IJICTR
International Journal of Information & Communication Technology Research

Volume 17- Number 3 – 2025 (19-33)

CNN Accelerator Adapted to Quasi Structured Pruning and Dense Mode

Amirhossein Sadough
Department of AI, Donders Center for Cognition,
Radboud University Netherlands
amirhossein.sadough@donders.ru.nl

Hossein Gharaee Garakani*
ICT Research Institute (ITRC)
Tehran, Iran
gharaee@itrc.ac.ir

Parviz Amiri
Department of Electrical Engineering
Shahid Rajaee Teacher Training University
Tehran, Iran
pamiri@sru.ac.ir

Mohammad Hossein Maghami
Department of Electrical Engineering
Shahid Rajaee Teacher Training University
Tehran, Iran
mhmaghami@sru.ac.ir

Received: 7 December 2024 – Revised: 7 February 2025 - Accepted: 22 March 2025

Abstract—**In recent years, Convolutional Neural Networks (CNN) have been extensively used in machine learning algorithms related to images due to their exceptional accuracy. The multiplication-accumulation (MAC) in convolutional layers makes them computationally expensive, and these layers account for 90% of the total computation. Several researchers have taken advantage of pruning the weights and activations to overcome high computation bandwidth. These techniques are divided into two categories: 1) unstructured pruning of the weights can achieve heavy pruning, but in the process, it unbalances data access and computation processes. Consequently, compression coding for indexing non-zero data increases, which causes much more memory volume. 2) Structured pruning by the specified pattern prunes the weights and regularizes both computations and memory access but does not support high pruning amounts compared to unstructured pruning. In this paper, we proposed Quasi Structured Pruning (QSP) that profits from the high pruning ratio of unstructured pruning. The load balancing property in structured pruning has also been included in the QSP scheme. Implementation results of our accelerator using VGG16 on a Xilinx XC7Z100 indicate 616.94 GOP/s and 1437.7 GOP/s at just 7.8 watts power consumption for dense and sparse mode, respectively. Experimental results show that the accelerator is 1.38×, 1.1×, 2.77×, 2.87×, 1.91×, and 1.18× better in terms of DSP efficiency than previous accelerators in dense mode. As well, our accelerator has achieved 1.9×, 2.92×, 1.67×, and 1.11× higher DSP efficiency besides 4.52×, 5.31×, 10.38×, and 1.1× better energy efficiency than other state-of-the-art sparse accelerators.**

Keywords: Load balance, convolutional neural network (CNN), hardware accelerator, zero-skipping, quasi-structured pruning (QSP).

Article type: Research Article

Publisher: ICT Research Institute

* Corresponding Author

I. INTRODUCTION

Convolutional Neural Network (CNN) has created an extraordinary revolution in deep learning and artificial intelligence applications such as machine vision, image classification, and sound detection. In recent years, CNNs have become a consistent method for learning and detecting problems [1], [2]. However, the advantages of engaging CNNs are challenged by the scope and complexity of computations and significant data volume. Recently, accelerators have become a crucial part of the real-time inference for CNNs. Accelerators based on GPUs, ASICs, and FPGAs have been investigated to attain heavy computation capability. FPGA-based accelerators have been used in [3-5] due to their reconfigurability, parallel and pipeline computation aptitude, performance, and efficient power consumption. Since hardware resource restrictions (computation resources and memory) are a potential adversity in realizing CNNs, specialization of accelerator structure is considered to maximize computation potency and power efficiency [6-9].

Recent research suggests compression methods that reduce the weight density of the network and subsequently achieve high inference speed. These strategies include computation load reduction by pruning the weights of CNN and weight quantization. [10] expressed that dropping the network weights to 10% by pruning slightly affects accuracy. Weight quantization and representing weights with low bit widths can reduce storage bandwidth and computation complexity. Binary Neural Networks (BNN) [11] have afforded an extreme discount on inference time and power dissipation through concise bit representation to +1 and -1.

The pruning exploiting space can be discussed in two approaches: 1) structured pruning and 2) unstructured pruning. Unstructured pruning can prune up to 90% of network weights [10, 12] but brings load imbalance and computation irregularity problems. In addition, non-zero indexing overhead and memory access hierarchy devastation derived from unstructured pruning does not approve of improving accelerator performance. In contrast, structured pruning [13-15] pursues a pruning process with a definite pattern that produces balancing on the load for hardware adaption. A software-based approach is contributed in this manner where the weights diminish first, and then the accelerator will be designed as particular for the adopted pruning strategy. Although the load-balancing nature of structured pruning is hardware-friendly, weight pruning practically does not exceed 60% in this method. Weight pruning requests proper criteria to retain accuracy when removing weights during the pruning. Weight valuation by the absolute magnitude criterion has been applied in [10] and [16] which removes the lowest elements by considering the desired pruning through layer-wise and model-wise comparison, respectively. [17] has inspected various pruning criteria and proved that random pruning has respectable and competitive alongside other schemes.

The CNNs are constructed by variant layers: convolutional layer, pooling layer, activation functions, and fully connected layer (FC). The consistent truth is that convolutional layers have a conquering contribution to CNN computation complexity. Hence, fast convolution techniques such as FFT convolution [8], [18], [19], and Winograd [20] have developed to speed up computation alongside pruning methods. Nevertheless, once an accelerator wants to be designed, designating a conventional or fast convolution algorithm to complete the convolutional layer computations depends on the pruning strategy.

As a result of the abovementioned arguments, structured pruning is hardware-friendly due to its data computation balancing. Still, it has a limited sparsity, while unstructured pruning confirms a heavy pruning level at the cost of more complexity, coding burden, and unbalancing on load. We proposed Quasi Structured Pruning (QSP) to obtain a colossal pruning expanse close to unstructured pruning that assures computation load balancing and regular memory access hierarchy. In addition, this pruning approach does not demand compression coding complexity and process management burden on the hardware. On the other hand, this is a genuinely right case that the pruning process by a conventional method which prunes the weights in a software environment by CPU and then the hardware accelerator employs them for inference, takes much time for preparation. In this work, an accelerator block has been designed for pruning the network weights based on the QSP approach on the hardware, which addresses the vast elapsed time by the pruning process. This block minimizes the time of pruning with negligible resource utilization. This paper presents a load-balanced accelerator that performs efficiently in dense and sparse networks. The main contribution of the paper will be as follows:

- Proposed QSP modifies load imbalance defect of unstructured pruning and covers its accessible pruning amount.
- A simple weight coding has been projected to comprehend the accelerator to skip computations for zero values without complicated procedures and pressure on hardware.
- Weight-shared computation flow has been proposed by establishing two parallelism approaches that promise performance efficiency for dense and sparse modes.
- Active Pruning Block accelerates the QSP-based pruning process on the chip.

The paper is organized as follows: Section II reviews related works; Section III presents proposed accelerator architecture; Section IV is focused on

analyzing accelerator performance and extending results and comparisons; Section V is the conclusion.

II. Related Works

A. Pruning

Accelerator designing based on network optimization frameworks has been regarded in many works. For instance, [21] has introduced a framework to combine software optimization and hardware for sparse CNNs. As mentioned in section I, structured and unstructured pruning approaches have developed recently. **Unstructured Pruning:** [12] has achieved plenty of pruning rates by appointing a threshold parameter and eliminating values less than the threshold. Although this method inaugurates a chapter of entrancing to mobile applications by offering weight reductions of up to 13× for VGG16 without any damage to the accuracy, it demands access to column indexes in CSR format to locate the required inputs. On the other side, load unbalancing of Processing Elements (PE) and the case mentioned earlier has formed an efficiency descent. Compression formats such as CSR [22] and CSC [23] to signify non-zero values when sparsity is applied on both weight and activation bother the hardware for handling the computation flow. One of the significant drawbacks of unstructured pruning is the imbalance of computational loads to PEs since the reduction of weights occurs asymmetrically and eventuates disruption in data scheduling. The idle time of some PEs owing to load imbalance issues does not make it possible for the accelerator to utilize maximum resources in run-time and begets decreasing computational efficiency. [24] has proposed dynamic scheduling that balances the PE loads by placing multiplexers on the input and output of PEs. This operates through managing the multiplexer control states that decide which FIFO should inject into which specific PE. In addition, the multiplexer control states in the output of each PE steer the partial results in their path to accumulation. However, this idea demands high power consumption for efficient utilization of PEs. **Structured Pruning:** recent research has revealed that structured pruning can aid the accelerator to skip computations for zeroes in processing cores conveniently and improves performance and energy efficiency. To obviate the irregularity of pruning, [25] suggests channel-wise, filter-wise, and shape-wise pruning that obeys specific patterns in the pruning process (Fig. 1). Similarly, [26] proposes a shape-wise pruning that prunes similar spatial points along channel kernels and applies this method to regulate the memory access hierarchy and enhance computation efficiency of PEs.

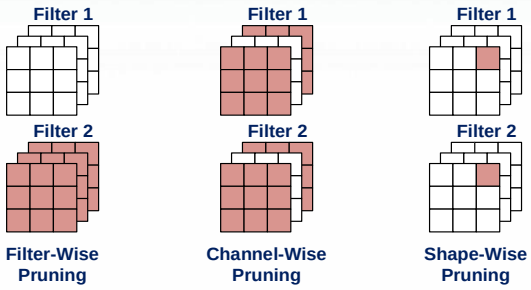


Figure 1. Filter-Wise, Channel-Wise, and Shape-Wise pruning [26], depicted from left to right

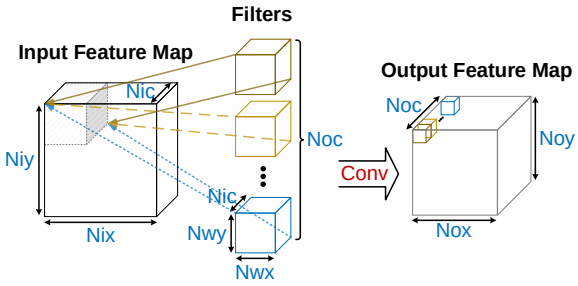


Figure 2. Convolutional layer including input feature map, filters and output feature map

Algorithm 1 Convolutional Layer Computation

Input: Input activation I_{y,x,c_i}, Weight W_{k_y,k_x,c_i,c_o} and Bias B_{c_o}

Output: Output Activation O_{y,x,c_o}

1: **for** c_o **from** 1 **to** N_{oc} **do**
2: **for** x **from** 1 **to** N_{ox} **do**
3: **for** y **from** 1 **to** N_{oy} **do**
4: **for** c_i **from** 1 **to** N_{ic} **do**
5: **for** k_x **from** 1 **to** N_{wx} **do**
6: **for** k_y **from** 1 **to** N_{wy} **do**
7: $\text{Partial\_Sum} += I_{S\times y+k_y, S\times x+k_x, c_i} \times W_{k_y,k_x,c_i,c_o}$
8: $O_{y,x,c_o} += \text{Partial\_Sum} + B_{c_o}$

B. Loop Executing Optimization

The convolutional layer with significant multiplication-accumulation in a few operations loops should accelerate with loop unrolling techniques. A convolutional layer operates by convolving a filter with the dimension of *Nwx*, *Nwy*, *Nic*, and *Noc* as kernel width, kernel height, number of input channels, and number of output channels, respectively, on an input feature map with a dimension of *Nix*, *Niy*, and

Nic where *Nix* and *Niy* demonstrate input width and input height, respectively, and produces an output feature map with a size of *Nox*, *Noy*, and *Noc* where *Nox* and *Noy* represent output width and output height, respectively. A relation between the input and output feature map is always steady, which will be as follows:

$$Nix=Nwx+S(Nox-1)\ ,\ Nox=(Nix-Nwx+S)/S \quad (1)$$
$$Niy=Nwy+S(Noy-1)\ ,\ Noy=(Niy-Nwy+S)/S \quad (2)$$

Where *S* is stride, and zero-padding has been included in Nix and Niy. Fig. 2 illustrates convolutional layer characteristics. Algorithm 1 introduces a convolutional layer in loop format. To compute a convolution layer, lines 1 to 6 must be executed. As the computation of these loops serially takes a long time, loop unrolling and parallelism techniques are essential to accelerate the loops. Loop unrolling repeatedly has been used in State-Of-The-Art accelerators. To better classify these accelerators, in Algorithm 1, line 1 is called loop-1, lines 2 and 3 are called loop-2, line 4 is called loop-3, and lines 5 and 6 are called loop-4. In [9] loops 1, 2, and 4, [6] loops 2 and 4, [27] loops 1 and 3, and [28] loops 3 and 4 have been unrolled. However, these loop-unrolling strategies have been explicitly considered for dense networks and do not bring adequate efficiency for sparse networks. Unrolling of loops 3 and 4 alongside sparse-wise data flow has been offered in [26], making it conceivable to employ both parallelism and zero-skipping in shape-wise format simultaneously. Loop tiling methods to diminish frequent access to external memory and maintain the under-computation data on internal buffers often apply in the accelerators, leading to higher performance and efficiency. [26] and [7] demonstrate that optimization in off-chip memory access and data-sharing ability on weights and activations have been practicable by the loop tiling technique.

An efficient state-of-the-art accelerator should pass these qualifications: 1) loop unrolling and tiling must satisfy different kernel sizes and strides in widely used CNN networks; 2) the designed structure should be scalable and flexible to adjust for corresponding hardware with limited resources; 3) accept both dense and sparse modes; 4) establishing proper pruning approach to speed up the accelerator performance; 5) choosing loop unrolling and tiling techniques based on determined pruning approach to profit from zero-skipping abundantly.

III. Proposed Accelerator

A. Loop Unrolling and Tiling Strategy

The proposed accelerator benefits from two parallelism approaches. These approaches promise performance and energy efficiency in dense and sparse modes. Moreover, according to the proposed pruning, the zero-value computation skipping scheme is acknowledged throughout parallelism approaches. Afterward, loop tiling can be illuminated based on parallelism.

1) Loop Unrolling

Parallelism Approach 1 (PA1) is shown in Fig. 3, where one weight from one input channel of the filter kernel is shared along *Py* pixel in the same column from the input feature map. In each computation cycle, these activations and weights are multiplicated as parallel. Therefore, *Py* is the parallelism coefficient in PA1, representing the number of parallel pixels under computation in an identical x location of the input feature map. Since *Py* parallel multiplication generates *Py* separate partial results in the output feature map that must be accumulated serially, accumulating these partial results entails inaugurating *Py* accumulator.

Parallelism Approach 2 (PA2) is shown in Fig. 4, where *Pic* weight in different input channels of the filter kernel with identical (x, y) locations are multiplicated with *Pic* pixel in the identical (x, y) locations along the activation channels. Hence, *Pic* parallel multiplication along the input channel exposes the necessitation of placing an adder tree to accumulate their results. The proposed parallelism containing PA1 and PA2 is shown in Fig. 5.

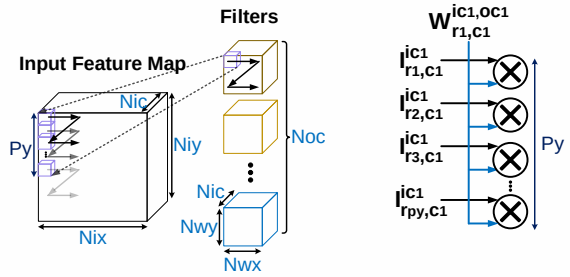


Figure 3. Parallelism Approach 1.

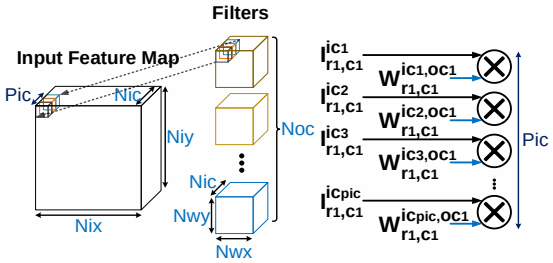


Figure 4. Parallelism Approach 2.

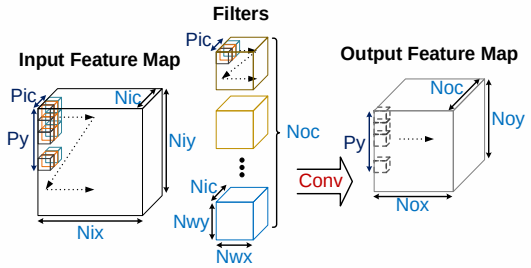


Figure 5. The proposed parallelism with PA1 and PA2

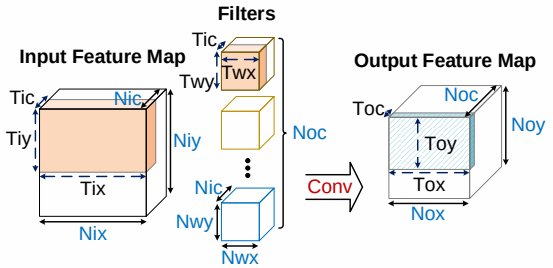


Figure 6. Loop tiling adapted to the parallelism approaches.

2) Loop Tiling

Consistent with PA1 and PA2, data required for computing based on these approaches must be prepared on the chip. A clever tiling strategy would acquire minimum transfers among external memory and on-chip buffer to lessen off-chip memory access, consequently abating power consumption and latency. As can see in Fig. 6, input feature map tiling with a size of $Tix \times Tiy \times Tic$, filter tiling with a size of $Twx \times Twy \times Tic$, and output feature map tiling with a size of $Tox \times Toy \times Toc$ is assumed. Pipeline computation besides PA1 and PA2 to achieve PE efficiency requires constituting suitable tile sizes. To address these, the input feature map tile size has been selected as follows:

$$\begin{cases} Tix=Nix \\ Tiy=Nwy+S(Py-1) \\ Tic=Pic \end{cases} \quad (3)$$

$Tix=Nix$ creates the pipeline computation possibility to form the output tile equal $Tox=Nox$. As well, $Toy=Py$ obtains by similarity relation of line 2 in (3) and (2). Parallelism approaches have not been applied for parallel computation along the output channel. Accordingly, Toc is equal to 1, and remained dimension of the filter tile is designated to be $Twx=Nwx$, $Twy=Nwy$, and $Tic=Pic$. These specifications ensure that the PE arrays continue computation without idle cycles.

B. Accumulation Scheme and Convolution Process

To address PA2 requirements, the adder tree with fan-in of Pic sums the partial results from an array of PE. This array realizes the PA2 strategy. Each operating cycle computes one spatial weight in the kernel means that after passing $Nwx \times Nwy$ cycles, convolution for a region in the input feature map will be finished. Then, the kernel slides along the x-direction on the input feature map with respect to the convolution stride and generates partial results concerning the output feature map. The accumulation scheme in Fig. 7 has been constructed to sustain pipeline computations for kernel weights and the x-direction of the output feature map. The proposed accumulator contains three adder stages, a Partial Sum Temp (PST) buffer, and an accumulator controller. Stage 1 accumulates partial results of convolving spatial weights on an area of the input feature map in $Nwx \times Nwy$ cycles. Since $Pic \leq Nic$, after each $Nwx \times Nwy$ cycle, obtained results are just for Pic channel and not completed.

Hereupon, $\lceil Nic/Pic \rceil$ appoints the number of computation phases to finalize the output result. Therefore, the PST buffer preserves incomplete results in each phase. The held results in the PST buffer and corresponding results in the current phase of computation accumulate by stage 2. In addition, the PST buffer saves the partial result generated by sliding the kernel in the x-direction. In other words, each address of the PST buffer is dedicated to one pixel of the output feature map with the same row. The number of PST buffer addresses should be compatible with different Nix in all network layers. The following equation has been founded to meet the compatibility:

$$Nix\_Max=Max(Nix(L)) \quad (4)$$

where Nix_Max is the number of PST buffer addresses, and L is the variable representing different network layers. When all computation phases are completed, stage 3 accumulates the filter bias with the final result. The accumulator controller with an address signal and a reset signal assists the convolution process in the accumulator. The address signal handles PST buffer addresses for writing/reading data to/from corresponding addresses. On the other hand, the reset signal applies at the first operating cycle of the new region process to ensure the remaining data in the previous cycle does not contribute to accumulating in stage 1.

Fig. 8 illustrates an instance of a convolutional process by the proposed scheme. The accelerator parameters, including Py and Pic, have been assumed to be 1 and 2, respectively.

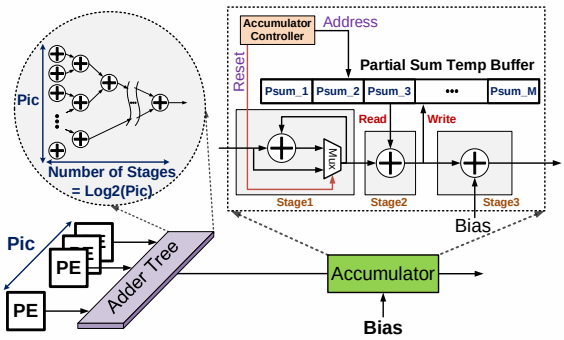
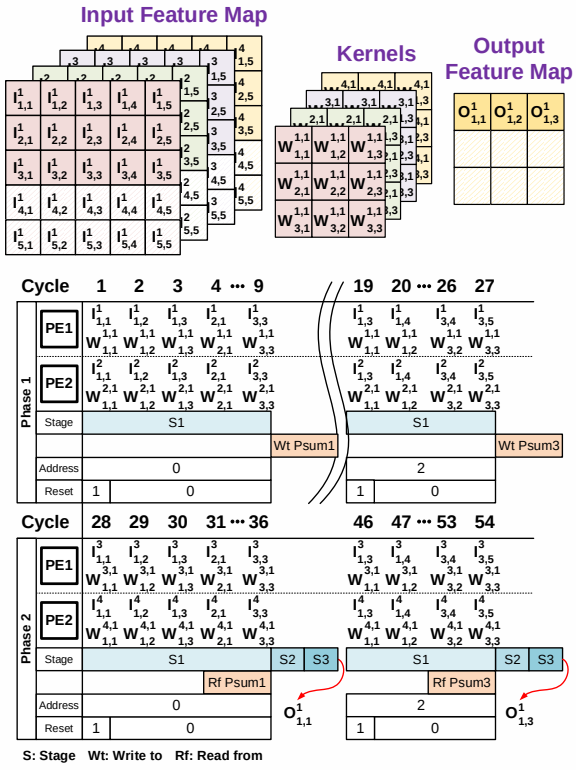


Figure 7. Accumulation scheme including accumulator and adder tree.



Volume 17, Number 3, 2025 (19-33)

Figure 8. Computation of a convolutional layer by proposed accelerator.

The convolution of an input feature map with a size of 5×5×4 and a filter with a dimension of 3×3×4 with stride=1 has led to the output feature map of 3×3×1. As *Pic=2* is fewer than *Nic=4*, the computation phases are divided into ⌈*Nic/Pic*⌉*=2* phases. In phase 1, PE1 and PE2 process the kernels of channel 1 and channel 2, respectively. In cycles 1 and 19, similarly, the reset signal has been applied to refresh the accumulation operation of stage 1 since these cycles are the beginning of the new region process. After elapsing nine cycles, the convolution result for two filter kernels on one region of the input feature map is obtained and should write to its address in the PST buffer. In phase 2, PE1 and PE2 calculate the kernels of channel 3 and channel 4, respectively. As this phase is the last processing phase to get the final results of the output feature map, the corresponding address of the region under computation in the PST buffer must be read before coming to the ninth cycle of the region. For example, $O^l_{1,1}$ and $O^l_{1,3}$ have been computed by accumulating Psum1 and Psum3 from the PST buffer with the achieved output result of stage 1 in cycles 36 and 54, respectively. Stage 2 accumulates these values and guides them to stage 3 for finalizing. As a result, the proposed accumulator utilizes three adders that diminish LUTs practically.

C. Quasi Structured Pruning

The network sparsity based on weight pruning extraordinarily optimizes performance and efficiency. However, structured or unstructured pruning has some strengths and weaknesses that engender challenges to fully utilizing the leverages of the pruning. This paper offers a Quasi Structured Pruning method, which powers from structured and unstructured pruning advantages. In this method, two fundamental principles are admitted: 1) each filter can possess its pruning rate; 2) in each filter, the channels can adopt different zero positions only by noticing an equal number of zeros along channels. Two irregularities created by these guidelines aid grow the pruning rate in the convolutional layer. Fig. 9 shows an instance of QSP-based pruning for a filter with *Nwx=Nwy=3*, *Nic=4*, and *Noc=2*. Filter 1 and 2 have pruning rates of 5 and 6, respectively. Even so, each channel benefits different zero locations with respect to the pruning rate of its filter. According to restrictions by shape-wise pruning [26] in terms of the identical location of non-zero weights per channel, QSP eliminates this boundary to realize vast pruning capacity.

1) Weight Status Coding

Sparse accelerators widely use compression coding and non-zero indexing. This option cuts memory storage, but code/decoding of the data brings a redundant involvedness burden on the accelerator hardware. This work has endeavored to considerably discount resource utilization and hardware complexity to reach higher clock frequency besides lower power consumption. Thus, a simple coding called Weight Status Coding (WSC) has been used to simplify detecting non-zero weights in hardware. As revealed in Fig. 10, a WSC with *Nwx×Nwy* bits determines the status of each weight in the kernel. When a location has a non-zero weight, logic equals 1; when it has a zero value, logic equals 0.

2) Load Balancing

The load imbalance nature of unstructured pruning induces a lack of maximum PE utilization efficiency, which emanates from idle cycles in some of them. Furthermore, organizing data flow to PE will be extra complicated. The QSP-based pruning, PA1, and PA2 constitute an accelerator with a balanced load. To express this feature, Fig. 11 has depicted a convolution layer process for an input feature map dimension of 5×5×2 and the filter dimension of 3×3×2×2 with stride=1.

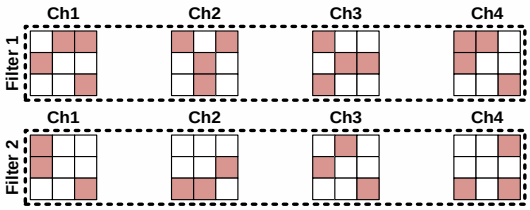


Figure 9. QSP-based pruning for a filter with Nwx=Nwy=3, Nic=4, and Noc=2.

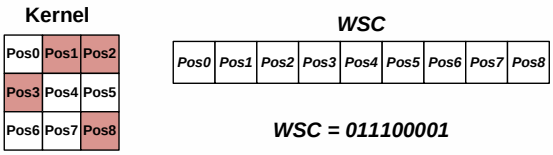


Figure 10. WSC for a 3×3 kernel.

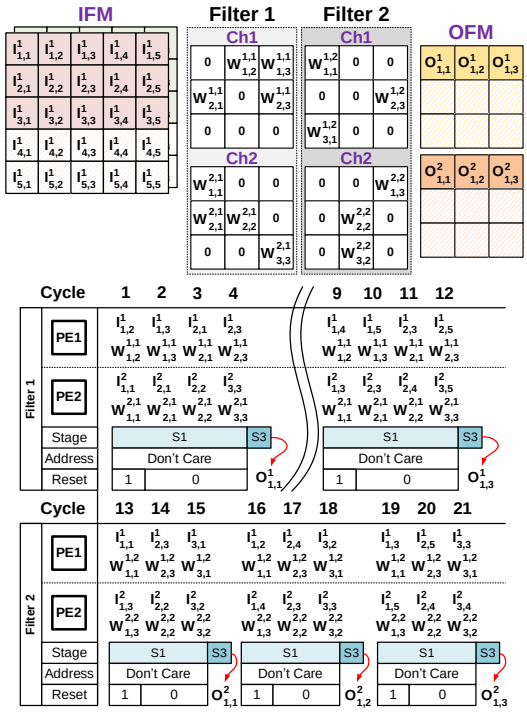


Figure 11. Load balancing and zero-skipping based on QSP.

The accelerator configuration has also been assumed to be *Pic=2* and *Py=1*. Since *Pic* equals *Nic*, each filter has just one computation phase and does not require holding partial results in the PST buffer. So, stage 2 has been ignored. Filter 1 and 2 have a pruning rate of 5 and 6, respectively. Filter 1 has two input channels with non-zero weights equal to four, which balances the effective operating cycle of the channels against each other. Cycles corresponding to zero weights have been ignored in PE1 and PE2 equally in terms of the number by a data flow. The proposed data flow will be introduced later, which brings collaboration to leverage the benefits of cycle equality in parallel PEs. Filter 1 and 2 have spent 12 and 9 cycles to obtain three output pixels, respectively. Meanwhile, Filter 1 and 2 have been computed serially and ensure that different pruning rates of filters have not affected the process. Based on the details above, parallel processing elements are permanently fed by identical load volume that guarantees PE utilization and computation efficiency.

D. Accelerator Architecture

Fig. 12 demonstrates the overall architecture of the proposed accelerator. An array of Processing Units (PU) erects the central core of the convolutional computation. According to Fig. 12, *Py* PUs simultaneously produce *Py* row of the output feature map, which emphasizes PA1. An array of PEs with a length equal to *Pic* as multiplier units has been equipped in PU. An adder tree, an accumulator based on the accumulation scheme, a normalization, and a ReLU have also been embedded in PU. Input activation from *Pic* input channel is stored in *Pic* Input Feature Map (IFM) buffer. Similarly, *Pic* kernel is stored in *Pic* weight buffer.

Gatherer block puts the convolutional results of *Py* PUs in one column and writes them on Output Feature Map (OFM) buffer. Required data transfers between buffers and external memory through DMA. Whenever the layer under process needs a pooling operation, OFM buffer data feeds into Max-Pooling block, and the results will be directed to DMA for storing in external memory. Otherwise, Max-Pooling operation will be disabled. WSC buffer maintains weight status codes related to kernels written in weight buffers. IFM Router dispatches data from IFM buffers to PE arrays. Address Generator (AG) block gets weight status codes from WSC buffer and then creates proper addresses for IFM buffers and Weight Buffers. Also, IFM Router is controlled by the AG block.

1) Data Buffering

According to the projected loop tiling, kernels and activations must prepare on the chip. The pattern of data storing on buffers is a significant matter that characterizes the data flow procedure into PEs.

a) Weight Storing

The pattern of storing the kernels in weight buffers for an instance of *Tic=Pic=2* has been revealed in Fig. 13. Each spatial weight gets one address in weight buffer. The weight buffer depth will be as follows:

$$Weight\_Buffer\_Depth = 2 \times (Nwx \times Nwy) \quad (5)$$

The selected depth has two aspects: 1) the number of addresses to hold all kernel weights equals *Nwx×Nwy*. 2) a ping pong buffer has been realized by creating two sections in each weight buffer. The buffers are based on Dual Port RAM (DPRAM) that enables write and read operations independently. Whenever a section is under computing (reading state), another can prepare the required data (writing state) for the next computation phase. Thus, coefficient 2 in (5) is set for buffer segmentation. Using (5), memory will be allocated in the following manner:

$$WB\_Bits = Tic \times Weight\_Buffer\_Depth \times WB\_Bit\_Width \quad (6)$$

Where *WB_Bit_Width* is the number of bits used to represent the weights, and *Tic* shows the number of weight buffers.

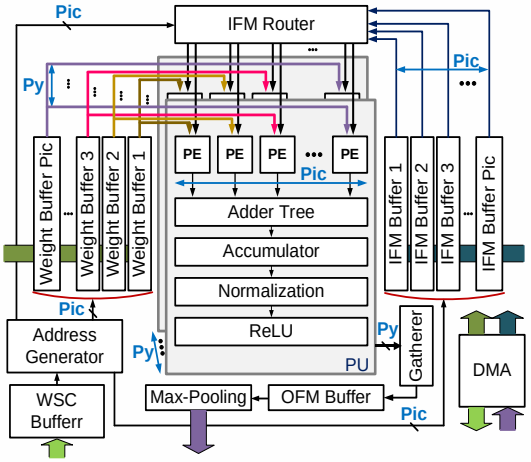


Figure 12. The proposed accelerator architecture.

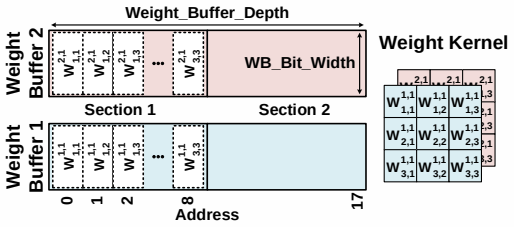


Figure 13. The weigh buffer pattern for an accelerator parameter of pic=2.

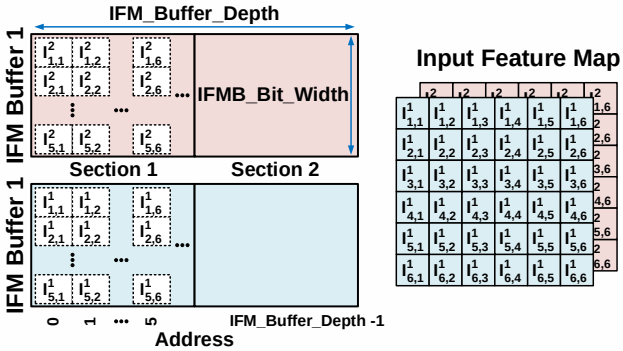


Figure 14. The IFM buffer pattern for an accelerator parameter of Pic=2.

b) Input Feature Map storing

An example of input feature map storing in IFM buffers for *Tic=Pic=2* has depicted in Fig. 14. The determined input tile size in (3) helps to find IFM buffer width in the following:

$$IFMB\_Bit\_Width = Tiy \times Feature\_Bits \quad (7)$$

Where *Feature_Bits* is the number of bits used to represent the activations, and *Tiy* is for confirming the PA1 strategy. Fig. 14 assumes *Py=3* and stride=1 by using (3) results in *Tiy=5*. As convolutional layers of the considered network may have different *Nix*, *Tix* should equal *Nix_Max* in (4) to be compatible with all convolutional layers. The ping pong buffer technique by DPRAM segmentation has also been used for IFM buffers. As a result, IFM buffer depth is:

$$IFM\_Buffer\_Depth = 2 \times Nix\_Max \quad (8)$$

Using (8), memory allocation to IFM buffers will be as follows:

$$IB\_Bits = Tic \times IFM\_Buffer\_Depth \times IFMB\_Bit\_Width \quad (9)$$

c) output Feature Map storing

The final results from *Py* PU being gathered will be written in OFM buffer as one address. Figure 15 illustrates an instance of output feature buffering with supposing *Py=3*. OFM buffer width will be as follows:

$$OFMB\_Bit\_Width = Py \times Feature\_Bits \quad (10)$$

As *Tox* equals *Nox* in the loop tiling, for compatibility of different convolutional layers, *Tox* should be as follows:

$$Tox\_Max = Max(Nox(L)) \quad (11)$$

Where *L* shows different convolutional layers of the network. The depth of OFM buffer with respect to the ping pong buffering method and using (11) is given below:

$$OFM\_Buffer\_Depth = 2 \times Tox\_Max \quad (12)$$

Finally, the allocated memory for OFM buffer using (12) is below:

$$OB\_Bits = OFM\_Buffer\_Depth \times OFMB\_Bit\_Width \quad (13)$$

2) Weight-Shared Data Flow to PE

Data flow to PEs is handled by AG block for weights and activations. The weights read from weight buffers are routed to PEs as straight, where each weight is shared among *Py* PUs. For the activations, an IF Router block scatters the inputs read from IFM buffer between PEs.

d) IF Router

Fig. 16 determines a sample of IF Router block for the accelerator parameters of *Pic=2* and *Py=3*. *IF_Router_Sub_Block1* and *IF_Router_Sub_Block2* distribute data from IFM Buffer 1 and IFM Buff 2 to corresponding PEs, respectively. Each IFM buffer address is placed in a partition register with a division coefficient equal to *Nwy* and the partition stride equal to the convolution stride. For example, when the stride=2, the jump offset of picking the activations and putting them in the partition will equal 2. Consequently, each partition contains *Py* sectors. A multiplexer (MUX) with *Nwy* inputs selects among partitions by the control state of IFR_S and transfers to the distribution register. The distribution register that includes *Py* sectors routes the partition sectors to PEs.

e) Address Generator and Zero-Skipping

AG block adopts two modes for accessing data to PEs: 1) dense mode operation, in which all cycles without ignoring will be done; 2) sparse mode based on QSP pruning that will skip addresses for zero weights. In sparse mode, the zero-weight locations differ in each weight buffer. This comes from the QSP kernel irregularity, which demands particular address generator subblocks for each weight buffer to generate non-zero weight addresses. In addition to the weight buffer, the AG subblock generates ignored addresses and ignored states for IFM buffer and IFR_S, respectively. Hence, *Pic* AG subblock produces fitting characteristics based on WSC codes related to its own kernel. Fig. 17 shows an example AG block for *Pic=2*.

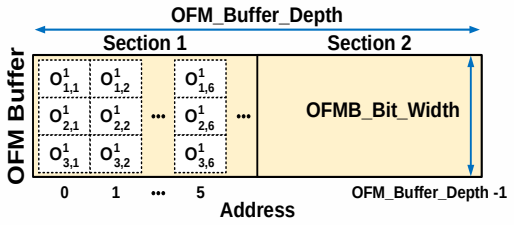


Figure 15. The IFM buffer pattern for an accelerator parameter of Py=3.

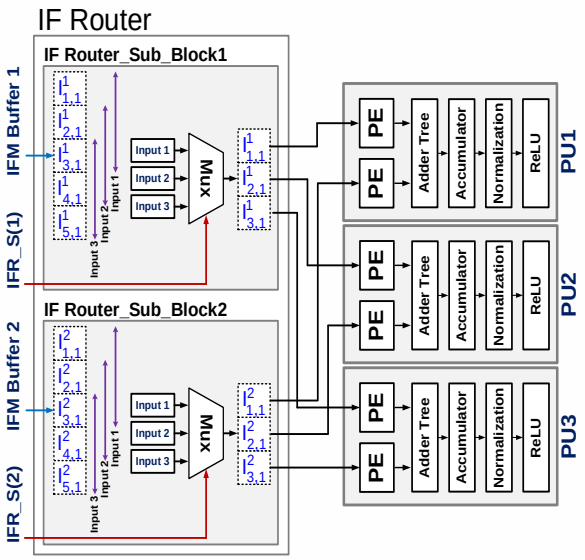


Figure 16. IF Router containing Pic=2 subblock for Py=3.

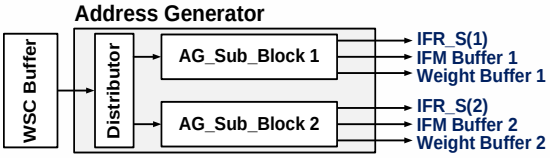


Figure 17. AG block with Pic=2 subblock.

The AG block is illustrated in Fig. 18 as a data flow controller by configuring *Pic=2* and *Py=3* based on weigh buffers in Fig. 13 and IFM buffers in Fig. 14. The pruning rate equals five, and for each computation area, four-cycle have been ignored. Data flow to PEs is controlled by two AG subblocks due to *Pic=2*. In the WSC, the Most Significant Bit (MSB) indicates the status of the first computation cycle since it contains the first weight of the kernel; accordingly, the Least Significant Bit (LSB) shows the last computation cycle. The AG subblock skips a computation cycle when its cycle status in WSC is zero.

AG block achieves different strides by varying the jump offset of addresses. Moreover, different kernel sizes will confirm by altering the number of computation cycles in AG block.

3) Fixed-Point Representation and PE Merging

The activations and weights in the proposed accelerator have been fixed to 8 bits. This item, alongside the weight-sharing feature in the proposed accelerator, makes it possible to merge a pair of PEs. Digital Signal Processing (DSP) units in Xilinx FPGAs such as DSP48E1 have a 25×18 multiplier. Recently, INT-8 multiplication by DSP has been used to decrease DSP utilization in some accelerators [29-31]. However, signed multiplication has remained ambiguous in these works. Adding 8-bit zeros between two 8-bit inputs as one input port of the multiplier and multiplicate with an 8-bit weight, such as Fig. 19 (a), results in a 32-bit output in which the lower 16 bits and the upper 16 bits are the results of multiplying the weight by the first and second inputs, respectively. This technique works well for unsigned multiplication, but various faults will happen when these values are signed, such as in Fig. 19 (b). According to Fig. 20, Merged-PE has been proposed to eliminate signed multiplication problems. Merged-PE is formed by merging two PEs from two adjacent PUs with a shared weight and different inputs. Input values in Merged-PE are converted to absolute values and multiplied according to Fig. 19. Conversely, XOR between the most significant bit of both inputs and weight will control the MUX. The MUX is fed by the positive state and negative state of the output. The positive form of output is ready Inherently, but the negative state obtains by subtracting the positive state from zero. With the lowest component, the Merged-PE scheme multiplies two signed inputs by a signed weight.

E. Active Pruning

The conventional software-based pruning process is realized by pruning the pre-trained weights through chosen pruning approach and then validates by datasets. The pruning process will terminate when the considered accuracy obtains in the validation regarding the desired pruning rate. Then, the pruned weights will be used in the hardware accelerator. This paper offers an Active Pruning (AP) block, which accelerates the pruning process on the accelerator. The AP block prunes the pre-trained weights based on the proposed QSP. To lacking overhead on the hardware, random pruning has been settled in the AP block. Fig. 21 introduces the pruning process by the AP block where *Pic* Random Generator (RG) produces *Pic* WSC for kernels intended to begin convolutional operation. Pruning rate and threshold accuracy are two inputs of the AP block. The AP block operates in four steps: 1) WSC codes for *Pic* channel of the filter generates, then will direct to the WSC buffer; 2) datasets applies to the accelerator for inference operation; 3) the classification result of datasets validates by validation labels; 4) validation result determines whether must save WSCs in external memory and finish the pruning process or return to step 1. Each *Pic* channel from *Nic* channel may prune at its own pruning rate based on the QSP-based pruning process in the AP block. For generating WSCs, RG blocks just spend *Nwx×Nwy* cycles, which have negligible latency for the pruning process.

F. Computational Latency

The convolutional layer latency depends on looping tiling and loop unrolling parameters. The number of multiply for a convolutional layer is as follows:

$$N\_Mult = Nwx \times Nwy \times Nic \times Nox \times Noy \times Noc \qquad (13)$$

Since computations are tiled, considering the size of input and filter beside tile size will have:

$$N\_Tile = \lceil \frac{Nwx}{Twx} \rceil \lceil \frac{Nwy}{Twy} \rceil \lceil \frac{Nic}{Tic} \rceil \lceil \frac{Nox}{Tox} \rceil \lceil \frac{Noy}{Toy} \rceil \lceil \frac{Noc}{Toc} \rceil \qquad (14)$$

Where *N_Tile* is the total number of convolutional tiles that should be computed. The proposed accelerator has chosen tile size of *Twx=Nwx*, *Twy=Nwy*, *Tic=Pic*, and *Toc=1* that modifies (14) to following form:

$$N\_Tile = \lceil \frac{Nic}{Pic} \rceil \lceil \frac{Noy}{Py} \rceil \times Noc \qquad (15)$$

On the other hand, maximum parallel multiplication in a convolutional layer will be as follows:

$$Max = Pwx \times Pwy \times Pic \times Pox \times Poy \times Poc \qquad (16)$$

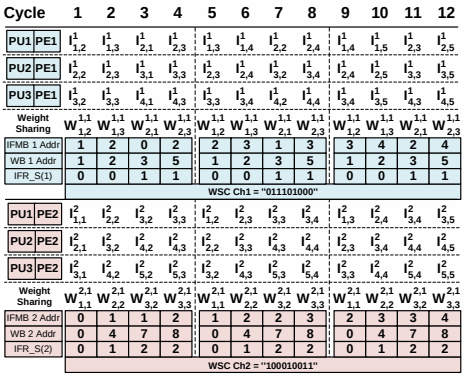


Figure 18. An instance of proposed dataflow to PEs and corresponding AG block outputs

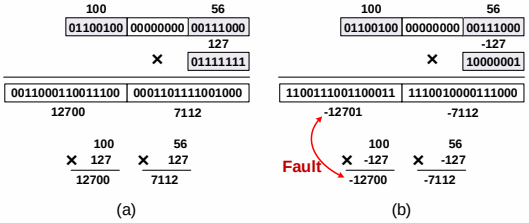


Figure 19. A 24×8 dual multiplication: a) right result for unsigned multiplication b) Fault result for signed multiplication.

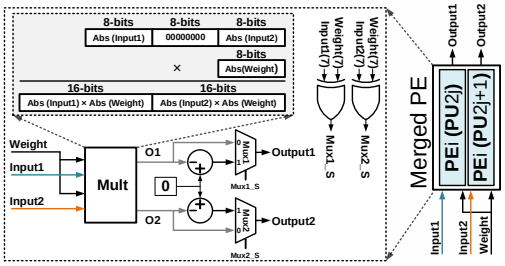


Figure 20. Internal structure of the Merged PE.

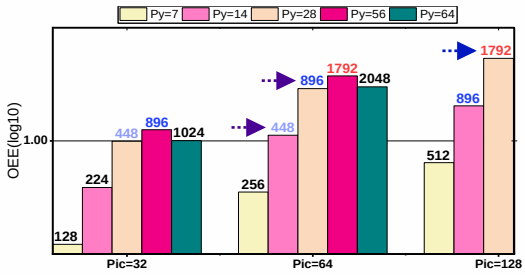


Figure 22. Computation efficiency of all convolutional layer of VGG16 for a) Pic=32, b) Pic=64, and c) Pic=128.

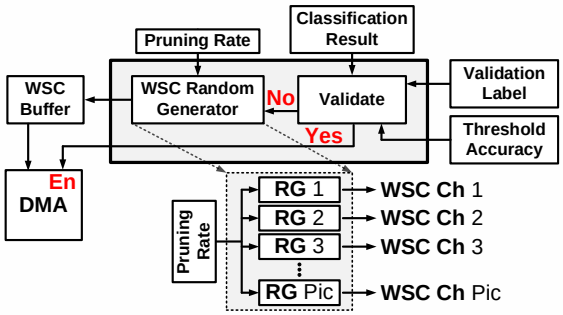


Figure 21. The proposed Active Pruning process.

TABLE I. COMPARISONS WITH OTHER PRUNING METHODS FOR VGG16 CONVOLUTIONAL LAYERS

	Sparsity(%)	Top-1
Deep	67.24	31.17
Cambricon-S[15]	64.83	31.33
OMNI[21]	88.70	31.10
QSP	79.68	30.59

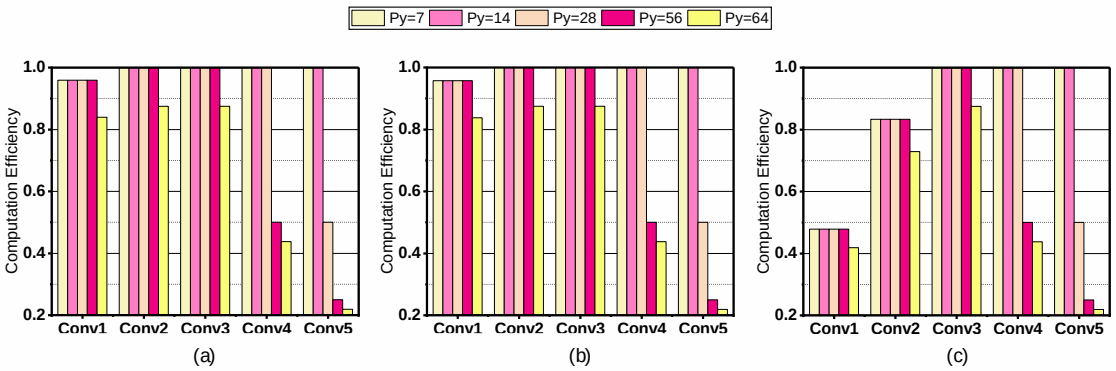


Figure 23. OEE and DSP usage of different configuration.

TABLE II. COMPARISONS WITH STATE-OF-THE-ART DENSE ACCELERATORS USING VGG16

	TVLSI'20 [31]	TCAD'22 [32]	TVLSI'19 [33]	TNNLS'22 [34]	TCAS-II'22 [35]	TCAS-I'22 [36]	Proposed Imp.I	Proposed Imp.II
FPGA	XC7K325t	ZC706	VX690T	VX980T	ZC706	Aria 10	XC7Z100	
Frequency(MHz)	200	150	200	150	200	200	240	
Precision	8bit fixed	8bit fixed	16bit fixed	8bit fixed	8bit fixed	8bit fixed	8bit fixed	
DSP Utilization	516(61%)	900(100%)	1436(40%)	3395(94%)	448(50%)	607(36%)	448(22%)	768(38%)
Logic Utilization(K)	94.7(46%)	-	468(67%)	335(54%)	78 (37%)	207(82%)	38.2(13%)	63(22%)
BRAM	165(37%)	545(100%)	1465(99%)	1492(99%)	168(31%)	769(36%)	202(26%)	271(35%)
Performance(GOP/s)	354	374.98	407.23	1000	326	352.06	334.26	616.94
DSP Efficiency (GOP/s/DSP)	0.68	0.42	0.28	0.29	0.73	0.58	0.746 (0.373[a])	0.803 (0.401[a])
Power (W)	16.5	-	-	14.36	3.338	-	5.117	7.836
Energy Efficiency (GOP/s/W)	21.45	-	-	69.64	97.66	-	65.32	78.73

[a] **The results normalized to 16-bit precision for fair comparisons with [33].**

TABLE III. COMPARISONS WITH STATE-OF-THE-ART SPARSE ACCELERATORS USING VGG16

	TCAD'21 [21]	TVLSI'20 [26]	TVLSI'22 [37]	TCAS-I'21 [38]	IEEE Access'20	Proposed Imp.II	Proposed Imp.III
Sparsification Type (Weight Density)	Structured-Like (12%)	Structured (36.75%)	Unstructured -	Structured (13.2%)	Structured (25%)	QSP (28%)	
FPGA	ZC706	ZCU102	XCVU9P	ZCU102	XC7Z045	XC7Z100	
Frequency(MHz)	166	200	300	300	150	240	
Precision	16bit fixed	8bit fixed	8bit fixed	8bit fixed	8(4)-16[a]	8bit fixed	
DSP Utilization	-	2520(100%)	512(7%)	654(26%)	450(50%)	768(38%)	1536(76%)
Logic Utilization(K)	-	405(67%)	822(70%)	71.992(26%)	163(75%)	63(22%)	124.2(44%)
BRAM	-	1460(80%)	1024(47%)	851(93%)	512(94%)	271(35%)	527(69%)
Power(W)	9.6	17.1	-	23.7	12.85	7.836	15.39
FPS	12.60	92	-	13.54[b]	14	46.50	63.25
Peak Performance	43.70	990.80	307.20	318.22	112.80	475.97	647.13
Achieved (GOP/s)	389.85[c]	2846.50[c]	862.16	418.98[c]	443.60	1437.70	1957
Achieved	-	1.12	1.68	0.64	0.98	1.87	1.27
Achieved (GOP/s/W)	40.60	166.46	-	17.68	34.52	183.47	127.16

[a] Means that 16-bit for input and output feature maps, 8-bit for weights of CONV layers, and 4-bit for weights of FC layers.

[b] 13.54 FPS is calculated according to given latency equal to 73.848ms in the paper.

[c] The value is estimated according to the number of frames per second in the paper.

Division of the tile size by the maximum parallel multiplication shows the number of cycles for the computation of a tile:

$$Cycle\_Tile=\lceil\frac{Twx}{Pwx}\rceil\lceil\frac{Twy}{Pwy}\rceil\lceil\frac{Tic}{Pic}\rceil\lceil\frac{Tox}{Pox}\rceil\lceil\frac{Toy}{Poy}\rceil\lceil\frac{Toc}{Poc}\rceil \quad (17)$$

The proposed accelerator has configured by $Toy=Poy=Py$ and $Pwx=Pwy=Pox=Poc=1$, so (17) will correct as follows:

$$Cycle\_Tile=Nwx \times Nwy \times Nox \quad (18)$$

Eq. (18) is for dense mode. On the opposite, the number of QSP-based computation cycles for a tile will obtain by the following:

$$Cycle\_Tile=(Nwx \times Nwy) \times Pruning\_Rate \times Nox \quad (19)$$

Pruning_Rate will define as dividing the number of non-zero weights by the number of zero weights in the kernels for the tile. Finally, using (18) for dense mode or (19) for sparse mode and using (15), the total number of cycles per layer can be calculated as follows:

$$Cycle\_Layer=N\_Tile \times Cycle\_Tile \quad (20)$$

IV. CONFIGURATION EXPLORATION AND EXPERIMENT

A. A. Experiment Setup

In this work, we used VHDL for RTL implementation of the proposed structure as well as Xilinx Vivado 2021.1 for compile of the codes. To evaluate the proposed accelerator, we used Xilinx XC7Z100 FPGA that includes 277K Look Up Tables (LUTs), 2020 DSP slices, and 755 Block RAM (BRAM) alongside 4GB DDR3 DRAM as an external memory. VGG16 has widely been used to benchmark the performance of state-of-the-art accelerators due to its complexity of computations and vast data. Hence, we exploited VGG16 to evaluate and compare the performance of the proposed accelerator in dense and sparse modes. A theoretical method has been used to estimate the performance and efficiency of different accelerator configurations. Three configurations with the highest performance and efficiency have been selected for comparison. We categorize these configurations in the aspect of resource utilization into low, middle, and high to demonstrate the scalability capacity and performance of the accelerator on low-resource, mid-resource, and high-resource hardware. The Xilinx Power Estimator (XPE) has been used to compare power consumption and energy efficiency.

B. Computation Efficiency

According to the previously-mentioned contents, the proposed accelerator in dense and sparse modes has a balanced load, and sparsity does not contribute to computation inefficiency. In our work, the computation inefficiency comes from the size of the input channel and the output feature map height. AP1 divides *Noy* by *Py*, and AP2 divides *Nic* by *Pic*. So, the computation efficiency of the accelerator will be as follows:

$$Comp\_Eff = \frac{Nic/Pic}{\lceil Nic/Pic\rceil} \times \frac{Noy/Py}{\lceil Noy/Py\rceil} \quad (21)$$

Fig. 22 shows the computation efficiency of the VGG16 convolutional layers for different accelerator configurations. According to Fig. 22 (a), (b), the computation efficiency for the configuration of *Pic=32* and *Pic=64* with a different *Py* is genuinely close. In Conv1 of VGG16, the number of input channels (*Nic*) for the first and second layers equals 3 and 64, respectively, with a 5% and 95% contribution. Based on the first term in (21), the inefficiency difference of *Pic=32* and *Pic=64* is just in the first layer of Conv1, where *Nic=3* causes better efficiency for *Pic=32* than *Pic=64*. Still, this layer has just a 5% contribution of the Conv1 and only creates a little difference in efficiency between these configurations. For other layers, *Nic* is bigger than *Pic=32* and *Pic=64* and divisible. In the case of *Pic=128* based on Fig. 22 (c), Conv1 and the first layer of Conv2 have *Nic<128*, which has caused a computation efficiency drop in

Conv1 and Conv2. Since *Noy≥14* in all layers, the second term of (21) is equal to 1 when *Py* is 7 or 14, which shows maximum computation efficiency by these configurations for *Py*. For *Py=28*, *Py=56*, and *Py=64*, *Noy* is lower than *Py* in Conv5, Conv4-5, and Conv3-5, respectively, which makes the inefficiency based on the second term in (21). Thus, computation efficiency has been degraded by the difference between *Noy* and *Py*.

C. Determine Configuration Using OEE Method

The best accelerator configuration with respect to resource utilization limits needs to be explored based on two criteria comprising computation efficiency and performance. Using (13), (18), and (22), we proposed a measurement method called Overall Efficiency Estimation (OEE) that will be defined as follows:

$$\text{OEE}=\sum_{i=1}^{L}\left[\frac{Comp\_Eff(i)}{Cycle\_Layer(i)}\left[\frac{N\_Mult(i)}{\sum_{j=1}^{L}N\_Mult(j)}\right]\right] \quad (22)$$

Where *L* equals the number of convolutional layers in VGG16. The OEE is affected by *Comp_Eff* per layer, the number of computation cycles per layer, and the computation contribution of each layer toward overall computations. DSP usage will estimate as follows:

$$DSP\_Estimated=Pic\times\lceil Py/2\rceil \quad (23)$$

Where $\lceil Py/2\rceil$ represents Merged-PE scheme. Hence, the combination of efficiency and latency in OEE, and DSP utilization, clarifies the best configuration of the accelerator. Fig. 23 depicts the OEE for different accelerator configurations. In addition, DSP usage in each configuration using (23) has been mentioned above its bar. We have considered three DSP usage for implementation, including 448, 896, and 1792 DSP. As Fig. 23 shows, for DSP utilization of 448, 896, and 1792, *Pic=64* and *Py=14*, *Pic=64* and *Py=28*, and *Pic=128* and *Py=28* are the best configurations, respectively. For DSP usage of 1796, OEE in Pic=128 and Py=28 is better than Pic=64 and Py=56, while Fig. 22 (b) and (c) showed the opposite of that. Therefore, this method truly assists in deciding among different configurations.

D. VGG16 QSP-based Pruning

The QSP-based pruning result for convolutional layers of VGG16 is placed in Table l. The result has been compared with the topmost pruning methods, such as unstructured pruning in Deep Compression [12], structured pruning of Cambricon-S [15], and OMNI [21]. Deep Compression and Cambricon-S have achieved under 70% sparsity, while the dominant consumption of the inference time is related to the convolutional layers. Pattern-Aware pruning of OMNI has concentrated on convolutional layers and achieved 88% sparsity. Although the QSP-based pruning has obtained lower sparsity than OMNI, QSP has a lower top-1 error than OMNI and others. QSP with almost 80% sparsity on convolutional layers has also reached 1.18× and 1.23× sparsity growth up than Deep Compression and Cambricon-S, respectively. We used these pruned weights for sparse mode comparison in Section IV-E.

E. Performance Analysis

This work is focused on gaining sparsity benefits and adapting the accelerator to the QSP approach. However, it has a deserved performance and efficiency in dense mode. Therefore, we present a comparison of dense and sparse modes.

1) Dense Comparison

The proposed accelerator will be evaluated in dense mode by **imp. 1** (*Pic=64* and *Py=14*) and **imp. 2** (*Pic=64* and *Py=28*), which exhibit compact and mid-level configurations, respectively. Results and comparison have shown in Table II. Similar to what we proposed in Merged-PE, [35] leverage from SMF-INT8 that uses from each DSP for implementing two INT-8 multiplication and accumulation. However, SMF-INT8 uses input activation sharing, while Merged-PE uses weight sharing. Of course, [35] shows a better energy efficiency that comes by two accumulation operations in each DSP, consequently LUT reduction, but LUT utilization of **imp. 1** is better. In addition, **imp. 1** has better performance and DSP efficiency in an identical DSP utilization. Albeit **imp. 1** has utilized 13%, 50%, and 26% DSP lower than [31], [32], and [36], respectively, and also has a 60% and 81% lower usage of LUT than [31] and [36], respectively, achieved very close performance as well as 1.09×, 1.8×, and 1.28× better DSP efficiency toward them that shows performance gain of them comes by higher usage of DSP than our imp. 1. In addition to higher DSP utilization [32] than **imp. 1**, it has powered from non-overlap convolution and the block convolution idea so that few convolutional layers can be pipeline to decrease transactions between chip and memory. Apart from [34], **imp. 2** has attained superior results than others, whereas 1.74×, 1.64×, 1.51×, 1.89×, and 1.75× performance speed-up as well as 1.18×, 1.91×, 2.77×, 1.1×, and 1.38× better DSP efficiency against [31], [32], [33], [35], and [36] has obtained, respectively. As well, **Imp. 2** has a 3.67× speed-up in terms of energy efficiency than [31]. While [34] has achieved 1.62× better performance than our **imp. 2**, it is reasonable that it is due to the use of 4.42× DSP resources relative to our implementation. So, this case can be confirmed by the superiority of 1.13× and 2.77× **imp. 2** in energy efficiency and DSP efficiency. Since [33] uses 16 bits precision, the DSP efficiency of our implementation has been normalized by 8/16 for a fair comparison. However, considering DSP efficiency equals 0.401, **imp. 2** has a 1.41× better DSP efficiency.

2) Sparse Comparison

Sparse mode and dense mode do not differ in hardware implementation but differ in the operational task. Thus, resource utilization of the accelerator in dense and sparse modes will be alike. **Imp. 2** (*Pic=64*, *Py=28*) and **imp. 3** (*Pic=128*, *Py=28*) has been selected to compare with other state-of-the-art sparse accelerators. The comparison and results of our work have declared in Table III. In [37], a peak performance evaluation method has been offered to extract the

theoretical computation power of sparse accelerators and the achieved performance used for evaluating computational and performance efficiency to reflect the competence of the sparse accelerators. The achieved performance equals dividing the inference latency by the workload of dense mode. We used these evaluation parameters besides achieved energy efficiency that introduces the real energy efficiency of the accelerators. **Imp. 2** has minimum power consumption (7.836 W) than other sparse accelerators. Furthermore, the energy efficiency of **imp. 2** with 183.47 is the best, so that has 4.52×, 5.31×, 10.37×, and 1.1× better energy efficiency toward [21], [39], [38], and [26], respectively. In terms of peak performance, **imp. 2** has achieved 10.89×, 4.22×, 1.50×, and 1.55× speed-up compared to [21], [39], [38], and [37], respectively, as well as the superiority in DSP efficiency that corroborates the preponderance of **imp. 2** meaningfully. **Imp. 2** has obtained 1.67× superior DSP efficiency than [26], which authenticates that 1.98× peak performance speed-up for [26] caused by 3.27× higher DSP utilization of them versus **imp. 2**. **Imp. 3** as a high throughput implementation, except for [26], has shown better energy efficiency toward others. In terms of the achieved performance, **imp. 3** has achieved a speed-up of 5×, 4.4×, and 4.7× than [21], [39], and [38], respectively. [37] has a better DSP efficiency than **imp. 3**, but 822K LUT and 1024 BRAM usage occupy massive space in hardware and are not compatible with implementing their structure on mid-level or semi-heavy FPGAs. Although [37] has achieved 1.47× higher DSP efficiency than **imp. 3** at the cost of extraordinarily higher utilization of LUT and BRAM, which assists in using lower DSP in their implementation, nevertheless, **imp. 3** has 2.27× speed-up in terms of the achieved performance. [37] uses 1.64× more DSP slice, 3.26× more LUT, and 2.77× more BRAM than our **imp. 3** but acquires just 1.45× better achieved performance, and even **imp. 3** has better DSP efficiency.

As a result, **imp. 2**, as a low-power and efficient configuration with the capacity to implement on low-cost hardware, achieves 1.473 TOP/s in terms of performance, which has superior energy efficiency and greater DSP efficiency among other sparse accelerators. Another side, **imp. 3,** as a high throughput approach, reaches 1.957 TOP/s in terms of performance by consuming 15.39 W.

V. CONCLUSION

In this paper, we proposed QSP-based pruning to attain a high pruning rate and establish sparse accelerators. QSP-based pruning for VGG16 achieved almost 80% sparsity in convolutional layers with an accuracy improvement toward other methods. In addition, an accelerator block for QSP-based random pruning (AP) has been offered, which has not put any redundant burden on the hardware. Finally, the proposed accelerator structure based on QSP has been balanced in PE loads, which does not leave any idle time for PEs. The scalability feature of the proposed parallelism enables the accelerator to implement on embedded or edge-computing platforms. The OEE method has been presented to evaluate different accelerator configurations before implementation. The introduced accelerator in dense and sparse modes without change in the structure has a very respectable performance. Using VGG16 for evaluation, our most optimum implementation (**imp. 2**) has achieved 1.38×, 1.1×, 2.77×, 2.87×, 1.91×, and 1.18× better DSP efficiency than other state-of-the-art dense accelerators. As well, **imp. 2** has achieved 1.9×, 2.92×, 1.67×, and 1.11× higher DSP efficiency besides 4.52×, 5.31×, 10.38×, and 1.1× better energy efficiency than other state-of-the-art sparse accelerators. The proposed accelerator has a minimal power consumption equal to 7.8 W and achieves 616.94 GOP/s and 1437.7 GOP/s in terms of performance for dense and sparse modes, respectively.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May 2015.

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

[3] J. Li, et al., An FPGA-based energy-efficient reconfigurable convolutional neural network accelerator for object recognition applications, IEEE Trans on Circuits and Systems II: Express Briefs 68 (9) (Sept, 2021) 3134–3147.

[4] D. Pestana, et al., A full featured configurable accelerator for object detection with YOLO, IEEE Access 9 (2021) 75864–75877.

[5] C. Park, S. Park, C.S. Park, Roofline-model-based design space exploration for dataflow techniques of CNN accelerators, IEEE Access 8 (2020) 172509–172523.

[6] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of convolutional neural networks onto FPGA," in Proc. 26th Int. Conf. Field Program. Log. Appl. (FPL), Aug. 2016, pp. 1–8.

[7] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing the convolution operation to accelerate deep neural networks on FPGA," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 7, pp. 1354–1367, Jul. 2018.

[8] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in Proc. IEEE 25th Annu. Int. Symp. Field-Programmable Custom Comput. Mach. (FCCM), Apr. 2017, pp. 101–108.

[9] K. Guo et al., "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 37, no. 1, pp. 35–47, Jan. 2018.

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

[11] M. Courbariaux, Y. Bengio, and J. P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations," in Proc. Adv. Neural Inf. Process. Syst., 2015, pp. 3123–3131.

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

[13] C. Ding et al., "CIRCNN: accelerating and compressing deep neural networks using block-circulant weight matrices," in MICRO, 2017.

[14] D. Chunhua et al., "PERMDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices," in MICRO, 2018.

[15] Z. Xuda et al., "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through a Cooperative Software-Hardware Approach," in MICRO, 2018.

[16] J. Frankle and M. Carbin, “The lottery ticket hypothesis: Finding sparse, trainable neural networks,” 2018, arXiv:1803.03635.

[17] D. Mittal, S. Bhardwaj, M. M. Khapra and B. Ravindran, "Recovering from Random Pruning: On the Plasticity of Deep Convolutional Neural Networks," *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018, pp. 848-857.

[18] Lin, J., Yao, Y.: A fast algorithm for convolutional neural networks using tile-based fast Fourier transforms,Neural Process. Lett. (2019).

[19] Zhang, C., Prasanna, V.: Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system.In: Fpga 2017-Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, February 2017 (2017).

[20] Huang, Y. et al.: A high-efficiency FPGA-based accelerator for convolutional neural networks using Winograd algorithm. In:Journal of Physics: Conference Series, 6–8 March 2018 Location: Avid College, Maldives (2018)

[21] Y. Liang, L. Lu and J. Xie, "OMNI: A Framework for Integrating Hardware and Software Optimizations for Sparse CNNs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 8, pp. 1648-1661, Aug. 2021, doi: 10.1109/TCAD.2020.3023903.

[22] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, andW. 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.

[23] 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.

[24] D. Wu, X. Fan, W. Cao and L. Wang, "SWM: A High-Performance Sparse-Winograd Matrix Multiplication CNN Accelerator," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 5, pp. 936-949, May 2021, doi: 10.1109/TVLSI.2021.3060041.

[25] Wen, Wei, et al. "Learning structured sparsity in deep neural networks." Advances in neural information processing systems 29 (2016).

[26] 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," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1953-1965, Sept. 2020, doi: 10.1109/TVLSI.2020.3002779.

[27] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA), Jun. 2016, pp. 367–379.

[28] A. Rahman, J. Lee, and K. Choi, “Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array,” in Proc. IEEE Design, Auto. Test Eur. Conf. (DATE), Mar. 2016, pp. 1393–1398.

[29] J. Guo et al., “Bit-width adaptive accelerator design for convolution neural network,” in Proc. IEEE Int. Symp. Circuits Syst. (ISCAS), 2018, pp. 1–5.

[30] S. Yin et al., “A high throughput acceleration for hybrid neural networks with efficient resource management on FPGA,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 38, no. 4, pp. 678–691, Apr. 2019.

[31] Y. Yu, C. Wu, T. Zhao, K. Wang and L. He, "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 1, pp. 35-47, Jan. 2020, doi: 10.1109/TVLSI.2019.2939726.

[32] G. Li, Z. Liu, F. Li and J. Cheng, "Block Convolution: Toward Memory-Efficient Inference of Large-Scale CNNs on FPGA," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 5, pp. 1436-1447, May 2022, doi: 10.1109/TCAD.2021.3082868.

[33] S. Kala, B. R. Jose, J. Mathew and S. Nalesh, "High-Performance CNN Accelerator on FPGA Using Unified Winograd-GEMM Architecture," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 12, pp. 2816-2828, Dec. 2019, doi: 10.1109/TVLSI.2019.2941250.

[34] W. Huang et al., "FPGA-Based High-Throughput CNN Hardware Accelerator With High Computing Resource Utilization Ratio," in IEEE Transactions on Neural Networks and Learning Systems, vol. 33, no. 8, pp. 4069-4083, Aug. 2022, doi: 10.1109/TNNLS.2021.3055814.

[35] X. Hu, X. Li, H. Huang, X. Zheng and X. Xiong, "TiNNA: A Tiny Accelerator for Neural Networks With Efficient DSP Optimization," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 69, no. 4, pp. 2301-2305, April 2022, doi: 10.1109/TCSII.2022.3150980.

[36] X. Wu, Y. Ma, M. Wang and Z. Wang, "A Flexible and Efficient FPGA Accelerator for Various Large-Scale and Lightweight CNNs," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 69, no. 3, pp. 1185-1198, March 2022, doi: 10.1109/TCSI.2021.3131581.

[37] C. Yang, Y. Meng, K. Huo, J. Xi and K. Mei, "A Sparse CNN Accelerator for Eliminating Redundant Computations in Intra- and Inter-Convolutional/Pooling Layers," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 30, no. 12, pp. 1902-1915, Dec. 2022, doi: 10.1109/TVLSI.2022.3211665.

[38] X. Chang, H. Pan, W. Lin and H. Gao, "A Mixed-Pruning Based Framework for Embedded Convolutional Neural Network Acceleration," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 68, no. 4, pp. 1706-1715, April 2021, doi: 10.1109/TCSI.2020.3048260.

[39] W. You and C. Wu, "RSNN: A Software/Hardware Co-Optimized Framework for Sparse Convolutional Neural Networks on FPGAs," in IEEE Access, vol. 9, pp. 949-960, 2021, doi: 10.1109/ACCESS.2020.3047144.

Amirhossein Sadough received his B.Sc. degree in Electrical Engineering from Semnan University, Iran, in 2019, and the M.Sc. degree in Electrical Engineering from Shahid Rajaee Teacher Training University, Iran, in 2022. He is currently pursuing his Ph.D. at Radboud University, The Netherlands. His research interests include computer architectures, with a focus on Hardware Acceleration of Deep Learning Algorithms Using FPGA and ASIC platforms.

Hossein Gharaee Garakani received his B.Sc. degree in Electrical Engineering from Khajeh Nasi Toosi University of Technology (KNTU), in 1998, and M.Sc. and Ph.D. degrees in Electrical Engineering from Tarbiat Modares University, Tehran, Iran, in 2000 and 2009respectively. Since 2009, he has been with the Department of Network Technology in ICT Research Institute (ITRC) and he is accociate Prof. at ITRC. His research interests include general area of VLSI with emphasis on Basic Logic Circuits for Low-Voltage Low-Power Applications, DSP Algorithm, Crypto Chip, Intrusion Detection and Prevention Systems.

Parviz Amiri received his B.Sc. degree from the University of Mazandaran in 1994, M.Sc. from K. N. Toosi University, Tehran, Iran in 1997, and Ph.D. from Tarbiat Modares University, Tehran, Iran in 2010, all degrees in Electrical Engineering (Electronic). His main research interest is in RF and Power Electronic Circuits, With Focus on Highly Efficient And Highly Linear Power Circuit Design.

Mohammad Hossein Maghami received his B.Sc. degree from Ferdowsi University of Mashhad, Mashhad, Iran, in 2006, the M.Sc. degree from Amirkabir University of Technology, Tehran, Iran, in 2009, and the Ph.D. degree from K.N. Toosi University of Technology, Tehran, Iran, in 2015, all in electrical engineering. Since September 2016 he is with Shahid Rajaee Teacher Training University, Tehran, Iran, as an assistant Professor. He carried out part of his PhD research work at Polytechnique Montreal as a visiting research scholar under supervision of Prof. Sawan. His main areas of interests are implantable Biomedical Microsystems, High-Speed Low-Power A/D Converters and Mixed-Mode Integrated Circuits.