IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 19, 2021, accepted November 4, 2021, date of publication November 8, 2021, date of current version November 15, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3126708

Sparse-PE: A Performance-Efficient Processing Engine Core for Sparse Convolutional Neural Networks

MAHMOOD AZHAR QURESHI AND ARSLAN MUNIR, (Senior Member, IEEE)
Department of Computer Science, Kansas State University, Manhattan, KS 66506, USA
Corresponding author: Arslan Munir (amunir@ksu.edu)

ABSTRACT Sparse convolutional neural network (CNN) models reduce the massive compute and memory bandwidth requirements inherently present in dense CNNs without a significant loss in accuracy. Sparse CNNs, however, present their own set of challenges including non-linear data accesses and complex design of CNN processing elements (PEs). Recently proposed accelerators like SCNN, Eyeriss v2, and SparTen, exploit the *two-sided* sparsity, that is, sparsity in both the input activations and weights to accelerate the CNN inference. These, accelerators, however, suffer from a multitude of problems that limit their applicability, such as inefficient micro-architecture (SCNN, Eyeriss v2), complex PE design (Eyeriss v2), no support for non-unit stride convolutions (SCNN) and FC layers (SparTen, SCNN). To address these issues in contemporary sparse CNN accelerators, we propose *Sparse-PE*, a multi-threaded, and flexible CNN PE, capable of handling both the dense and sparse CNNs. The Sparse-PE core uses binary mask representation and actively skips computations involving zeros and favors non-zero computations, thereby, drastically increasing the effective throughput and hardware utilization. Unlike previous designs, the Sparse-PE core is generic in nature and not targeted towards a specific accelerator, and thus, can also be used as a standalone sparse dot product compute engine. We evaluate the performance of the core using a custom built cycle accurate simulator. Our simulations show that the Sparse-PE core-based accelerator provides a performance gain of 12× over a recently proposed dense accelerator (NeuroMAX). For sparse accelerators, it provides a performance gain of 4.2×, 2.38×, and 1.98× over SCNN, Eyeriss v2, and SparTen, respectively.

INDEX TERMS Convolutional neural networks (CNNs), hardware accelerators, multi-threaded, sparsity, high-throughput.

I. INTRODUCTION

Deep neural networks (DNNs) have enabled the deployment of artificial intelligence (AI) in many modern applications including autonomous driving, image recognition, speech processing, and language translation. One of the most popular algorithmic approach for many AI applications is the convolutional neural network (CNN). Deployment of a CNN model is usually carried out in two stages: (1) training - the CNN parameters are learned by extracting key features from a set of training data, and (2) inference - the trained CNN model is deployed on the field and subjected to the *real-world* data for making predictions. The CNN accuracy is determined by observing the total number of accurate classifications in the inference phase.

High accuracy CNN models [1]–[3] proposed in recent years have further strengthened the notion of employing CNNs for various vision-based AI applications. These CNN models require massive amounts of convolution operations over a series of network layers to perform a classification task during the inference phase. These tremendous number of computations (typically in tens of millions) present a huge challenge for the devices employing these CNN models. In addition, because of the large number of network layers and varying layer dimensions, the massive CNN model cannot be stored in the on-chip memory of the device, and, therefore, requires off-chip DRAM which presents high DRAM access cost. To put this in perspective, the energy cost per fetch for 32b coefficients in an off-chip LPDDR2 DRAM is

The associate editor coordinating the review of this manuscript and approving it for publication was Thomas Canhao Xu.

about 640pJ, which is about 6400× the energy cost of a 32b integer ADD operation [4]. The energy cost from just the DRAM accesses would be well beyond the limitations of an embedded mobile device employing the CNN.

Various techniques have been developed to address the compute and memory bandwidth issues of a neural network accelerator, running a CNN. *Mobilenets* [5], [6] were developed to reduce the total number of computations by splitting a regular convolution operation into separable convolutions (depthwise and pointwise), without incurring a loss in accuracy. Another widely used approach for decreasing the model size is the reduction in precision of both weights and activations using various quantization strategies [7]–[9]. This again does not result in a significant loss in accuracy and reduces the model size by a considerable amount. Hardware implementations like Envision [10], NeuroMAX [11], UNPU [12], and Stripes [13] show how reduced bit precision, and quantization, translates into increased throughput and savings in energy.

Non-linear activation functions [14], in addition to deep layers, are one of the key characteristics that improve the accuracy of a CNN model. Typically, non-linearity is added by incorporating activation functions, the most common being the rectified linear unit (ReLU) [14]. The ReLU converts all negative values in a feature map to zeros. Since the output of one layer is the input to the next layer, many of the computations, within a layer, involve multiplication with zeros. These feature maps containing zeros are referred to as *one-sided* sparse feature maps. The multiplications resulting from this one-sided sparsity wastes compute cycles and decreases the *effective* throughput and hardware utilization, thus, reducing the overall performance of the accelerator. It also results in high energy cost as the transfer of zeros to/from off-chip memory is a wasted memory access. In order to reduce the computational and memory access volume, previous works [15]–[17] have exploited this one-sided sparsity and displayed some performance improvements.

Compression of DNN models was introduced the first time in [18]. Han *et al.* [18] iteratively pruned the connections based on parameter threshold, and performed retraining to retain accuracy. This process resulted in *two-sided* sparsity, i.e., sparsity in both weights and activations, which led to approximately 9× model reduction for AlexNet, and 13× reduction for VGG-16. It also resulted in 4 - 9× *effective* compute reduction (depending on the model). These gains seem very promising, however, designing an accelerator architecture to leverage them is quite challenging because of the following reasons:

A. DATA ACCESS INCONSISTENCY

Computation gating is one of the most common ways by which sparsity is generally exploited. Whenever a zero in the activation or the weight data is read, no operation is performed. This results in energy savings but has no impact on the throughput because of the wastage of compute cycle. Complex read logic needs to be implemented to discard the zeros, and instead, perform effective computations on non-zero data. Some previous works [19], [20] use sparse compression formats like compressed sparse column (CSC) or compressed sparse row (CSR) to represent sparse data. These formats have variable lengths and make *"looking ahead"* difficult if both the weight and the activation sparsity is being considered. Other than that, developing the complex control and read logic to process these formats can be quite challenging.

B. LOW UTILIZATION OF THE PROCESSING ELEMENT (PE) ARRAY

Convolution operations for CNN inference are usually performed using an array of two-dimensional PEs in a CNN accelerator. Different dataflows (input stationary, output stationary, and row stationary) have been proposed that efficiently map the weight data and the activation data on to the PE array to maximize the throughput [14]. Sparsity introduces inconsistency in the scheduling of data thereby reducing the hardware utilization. The subset of PEs provided with more sparse data have idle times while those provided with less sparse (or more dense) data are fully active. This bounds the throughput of the accelerator to the most active PEs, and therefore, leads to the under utilization of the PE array.

Considering the above mentioned issues, many accelerators have been proposed in the past that attempt to strike a balance between hardware resource complexity and performance improvements. Cnvlutin [15] attempts to exploit sparsity by skipping the computations during *zero* activation data. It, however, does not avoid transfer of zeros and only skips cycles for zero-activations but not zero-weights. This results in exploitation of only *one-sided* sparsity. Eyeriss [17] only gates computations for sparse activations. Eyeriss v2 [19] attempts to address the two-sided sparsity by using CSC format for both the activations and weights. It, however, requires complex read logic embedded within a PE that drastically increases the area by $\sim$ 93% when compared to the original Eyeriss [17]. Cambricon-X [16] does not store activations in compressed format while Cambricon-S [21] forces regularity by employing coarse grain pruning that affects accuracy. Even though it discards zeros during computation, it still retrieves and stores them. EIE [20] exploits the two-sided sparsity, albeit *only* in fully-connected layers. EIE's performance is equivalent to one-sided sparsity as it discards zeros in the filter but wastes compute cycles due to being idle. Sparse CNN (SCNN) [22] targets two-sided sparsity but suffers heavily from inefficient microarchitecture and systematic load imbalance as explained in [23]. It, also, cannot handle non-unit stride convolutions and FC layers. To address the complexity associated with the CSC compression format, SparTen [23] uses sparse bit mask to represent the location of zeros and non-zero data values. SparTen, however, needs an offline load balancing strategy, which it refers to as *Greedy Balancing*, to address the systematic load imbalance. This form of balancing adds extra latency and complicates the synchronization of various

compute threads. SparTen also employs *Permuter* and *Output Collector Units* for the computation clusters to merge and/or accumulate the outputs from *independently* running compute units. These circuits require rather complex hardware and the complexity grows exponentially as the number of compute units are increased. In addition, SparTen, like SCNN, has no support for FC layers.

In this paper we introduce Sparse-PE, a high performance, multi-threaded, generic processing engine (PE) core for sparse CNN computations. Unlike many previous works, this PE design can exploit full or *two-sided* sparsity and can be used for sparse computations in any layer in a typical CNN model. While the previous approaches use complex PE designs targeted towards their specific accelerator architectures, the Sparse-PE core is generic in nature and can be used as a general purpose, sparse dot product compute core. Our main contributions can be summarized as follows:

- We present Sparse-PE, a multi-threaded, high performance, processing engine core, ideal for *two-sided* sparse computations. The core works by actively skipping a huge number of *ineffective* computations ($zero_w \times zero_a$, $zero_w \times non\text{-}zero_a$, $non\text{-}zero_w \times zero_a$) involving zeros, while only favoring *effective* computations ($non\text{-}zero_w \times non\text{-}zero_a$). This is accomplished by the use of novel selection, computation, and accumulation blocks to dynamically allocate maximum, *non-zero* computations, on to a thread matrix inside the core to drastically improve the hardware utilization. The presented PE core does not target a specific architecture, and thus, can be modified for any accelerator design.
- Unlike previous approaches that use compressed sparse column (CSC) format for their PEs, the Sparse-PE core uses bit mask (BM) representation for sparse computations. We show that, on average, the CSC format has 3× higher DRAM memory accesses compared to the BM representation which directly translates into higher energy requirements for the CSC format.
- We develop a cycle-accurate performance simulator for an accelerator that uses the Sparse-PE cores and show drastic performance improvement over various recently proposed dense and sparse CNN accelerators, and high hardware utilization over a range of sparsity levels. Our experiments show that the Sparse-PE core-based accelerator has a performance gain of 12× over a recently proposed dense accelerator (NeuroMAX). For sparse accelerators, it provides a performance gain of 4.2×, 2.38×, and 1.98× over SCNN, Eyeriss v2, and SparTen, respectively. We also do an RTL implementation of the core on Xilinx Z-7100 SoC and show a detailed module level breakdown of the FPGA primitives cost, SRAM cost, and power consumption of the core.

The remainder of the paper is organized as follows. Section II gives a background on CNN sparsity and layer by layer sparsity associated with various popular CNN models. Section III presents the proposed Sparse-PE core and its inner workings. Experimental methodology, simulation results, comparisons, and implementation cost is given in Section IV. Section V summarizes recent literature related to our work. Lastly, Section VI concludes this paper.

II. RELATED WORK

Many dense architectures have been introduced in the past for acceleration of CNN inferences. Accelerators proposed in [9], [24], [25] optimize compute, whereas, [26], [27] optimize memory bandwidth. Quantization (linear [9], [28] and log [7], [8]) of weights and activations provides additional benefits for memory footprint and compute reductions. Accelerators like NeuroMAX [11], VWA [29], UNPU [12], and Stripes [13], show how reduction in bit precision, improved dataflow, and quantization, increases throughput and saves energy. Another set of accelerators [30], [31] provide efficient implementation of separable convolutions on FPGA hardware. These, however, cannot handle regular convolution and fully connected (FC) layers which are almost always a part of CNN models. Accelerators proposed in [32], [33] use Booth encoding to avoid the use of zeros to reduce the total computations. They, however, still transfer zeros to and from memory which incurs SRAM area and energy. Block circulant matrices for weights were introduced in CirCNN [34]. CirCNN, however, requires complex fast Fourier transform (FFT) operations in its PE design. It also does not capture two-sided sparsity. In-memory accelerators [35], [36] have also been presented that use analog logic design to perform matrix multiplications within memory. Sparse multiplications, however, cannot be performed in these accelerators as they require complex ALU and buffering logic. Analog circuits are also impacted by noise and variations during manufacturing process which can significantly impact the CNN model accuracy during inference.

Sparse architectures reduce the compute and memory access volume by exploiting the zeros in activations (one-sided), or both activations and weights (two-sided). Cnvlutin [15] and Cambricon-X [16] exploit one-sided sparsity by ignoring zeros in weights or input maps, but not both. Cnvlutin also does not avoid transferring of zeros and only skips cycles for activations. Tensaurus [37] accelerates sparse and dense tensor factorizations by introducing compressed interleaved sparse slice (CISS) dataflow. It, however, only supports one-sided sparsity. Recent sparse GEMM (SpGEMM) accelerators [38]–[42] target generalized sparse-matrix, sparse-matrix multiplications. Sigma [40] and ExTensor [39] use inner-product (output stationary) dataflow for sparse matrix multiplications. Inner product, however, is inefficient for highly sparse matrices because every element of the rows and the columns must be traversed even though there are less effectual computations (non-zero × non-zero). This leads to a significant amount of wasted computations. SpArch [41] and OuterSPACE [42] use outer-product (or input stationary) dataflow to avoid the inefficiencies associated with the traversals inherent in the inner-product dataflow. Outer-product, however, gives poor output reuse as the partial outputs generated are much

more than the final outputs causing significant memory traffic. Finally, MatRaptor introduces a modified version of the CSR format referred to as channel cyclic sparse row (C^2SR) for better reuse and memory efficiency but requires complex encoding for output matrices. Eyeriss v2 [19] uses the CSC format for both weights and activations to address the two-sided sparsity. It, however, suffers from systematic load imbalance due to variations in the density of the sparse matrices. The PE design of Eyeriss v2 also requires complex buffering logic that drastically increases the area by $\sim$ 93% when compared to the original Eyeriss [17]. EIE [20] exploits the two-sided sparsity in FC layers and does not address the CONV layers. EIE essentially discards zeros in weights but remains idle, thus, wasting compute cycles. Sparse CNN (SCNN) [22] targets two-sided sparsity but its PEs suffer from inefficient microarchitecture and system-level load imbalance (also pointed out in [23]). SCNN, also, is incapable of handling non-unit stride convolutions and FC layers. Although some previous accelerator architectures attempt to exploit sparsity in CNNs, they do not address the issues related to high PE cost, inefficient microarchitecture, and dependence of PE on accelerator design. We design a multi-threaded PE, referred to as Sparse-PE, which not only addresses the issues present in the previous designs, but also, can carry out general sparse dot product computations for any application.

III. SPARSITY IN CNNS

Sparsity refers to the fraction of zeros in a CNN layer's weight and input activation matrices. Weight sparsity is static and is introduced while pruning a network during training. Han *et al.* developed an iterative scheme for pruning a network and retraining to retain the network's accuracy [18]. Activation sparsity is introduced dynamically during the inference phase and is highly dependent on the input being processed. This sparsity occurs because of the ReLU activation function, most commonly found in many CNNs, which converts all the negative outputs of a layer to zero.

Figure 1 shows the weight and the activation sparsity among two of the most commonly used CNNs. We randomly select 50,000 images from the ImageNet dataset [43] and run pretrained sparse AlexNet and VGG16. The weight sparsity values in Figure 1 were obtained directly from the sparse CNNs, whereas, the activation sparsity values were obtained by averaging out the layer-by-layer activation sparsity during processing of the input images. The whole process was done using the Keras framework [44]. It can be seen that the weight sparsity for AlexNet and VGG-16 can reach as high as 70% and 80%, respectively, for some layers. Activation sparsity tends to be lower in the initial layers but rises considerably in later layers with some layers of VGG-16 having activation sparsity as high as 85%. This shows that many neural nets, though seemingly compute and memory bandwidth-intensive, are incredibly sparse with huge amounts of redundant computations. A PE design which efficiently exploits this redundancy can provide immense gains in both performance and energy efficiency.

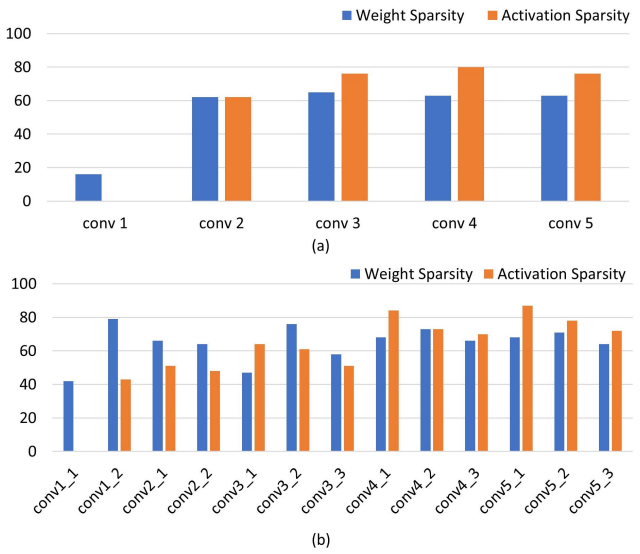

FIGURE 1. Sparsity in CNNs (a) Sparsity in AlexNet (b) Sparsity in VGG-16.

IV. SPARSE-PE

Figure 2 shows a typical convolution operation in a CNN. Here, a 3×8 sparse input is convolved with a 3×3 sparse weight to generate a 1×6 output. The input and weight matrices have sparsity of 42% and 45%, respectively. The convolution operation can be broken down into 6 smaller convolution chunks (C0-C5), as shown in Figure 2. Each of the 9 multiplications in a single convolution chunk are performed by a compute thread within a 3×3 compute thread matrix. The multiplications in red are ineffective computations which means that either one or both the multiplication operands are zeros, resulting in a wasted computation, whereas, the multiplications in black are effective. It can be seen that, on average, 66% multiplications in a convolution chunk are ineffective (Output Sparsity OS = 6/9) which corresponds to an *effective* hardware utilization of only 33%. This represents

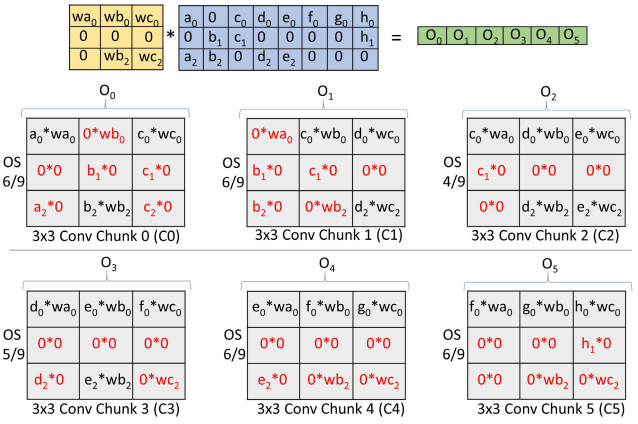

FIGURE 2. Dense convolutions.

a significant loss in computational efficiency as most of the compute cycles are wasted on ineffective computations. The Sparse-PE core addresses this issue and increases the hardware utilization, consequently the throughput, by minimizing the total number of ineffective computations performed by the 3×3 compute matrix. It does this by *looking-ahead* into the computations beforehand and scheduling only the valid computations to minimize the total compute cycles.

Figure 3 shows the high-level block diagram of the Sparse-PE core. The core takes the binary mask (BM) and data input and performs sparse computations to generate output data and binary mask. The Sparse-PE core consists of three main components: Selection (SL), Computation (CM), and Accumulation (AM). The SL block uses the sparse binary masks of input data/feature maps and weights to perform selection of valid computations ($non\text{-}zero_w \times non\text{-}zero_a$). These valid computations are represented by a set of binary matrices referred to as select matrices. The CM block uses the select matrices to map the sparse input and weight data on to a 3×3 matrix of compute threads. The mapping is performed in such a way as to maximize the utilization of individual compute threads. The AM block accumulates the CM outputs to produce valid output results. The output sparse binary mask (BM) is also generated which will be used for the next CNN layer.

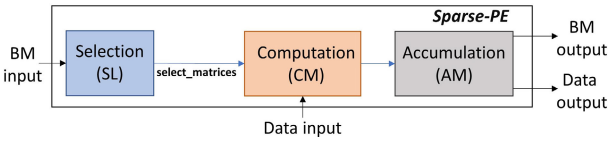

FIGURE 3. Sparse-PE architecture.

A. SPARSE BINARY MASK

Many previous approaches use compressed sparse row (CSR) or column (CSC) formats to represent sparse data [15], [16], [20]. We, instead, use a binary representation referred to as sparse binary mask (BM) for representing both weights and activations. The BM representation provides a simplistic, and a more convenient method for representing the *unstored* zero data and the *stored*, non-zero data. Unlike the CSR/CSC formats, this representation does not require storage of *count* and *data pointers* which significantly decreases the memory footprint of the BM representation. Figure 4 shows the dense, BM and CSR format representation of the 3×3 weight matrix and the 3×8 input data/feature map given in Figure 2. In the dense representation, both the zero and non-zero data is stored in the memory along with the indices, as shown in Figure 4(a). Figure 4(b) shows the equivalent BM representation where only the non-zero data is stored. The BM representation represents non-zero (stored) data with the binary 1, and zero (unstored) data with the binary 0. Finally, Figure 4(c) shows the CSR format representation, where the relative locations of the non-zero data are represented by the row and col pointers. We will further analyze the

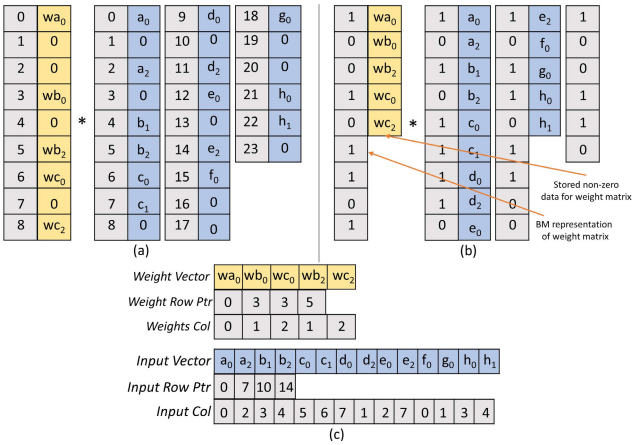

FIGURE 4. Sparse data representations (a) Indexed-based representation (b) BM-based representation (c) CSR format.

superiority of the BM representation over the CSR format in Section IV-2.

To process the input sparse data, the Sparse-PE core is provided the BM and the associated data in the form of chunks for processing of a particular CNN layer. Although, the Sparse-PE core can work on any type of convolution or FC layer, for convenience and ease of understanding, we will show the working of the core using the input and the weight matrix in Figure 2. We also assume that the input and the weight data is 8 bits wide.

B. SELECTION

The convolution operation works by performing dot product between two vectors. In a *two-sided* sparse CNN model, the dot product can result in four possible multiplication outputs.

i) $zero_o = zero_w \times zero_a$
ii) $zero_o = zero_w \times non\text{-}zero_a$
iii) $zero_o = non\text{-}zero_w \times zero_a$
iv) $non\text{-}zero_o = non\text{-}zero_w \times non\text{-}zero_a$

It can be seen that the only valid multiplication is the $non\text{-}zero_o$ which results when a non-zero weight ($non\text{-}zero_w$) is multiplied by a non-zero input/activation ($non\text{-}zero_a$). The SL block has two user-defined parameters, n and k. The main purpose of the SL block is to determine k, $non\text{-}zero_o$ computations in a set of n convolution chunks. The value of k represents the total number of multiplier threads in the CM block. In this design, we have a 3×3 matrix of multipliers, making the value of $k = 9$. We define n as the *lookahead factor*, which represents the number of convolution chunks the core looks into to determine k multiplications. There are a total of 6 convolution chunks (C0-C5) in the example input, as shown in Figure 2. For this design, we consider the value of n to be 3. This means that during one cycle, the core looks into $k = 9$ valid multiplications in a set of $n = 3$, 3×3 convolution chunks. The SL block does this by using a series of n, $2k$-input AND gates followed by a selector, as shown in Figure 5. Figure 6 shows the process of ANDing. During the first cycle, the sparse masks of the weight and $n = 3$

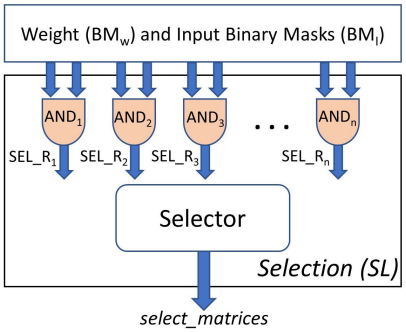

FIGURE 5. Selection (SL) block diagram.

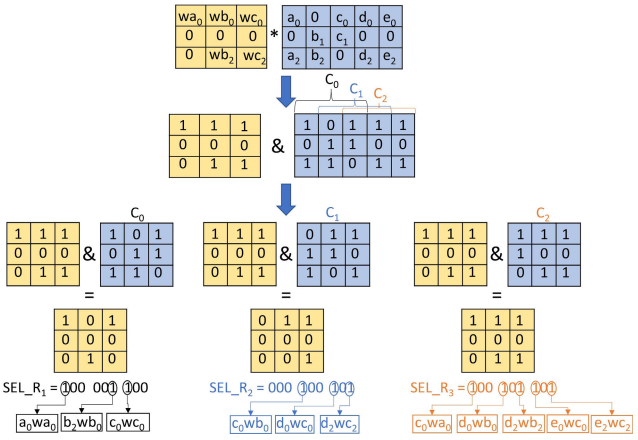

FIGURE 6. Process of ANDing between the weight binary mask and the input binary mask chunks in the SL block.

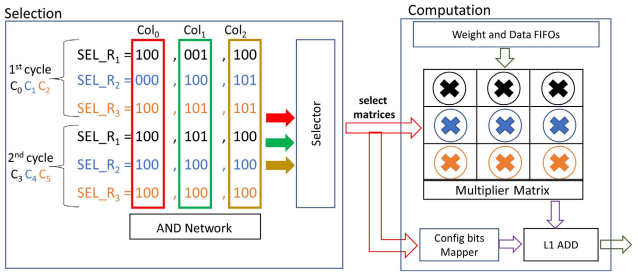

FIGURE 7. Selection for maximum multiplier utilization.

chunks of the input matrix are loaded into the core. Bit by bit ANDing is performed between the sparse masks of the weight and the input to generate SEL_R1, SEL_R2, and SEL_R3, representing the ANDed output of the first, second, and third, convolution chunks, respectively. The ones in the SEL_R outputs represent the location of valid non-zero multiplications, whereas, zeros represent in-effective computations involving zero operands. To process the six convolution chunks, two cycles are needed and the final SEL_R outputs are shown in Figure 7. The first cycle generates the ANDed outputs for the first three convolution chunks (C0, C1, C2), whereas, the second cycle generates the ANDed outputs for the last three convolution chunks (C3, C4, C5).

```
init = #ones(val_1)
accum_1 = init + #ones(val_2)
accum_2 = init + #ones(val_3)
accum_3 = init + #ones(val_2) + #ones(val_3)
if (accum_3) <= k/3
        select val_1, val_2, val_3
elseif (accum_1) <= k/3
        select val_1, val_2
        init = val_3
elseif (accum_2) <= k/3
        select val_1, val_3
        init = val_2
else
        select val_1
        init = val_2
```

FIGURE 8. Selection algorithm employed by the Selector to maximum the utilization of the multiplier matrix.

The ANDed outputs are provided to the selector which selects the valid multiplications in a column major format. There are a total of n selectors each processing a particular *comma-separated* column. For this design, the three columns Col_0, Col_1, Col_2 (shown in Figure 7) are processed in parallel by the three selectors. The selection process occurs iteratively in a non-linear (or out-of-order) fashion. The purpose of the selector is to schedule the effective computations in each of the input columns (Col_0, Col_1, Col_2) on to the respective columns of the multiplier matrix. Since, at any point, the maximum number of scheduled multiplications per column cannot exceed the total number of multipliers per column of the multiplication matrix, the selector has to make sure that during any cycle the total number of selected multiplications are equal to multiplier threads per column of the multiplier matrix ($(k = 9)/3 = 3$) to ensure maximum hardware utilization. Since every 1 in the input columns (Col_0, Col_1, Col_2) represents a valid computation, the selection algorithm works by counting the number of ones in the individual entries and selecting the entries that maximize the utilization of the multiplier matrix. Consider the second column (Col_1) in Figure 7 on which the second selector operates. The selection algorithm is shown in Figure 8. The selector iterates over the first $n = 3$ values (val_1, val_2, val_3) and generates three accumulation values (accum_1, accum_2, accum_3). **#ones** function calculates the total number of ones in a particular val. The first iteration comprises of the first three values 001 (val_1), 100 (val_2), 101 (val_3). The first value, i.e., 001 is assigned the highest priority and the selector counts the total number of ones in this entry and stores the result, which in this case is 1, in the *init* variable. It then computes the accumulator variables (accum_1, accum_2, accum_3) using the next two values (val_2, val_3). The selector then selects the values based on whether the total number of ones in accumulated values exceed the total number of compute threads in one column of the thread matrix (i.e. 3). The working of the algorithm for Col_1 is shown in Figure 9. In the first iteration (cycle 1), the selector selects val_1 and val_2, based on the

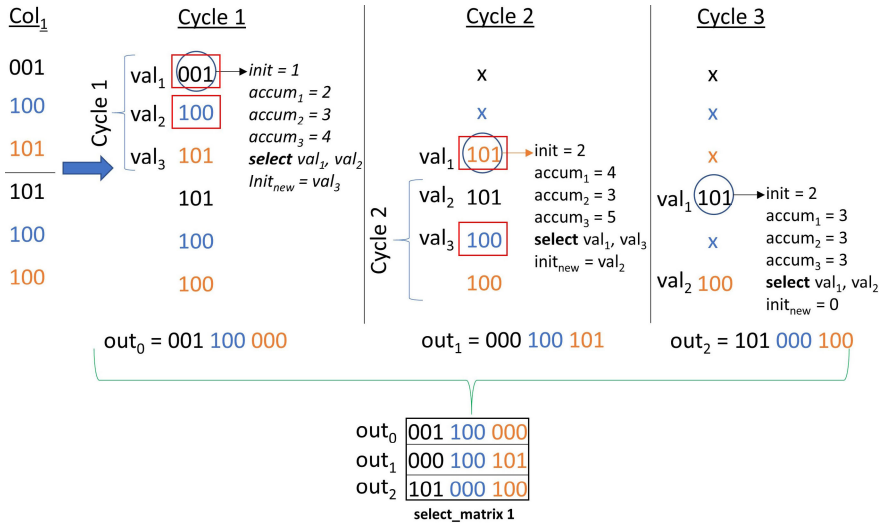

FIGURE 9. Selection process for column 1 (Col_1 in Figure 7) after running the algorithm in Figure 8.

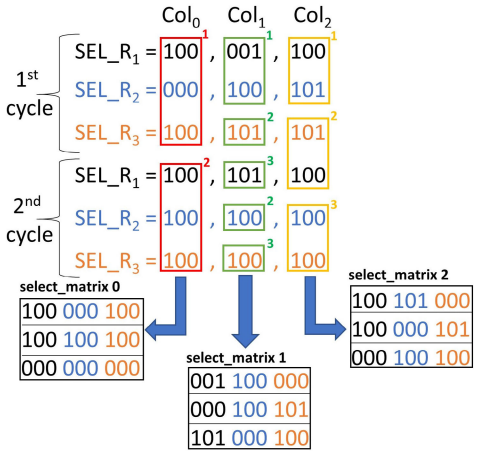

FIGURE 10. Generation of selection matrices.

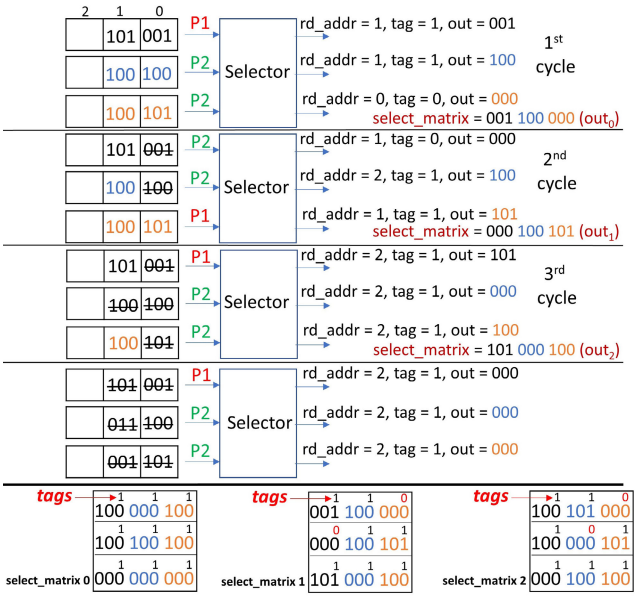

FIGURE 11. FIFO-based Selector implementation in hardware for generating select_matrices.

algorithm in Figure 8. The selector generates the output (out_0) by creating a single row of the selected values. Since a total of three values were considered, the selector replaces the last unselected value (val_3) by zeros in the out_0 output. In the next iteration (cycle 2), the selector prioritizes the unselected value (val_3) from the previous cycle and repeats the same process to generate out_1. It takes a total of three cycles for the selector to process the entire input column. At the end, a total of three select matrices are generated by the three selectors operating in parallel, as shown in Figure 10.

As indicated earlier, in this design example, we have considered the value of n to be 3. Therefore, in Figure 9, the selector considers three values (val_1, val_2, val_3) for selection in a particular cycle and uses three accumulators ($accum_1$, $accum_2$, $accum_3$ in Figure 8). For a higher value of n, let's say 6, the selector will consider all the values in Col_1 in Figure 9. At higher levels of sparsity, the increased n will result in an increase in throughput as more valid, non-zero computations will be scheduled and more invalid, zero computations will be skipped by the core. This dependency of the throughput on sparsity and n will be explored later in Section V-A. We should also mention that increasing the value of n will also result in an increase in the hardware resource count as more logic will be required for the selector implementation. For $n > 3$, $(2n - 2)$ accumulators and conditional statements will be required by the algorithm in Figure 8 for the selector implementation. Therefore, the value of n is chosen in such a way as to keep a balance between performance and area overhead.

Figure 11 shows the implementation of the selection process for Col_1 in Figure 7. The outputs from the AND network are stored in a local SRAM memory. On every cycle, a new

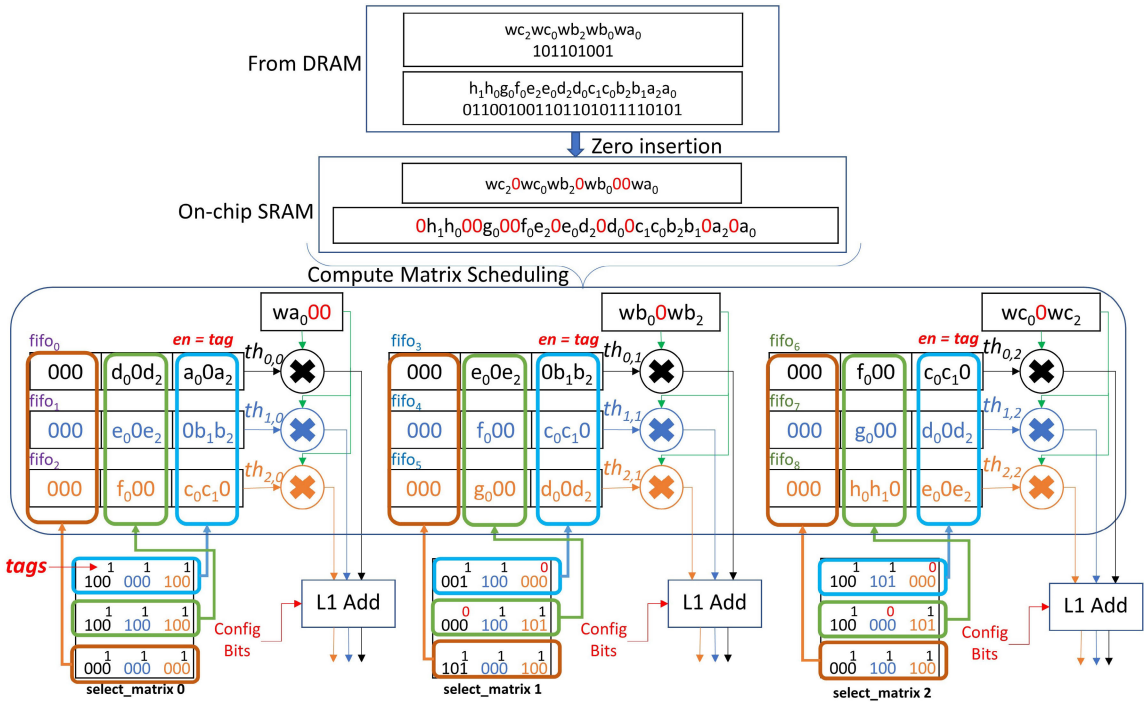

FIGURE 12. Process of computation in the first row of the multiplier matrix (Figure 7). The select_matrices (generated in Figure 11) are used to schedule the computations on to the multiplier threads.

value is written into the memory and values are read based on the priority of selection, with P1 being the higher priority. The selector block implements the algorithm presented in Figure 8. After every iteration, the read address (rd_addr) gets incremented to read the next value in the memory and the priorities are reversed. The outputs get stored into the selection matrix based on the row numbers. The tag bits are a crucial part of this process since they are used by the CM and the AM block for accumulation of data (explained in subsequent sections). Whenever a particular value is not selected during a selection run, the tag bit associated to that value is resetted to 0. The selection process ends when no more values need to be read from the memory, the read addresses for all the memories are equal, and all the tag bits are set to 1. These conditions raise a termination flag which ends the process of selection. The final select matrices, along with the associated tag bits for every value are shown in Figure 11.

C. COMPUTATION

Figure 12 shows the process of computation in the CM block. The actual sparse data and its BM representation for a particular layer is loaded into the on-chip SRAM. Based on the BM of the input and the weight matrix, zeros are inserted at various locations for length equalization and proper indexing, as shown in Figure 12. The CM block consists of a series of fifos connected to the multiplier threads of the thread matrix. Since the size of multiplier matrix is 3×3, a total of 9 fifos ($fifo_0$-$fifo_8$) are connected to the 9 individual multiplier threads. Each column of the multiplier matrix gets a portion of the zero-inserted weight matrix. This can be seen in Figure 12, where the first column of multiplier threads ($th_{0,0}$, $th_{1,0}$, $th_{2,0}$) gets the weight vector $wa_0 00$, the second column ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) gets $wb_0 0 wb_2$ and the third column ($th_{0,2}$, $th_{1,2}$, $th_{2,2}$) gets $wc_0 0 wc_2$. Zero-inserted data is also scheduled to the threads from the fifos as shown in Figure 12. The enable pin of the fifos are connected to the tag bits of the select matrices. Whenever the tag bit for a particular entry is 1, the fifo is enabled and a new value from the fifo is passed to the multiplier which performs the computation. Figure 13 shows the process of computation for the second column ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) of the thread matrix. In the first cycle, the first three values in the three fifos ($fifo_3$, $fifo_4$, $fifo_5$) are controlled by the first row of the select_matrix 1 (001 100 000). The first three bits (001) along with the associated tag bit (1) controls $fifo_3$. Similarly the next 2 sets of the three bits control $fifo_4$ and $fifo_5$, respectively. Since the tag bits for the first two values (001 and 100) are set to 1, the two corresponding fifos are enabled and the data is moved to the corresponding multiplier threads. The multiplier determines the valid non-zero computation based on the values from the select matrix. For the first value (001), the multiplier extracts the last 8 bits corresponding to the weight value (wb_2) and the data (b_2) and performs the multiplication to generate the output value $b_2 wb_2$. Similarly the second multiplier thread ($th_{1,1}$) generates the value $c_0 wb_0$. Since the last three bits of the first row of the select matrix 1 have a tag 0 associated to it, the $fifo_5$ enable is off and the multiplier $th_{2,1}$ does not perform a valid multiplication. The same process repeats for

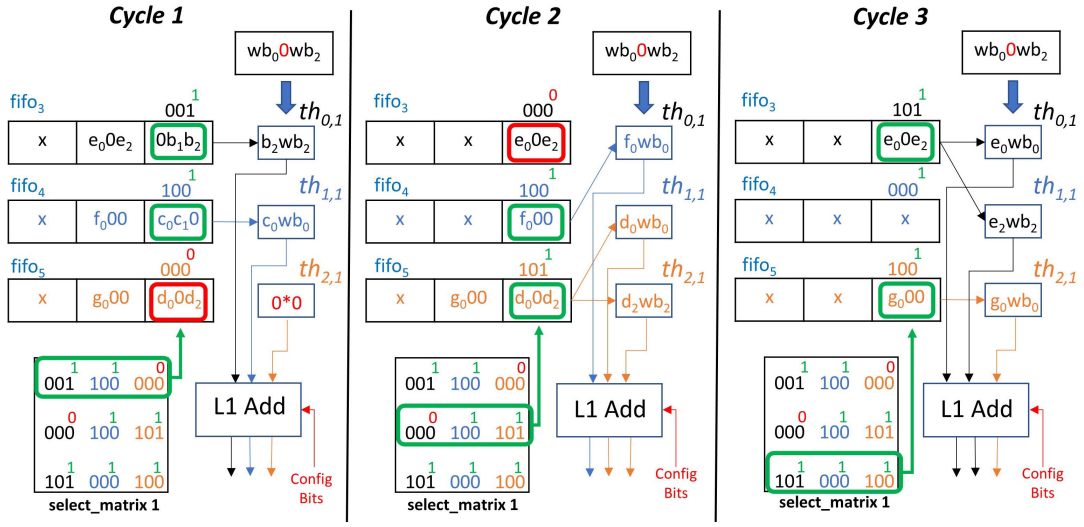

FIGURE 13. Cycle-by-cycle computation of the partial products from the second column thread matrix ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) in Figure 12.

the next two rows of the select matrix for cycles 2 and 3. The green rectangles on the fifo entries in Figure 13 represent the fifo entries that have tag bits of 1 and are thus utilized in the current cycle, whereas, the red rectangles indicate the entries that have tag bit of 0, and thus, are not utilized in the current cycle. Parallel to this multiplier column, the first ($th_{0,0}$, $th_{1,0}$, $th_{2,0}$) and the third ($th_{0,2}$, $th_{1,2}$, $th_{2,2}$) multiplier columns process the data in their own respective fifos using their own select matrices to generate the outputs.

From Figure 12, it can be seen that the output of the multiplier threads are fed into the level 1 (L1) Adder circuit. The purpose of this adder is to accumulate the outputs which belong to the same convolution chunk and to bypass the addition if the outputs belong to different convolution chunks. To further illustrate this, consider the Cycle 2 in Figure 13. The outputs from $th_{1,1}$ (d_0wb_0) and $th_{2,1}$ (d_2wb_2) belong to the same convolution chunk (C2 in Figure 2) and therefore need to be accumulated, whereas, the output from $th_{0,1}$ (f_0wb_0) belongs to a completely different convolution chunk (C4 in Figure 2) and has no relevance to the outputs from $th_{1,1}$ and $th_{2,1}$. This happens because the selector in the previous step is selecting the valid non-zero computations in a non-linear fashion to maximize the multiplier utilization and does not necessarily care about maintaining a proper flow of multiplications. To circumvent this issue and to determine which multiplier outputs need to be accumulated, the L1 adder uses a bit mapper that encodes the select matrix rows to appropriate values, as shown in Figure 14(a). The two bits at the output encode the following information:

00 -> The thread outputs within the multiplier column are not added and passed as is.

01 -> The outputs of $th_{0,x}$ and $th_{1,x}$ are added, whereas, the $th_{2,x}$ output is passed as is.

10 -> The outputs of $th_{1,x}$ and $th_{2,x}$ are added, whereas, the $th_{0,x}$ output is passed as is.

11 -> The outputs of all the threads within a multiplier column are added.

The mapping from the 9 bits of select matrix to 2 bit output is straightforward. Since every color coded set of three bits represent a multiplication within a particular convolution chunk, whenever there are multiple ones within the same set (black, blue, or orange), the thread outputs associated to those values need to be added together. This can be seen from entry 4 of Figure 14(a) (000 000 011 -> 01). The orange set has value 011, so according to the above mapping, the first two thread outputs are added together because they belong to the same convolution chunk. Similarly, the last entry (111 000 000) is mapped to 11, meaning that all three thread outputs within a multiplier column belong to the same convolution chunk, and therefore, need to be accumulated. Using this mapping, the L1 adder can keep track of which thread outputs need to be accumulated and which do not. Figure 14(b) shows the internal structure of the L1 Adder. It comprises of three adders which add different combinations of the thread outputs. It also comprises of an output multiplexer whose select line is connected to the mapper output (2 bits). The multiplexer has three outputs which are determined by the mapper output bits. The L1 Adder operation for the second column of the thread matrix ($th_{0,1}$, $th_{1,1}$, $th_{2,1}$) is shown in Figure 14(c). The mapper maps the individual rows of the select matrix to 2 configuration bits. Using the configuration bits, the L1 Adder generates the outputs. In the first cycle, all three outputs (b_2wb_2, c_0wb_0, 0) belong to different convolution chunks and therefore the config bits are 00, which passes the inputs to outputs without accumulation. In the second cycle, the select matrix row maps to 10, which adds the last two thread outputs (d_0wb_0, d_2wb_2) together because they belong to the same convolution chunk and forwards the first thread output (f_0wb_0) as is because it belongs to a different convolution chunk. In the last cycle,

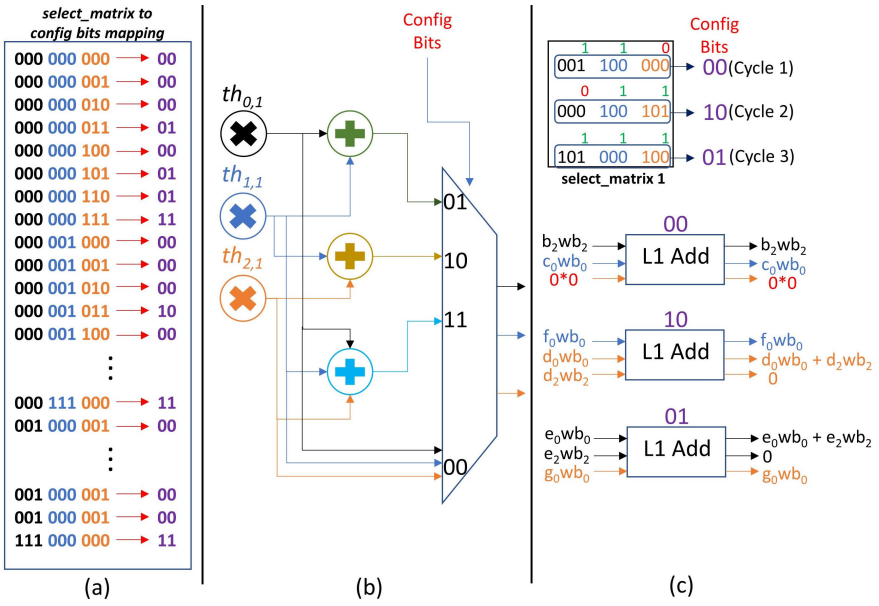

FIGURE 14. L1 Adder computational mapping and partial sum generation (a) select_matrix to config bits mapper (b) L1 Adder internal structure (c) Partial sum generation through accumulation from the L1 adders.

the first two thread outputs are added and the last is passed as is. It should be noted that there are a total of three identical mappers (Figure 14(a)), each belonging to a particular L1 adder in Figure 12.

Even though the row length of the select matrix is 9 which produces $2^9 = 512$ combinations, the mapper only needs to store those combinations for which the total number of ones in the select matrix rows are less than or equal to the multiplier threads per column of the multiplier matrix (3 in this case). Therefore, the total combinations that need to be stored are only 130. Generally, the total combinations that need to be stored can be found using the following equation:

$$Combinations = \binom{k}{0} + \binom{k}{1} + \binom{k}{2} + \ldots + \binom{k}{n} \quad (1)$$

where, k represents the length of a select matrix row (9 in this case) and n represents the total multiplier threads per column of the thread matrix (3 in this case). Therefore, the total memory required for storing the three mappers is only 780 bits ($130 \times 2 \times 3$).

Figure 15 shows the scheduling of threads for computation of the output for the example in Figure 2. All the threads are connected to their respective fifos and receive the data. The select matrices are used to schedule the data and for generating the 2-bit L1 Adder configuration bits (also shown in Figure 15). To compare the performance of a dense design which does not exploit sparsity, we can see from Figure 2 that a dense approach would take a total of 6 cycles to generate the output using the same number of multipliers (9 arranged in a 3×3 matrix). The approach presented here utilizes just three cycles to process the entire output by exploiting the sparsity and maximizing the scheduling of valid, non-zero

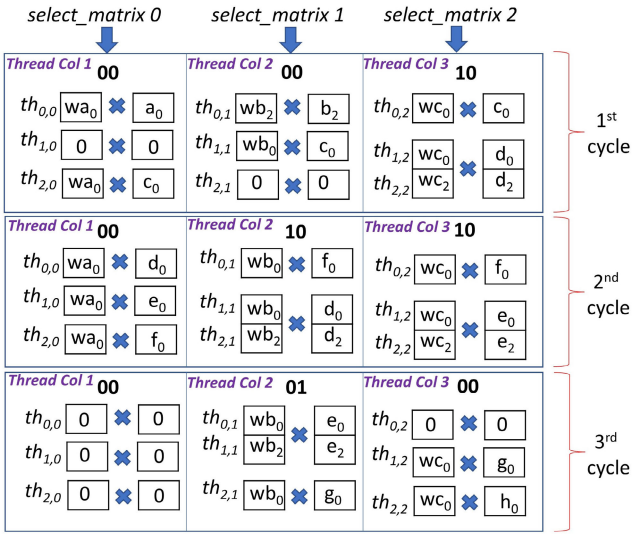

FIGURE 15. Multiplier matrix thread scheduling for the example in Figure 2.

multiplications during majority of the processing cycles. This represents a 50% increase in the hardware utilization which consequently represents a 50% increase in throughput. It should also be noted that the hardware utilization increases further when the input sparse matrix is larger. It can be seen from Figure 15 that the last three multiplications in the Thread Col 1 are all zeros because the input matrix has been exhausted. For a larger input matrix, the average hardware utilization and throughput would be higher. We will further explore the impact of the size of the input on the hardware utilization and throughput in Section IV.

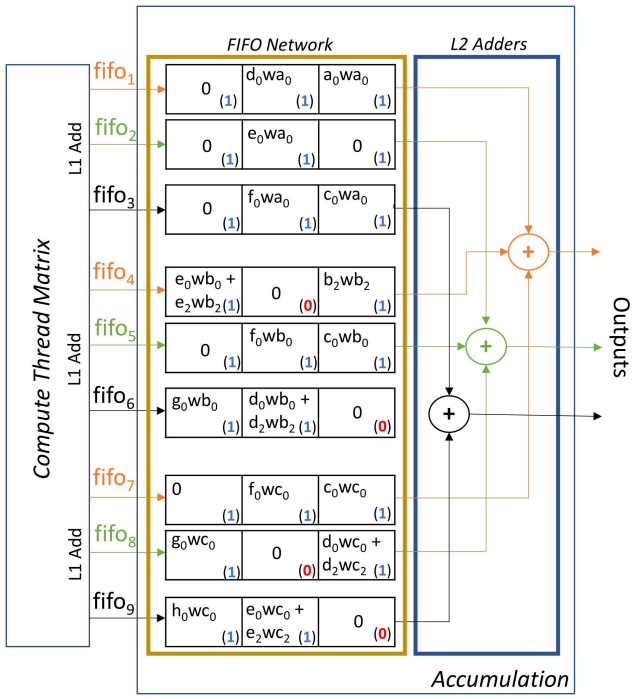

FIGURE 16. Accumulation for the example in Figure 2.

D. ACCUMULATION

The accumulation (AM) block buffers and accumulates the partial dot product outputs from the CM block to produce the final output results. The AM block consists of a series of fifos ($fifo_1$-$fifo_9$) connected at the output of the 3 L1 Adder circuits, as shown in Figure 16. The output of the fifos are provided to a level 2 (L2) adder which accumulates the dot products for a particular convolution chunk to generate the final output. The CM block outputs the partial dot products as well as the tag values associated with every product. The tag bits are shown in parenthesis in Figure 16. The process of accumulation occurs in two stages by the same color coded fifos ($fifo_1 + fifo_4 + fifo_7$), ($fifo_2 + fifo_5 + fifo_8$), and ($fifo_3 + fifo_6 + fifo_9$). The outputs are either valid or partial, based on the tag values associated to each accumulation stage. If the tag values for all the inputs are equal to **1**, the output is considered *valid*, otherwise, it is considered *partial*. Figure 17 shows the accumulation stage map for the example in Figure 2. On every cycle there are two accumulation stages. In the first stage, the previously generated partial outputs are added to the new entries to make the output *valid*. To do this, the AM block checks the tag bits in the partial output and adds the missing tag 1 input to make the partial output complete/valid. In the second stage, the AM replaces the already used tag 1 values with zeros and generates new partial outputs by summing the unused tag 1 values. This process can be seen in Figure 17. The O_3 partial value is generated by summing the fifos F3, F6, F9 in cycle 1. The output is partial because the $Input_2$ and $Input_3$ tag values are 0. O_3 is made valid by summing the O_3(partial) with the now available $Input_2$ and $Input_3$ tag 1 values in the stage 1 during the 2^{nd} cycle. In the second stage, during the same cycle, the already used tag 1 inputs are replaced by 0 and the new partial product (O_6) is generated. This process is repeated in subsequent cycles until all the inputs are utilized and all the outputs are *valid*.

Cycle #	Accumulation Fifos	$Input_1$/tag	$Input_2$/tag	$Input_3$/tag	Accumulation Stage	Output
1	F1 + F4 + F7	$(a_0wa_0)/1$	$(b_2wb_2)/1$	$(c_0wc_0)/1$	Stage 2	O_0(valid)
1	F2 + F5 + F8	(0)/1	$(c_0wb_0)/1$	$(d_0wc_0 + d_2wc_2)/1$	Stage 2	O_2(valid)
1	F3 + F6 + F9	$(c_0wa_0)/1$	(0)/0	(0)/0	Stage 2	O_3(partial)
2	F1 + F4 + F7	$(d_0wa_0)/1$	(0)/0	$(f_0wc_0)/1$	Stage 2	O_4(partial)
2	F2 + F5 + F8	$(e_0wa_0)/1$	$(f_0wb_0)/1$	(0)/0	Stage 2	O_5(partial)
2	F3 + F6 + F9	O_3(partial)	$(d_0wb_0 + d_2wb_2)/1$	$(e_0wc_0 + e_2wc_2)/1$	Stage 1	O_3(valid)
2	F3 + F6 + F9	$(f_0wa_0)/1$	(0)/0	(0)/0	Stage 2	O_6(partial)
3	F1 + F4 + F7	O_4(partial)	-	$(e_0wb_0 + e_2wb_2)/1$	Stage 1	O_4(valid)
3	F2 + F5 + F8	(0)/1	(0)/1	(0)/1	Stage 2	Outputs Exhausted
3	F2 + F5 + F8	O_5(partial)	$(g_0wc_0)/1$	-	Stage 1	O_5(valid)
3	F2 + F5 + F8	(0)/1	(0)/1	(0)/1	Stage 2	Outputs Exhausted
3	F3 + F6 + F9	O_6(partial)	$(g_0wb_0)/1$	$(h_0wc_0)/1$	Stage 1	O_6(valid)
3	F3 + F6 + F9	(0)/1	(0)/1	(0)/1	Stage 2	Outputs Exhausted

FIGURE 17. Accumulation stage map.

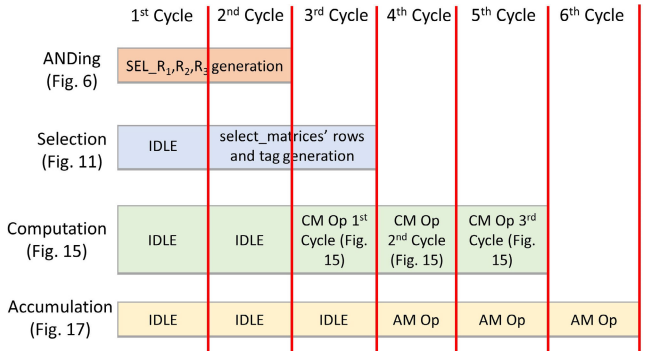

FIGURE 18. Initial latency of the Sparse-PE core operations.

Figure 18 shows the cycle-by-cycle processing latency of the various blocks of the Sparse-PE core for the example in Figure 2. The process of ANDing takes a total of two cycles for the generation of a pair of 3-tuples (SEL_R_1, SEL_R_2, SEL_R_3). After the generation of the first tuple of the SEL_R_1, SEL_R_2, and SEL_R_3 values, the selection block starts processing and takes a total of 3 cycles to generate the select_matrices and the associated tag values (Figure 11). The computation block gets triggered after the first row of the select_matrices is generated and takes a total of three cycles to process the three rows of the select_matrices (Figure 15). Finally, the accumulation block takes a total of three cycles to process the partial product outputs from the computation block (Figure 17) to generate the final outputs. Therefore, the initial processing latency to generate the first output is six cycles. This initial latency, however, is amortized over processing over a larger input.

E. OUTPUT ENCODING

Figure 19 shows the process of output sparse binary mask encoding. Unlike the weight masks, the output binary mask needs to be generated on-the-fly because of its dynamic nature. From Figure 6, we can see that the SEL_R outputs show the presence of the non-zero partial products for a particular convolution chunk. The presence of even a single *one* in the SEL_R outputs show that the output is non-zero. To determine the output binary mask, we can use the same metadata. The first step involves reduction of the individual SEL_R outputs to a single bit (SEL_R_{XR}), based on an *all-zero* check, as shown in Figure 19(a). This generates the sparse binary mask for the outputs before ReLU. Note that the SEL_R values are taken from the test example (Figure 2). Figure 19(b) shows the second step after ReLU, where the negative outputs, and their corresponding sparse mask locations, are converted into zeros. This final sparse mask is stored as is, whereas, the output is shifted first to omit zero data entries, and then stored.

This concludes the processing that goes on in a single Sparse-PE core when convolution is performed using a 3 × 3 filter. For a filter of size 5 × 5, 7 × 7, or higher, the kernel factorization [45] is employed to convert larger filters into a set of smaller 3 × 3 filters. This factorization makes it

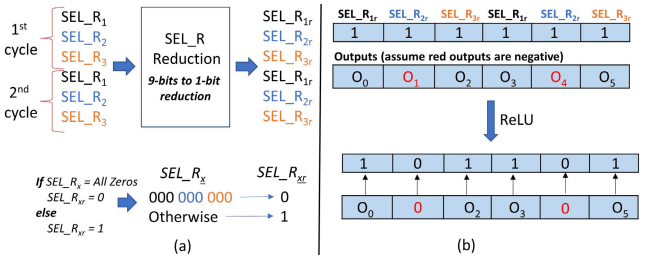

FIGURE 19. Output sparse mask generation (a) 9-bits to 1-bit reduction operation (b) Post-ReLU output generation.

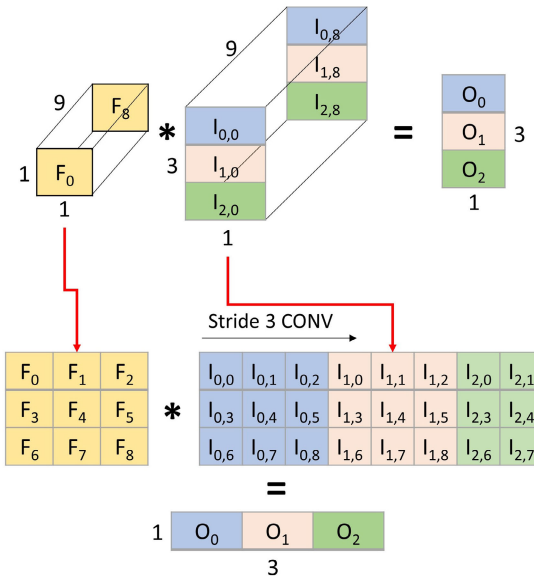

FIGURE 20. 1 × 1 convolution layer transformation into an equivalent 3 × 3, stride 3 convolution layer.

possible to keep the design of PEs relatively simple and not incorporate complex flow control to support larger filters. It should be noted that the kernel factorization is done during model generation and training. The final generated model with 3 × 3 filters is then used during the inference. Sparse-PE, however, also supports 5 × 5 filters by using a multiplier matrix of 5 × 5, instead of the current 3 × 3.

1 × 1 convolution, used in many modern CNN models, is performed the same way as the 3 × 3 convolution. Figure 20 shows an example of 1 × 1 convolution. Here, a 1 × 1 × 9 filter is convolved with a 3 × 1 × 9 input matrix to produce a 3 × 1 output. Figure 20 also shows the transformation employed by the architecture to transform the 1 × 1 convolution operation into an equivalent 3 × 3 stride 3 convolution. Here, individual channels of the filter and the input matrix are transformed into a 3 × 3 matrix and scheduled to the core for processing. The core then performs the selection, computation, and accumulation in the same manner as explained previously. For an input and a kernel with higher channel count, as is the case for most commonly used CNN models, the channels are broken down equally and scheduled among different Sparse-PE cores (Figure 22).

Sparse-PE architecture also allows the processing of FC layers. Figure 21 shows an example of FC layer processing.

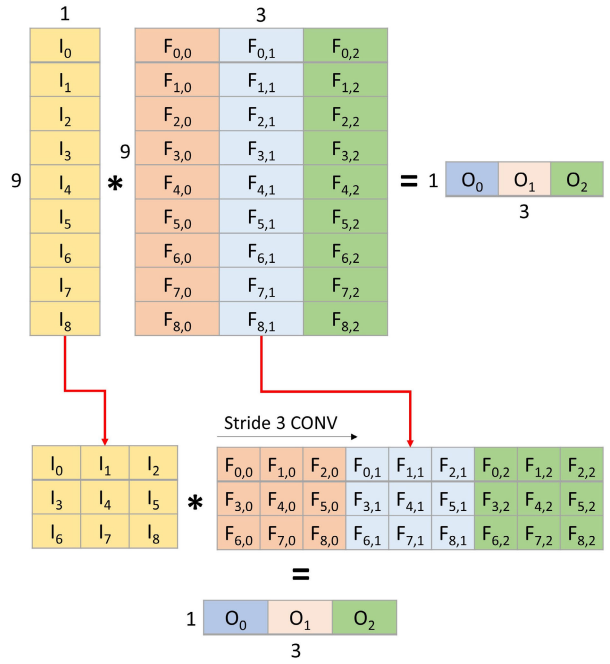

FIGURE 21. FC layer transformation into an equivalent 3 × 3, stride 3 convolution layer.

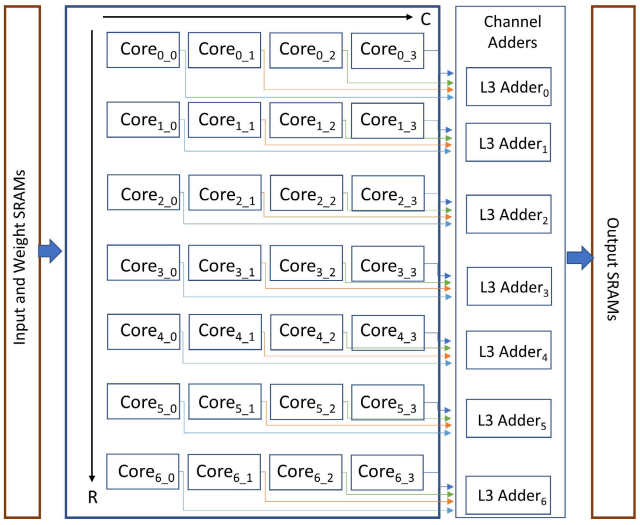

FIGURE 22. 3D Sparse-PE core architecture.

Here, a 9 × 1 input array and a 9 × 3 filter matrix produces a 1 × 3 output. The input and the filter matrices are again transformed into an equivalent 3 × 3 stride 3 convolution. Sparse-PE then processes the transformed input to generate the final outputs.

In the next section, we will show the performance improvement offered by an accelerator that uses a system of Sparse-PE cores to accelerate the CNN inference process.

V. IMPLEMENTATION AND RESULTS

This section describes the performance modeling and the implementation of the Sparse-PE core. An individual Sparse-PE core works by computing dot products between a subsection of the sparse input data and the sparse weight data to produce a subsection of the output data. To process multiple subsections, we envision an $R \times C$ matrix architecture of the Sparse-PE cores, where, R represents the rows and C represents the columns. From Figure 22, we can see that the value of R and C is 7 and 4, respectively, making the total number of Sparse-PE cores equal to $7 \times 4 = 28$. Since, a single Sparse-PE core has $3 \times 3 = 9$ multiplier threads, the matrix of cores in Figure 22 have a total of $9 \times 28 = 252$ multiplier threads. The cores get the sparse data and the binary masks from input SRAMs. The outputs of the cores are connected to level 3 (L3) adders that accumulate individual channel outputs to generate the final output. These outputs are stored in the output SRAMs and subsequently sent to the DRAM for next layer processing. We simulate the architecture using our cycle-accurate simulator and extract the throughput and hardware utilization results. We also implement the Sparse-PE core in RTL Verilog and provide an estimate of resource and power consumption.

A. CYCLE-ACCURATE SIMULATOR

To evaluate the performance of an individual Sparse-PE core and the 7 × 4 matrix of Sparse-PE cores as a whole, we develop a cycle-accurate performance simulator. The simulator generates the results using different values of n for the Sparse-PE cores in Figure 22. The simulator was built using MATLAB R2020a and the Sparse-PE functionality was implemented in software. The Sparse-PE cores in Figure 22 are implemented using a MATLAB function which is provided different data based on the current layer dimensions. The evaluation files in the simulator use the data outputted by the individual cores and schedulers to generate the throughput and the speedup results for various dense and sparse CNN models. The simulator has modifiable n parameter (See Section IV-B) for the individual cores. Recall that n (look-ahead factor) represents the total number of convolution chunks the core looks into during an input processing. For the design example previously presented, we considered the n value to be 3, which indicates that at any particular cycle, the core looks into 3 convolution chunks to determine valid computations. To evaluate the performance, we use different values of the n parameter which allows us to *look-ahead* into a different number of convolution chunks for processing in a particular cycle. The simulator also contains routines for SparTen [23], SCNN [22], and Eyeriss v2 [19] for performing comparisons.

The performance is evaluated on many widely used CNN models including Alexnet [1], VGG16 [2], MobileNet [5], and GoogleNet [46]. We use the sparse versions of VGG16 and MobileNet for comparison purposes. To create sparse versions of the CNNs, we use the approach presented in [18] for pruning using the MATLAB's *Deep Learning Toolbox*. For fair comparison, we achieve the same level of the weight and the input sparsity as the previous approaches and then evaluate the nets for performance comparisons.

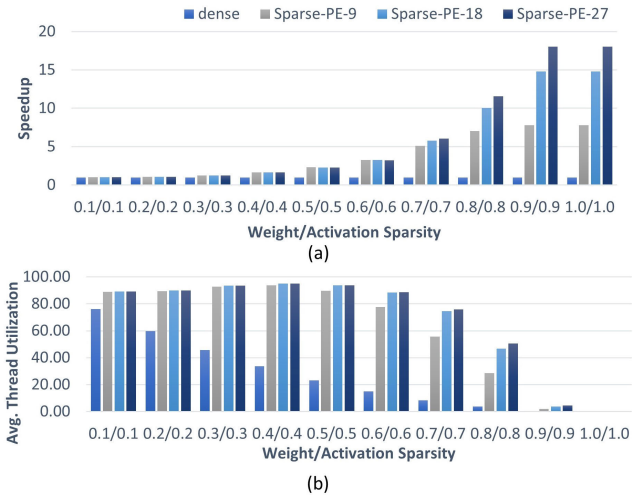

FIGURE 23. VGG16 performance with varying sparsity (a) Speedup results (b) Average thread utilization.

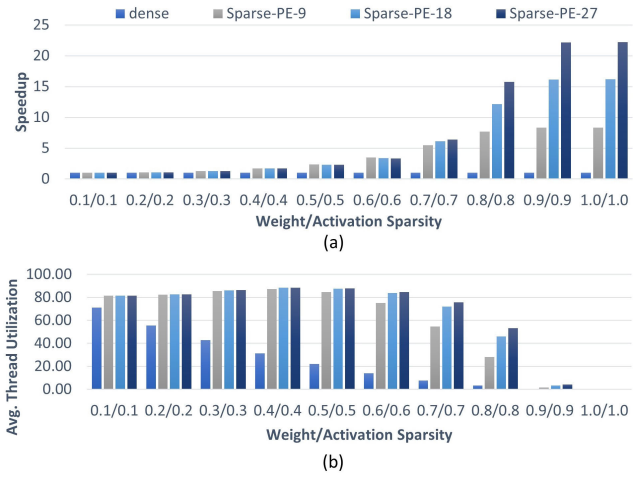

FIGURE 24. MobileNet performance with varying sparsity (a) Speedup results (b) Average thread utilization.

The activation sparsity changes dynamically during the inference process, therefore, we average out the input sparsity for a batch of 100 randomly selected inputs. After pruning of the network, we generate the sparse binary masks for every layer and generate a network containing only the binary masks, since only this information is needed to efficiently represent the MAC operations needed per layer for the accelerator.

1) PERFORMANCE WITH VARYING SPARSITY

Our core simulator has the ability to sweep both input activation/weight sparsity from high (95%/95%) to low (10%/10%). This gives us an accurate measure of how much performance improvement, in terms of speedup and hardware utilization, can be achieved using the Sparse-PE core. Figures 23 and 24 show the speedup results and the average thread utilization for VGG16 and MobileNet, respectively, under varying sparsity. These results are obtained by running the sparse neural nets layer by layer and then averaging out the speedup and the thread utilization for one complete run. It should be noted that all the layers, including the FC layers, are accounted for in this test. Three Sparse-PE core configurations (Sparse-PE-n) are considered with n (lookahead factor) changed from 9 to 18 to 27. At very low sparsity (0.1/0.1), we observe that a dense core and the Sparse-PE core perform somewhat similar in terms of performance. There is a slight increase in the thread utilization, with dense providing almost 80% utilization while all the Sparse-PE configurations providing almost 90% utilization for sparse VGG16 net. For MobileNet, the dense provides almost 70% utilization, while Sparse-PE cores provide almost 80%. For VGG16, the Sparse-PE cores consistently keep the utilization greater than 90% even at high sparsity levels (60%), whereas, as expected, the dense core's utilization decreases by 25-30% and then decreases sharply by 50% at high sparsity levels. This higher thread utilization directly correlates to improved throughput and speedup when compared against a dense design. At 80% sparsity, the Sparse-PE-9, Sparse-PE-18 and Sparse-PE-27 are 7×, 10×, and 11.5× faster, respectively, than a dense design for the sparse VGG16 net. Comparing different versions of the Sparse-PE cores, we see a 57% performance improvement when going from Sparse-PE-9 configuration to Sparse-PE-27 configuration. This shows that the higher the value of the lookahead factor n, the greater is the performance improvement delivered by our Sparse-PE core for sparse input activations/weights.

2) COMPARISON AGAINST PAST APPROACHES

We compare our core design against three previously proposed sparse CNN accelerators (SCNN [22], SparTen [23], Eyeriss v2 [19]) and one dense accelerator (NeuroMAX [11]). Among these designs, SCNN, SparTen, and NeuroMAX do not support FC layers, so we omit our FC layer results for fair comparison. SCNN, in addition, also does not support non-unit stride convolutions. We use the Sparse VGG16 net for comparison with these three accelerators. The average sparsity achieved without a significant loss in accuracy for the weights and activations is 77% and 68%, respectively. Figure 25 shows the comparison results obtained using our simulator. We observe that Sparse-PE-27, on average, performs 11.2×, 4.3×, and 1.96× better than NeuroMAX, SCNN, and SparTen, respectively.

Figure 26 shows the comparison of the Sparse-PE core configurations against Eyeriss v2 on selected layers of MobileNet. The average sparsity for the weights and activations is 73% and 64%, respectively. We observe that, on average, Sparse-PE-9, Sparse-PE-18, and Sparse-PE-27, perform 1.04×, 1.71×, and 2.85×, better, respectively, than Eyeriss v2. Comparing different configurations show that the Sparse-PE-27 configuration offers 108.8% increase in speedup over Sparse-PE-9 configuration for sparse MobileNet.

Energy comparison among different accelerators is somewhat challenging as it requires working RTL implementations. Since the energy consumption is dominated by the total number of DRAM accesses, therefore, by estimating

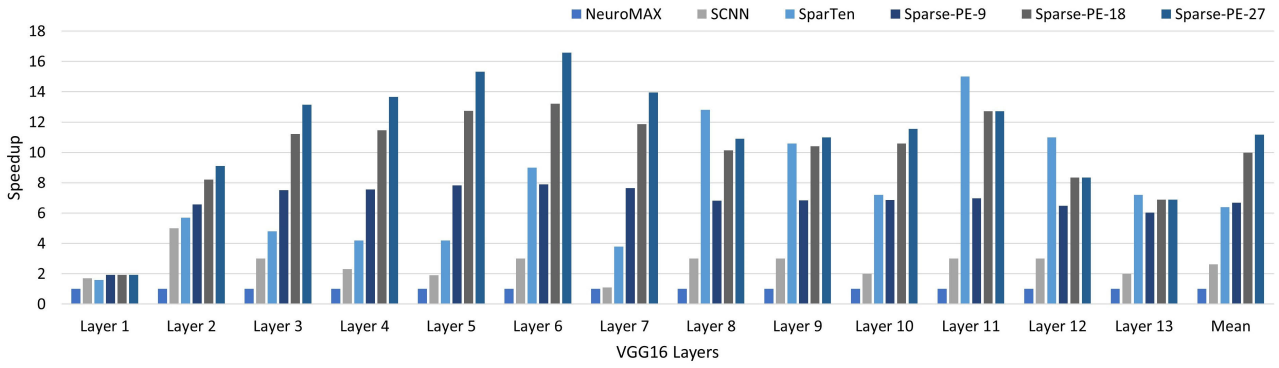

FIGURE 25. VGG16 speedup comparison.

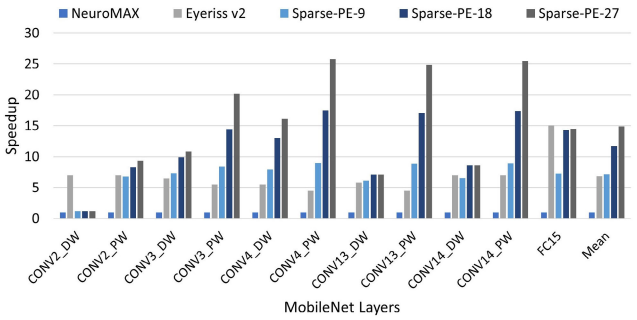

FIGURE 26. MobileNet speedup comparison.

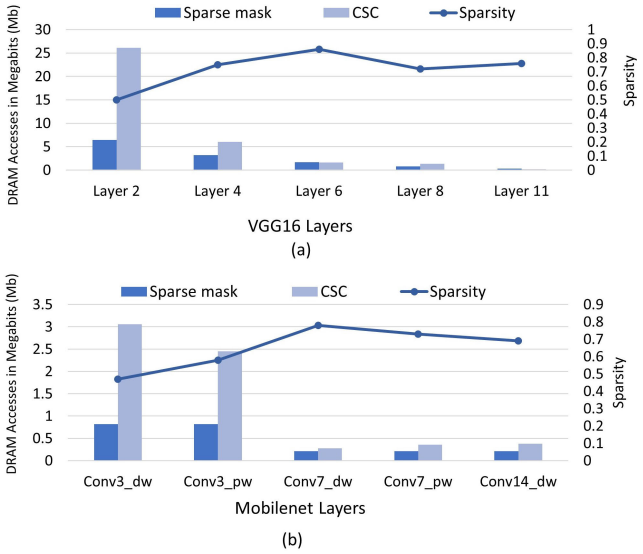

FIGURE 27. Sparse binary mask vs. CSC DRAM access for (a) Sparse VGG16 (b) Sparse MobileNet.

the DRAM accesses, energy difference among the accelerators can be approximated. Many of the recent works rely on the CSC/CSR formats for the storage of the non-zero data. We, therefore, compare the accessed memory for the CSC format against the sparse binary mask format. Figure 27 shows the intermediate activations' memory access comparison for selected sparse VGG 16 and MobileNet layers.[1] The activation sparsity for different layers is also shown. In the initial layers with low activation sparsity, the CSC format has approximately 4× and 3.7× higher DRAM memory accesses than the sparse mask for sparse VGG16 and Mobilenet, respectively. In the deeper layers with moderate to high sparsity, the memory requirement for the CSC format is around 1.7× that of the sparse mask.

This shows that the sparse binary mask format not only needs less encoding/decoding logic, but is also efficient when it comes to memory requirements when compared against the CSC format. This translates directly to higher energy, area, and compute savings for our accelerator which employs the sparse binary mask.

B. RTL IMPLEMENTATION

We use Xilinx Z-7100 SoC to implement the Sparse-PE core design. The SoC is divided into two parts, the programmable logic (PL), containing the FPGA fabric, and the processing system (PS), containing ARM cores. The two are connected using an AXI on-chip communication subsystem. We implement the Sparse-PE core on the PL and use the PS to transfer data to/from a desktop computer to PL. The test design is implemented for the Sparse-PE-27 configuration and runs at 200 MHz. Table 1 shows the resource utilization results for a single Sparse-PE core with $n = 27$. Figure 28 shows the breakdown of the utilization among various sub-blocks of the Sparse-PE core. The LUT and the FF cost is dominated by the selection (SL) block with SL taking almost 40% and 34% of the overall LUTs and FFs used, respectively. The SRAM utilization is dominated by the computation (CM)

TABLE 1. Resource utilization for a single Sparse-PE-27 core.

Property	Available	Used	Utilization (%)
LUTs	277k	3.4k	1.23%
FFs	554k	6k	1.1%
On-chip SRAM	26.5Mb	2.1kB	0.01%

[1]The memory requirement for the non-zero data is not shown since it is the same for both the CSC format and the sparse binary mask. The accessed memory comparison is made for the sparse binary mask and the location vectors (column, index) of the CSC format.

TABLE 2. Comparison of Sparse-PE-based accelerator with previous designs.

Property	Sparse-PE accelerator	Eyerissv2 [19]	SparTen [23]	ZASCA [47]	Zhu et al. [48]	Xie et al. [49]	Lu et al. [50]
Technology	Z-7100	65nm	Cyclone IV	65nm	ZCU102	Aria 10	ZCU102
Precision(bits)	8-bit	8-20 bits	8-bit	16-bit	16-bit	8-bit	16-bit
PE number	252	192	256	192	192	512	288
Frequency (MHz)	200	200	50	200	200	170	200
Accelerator type	$sparse_{wa}$	$sparse_{wa}$	$sparse_{wa}$	$sparse_{a}$	$sparse_{w}$	$sparse_{w}$	$sparse_{w}$
Resources (LUTs(a) , Gates(b))	112k(a)	2695k(b)	unreported	1036k(b)	390k(a)	102.6k(a)	132k(a)
Core only Power (W)	3.7	0.460	unreported	0.301	unreported	4.6	unreported
Supported layers	CONV + FC	CONV + FC	CONV only	CONV + FC	CONV + FC	CONV only	CONV only

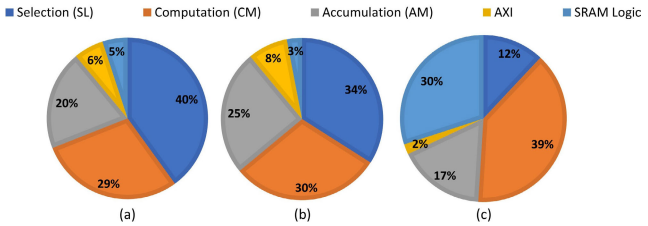

FIGURE 28. Breakdown of resource utilization for (a) LUTs (b) FFs (c) SRAM memory.

block because of the bit mapping and the buffering fifos (Figure 13). The design has a power consumption of 2.48W with the PS dominating the consumption (55%).

Table 2 shows the implementation details of the Sparse-PE accelerator (Figure 22) comprising of the Sparse-PE-27 cores. It should be noted that the PE number in Table 2 refers to the total number of multipliers in the design. Since, a single Sparse-PE-27 core has $3 \times 3 = 9$ multipliers, the total PEs (or multipliers) in the design are $R \times C \times 9 = 252$. The $R \times C$ matrix of the accelerator consumes roughly 85% of the available LUT resources, whereas, the rest 15% are consumed by the additional control logic. Table 2 also lists the details of some recently proposed sparse CNN accelerators. Eyeriss v2, implemented on a 65nm ASIC, supports two-sided sparsity ($sparse_{wa}$), i.e., sparsity in both weights and activations. Eyeriss v2, however, uses a total of 2695k gates which represents a 108% increase in area cost when compared to the original Eyeriss [17]. This is because of the relatively complex design of the PEs of Eyeriss v2. SparTen accelerator [23] is implemented on Intel Cyclone IV FPGA and operates at 50 MHz. It also supports two-sided sparsity but has no support for FC layers. Although the implementation cost of SparTen is not reported, the architecture of SparTen requires complex inter-PE synchronization circuits which would greatly increase its cost. Zero-Activation-Skipping Convolutional Accelerator (ZASCA) [47], implemented on a 65nm ASIC, uses a total of 192 PEs and runs at 200 MHz. Just like Sparse-PE, ZASCA accommodates both the CONV and the FC layers in its architecture. However, unlike Sparse-PE, ZASCA only exploits activation sparsity ($sparse_a$). In addition, because of architectural limitations, ZASCA cannot fully exploit its resources for 1×1 convolution, resulting in a significant decrease in hardware utilization for such convolutions. Zhu *et al.* [48] proposed a structured sparse CNN accelerator, implemented on Xilinx ZCU102 FPGA. The proposed accelerator uses structured pruning to reduce irregularities in sparse weights and employs a sparse wise dataflow scheme for high data reuse. The accelerator proposed in [48], however, only exploits sparsity in weights ($sparse_w$) and cannot exploit activation sparsity. Because of the complex design and the dataflow scheme, the accelerator proposed in [48] has a huge logic utilization cost (390k LUTs). Sparse-PE accelerator, even though, employs 32% more PEs than [48], it, however, utilizes 71% lesser resources. Xie *et al.* [49] proposed a flexible accelerator architecture that supports both structured and unstructured weight sparsity. The accelerator is implemented on Intel Aria 10 SoC and uses a hybrid parallel (HP) dataflow. Even though, the resource count of the accelerator is lower than Sparse-PE, the dataflow employed in the accelerator does not support FC layers. It, also, can only exploit weight sparsity ($sparse_w$) and has no support for activation sparsity. The accelerator proposed in [50] uses a weight-oriented dataflow that performs element-matrix multiplication. It also uses a tile lookup table (TLUT) to match the sparse weight with the input pixel to exploit weight sparsity ($sparse_w$). Similar to [49], the accelerator proposed in [50] is also a CONV only accelerator and has no support for FC layers. It also, cannot exploit activation sparsity ($sparse_a$).

VI. CONCLUSION

This paper proposes Sparse-PE, a multi-threaded, general purpose, dot product core for sparse convolutional neural networks. The designed core is capable of exploiting *two-sided* sparsity, that is, sparsity in both the weights and activations, to maximize the throughput and hardware utilization. Unlike contemporary approaches that use the CSC format and the associated complex PE design, the Sparse-PE core uses the sparse binary mask format and has a relatively low complexity. We also develop novel, low-cost circuits, including selection, computation, and accumulation, which, when used in conjunction, allows the core to skip huge number of computations involving *zero* data and only favor computations involving *non-zero* data to maximize the throughput. Our results show that the Sparse-PE core can effectively keep the hardware utilization above 85% at sparsity as high as 60%, for

both the input activations and weights. We also compared the performance of our core-based accelerator against previous state-of-the-art dense and two-sided sparse CNN accelerators. Sparse-PE offers, on average, 12×, 4.2×, 2.38×, and 1.98×, speedup over NeuroMAX (dense), SCNN (sparse), Eyeriss v2 (sparse), and SparTen (sparse), respectively.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, vol. 1. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015, *arXiv:1512.03385*.

[4] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, vol. 57, Feb. 2014, pp. 10–14.

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.

[6] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," 2018, *arXiv:1801.04381*.

[7] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," 2016, *arXiv:1603.01025*.

[8] S. Vogel, M. Liang, A. Guntoro, W. Stechele, and G. Ascheid, "Efficient hardware acceleration of CNNs using logarithmic data representation with arbitrary log-base," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*. New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 1–8, doi: 10.1145/3240765.3240803.

[9] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed point quantization of deep convolutional networks," 2015, *arXiv:1511.06393*.

[10] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 envision: A 0.26-to-10 TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28 nm FDSOI," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Feb. 2017, pp. 246–247.

[11] M. A. Qureshi and A. Munir, "NeuroMAX: A high throughput, multi-threaded, log-based accelerator for convolutional neural networks," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2020, pp. 1–9.

[12] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 173–185, Jan. 2019.

[13] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[14] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[15] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 1–13.

[16] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[17] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

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

[19] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerging Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019.

[20] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.

[21] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 15–28, doi: 10.1109/MICRO.2018.00011.

[22] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 27–40.

[23] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 151–165, doi: 10.1145/3352460.3358291.

[24] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: An efficient hardware accelerator for convolutional neural networks," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2017, pp. 1–4.

[25] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[26] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 92–104.

[27] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, "PuDianNao: A polyvalent machine learning accelerator," *SIGPLAN Notices*, vol. 50, no. 4, pp. 369–381, Mar. 2015, doi: 10.1145/2775054.2694358.

[28] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," 2015, *arXiv:1502.02551*.

[29] K.-W. Chang and T.-S. Chang, "VWA: Hardware efficient vectorwise accelerator for convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 1, pp. 145–154, Jan. 2020.

[30] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, "An FPGA-based CNN accelerator integrating depthwise separable convolution," *Electronics*, vol. 8, no. 3, p. 281, Mar. 2019.

[31] L. Bai, Y. Zhao, and X. Huang, "A CNN accelerator on FPGA using depthwise separable convolution," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 10, pp. 1415–1419, Aug. 2018.

[32] S. Sharify, A. D. Lascorz, M. Mahmoud, M. Nikolic, K. Siu, D. M. Stuart, Z. Poulos, and A. Moshovos, "Laconic deep learning inference acceleration," in *Proc. 46th Int. Symp. Comput. Archit. (ISCA)*. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 304–317, doi: 10.1145/3307650.3322255.

[33] A. Delmas, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, and A. Moshovos, "Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how," 2018, *arXiv:1803.03688*.

[34] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, X. Ma, Y. Zhang, J. Tang, Q. Qiu, X. Lin, and B. Yuan, "CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 395–408.

[35] A. Ankit, I. El Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-M. Hwu, J. P. Strachan, K. Roy, and D. S Milojicic, "PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference," 2019, *arXiv:1901.10351*.

[36] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with *in-situ*, analog arithmetic in crossbars," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 14–26.

[37] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, "Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 689–702.

[38] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "MatRaptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2020, pp. 766–780.

[39] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 319–333, doi: 10.1145/3352460.3358275.

[40] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 58–70.

[41] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2020, pp. 261–274.

[42] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2018, pp. 724–736.

[43] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, Dec. 2015, doi: 10.1007/s11263-015-0816-y.

[44] F. Chollet *et al.* (2015). *Keras*. [Online]. Available: https://keras.io

[45] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015, *arXiv:1512.00567*.

[46] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. Comput. Vis. Pattern Recognit. (CVPR)*, 2015, pp. 1–9.

[47] A. Ardakani, C. Condo, and W. J. Gross, "Fast and efficient convolutional accelerator for edge computing," *IEEE Trans. Comput.*, vol. 69, no. 1, pp. 138–152, Jan. 2020.

[48] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 28, no. 9, pp. 1953–1965, Sep. 2020, doi: 10.1109/TVLSI.2020.3002779.

[49] X. Xie, J. Lin, Z. Wang, and J. Wei, "An efficient and flexible accelerator design for sparse convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 7, pp. 2936–2949, Jul. 2021.

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

MAHMOOD AZHAR QURESHI received the B.S. degree in electrical engineering from the National University of Science and Technology (NUST), Pakistan, in 2013, and the M.S. degree in electrical engineering from the University of Engineering and Technology (UET), Taxila, Pakistan, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science, Kansas State University. From 2014 to 2018, he worked as a Senior RTL Design Engineer at Center for Advanced Research in Engineering (CARE) Pvt. Ltd., Islamabad, Pakistan. During the summer of 2020, he interned at MathWorks, USA, adding new features in the HDL Verifier toolbox product. During fall 2020, he interned at Tesla, working on the failure analysis of the infotainment hardware for the Tesla Model 3 and Model Y global feet. His contributions resulted in massive savings for the company in Q4 of 2020. He is actively pursuing research in the domain of hardware security and deep learning hardware accelerators.

ARSLAN MUNIR (Senior Member, IEEE) received the M.A.Sc. degree in electrical and computer engineering from The University of British Columbia, Vancouver, BC, Canada, in 2007, and the Ph.D. degree in electrical and computer engineering from the University of Florida, Gainesville, FL, USA, in 2012.

From 2007 to 2008, he worked as a Software Development Engineer at the Embedded Systems Division, Mentor Graphics Corporation. From May 2012 to June 2014, he was a Postdoctoral Research Associate with the Electrical and Computer Engineering Department, Rice University, Houston, TX, USA. He is currently an Associate Professor with the Department of Computer Science, Kansas State University. His current research interests include embedded and cyber-physical systems, secure and trustworthy systems, parallel computing, artificial intelligence, and computer vision. He received many academic awards, including the Doctoral Fellowship from the Natural Sciences and Engineering Research Council (NSERC) of Canada. He earned gold medals for best performance in electrical engineering, gold medals and academic roll of honor for securing rank one in pre-engineering provincial examinations (out of approximately 300,000 candidates).