIA, Peter Vingelmann, and Frank H. P. Fitzek. 2020. CUDA, release: 10.2.89. (2020). https://developer.nvidia.com/cuda-toolkit.

[30] Anshuman Parashar, Minsoo Rho, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In 2017 IEEE/ACM International Symposium on Computer Architecture (ISCA’17), 27–40. https://doi.org/10.1145/3079856.3080254

[31] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA’20), 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[32] Mahmood Azhar Qureshi and Arslan Munir. 2021. Sparse-PE: A performance-efficient processing engine core for sparse convolutional neural networks. IEEE Access 9 (2021), 151458–151475. https://doi.org/10.1109/ACCESS.2021.3126708

[33] 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
