Received: 1 March 2021 | Revised: 8 November 2021 | Accepted: 15 November 2021

DOI: 10.1049/cdt2.12038

IET Computers & Digital Techniques

ORIGINAL RESEARCH

# Sparse convolutional neural network acceleration with lossless input feature map compression for resource-constrained systems

Jisu Kwon[1] | Joonho Kong[1] | Arslan Munir[2]

[1]School of Electronic and Electrical Engineering, Kyungpook National University, Daegu, South Korea

[2]Department of Computer Science, Kansas State University, Manhattan, Kansas, USA

**Correspondence**

Joonho Kong, School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, South Korea.
Email: joonho.kong@knu.ac.kr

**Funding information**

National Research Foundation of Korea, Grant/Award Numbers: NRF-2018R1D1A3B07045908, NRF-2021R1I1A3A04037455

**Abstract**

Many recent research efforts have exploited data sparsity for the acceleration of convolutional neural network (CNN) inferences. However, the effects of data transfer between main memory and the CNN accelerator have been largely overlooked. In this work, the authors propose a CNN acceleration technique that leverages hardware/software co-design and exploits the sparsity in input feature maps (IFMs). On the software side, the authors' technique employs a novel lossless compression scheme for IFMs, which are sent to the hardware accelerator via direct memory access. On the hardware side, the authors' technique uses a CNN inference accelerator that performs convolutional layer operations with their compressed data format. With several design optimization techniques, the authors have implemented their technique in a field-programmable gate array (FPGA) system-on-chip platform and evaluated their technique for six different convolutional layers in SqueezeNet. Results reveal that the authors' technique improves the performance by 1.1×–22.6× while reducing energy consumption by 47.7%–97.4% as compared to the CPU-based execution. Furthermore, results indicate that the IFM size and transfer latency are reduced by 34.0%–85.2% and 4.4%–75.7%, respectively, compared to the case without data compression. In addition, the authors' hardware accelerator shows better performance per hardware resource with less than or comparable power consumption to the state-of-the-art FPGA-based designs.

**KEYWORDS**

accelerator, compression, convolutional neural networks, field programmable gate array, input sparsity

## 1 | INTRODUCTION

Convolutional neural networks (CNNs) have been widely deployed in real-world applications such as real-time object detection and image classification [1]. Along with continuous developments and improvements in CNN models, many hardware accelerators for CNNs have been introduced [2] for not only performance improvement but also energy reduction. In the meantime, though huge efforts have been expended to improve computational performance via hardware-based acceleration, there have been few works focussing on the reduction of off-chip data transfer overheads. Although CNN model parameters are continuously being reduced via model compression and quantization [3], the CNN hardware acceleration still entails huge data transfer between the hardware accelerator and memory via direct memory access (DMA). This data transfer overhead aggrandizes latency, memory bandwidth pressure, and energy consumption, which limits the feasibility of many contemporary CNN accelerators for resource-constrained systems such as embedded and Internet-of-things (IoT) devices. Even though many hardware accelerators attempt to reuse data to minimize off-chip data transfer, the limited on-chip memory of CNN hardware accelerators results in large amounts of data transfer between hardware accelerators and main memory.

The data transfer overhead in CNNs can be classified into two types: latency overhead and energy overhead. The latency overhead could be hidden by double buffering. However, double buffering requires twice more on-chip memory capacity to support the same processing element (PE) utilizations, thus inhibiting its suitability for resource-constrained systems. Furthermore, in the case of huge amounts of data transfer between the accelerator and main memory, data transfer latency cannot be hidden completely by overlapping computations with data transfer. Moreover, overlapping computations with data transfer do not hide the energy overhead of data transfer. The only way to minimize data transfer energy is to reuse on-chip data as much as possible. However, limited size of on-chip memory in CNN hardware accelerators restricts the amount of the data reuse, thus resulting in non-negligible energy overhead. Though processing in memory (PIM) has been emerged to minimize data transfer, PIM would be hard to be adopted in resource-constrained systems. Hence, it is imperative to consider both computational performance and data transfer between main memory and the CNN accelerator when designing CNN hardware accelerator for resource-constrained systems.

To tackle this problem, this paper proposes a novel CNN acceleration technique for resource-constrained systems by leveraging hardware-software (HW/SW) co-design. To reduce data transfer overhead between the hardware accelerator and main memory, it is proposed to use an efficient compression method for input feature map (IFM) data. The proposed scheme exploits the activation sparsity in CNN models, which is attributed to rectified linear unit (ReLU) activation function. The proposed IFM data compression scheme, which is performed on the software side, removes the transfer of zero-valued elements in IFMs. On the hardware side, it is proposed to use an accelerator architecture that efficiently performs convolution operations with our compressed data format. In addition, our proposed hardware accelerator adopts several techniques to further optimize our CNN hardware accelerator. Firstly, it exploits the parallelism that exists in searching and weight matching of non-zero elements (NZEs) from our compressed IFM format. Secondly, it reuses the input data inside of the internal buffer in our accelerator when performing the convolution with $N \times N$ kernels (N > 1), thus eliminating unnecessary internal data transfer inside of the accelerator. Thirdly, it replaces complex operations required for searching and weight matching of NZEs with simpler operations such as lookup table (LUT) and shift operations. The implementation of our HW/SW co-designed CNN accelerator on a field-programmable gate array-system-on-chip (FPGA-SoC) shows an average of 1.1×–22.6× performance improvement with 47.7%–97.4% energy reduction as compared to the case of using embedded/mobile CPU (ARM Cortex-A53 [4])-based CNN inference across the degree of the IFM sparsity (50%–90%). Note that the proposed technique is based on lossless compression of IFMs, meaning that there is no accuracy loss by adopting the technique. In addition, our hardware accelerator implementation shows better cost efficiency (attainable performance per cost) with less or comparable power consumption as compared to other state-of-the-art accelerators [5–9], which means that the proposed acceleration technique is more suitable for low-power, low-energy, and resource-constrained systems.

Our main contributions in this paper can be summarized as follows:

- Proposal of a software-based novel compression scheme that compresses IFM data to reduce the amount of data transfer between the memory and the accelerator
- Proposal of a hardware accelerator design that accelerates CNN inference with the proposed compressed data format
- Introduction of several optimization techniques for hardware accelerator, such as parallel searching and weight matching, input data reuse, and removal of complex operations
- Implementation of the proposed hardware accelerator in an FPGA-SoC platform with two different versions of precision supports, 32-bit floating-point and 16-bit fixed-point, which shows better cost efficiency with less or comparable power consumption compared to the state-of-the-art FPGA-based accelerator designs
- Analysis of the proposed HW/SW co-designed IFM compression technique with the accelerator. Results reveal a huge performance improvement of 1.1×–22.6× and energy reduction by 47.7%–97.4% as compared to a software-based CNN inference on an embedded/mobile processor

The remainder of this paper is organized as follows. Section 2 presents the relevant works in the literature related to CNN acceleration and data transfer reduction. Section 3 presents the background of CNNs and their input sparsity. The proposed HW/SW co-designed CNN accelerator and its implementation on an FPGA-SoC platform are discussed in Section 4. Section 5 presents the evaluation results. Section 6 discusses several limitations of our work. Finally, Section 7 concludes this paper with future research directions.

## 2 | RELATED WORK

There exist many works for sparse CNN accelerator designs. Cnvlutin [10] eliminates multiplication and addition operations with zero values by exploiting the sparsity of the inter-layer data (i.e., caused by activation) in CNNs. Cnvlutin accelerator does not bring zero data from on-chip eDRAM to internal buffers, removing the multiplication and addition with the zero values in the multiply-and-accumulate (MAC) units. Cambricon-X [11] focusses on removing ineffectual operations related to the pruned weights, resulting in the reduction of the required computations. SCNN accelerator [12] exploits both weight and activation sparsity by exploiting Cartesian-product-based internal dataflow architecture. In Ref. [13], to skip multiplications and additions with zero values, a new PE design has been introduced. Sparten [14] introduces an efficient accelerator architecture to find and match NZEs (i.e., inner-join) for efficient vector-vector multiplication with sparse weights and inputs. However, the works introduced above mostly focus only on the accelerator design while the data transfer reduction with the compressed data is ignored.

In Ref. [5], Aimar et al. have presented FPGA- and application-specific integrated circuits (ASIC)-based CNN accelerators that exploit the sparsity in feature maps. Although the CNN accelerators presented in [5] attained a performance of hundreds of giga operations per second (GOPs) in certain CNN models, the accelerators only utilized hardware optimizations for acceleration, whereas our proposed technique also introduces a software-based compression technique along with the hardware accelerator. Furthermore, according to the quantitative comparison that will be presented in Section 5.3, our proposed design shows better cost efficiency (i.e., performance per unit cost or performance per hardware resource) as compared to the FPGA-based implementation of [5]. In Ref. [6], Kala et al. have introduced the Winograd-based CNN accelerator design on FPGA. However, the sparsity in IFMs is not considered in Ref. [6] while our proposed technique utilizes HW/SW co-design methodology for sparsity-aware CNN inference acceleration and memory data transfer reduction. In addition, there have been several FPGA-based designs [7–9] for CNN acceleration. However, our proposed design shows better (or comparable) cost efficiency as compared to those designs as illustrated in Section 5.3. Furthermore, our design also contributes to memory energy reduction and effective bandwidth improvement by adopting our two-step IFM compression technique.

For data transfer reduction of deep neural networks (DNNs), Chen et al. have introduced the Eyeriss design [15] that exploits activation sparsity with data compression by run-length coding. Though data compression and DRAM access reduction have been considered in [15], the corresponding accelerator design cannot fully eliminate the latency caused by the multiplication operations with zero values while merely power gating the PEs with zero inputs. In Ref. [16], Li et al. have proposed an FPGA-based accelerator with an optimization for efficient off-chip memory bandwidth usage. However, their design only focusses on fully connected layers while our design can be applied to convolution layers that consume the largest execution time among the entire layers in CNNs. In Ref. [17], by exploiting activation sparsity during DNN training, Rhu et al. have introduced data compression methods and a hardware architecture for the DMA engine that enables data compression/decompression. However, their focus is DNN training with GPUs while our focus is CNN inference with hardware accelerator optimized for our novel compressed data format.

In Ref. [18], Shen et al. have proposed a CNN accelerator with a flexible buffering scheme that enables a balance between the input and weight transfer to efficiently utilize off-chip data transfer bandwidth. However, the sparsity of data in CNNs has not been considered in their work while our technique reduces both off-chip data transfer via compression and the amount of computations with our novel accelerator design. In Ref. [19], Li et al. have proposed a technique (SmartShuttle) to maximize OPS (operations per second) per DRAM access by layer tiling and scheduling. To find the optimal tiling factor, it elaborates an optimization problem for DRAM access minimization and adaptively applies a scheduling scheme for input-, output-, or weight-reuse while also considering the data compression ratio due to data sparsity. However, SmartShuttle is a software-based scheme to minimize the off-chip data transfer while the amount of the required computations for CNNs is not reduced. In Ref. [20], Zhang et al. have proposed a technique to reduce on-chip SRAM energy in systolic array-based CNN accelerators. By adopting and optimizing the run-length coding, it reduces dynamic energy consumed by accessing the on-chip SRAM memory. In addition, it also employs bank-level low-power modes (drowsy and sleep) to reduce leakage power. However, their proposed technique only focusses on energy reduction while performance improvement by exploiting the data sparsity is overlooked. In Ref. [21], Yuan et al. have proposed a CNN model compression method with the hardware accelerator design. For compression, the method proposed in Ref. [21] classifies the CNN layers into non-pruned (NP) and pruned (P) layers. For error resilience, different compression approaches are employed to NP- and P-layers. For efficient processing of the compressed weights in NP- and P-layers, the proposed hardware architecture introduces a hybrid PE design. For NP-layers and P-layers, finite impulse response-based PEs and shift-accumulator based PEs have been proposed, respectively. However, the compression method proposed in [21] focusses on the weight compression while the IFM compression is overlooked. In addition, due to the data losses during the weight compression, there is an inevitable accuracy loss (0.44% loss for top-5 accuracy in VGG-16 [22]) while ours does not incur any accuracy loss because of the lossless compression of IFMs.

## 3 | BACKGROUND

### 3.1 | Convolutional neural networks

CNNs are increasingly being used in many real-world applications. CNN models typically contain many different layers: convolution layers, pooling layers, fully connected layers etc. Typical CNN models have many convolutional layers and there are several pooling layers in-between the convolutional layers. Fully connected layers generally exist in the last layer of the CNNs. This paper focusses on accelerating convolutional (CONV) layers as CNNs spend most of the execution time in CONV layers when performing CNN inferences.

As shown in Figure 1, the CONV layer operations are composed of convolution, add (sometimes, and scale) bias, and activation. The input to the CONV layer is IFMs, which are comprised of three-dimensional tensors. There is also a weight input that is composed of multiple three-dimensional filters (thus, four-dimensional tensors). One filter has a format of $N \times N \times M$ tensor ($N \times N$ kernel with the depth of $M$). The convolution operations are typically implemented with MAC operations. After the convolution operations in CNNs, the bias can be added (and also multiplied for scale). For each filter, one can have different bias values. After the addition of bias, one performs an activation function. Though sigmoid function was used in the early CNN models, ReLU is currently more widely used due to its computation simplicity with negligible accuracy losses. ReLU activation function rectifies negative values while it merely forwards positive values.

### 3.2 | Sparsity of CNN IFMs

The ReLU activation function results in a huge sparsity in the output feature maps (OFMs) in CNNs. As the output data

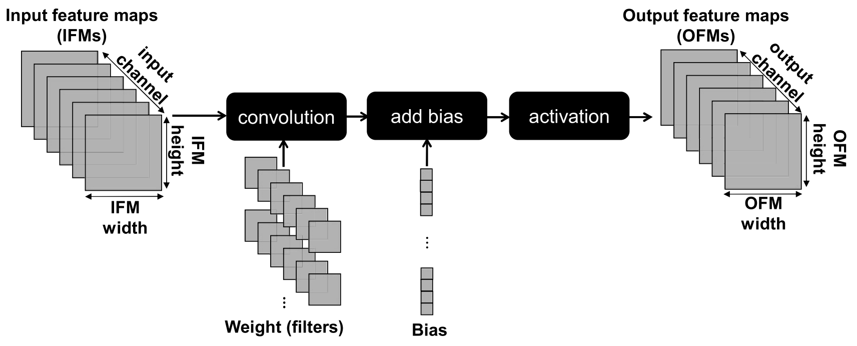

**FIGURE 1** The convolution layer operations in convolutional neural networks

from the previous layer is used as an input for the next layer in CNNs, there is a huge amount of zero data in the IFMs. It means that there could be huge opportunities for reduction of data size and computation by handling only NZEs in IFMs. As reported in Ref. [14], AlexNet [23] exhibits the input density of 20%–38%, meaning that 62%–80% of IFMs comprise zero elements. In VGGNet [22], it is also reported that the average input density is 13%–57%, which implies there could be a huge opportunity to exploit the input sparsity.

## 4 | CNN ACCELERATION AND DATA TRANSFER REDUCTION FOR SPARSE IFMs

### 4.1 | Overview

For CNN acceleration with data transfer overhead reduction, our technique exploits an HW/SW co-design methodology. Figure 2 shows an overall flow of the proposed CNN acceleration technique for sparse IFM. On the software side, the non-compressed IFM is converted into the compressed format before it is sent to the on-chip memory in the hardware accelerator via DMA. The weight data is not compressed and just sent to the hardware accelerator. Though there are also non-negligible zero-valued weights in CNNs when using network pruning [24], it is out-of-scope of this work and left for our future work. On the hardware side, the proposed CNN accelerator performs convolution layer operations with the weight and compressed IFMs. The following subsections present how IFMs are compressed in software (Section 4.2) and convolution operations are performed in the hardware accelerator (Section 4.3).

### 4.2 | Sparse IFM compression

Since there is a huge sparsity in CNN feature maps, the proposed technique only maintains non-zero values (elements) with metadata for a location while zero values are omitted in the IFM. In order to record the location of NZEs in IFMs, the proposed technique also maintains indices of the NZEs in IFMs. Figure 3 shows a format of compressed IFM data. The proposed compression algorithm maintains three entities: *non-zero elements* (32-bit floating-point for each element[1]), *indices* (8-bit for each element), and *count_table* (one element for each chunk[2]). The first part, that is, *non-zero element* part, stores all NZEs in the IFMs. The second part, *indices*, indicates the location of NZEs in a chunk of IFM. The third part, *count_table*, indicates the accumulated number of NZEs from the first chunk to the current one.

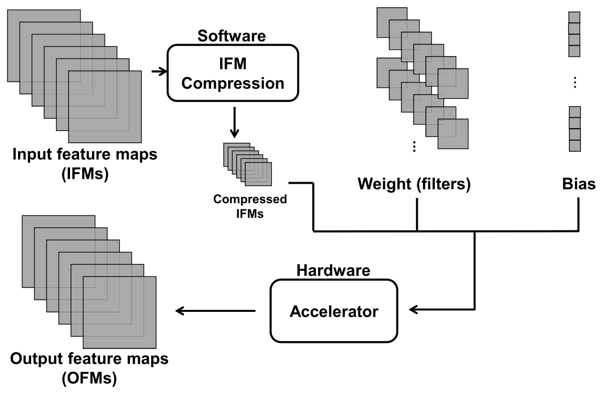

**FIGURE 2** The execution flow of the proposed acceleration technique

The proposed compression algorithm works as follows. Firstly, the 3D tensor IFM is converted into the 1D vector format, which is also divided into a granularity of the chunk (256 elements per chunk in this work). It then linearly searches for the NZE from the first chunk. If it finds the NZE in the chunk, it updates the *indices* by storing the location of the NZE in the chunk (i.e., IFM 1D vector index % 256). It also increases the number of the NZEs of the current chunk in the *count_table*. Also, it stores the element itself to *non-zero elements*. After it finishes the NZE search in a chunk, it starts to search for the NZEs in the next chunk and updates the cumulative number of the NZE in the next *count_table* entry. For example, as shown in Figure 3, since there are three NZEs in the first chunk (IFM[0]-IFM[255]), it records 3 in the *count_table*[0]. In the next chunk, it found another three NZEs, and thus records 6 (cumulative number)

[1]For explanation of the proposed technique, 32-bit floating-point precision for IFM elements is used. However, the proposed technique can also be applied to the cases of using 16-bit fixed-point and 8-bit integer elements by applying quantization.
[2]Although in this work, it is assumed that 256 elements correspond to one chunk, the chunk size depends on the design and can be extended.

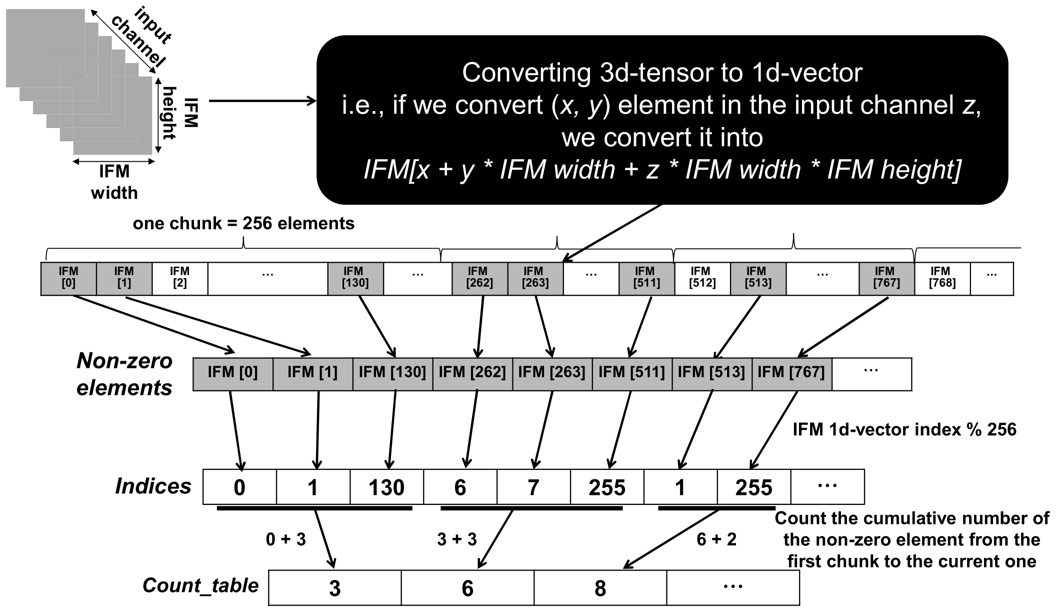
input channel
IFM height
IFM width
Converting 3d-tensor to 1d-vector
i.e., if we convert (x, y) element in the input channel z, we convert it into
IFM[x + y * IFM width + z * IFM width * IFM height]
one chunk = 256 elements
IFM [0]
IFM [1]
IFM [2]
IFM [130]
IFM [262]
IFM [263]
IFM [511]
IFM [512]
IFM [513]
IFM [767]
IFM [768]
Non-zero elements
IFM 1d-vector index % 256
Indices
0
1
130
6
7
255
1
255
0 + 3
3 + 3
6 + 2
Count the cumulative number of the non-zero element from the first chunk to the current one
Count_table
3
6
8

**FIGURE 3** The proposed two-step data compression technique. IFM, input feature map

in the *count_table*[1]. Similarly, it records 8 in the *count_table* [2] corresponding to the third chunk as it founds two NZEs in the third chunk. Following the same procedure, it searches for the NZEs in all the chunks till it encounters the last element in IFM.

Table 1 illustrates a comparison among the proposed two-step compression, conventional one-step compression that maintains NZEs and the corresponding indices (locations), and well-known compressed sparse row (CSR) compression. In Table 1, $N_{\overline{\phi}}$ denotes the number of NZEs and $N_c$ denotes the number of chunks. Table 1 indicates that for the proposed two-step compression scheme, 4B (i.e., 32-bits) are required for storing an NZE, 1B is required for storing the index of an NZE in a chunk (since it performs index%256), and 2B are required to store the cumulative number of NZEs in a chunk. Table 1 also indicates that for the conventional one-step compression schemes, 4B are required for storing an NZE and 2B are required for storing the index of NZEs. The proposed two-step compression scheme leads to a better compression ratio than the conventional one under the condition of $N_{\overline{\phi}} > 2 \times N_c$. Since the maximum allowable number of chunks ($N_c$) is 256 assuming that the maximum number of elements in the IFM is $2^{16}$, the proposed two-step compression scheme leads to a better compression ratio as long as the sparsity of the IFMs is lower than 99.2% ((65,536−512)/65,536 = 99.2%), which is a common case.[3] We also compare our proposed compression scheme with the CSR compression scheme. Note that we also assume the $256 \times 256$ matrix format where $2^{16}$ elements (at maximum) can exist since the CSR scheme can only be used to compress the 2D matrix format. The CSR format requires a column index for each NZE and row pointers to maintain (in a cumulative manner) how many NZEs exist in each row. Thus, we need $N_{\overline{\phi}} \times 1\text{B}$ (8-bits) for column indices because we need to represent 0–255 column indices. For the row pointers, we assume the worst-case requirements because the data size for row pointers depends on the sparsity and data distribution pattern in the matrix. Since the row pointer should record the cumulative number of NZEs until the corresponding row (i.e., from the first row to the corresponding row), the maximum value of the row pointer is $2^{16}$, meaning that we require 16-bits (2B) for each row pointer value (technically, though we need 17-bits for the case where all the elements in the matrix are non-zero value, we exclude this case because it is a very rare case and not desirable to apply CSR compression). In addition, the number of required row pointer elements is the same as the number of rows in the matrix. Thus, 256 ($2^8$) row pointer values are required. Thus, for the CSR format, we need $N_{\overline{\phi}} \times 4\text{B} + N_{\overline{\phi}} \times 1\text{B} + 2^8 \times 2\text{B}$. As we explained, since the maximum allowable number of chunks ($N_c$) of our proposed compression technique is 256 assuming that the maximum number of elements in the IFM is $2^{16}$, our two-step compression technique shows a compression ratio comparable to the CSR format.

On the other hand, the CSR format can only be applied to a 2D sparse matrix because it can only contain the column indices and row pointers, which are two-dimensional positional metadata. On the contrary, our compression technique can also be employed to compress the 3D tensors because we flatten the sparse tensor to a 1D vector before data compression. If we would like to use the CSR compression for a tensor (i.e., IFMs) compression, we can separately compress each 2D IFM and combine the compressed data of each IFM. However, when performing the compression with a large number of input channels, it may seriously increase the metadata (column indices and row pointers) size because the metadata size will be proportional to the number of input channels. In addition, depending on the IFM size (height and width) and the number

[3]Since the proposed acceleration technique is geared toward resource-constrained systems, it is assumed that the maximum number of elements in the IFM is $2^{16}$ (256 KB when using 4B floating-point format for an element).

**TABLE 1** Compressed data size comparison between conventional one-step compression, CSR-based compression, and the proposed two-step compression assuming the maximum number of elements in the IFM is $2^{16}$

| Compression scheme | Data stored | Memory required |
|---|---|---|
| Conventional one-step compression | Data element + indices | $N_{\overline{\phi}} \times 4\text{B} + N_{\overline{\phi}} \times 2\text{B}$ |
| Conventional CSR compression | Data element + column indices + row pointers | $N_{\overline{\phi}} \times 4\text{B} + N_{\overline{\phi}} \times 1\text{B} + 256 \times 2\text{B}$ |
| Proposed two-step compression | Data element + indices + Count_table | $N_{\overline{\phi}} \times 4\text{B} + N_{\overline{\phi}} \times 1\text{B} + N_c \times 2\text{B}$ |

*Note*: $N_{\overline{\phi}}$ and $N_c$ denote the number of non-zero elements and chunks, respectively.
Abbreviation: CSR, compressed sparse row.

of input channels, the metadata size will vary. It may hugely increase accelerator hardware complexity due to variable input dimensions. On the contrary, since our compression technique converts the 3D IFM tensor into a 1D vector format, our hardware accelerator operations are not affected by the tensor dimension, relieving the hardware design complexity. Since our acceleration technique is based on HW/SW co-design, we need a tradeoff between the compression ratio and hardware complexity. Even though some other compression techniques may be a little better in terms of the compression ratio, they can increase hardware complexity for decompression and convolution, deteriorating the overall performance.

In our compression technique, we remove the zero values to reduce the required storage size. At the same time, the non-zero values still remain in the compressed data with their location information within the tensor. Although we remove the zero values in the compressed data, it is not actually removed from the tensor. In other words, if we restore (i.e., decompress) our compressed data to the original data, we can completely restore the tensor same as before compression (i.e., without any data loss). This is because our compressed data only contains the information on non-zero values (i.e., values and locations), whereas all the remaining spots of the tensor can be filled with zero values, meaning that our compression technique is lossless compression.

## 4.3 | CNN accelerator for compressed IFMs

### 4.3.1 | Hardware design overview

Since the proposed acceleration technique compresses the IFMs, it is crucial to efficiently perform convolution operations with the compressed data format. The proposed hardware accelerator performs convolution operations with only NZEs and their location information (extracted from *Indices* and *Count_table*) and avoids the MAC operations with the zero operands,[4] eventually resulting in performance and energy benefits proportional to the degree of input sparsity. Figure 4 describes the overall execution flow of the proposed hardware accelerator.

The first part of the proposed hardware accelerator is searching and weight matching. It brings the compressed IFMs (*non-zero elements*) with metadata (*indices* and *count_table*) and stores them in on-chip memory in the accelerator (① in Figure 4). Firstly, it brings NZEs to the *non-zero element buffer* before performing the MAC operations (② in Figure 4). To perform MAC operations with sparse inputs, it is crucial to align the weights with the NZEs to be multiplied, which is done in the *Non-zero alignment logic*. In the *Non-zero alignment logic* (③ in Figure 4), there are *filter alignment logic* and *filter offset buffer*. *Filter alignment logic* (④ in Figures 4 and 5) finds the offset of the weights in the filter, which will be multiplied with non-zero IFM elements. Figure 5 shows a detailed operation of the filter alignment logic. To perform index matching, it first restores the original IFM index in the 1D vector format by using the *Indices* and *Count_table*. It then performs a comparison between the 1D vector format indices of the NZEs and the indices of the IFM area where it will perform the convolution operations (dark-grey shaded buffer in Figure 5). If there are matched indices, it stores the filter indices to the *filter offset buffer* (⑤ in Figures 4 and 5) in a first-in first-out (FIFO) manner.

After filling the *filter offset buffer*, the weights are loaded to the *filter buffers* (⑥ in Figure 4). The values in the *filter offset buffer* indicate the location of weight values in the filter. For example, Figure 4 shows that the weight value in the filter indices[5] 0, 5, 7, 9, 14, 21, 22, and 24 will be selected and loaded into the *filter buffers*. This process of loading filter weights into filter buffers is also depicted in the weight loading part of Figure 4. For example, filter0[0] corresponds to the weight value 6 in Figure 4, and thus, it loads 6 in the *filter buffer*. The rest of the weight values are also loaded into the *filter buffer* in the same manner. Note that the *filter buffer* holds 16 entries in our hardware design and implementation although only eight entries are shown in Figure 4. Once 16 weight values are loaded into the *filter buffer*, it performs MAC operations with the aligned NZEs and the corresponding weights in the PEs, followed by bias addition and activation (ReLU) (⑦ in Figure 4). The OFM elements are also stored in the on-chip memory. The proposed hardware accelerator design exploits output channel-level parallelism, which means each PE performs the MAC operations for each output channel. The

[4]Though we compute the MAC operations with only NZEs, it does not hurt accuracy as compared to the case without our compression and acceleration technique because MAC operations with the zero values are ineffectual (i.e., $N \times 0 = 0$).

[5]1D vector index is used instead of 3D tensor index in the filter. It means Filter[z][y][x] = Filter[x + y*width + z*width*height].

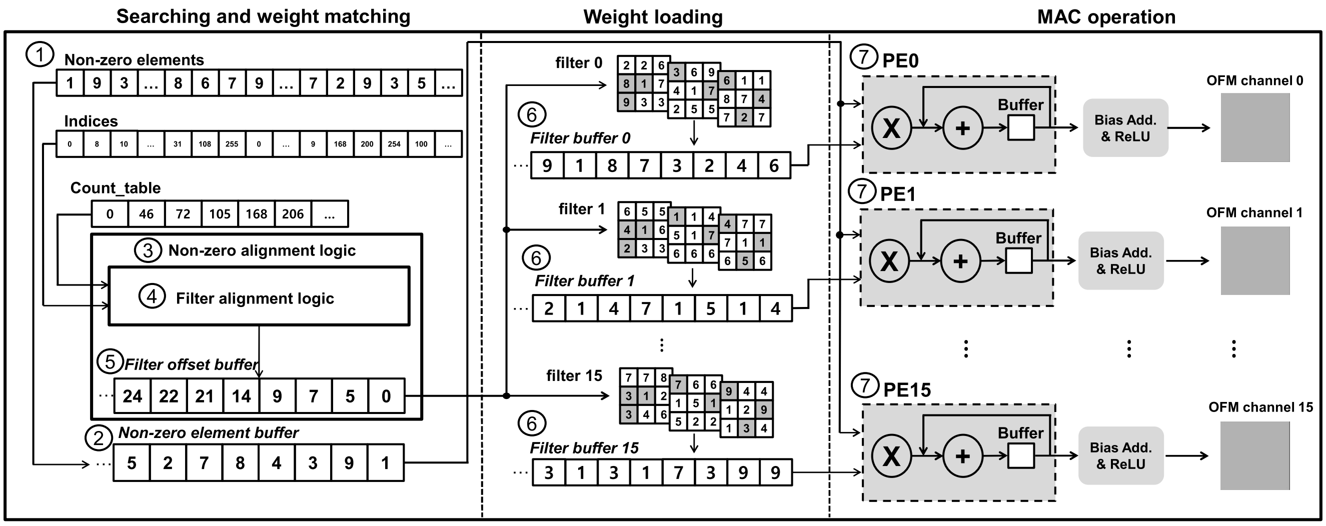

**FIGURE 4** Overall architecture of our convolutional neural network accelerator. OFM stands for output feature map

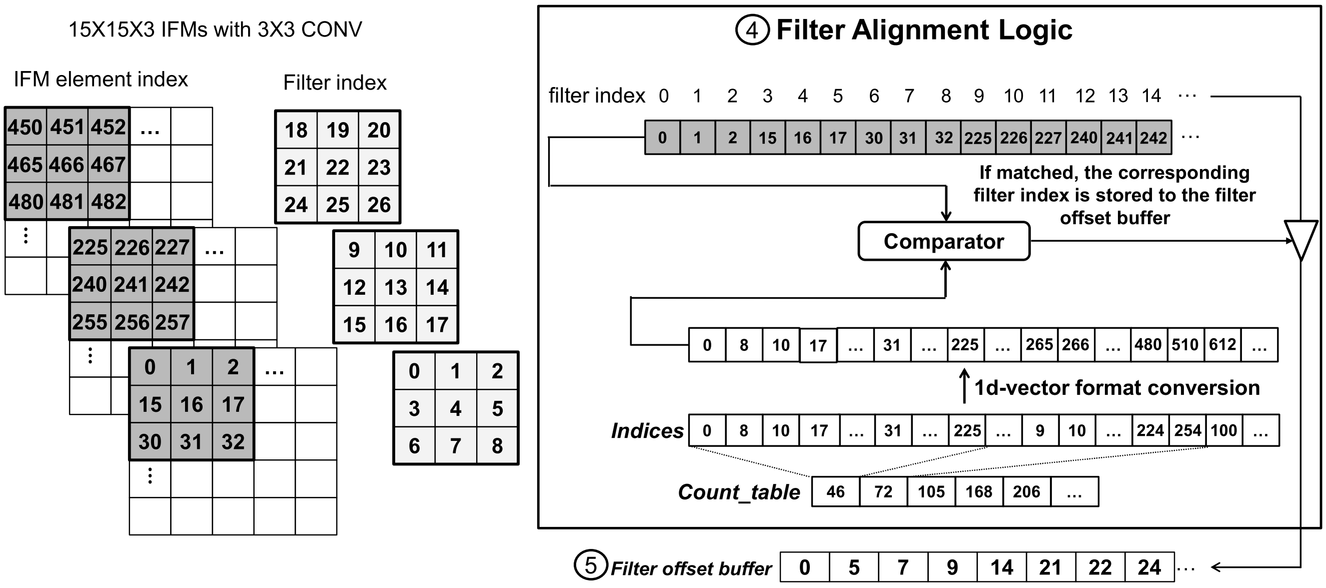
15X15X3 IFMs with 3X3 CONV
IFM element index
Filter index
Filter Alignment Logic
filter index
If matched, the corresponding filter index is stored to the filter offset buffer
Comparator
1d-vector format conversion
Indices
Count_table
Filter offset buffer

**FIGURE 5** Architecture and operations of filter alignment logic. IFM stands for input feature map

proposed hardware accelerator design for 32-bit floating-point precision[6] uses 16 PEs though the number of PEs is also design-dependent.

In the filter alignment logic, the storage for 1D vector format conversion may require the non-negligible storage overhead. Assuming that the maximum number of NZEs is $2^{16}$, we need 128 KB ($2B \times 2^{16}$) on-chip buffer to temporarily store the converted indices for 1D vector format. It accounts for the on-chip buffer size of 10.3% (=128 KB/1247 KB; 1247 KB is our total BRAM usage in our FPGA prototype with floating-point 32-bit precision support), which is not a significant overhead. To alleviate this overhead, the counter or on-demand conversion may help reduce the on-chip buffer requirements of our design, though it should be done with further design optimizations. Thus, we leave it as our future work.

### 4.3.2 | Design optimization techniques utilized in the hardware accelerator

While the back-end (MAC PEs, bias addition, and activation) logic is very similar to the conventional CNN accelerators, the front-end logic (searching and weight matching, and weight loading) is newly introduced in the proposed design. Though the baseline (i.e., introduced in the previous subsection) hardware design can bring a non-negligible performance benefit, one can further optimize the front-end of an accelerator for even better performance. The proposed accelerator

[6]We have also implemented 16-bit fixed-point precision-based accelerator with 32 PEs, which will be explained in Section 4.4.

design leverages three design optimization techniques for the front-end part of the accelerator.

- Parallel Searching and Weight Matching: The most performance-critical aspect of the proposed hardware accelerator is searching and weight matching that finds and aligns filter weights that will be multiplied with the NZEs in IFMs. It needs to compare the NZE indices to the original 1D vector index in the IFM, which is performed sequentially in the baseline design. For better performance, it uses multiple comparison logics (16 in our design) so that the searching and weight matching can be performed for multiple NZEs in parallel
- Input Data Reuse: $N \times N$ kernels with $N > 1$ provide an opportunity for IFM data reuse. For example, let us assume that it performs $3 \times 3$ convolution operations with the depth of 3 (i.e., $3 \times 3 \times 3$ filter). For the next adjacent OFM element, instead of searching 27 ($3 \times 3 \times 3$) elements and reloading all NZEs to the *non-zero element buffer*, it can search and load the data for only nine elements while the rest 18 elements are reused in the *non-zero element buffer*. This is because $3\times3$ CONV (in general, all $N \times N$ kernels with $N > 1$) is carried out in a sliding-window manner. Note that $3 \times 3$ CONV operation is performed with a sliding-window-based operation [25] while it does not mean that our hardware uses sliding-window logic. Assuming the kernel window slides in a row-major order, the IFM elements overlapped with the second and third columns of the $3 \times 3$ kernel can be reused in CONV operations for the next adjacent OFM element. On the other hand, when performing $1 \times 1$ convolution, it does not perform the input data reuse as $1 \times 1$ convolution has no opportunity for IFM data reuse in the proposed design[7]
- Removal of Complex Operations: Searching and weight matching require several complex operations such as division and modulo to calculate offset or index values. The proposed hardware design replaces those complex operations with simpler operations such as shift operations or LUT so that the latency for searching and matching can be reduced. For removal of the complex operations, it does not actually affect the data (e.g., NZEs) while it only reduces the complexity of the required operations, resulting in performance improvement. According to our evaluations, the removal of complex operations results in 1.7× and 2.4× speedups for $1 \times 1$ and $3 \times 3$ CONV, respectively, as compared to the case without the removal of complex operations, implying that the removal of complex operations affects performance significantly

## 4.4 | FPGA implementation

We implement the proposed hardware accelerator and software (for compression) in Xilinx ZCU106 FPGA-SoC platform [26], which is equipped with a quad-core ARM Cortex-A53 CPU [4], programmable logic (PL) elements, and various hard intellectual properties. The proposed hardware accelerator is implemented in the PL part while the proposed compression algorithm is implemented in software and executed on the CPU onboard Xilinx ZCU106 FPGA-SoC platform.

For hardware implementation, Xilinx HLS and Vivado design suite are used. The implemented hardware accelerator operates at 150-MHz clock frequency. Although we have explained our compression technique based on 32-bit floating-point elements, the proposed hardware accelerator has been implemented in this work with two different versions of precisions, 32-bit floating-point and 16-bit fixed-point. In the resource-constrained mobile or edge platforms, the data quantization is a common technique for efficient on-device CNN inferences. This implementation of the proposed CNN accelerator with 32-bit floating-point and 16-bit fixed-point is expedient as many CNN hardware accelerators have been implemented to process 16-bit fixed-point as well as 32-bit floating-point elements for CNN inferences. Note that the compression technique can be identically employed to both versions of the accelerator. Furthermore, our proposed IFM compression technique can be applied for other quantization levels, such as 5-bit quantization or log quantization. However, in this work, we have explained our proposed approach with two different precisions (32-bit floating-point for the compression technique and accelerator and 16-bit fixed-point for the accelerator). Depending on the CNN applications, a system designer can choose an appropriate version of the accelerator. Due to the reduced resource usages of the 16-bit fixed-point version of the accelerator under the same number of PEs, we increase the number of PEs from 16 to 32 in the 16-bit fixed-point version. Table 2 summarizes the resource utilization of the proposed hardware accelerator's implementation in Xilinx ZCU106. The results in Table 2 correspond to the proposed hardware accelerator implementation that incorporates all the optimization techniques discussed in Section 4.3.2. For both versions of the accelerators, the implementation can execute both $1 \times 1$ CONV and $3 \times 3$ CONV versatilely. The proposed accelerator design focusses on cost efficiency, which means the main goal is to improve performance per resource usage (cost). As seen from Table 2, the absolute amount of the resource usage as well as usage rates of our implementation is very low, meaning that our design and implementation is well suited for resource-constrained systems. When comparing the 16-bit fixed-point version to the 32-bit floating-point version, the former uses less BRAM blocks as it uses 16-bit elements instead of 32-bit elements. However, the 16-bit fixed-point version uses more DSPs, FFs, and LUTs because of the increased number of PEs as compared to the 32-bit floating-point version.

For both the 32-bit floating-point and 16-bit fixed-point versions, the resource utilization of 18-kb BRAM (i.e., on-chip memory) blocks is still higher than that of the other components such as LUTs and FFs. Our implementation uses BRAM blocks to store the compressed IFMs, weights, and OFMs. To exploit the parallelism between the output channels with multiple PEs, we need to maintain as many IFMs, weights,

[7]For $1 \times 1$ CONV, since there is no overlapped IFM element when sliding the kernel window, it cannot reuse the input data.

**TABLE 2** Field-programmable gate array resource utilization of the proposed hardware accelerator implementations

| | 32-bit floating-point | 16-bit fixed-point | Available resources |
|---|---|---|---|
| BRAM 18Kb | 554 (88%) | 458 (73%) | 624 |
| DSP48E | 136 (8%) | 540 (31%) | 1728 |
| Flip-flop | 64,600 (14%) | 35,458 (7%) | 460,800 |
| Lookup table | 66,481 (29%) | 132,143 (57%) | 230,400 |

and OFMs on-chip as possible, resulting in high resource utilization of the BRAM blocks. While hardware resource utilization of the searching and weight matching part, which is mainly composed of the LUTs and FFs, is hardly proportional to the number of PEs (because it is shared between the PEs), that of the other parts including BRAM blocks is linearly proportional to the number of PEs. For the best performance in our platform, we have implemented our accelerator to have as many PEs (note that the number of PEs is $2^N$ where integer $N \geq 0$) as possible, resulting in high BRAM block utilizations in our platform.

## 5 | EVALUATION RESULTS

Since we have prototyped the proposed sparse CNN accelerator in FPGA-SoC, direct quantitative comparison with many of the previously proposed CNN accelerators would be infeasible as some of the previously proposed CNN accelerators have been implemented in ASIC [11, 15], which hugely differs from an FPGA in terms of the degree of logic optimizations, or some of the accelerators have been evaluated in simulators [12, 13]. Hence, Sections 5.1 and 5.2 compare the proposed technique to CPU-based (i.e., fully software-based with 1.2-GHz clock frequency) CNN inference with Darknet framework [27] (by only comparing CONV layer execution time) with a single-core execution. Though the ZCU106 platform has a quad-core Cortex-A53 CPU, the reason why we assume a single-core CPU execution as a baseline is that the main target for our technique is a resource-constrained system. For example, the resource-constrained edge devices (e.g., IoT devices) often use a single-core CPU due to resource and power constraints (e.g., Raspberry Pi Zero [28] still uses a single-core CPU, ARM1176 series CPU [29]). We have compiled the software code for two different versions: without and with single instruction multiple data (SIMD) vector instruction supports for both baseline and our acceleration technique. Section 5.3 provides quantitative comparison results of our hardware accelerator implementation as compared to the previous FPGA-based hardware accelerators.

For benchmarks, six convolution layers from SqueezeNet [30] have been selected: CONV layers 15, 17, 26, 28, 41, and 43. Table 3 summarizes the configurations of the convolution layers used for the evaluation. When executing the selected six layers, all the data (input, output, and weight) required for processing a single CONV layer can be fit into the accelerator on-chip memory (i.e., BRAM in the FPGA-SoC). Though we run our experiments with the selected six layers, we believe that it does not hurt the generality of the evaluation. For the layers in which all the data cannot be fit into the accelerator on-chip memory, we can perform the accelerator execution along with the DMA data transfer multiple times. In this case, performance can be easily estimated by adding the results of accelerator execution time and DMA data transfer time obtained from executing the accelerator and DMA transfer multiple times. For IFMs used for the evaluations, we have synthetically generated 100 random-valued IFMs for each degree of sparsity (50%, 60%, 70%, 80%, and 90%). For evaluations of performance, energy, and data transfer, a 32-bit floating-point version of the hardware accelerator is used since the baseline CNN framework [27] only supports 32-bit floating-point precision while not supporting the 16-bit fixed-point precision [31].

### 5.1 | Performance and energy

Figure 6 summarizes the performance comparison across various levels of IFM sparsity. Note that the performance is normalized to the case of CPU-based CNN inference without the IFM compression. For a fair comparison, the DMA data transfer time (both read and write) and compression time[8] are also included in the total execution time in the case of our design. In the cases of executing 1 × 1 CONV and 3 × 3 CONV, the proposed technique leads to better performance by 3.4×–5.4× and 11.3×–22.6×, respectively, as compared to the CPU-based execution without SIMD supports. Even when comparing with the CPU-based execution with SIMD supports, the proposed technique still leads to better performance by 1.1×–1.7× and 3.5×–7.1× in the cases of 1 × 1 CONV and 3 × 3 CONV, respectively. As demonstrated in the results, the proposed acceleration technique shows better performance (compared to the CPU-based execution) in 3 × 3 CONV rather than 1 × 1 CONV. This is because the proposed hardware design is versatile for both 1 × 1 CONV and 3 × 3 CONV. Since 3 × 3 CONV has more complex operations compared to 1 × 1 CONV, the proposed hardware design is mainly optimized for 3 × 3 CONV rather than 1 × 1 CONV. Moreover, the hardware logic for data reuse is only for 3 × 3 CONV, which is merely regarded as hardware overhead in the case of 1 × 1 CONV.

[8]The OFMs of the previous layer are IFMs of the current layer in CNNs, thus inherently incorporating the recompression latency for the OFMs of the previous layer in our latency evaluation. According to our evaluation, the compression latency on the software side occupies only a small portion of the entire latency (3.5%–9.5% of the entire latency).

**TABLE 3** The configurations of convolutional neural network layers for evaluations

| | Layer 15 | Layer 17 | Layer 26 | Layer 28 | Layer 41 | Layer 43 |
|---|---|---|---|---|---|---|
| Input feature map size | 29 × 29 × 32 | 29 × 29 × 32 | 15 × 15 × 48 | 15 × 15 × 48 | 15 × 15 × 64 | 15 × 15 × 64 |
| Output feature map size | 29 × 29 × 128 | 29 × 29 × 128 | 15 × 15 × 192 | 15 × 15 × 192 | 15 × 15 × 256 | 15 × 15 × 256 |
| Filter size | 1 × 1 × 32 | 3 × 3 × 32 | 1 × 1 × 48 | 3 × 3 × 48 | 1 × 1 × 64 | 3 × 3 × 64 |
| # of filters | 128 | 128 | 192 | 192 | 256 | 256 |
| Stride | 1 | 1 | 1 | 1 | 1 | 1 |
| Pad | 0 | 1 | 0 | 1 | 0 | 1 |

*Note*: The layers' configurations are based on SqueezeNet implementation in Darknet Framework [27].

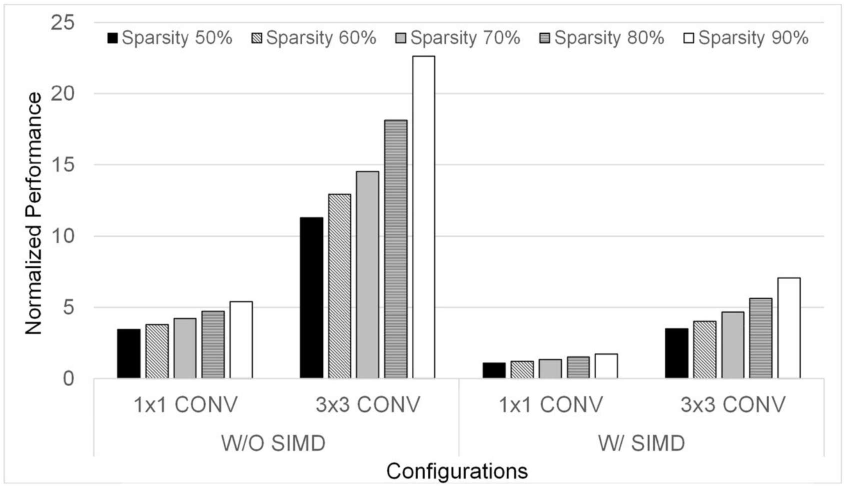

**FIGURE 6** Performance comparison of the proposed acceleration technique normalized to the CPU-based execution w/o and w/SIMD supports

The proposed acceleration technique also reduces energy consumption due to a significant reduction of execution time for CNN inferences. For energy calculation, we utilize estimated average power results (shown in Table 4) from the Vivado tool. The static power is always consumed as long as the system is turned on while the dynamic power is only consumed when the component is running. Thus, in order to obtain power consumption of the baseline (i.e., only CPU execution), we add the dynamic and static power of the processor system (PS; CPU part) and only static power of the PL (FPGA part). For the energy calculation of the baseline, we multiply average power consumption by the execution time. To obtain the power consumption when employing our acceleration technique, we break down the execution into two phases: data compression and CONV, which are performed in the CPU and accelerator, respectively. When performing the data compression, the total system power consumption is equal to an addition of dynamic and static power of the PS and only static power of the PL (excluding the dynamic power of the PL). On the other hand, when performing the CONV operations in the accelerator, the total system power consumption is equal to the addition of static power of the PS (excluding the dynamic power of the PS) and dynamic and static power of the PL. To calculate the total energy consumption, for each execution phase, we multiply the execution time by the total system power consumption and aggregate the energy consumptions of the two phases.

**TABLE 4** Average power results obtained from Xilinx Vivado

| | Processor system | | Programmable logic | |
|---|---|---|---|---|
| | Static | Dynamic | Static | Dynamic |
| Power (W) | 0.100 | 2.701 | 0.602 | 1.133 |

Figure 7 shows the energy results normalized to the CPU-based CNN inference across various levels of IFM sparsity. As compared to the CPU-based execution without SIMD supports, the proposed technique reduces system-level energy consumption by 83.4%–89.3% and 95.0%–97.4% in the cases of 1 × 1 CONV and 3 × 3 CONV, respectively. When the SIMD supports in the CPU are available, the proposed technique still reduces energy consumption by 47.7%–66.9% and 84.0%–91.9% in the cases of 1 × 1 CONV and 3 × 3 CONV, respectively. As demonstrated in the energy results, the proposed technique leads to a huge energy reduction, implying the proposed technique is suitable for resource-constrained embedded systems such as tiny IoT devices.

## 5.2 | Data transfer

Figure 8 shows the comparison of compressed IFM size normalized to the non-compressed IFM size. The proposed technique reduces the IFM data size by 34.0%–85.2% across

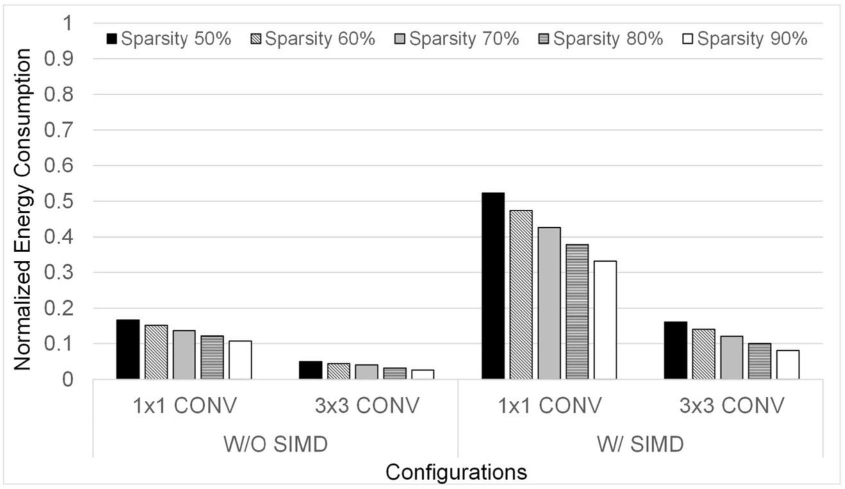

**FIGURE 7** Energy comparison of the proposed acceleration technique normalized to the CPU-based execution w/o and w/SIMD supports

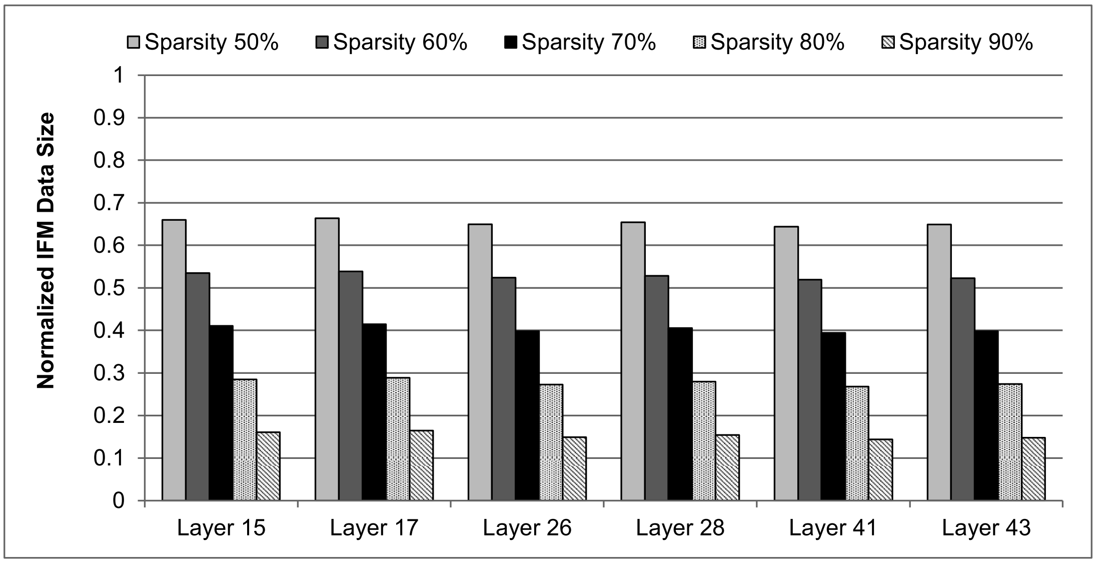

**FIGURE 8** Normalized compressed data size (including metadata) comparison of the proposed acceleration technique across various CONV layers

the degree of the IFM sparsity from 50% to 90%. As illustrated in Figure 8, there are remarkable differences in compressed data size depending on the degree of sparsity in IFM. The reduced IFM data size can also contribute to the reduction of latency, memory bandwidth pressure, and energy. As shown in Figure 9, the proposed technique reduces the DMA transfer latency by 41.6%, on average as compared to the case of transferring non-compressed IFMs. These results imply that one can significantly reduce the data transfer latency caused by the large batch of the inputs depending on the input data sparsity. In the case of the input sparsity of 50%, one can only obtain the data transfer latency reduction by 4.4% with the proposed technique. This is because additional latency for compression is required when applying the proposed technique. Although the proposed compression technique provides only a small data transfer latency benefit for 50% input sparsity, the data transfer latency improvements imparted by the proposed compression technique increase as the degree of IFM sparsity increases. For input sparsity of 90%, results indicate that one can obtain data transfer latency reductions by 75.7%, on average, with the proposed technique.

Though the DMA transfer time may be hidden by double buffering, the reduced data transfer latency can also reduce the required memory bandwidth, is very crucial for low-power resource-constrained edge systems. Moreover, as observed in Ref. [32], since power and energy consumption from the DRAM interface is significant in embedded platforms, the data transfer reduction by the proposed technique can hugely contribute to the energy efficiency of the system. Apart from the data transfer latency, since the proposed technique compresses the IFMs, the decoding latency should also be taken into account when performing CONV layer operations. However, the proposed hardware accelerator performs the decoding of the compressed IFMs in an in-situ manner. Thus, the decoding latency is inherently included in the latency of our

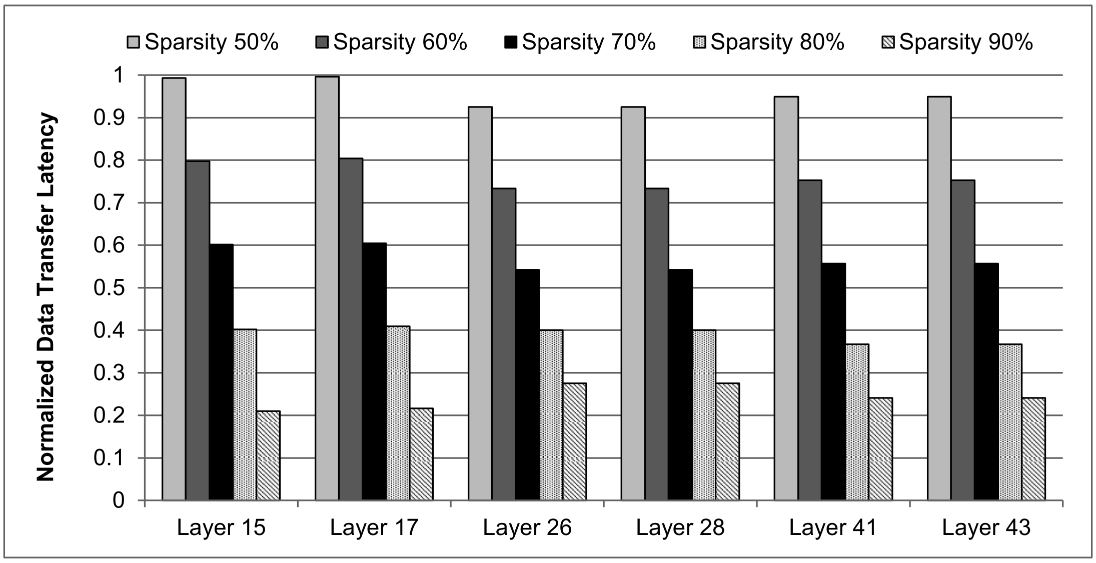

**FIGURE 9** Normalized data transfer latency comparison of the proposed acceleration technique across various CONV layers

proposed hardware accelerator, and therefore, we do not separately report the decoding latency in convolution operations.

## 5.3 | Comparison with state-of-the-art hardware accelerator implementations

We compare the two versions (32-bit floating-point and 16-bit fixed-point precision) of the proposed hardware accelerator implementation with several recent FPGA-based CNN accelerator implementations [5–9]. For quantifying the cost (area) efficiency, the most fair way to perform comparison would be based on the hardware implementations of [5–9] under the same or similar costs. However, it is not possible to implement the hardware designs in Refs. [5–9] to have hardware cost (or resource usages) same or similar to the proposed accelerator (i.e., our own implementations of the accelerators in Refs. [5–9] that have the costs similar to the proposed accelerator may distort the results due to suboptimal or different implementations compared to those in Refs. [5–9]). Thus, we introduce the metric, GOPs per unit cost (GOPs/cost), which quantifies the performance of the accelerator at iso-cost. We clarify that the cost here estimates the area required for implementing the accelerator in an FPGA fabric and BRAM blocks.

The cost is estimated based on resource usage and the number of the transistors required for each hardware resource component. Firstly, we calculate the number of metal–oxide–semiconductor (MOS) transistors required for implementing each type of components. For calculation of the 18-kb BRAM block, we multiply 6 (6T SRAM cell) by the number of cells (18*1024) in the block. For flip-flops (FFs), we assume that the FF is composed of two cascaded latches (SRAMs), requiring 12 transistors for each FF. For LUTs, we assume 5 × 1 LUTs that are commonly used in the modern FPGAs. A single 5 × 1 LUT can be implemented with 32 SRAMs and 31 2 × 1 MUXes [33]. For LUT cost calculations, we only consider the SRAM cost because SRAM cells typically occupy a large area (due to the sizing of the transistor) than 2 × 1 MUXes. Note that excluding the 2 × 1 MUX area is still a conservative assumption as our implementation uses a smaller number of the LUTs (see Table 5). Due to the lack of the detailed design and implementation of DSPs, we omit the cost of the DSPs. For the technology-dependent (TD) cost, we multiply the square of the process technology ($T^2$) [34] (e.g., 16 nm, 28 nm etc.) with the number of required MOS transistors ($N_{Tr}$) for each type of component. After that, by multiplying the number of used blocks or elements in the FPGA with $T^2 \times N_{Tr}$, we can estimate the relative cost (area) of the three different components (i.e., BRAM blocks, FFs, and LUTs) in the FPGA. By aggregating the relative costs of the three types of the components, we can figure out the entire relative cost of a certain accelerator implementation. Finally, we normalize the relative cost of each implementation to that of our 16-bit fixed-point implementation.

As shown in Table 5, although our implementations attain lower absolute GOPs as compared to the implementations in Refs. [5–9], the 16-bit fixed-point version of our hardware implementation achieves 1.9× (on average) better technology-dependent performance per unit cost as compared to that in Refs. [5–9]. Similarly, the 32-bit floating-point version of our hardware implementation also attains better technology-dependent performance per unit cost as compared to that in Refs. [5, 7]. On the other hand, our implementations use much less DSP blocks compared to the other designs. Considering that the cost estimated here excludes the cost of the DSP blocks, the actual (i.e., DSP-included) technology-dependent cost efficiency of our implementations will be much better as compared to Refs. [5–9]. In summary, the proposed hardware accelerator is very suitable for resource-constrained systems where performance under the limited hardware cost is

**TABLE 5** Quantitative comparison of the proposed hardware accelerator with the state-of-the-art designs [5–9]

| Comparison metrics | [5] | [6] | [7] | [8] | [9] | Proposed accelerator (32-bit floating-point) | Proposed accelerator (16-bit fixed-point) |
|---|---|---|---|---|---|---|---|
| Platform | Zynq7100-based | Virtex7-based | Zynq7100-based | VC709 | VC707 | ZCU106 | ZCU106 |
| Process technology | 28 nm | 28 nm | 28 nm | 28 nm | 28 nm | 16 nm | 16 nm |
| Precision | 16-bit fixed | 32-bit float | 32-bit float | 32-bit float | 32-bit float | 32-bit float | 16-bit fixed |
| Clock (MHz) | 60 | 200 | 100 | 200 | 100 | 150 | 150 |
| BRAM blocks (18Kb) | 772 | 1515 | 708 | 1121 | 1024 | 554 | 458 |
| Flip-flop | 107K | 383 K | 187,146 | 292,016 | 205,704 | 64,600 | 35,458 |
| Lookup table | 229K | 252 K | 142,291 | 192,493 | 186,251 | 66,481 | 132,143 |
| DSP | 128 | 1123 | 1926 | 1032 | 2240 | 136 | 540 |
| GOPs | 17.2 | 78.3 | 17.1 | 66.4 | 61.6 | 6.8 | 10.9 |
| Power (W) | 2.3 | 6.0 | 3.3 | 5.3 | 3.6 | 2.4 | 2.7 |
| TD relative estimated cost except DSP | 5.23 | 8.83 | 4.32 | 6.59 | 6.07 | 0.98 | 1.00 |
| TD GOPs/cost | 3.29 | 8.87 | 3.96 | 10.07 | 10.15 | 6.94 | 10.90 |

*Note*: Comparison with Ref. [5] is provided for the FPGA-based accelerator introduced in Ref. [5]. Comparison with Ref. [6] is provided for the 32-bit floating-point accelerator introduced in Ref. [6] because 16-bit fixed-point version accelerator in Ref. [6] has huge resource usage (e.g., LUTs and FFs usage over 460 K and 640 K, respectively, and a 99.6% BRAM usage in Virtex7 FPGA), which would not be suitable for resource-constrained systems. 'TD' means the technology-dependent, considering the process technology impact
Abbreviation: GOPs, giga operations per second.

the most important design metric. Moreover, in terms of power consumption (obtained from the Xilinx power estimator tool [35]), Table 5 indicates that our implementations show lower power consumption than Refs. [6–9] assuming that the accelerators are implemented in the same FPGA device (i.e., the implementation is technology-independent). These power results verify that our proposed acceleration is more suitable for low-power, low-energy, and resource-constrained systems as compared to the contemporary CNN accelerators. Though our proposed accelerator shows a little higher power consumption than that in Ref. [5], the TD GOPs/cost of our accelerator is much better than that in Ref. [5], leading to a better tradeoff between cost efficiency and power consumption.

Unlike the works that only present hardware accelerators, the proposed technique also has additional benefits for end-to-end system performance and energy (i.e., considering not only hardware but also software stack) by using the software-based compression. The end-to-end performance and energy comparison of our work with other works are not presented in Table 5 because the proposed technique combines the software-based compression with the tailored hardware accelerator design, whereas the other works only present hardware accelerators. It also means end-to-end performance comparisons to the other accelerators [5–9] with the same software optimizations would result in misleading because the software optimizations would affect differently to the various accelerators. Meanwhile, as presented in Section 5.2, the proposed technique reduces memory access latency and the data size to be transferred between the accelerator and the off-chip memory, which leads to significant performance and energy efficiency benefits. We leave the quantitative end-to-end performance and energy comparison of the proposed HW/SW co-design-based CNN accelerator against other state-of-the-art designs and techniques as one of our future work.

## 6 | LIMITATION AND DISCUSSION

In this section, we present limitations and discussions on our acceleration technique and evaluation.

### 6.1 | Compression of output feature maps

Our technique can likely have better performance if our accelerator supports in-situ compression of the OFMs, enabling the reuse of the OFMs in the next layer as an input. However, typically in resource-constrained systems, there is a limited size of the accelerator on-chip memory (private local memories). This implies that we need to call the accelerator multiple times in many CNN CONV layers. In this case, even for a single CONV layer execution, the partial OFMs should be transferred to the main memory multiple times due to the limited on-chip memory space in the accelerator. For the next layer processing, the partial OFMs are gathered in the main memory (or CPU cache) and can be compressed with our two-step compression technique, which will be sent to the accelerator on-chip memory as an input. Thus, in this work, our technique does not support the in-situ compression of the OFMs.

## 6.2 | Assumptions on sparsity in evaluations

In our evaluations, we assume the synthetically determined sparsity (50%, 60%, 70%, 80%, and 90%) instead of the empirically observed sparsity from the real-world CNN models. The main reason why we chose synthetically determined sparsity is that the CNN models are diverse (they have different model architecture and different levels of sparsity), meaning that there exist a huge number of real-world CNN models. In addition, the CNN models are continuously evolving, implying that the sparsity of a certain model can also be changed across the model versions.

In Section 5, we have only shown the results with 50%–90% sparsity. In fact, the performance of our acceleration technique is linearly proportional to the input sparsity, implying that performance will be worse with the sparsity less than 50% than that with the sparsity over 50%. When the input sparsity is less than 50% in a certain CNN model, we could use other processing units such as CPUs or GPUs in a system. Furthermore, many research efforts have been devoted to increasing sparsity in both weights and inputs. In addition, lightweight CNN models with high sparsity are preferred in resource-constrained systems. Thus, we believe that our acceleration technique can be highly useful for resource-constrained systems.

## 6.3 | Comparison with the GPU-based acceleration

In our evaluation results, we do not present the quantitative comparison results with GPU-based acceleration while only providing the comparison results with the CPU-based execution. The main reason why we compare our acceleration technique with the CPU is that DNN inferences at the edge devices are commonly performed in the CPUs [36]. According to Ref. [36], mobile GPUs employed in typical Android OS-based devices show similar performance as compared to mobile CPUs. For only 11% of the Android OS-based devices, their GPUs show 3× better performance as compared to their CPU. As shown in our evaluation results, our acceleration technique shows 1.1×—22.6× speedup as compared to the CPU, meaning that our acceleration techniques will still show much better performance in most cases as compared to the GPU. In addition, GPUs typically consume much higher power as compared to the accelerator because of their large memories and additional logic to support the graphics pipeline. Thus, considering that our acceleration technique is geared towards resource-constrained systems, our acceleration technique is more suitable and cost-efficient as compared to GPUs.

## 7 | CONCLUSION

Sparsity in CNNs provides a huge opportunity for optimizing CNN accelerators and reducing data size. In this work, we have proposed a novel HW/SW co-design approach for CNN acceleration that exploits sparsity in IFMs to enhance performance, reduce energy consumption, and curtail data transfer between the accelerator and the off-chip main memory. The proposed lossless compression scheme implemented in software compresses IFM data resulting in data transfer reduction between the memory and the accelerator. The proposed hardware accelerator performs convolution layer operations with the compressed IFM engendering performance improvement and energy reduction. We have implemented our HW/SW co-designed CNN accelerator on an FPGA-SoC platform (Xilinx ZCU106). Evaluation results from our prototype implementation demonstrate that the proposed technique improves the performance by 1.1×–22.6× depending on the degree of the sparsity and filter size as compared to the CPU-based convolution layer execution. In terms of energy, the proposed technique leads to 47.7%–97.4% energy reduction as compared to the CPU-based execution. The proposed technique also reduces the data size and latency for IFM data transfer by 34.0%–85.2% and 4.4%–75.7%, respectively, as compared to the design that does not incorporate data compression. Moreover, the proposed hardware accelerator design attains 1.9× (on average) better cost efficiency with less or comparable power consumption as compared to several state-of-the-art CNN accelerator designs. Results verify the suitability of the proposed technique for resource-constrained artificial intelligence systems such as intelligent embedded and IoT devices that require performance- and energy-efficient CNN inference.

In our future work, we plan to extend the proposed CNN acceleration technique by compressing the CNN weights (along with weight pruning) and then performing the CNN inference with the compressed weights as well as with the compressed IFMs. We also plan to employ our acceleration techniques for different CNN-based object detection models (e.g., YOLOv3 [37], SSD [38], and Faster R-CNN [39]). We further plan to implement the proposed hardware accelerator in an ASIC with further optimizations (e.g., reducing the storage requirements for filter alignment logic) and quantitatively evaluate it. We intend to perform the quantitative end-to-end performance and energy comparison of the proposed HW/SW co-design-based technique against other state-of-the-art designs and techniques.

## ACKNOWLEDGEMENTS

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A3B07045908 and NRF-2021R1I1A3A04037455).

## CONFLICT OF INTEREST

The authors declare no conflict of interest.

## DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

## ORCID

*Joonho Kong* https://orcid.org/0000-0002-9013-9561

## REFERENCES

1. Lee, K., Kong, J., Munir, A.: Hw/sw co-design of cost-efficient cnn inference for cognitive iot. In: Proc. of IEEE International Conference on Intelligent Computing in Data Sciences. ICDS (2020)
2. Sze, V., et al.: Efficient processing of deep neural networks: a tutorial and survey. In: Proceedings of the IEEE (2017)
3. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding. arXiv preprint arXiv:1510.00149 (2015)
4. ARM: Arm cortex-a53 mpcore processor. Techical Reference Manual
5. Aimar, A., et al.: A flexible convolutional neural network accelerator based on sparse representations of feature maps. IEEE Trans. Neural Netw. Learn. Syst. 30(3), 644–656 (2019)
6. Kala, S., et al.: High-performance cnn accelerator on FPGA using unified winograd-gemm architecture. IEEE Trans. Very Large Scale Integrat. Syst. 27(12), 2816–2828 (2019)
7. Liu, B., et al.: An FPGA-based cnn accelerator integrating depthwise separable convolution. MDPI Electron. 8(3), 1–18 (2019)
8. Shen, J., et al.: Towards a multi-array architecture for accelerating large-scale matrix multiplication on FPGAs. In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5 (2018)
9. Zhang, C., et al.: Optimizing FPGA-based accelerator design for deep convolutional neural networks. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 161–170 (2015)
10. Albericio, J., et al.: Cnvlutin: ineffectual-neuron-free deep neural network computing. In: Proceedings of the 43rd International Symposium on Computer Architecture, pp. 1–13 (2016)
11. Zhang, S., et al.: Cambricon-X: an accelerator for sparse neural networks. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–12 (2016)
12. Parashar, A., et al.: SCNN: an accelerator for compressed-sparse convolutional neural networks. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, pp. 27–40 (2017)
13. Kim, D., Ahn, J., Yoo, S.: A novel zero weight/activation-aware hardware architecture of convolutional neural network. In: Proceedings of the Conference on Design, Automation & Test in Europe, pp. 1466–1471 (2017)
14. Gondimalla, A., et al.: Sparten: a sparse tensor accelerator for convolutional neural networks. In: Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 151–165 (2019)
15. Chen, Y., et al.: Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. IEEE J. Solid-State Circ. 52(1), 127–138 (2017)
16. Li, H., et al.: A high performance FPGA-based accelerator for large-scale convolutional neural networks. In: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–9 (2016)
17. Rhu, M., et al.: Compressing dma engine: leveraging activation sparsity for training deep neural networks. In: 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 78–91 (2018)
18. Shen, Y., et al.: Escher: a cnn accelerator with flexible buffering to minimize off-chip transfer. In: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 93–100 (2017)
19. Li, J., et al.: SmartShuttle: optimizing off-chip memory accesses for deep learning accelerators. In: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 343–348 (2018)
20. Zhang, J.J., et al.: Compact: on-chip compression of activations for low power systolic array based cnn acceleration. ACM Trans. Embed. Comput. Syst. 18(5s), 1–24 (2019)
21. Yuan, T., et al.: High performance cnn accelerators based on hardware and algorithm co-optimization. IEEE Trans. Circ. Syst. I Regul. Pap. 68(1), 250–263 (2021)
22. Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)
23. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, pp. 1097–1105 (2012)
24. Han, S., et al.: Learning both weights and connections for efficient neural network. Adv. Neural Inf. Process. Syst. 28, 1135–1143 (2015)
25. Gao, Z., Wang, L., Wu, G.: Lip: local importance-based pooling. In: 2019 IEEE/CVF International Conference on Computer Vision (ICCV), pp. 3354–3363 (2019)
26. Xilinx: Xilinx zynq ultrascale+ mpsoc zcu106 evaluation kit
27. Redmon, J.: Darknet: open source neural networks in c (2013-2016). http://pjreddie.com/darknet/
28. Raspberry Pi Foundation: Raspberry Pi zero. https://www.raspberrypi.org/products/raspberry-pi-zero/. Accessed 28 January 2021
29. ARM: ARM1176JZF-S. Technical Reference Manual. https://developer.arm.com/documentation/ddi0301/h/. Accessed 28 January 2021
30. Iandola, F.N., et al.: Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. arXiv preprint arXiv:1602.07360 (2016)
31. Zhong, G., et al.: Synergy: an hw/sw framework for high throughput cnns on embedded heterogeneous soc. ACM Trans. Embed. Comput. Syst. 18(2) (2019)
32. Lee, K., et al.: Memory streaming acceleration for embedded systems with cpu-accelerator cooperative data processing. Microprocess. Microsyst. 71, 102897 (2019)
33. Altera: FPGA architecture. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf. Accessed 31 January 2021
34. Rabaey, J.M., Chandrakasan, A., Nikolic, B.: Digital Integrated Circuits, 2nd ed. Prentice Hall Press (2003)
35. Xilinx: Xilinx power estimator. https://www.xilinx.com/products/technology/power/xpe.html. Accessed 31 January 2021
36. Wu, C.-J., et al.: Machine learning at facebook: understanding inference at the edge. In: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 331–344 (2019)
37. Redmon, J., Farhadi, A.: Yolov3: an incremental improvement. arXiv preprint arXiv:1804.02767 (2018)
38. Liu, W., et al.: Ssd: single shot multibox detector. In: Lecture Notes in Computer Science, pp. 21–37 (2016)
39. Ren, S., et al.: Faster R-CNN: towards real-time object detection with region proposal networks. IEEE Trans. Pattern Anal. Mach. 39(6), 1137–1149 (2017). https://doi.org/10.1109/tpami.2016.2577031

**How to cite this article:** Kwon, J., Kong, J., Munir, A.: Sparse convolutional neural network acceleration with lossless input feature map compression for resource-constrained systems. IET Comput. Digit. Tech. 16(1), 29–43 (2022). https://doi.org/10.1049/cdt2.12038