A Novel Zero Weight/Activation-Aware Hardware Architecture of Convolutional Neural Network

Dongyoung Kim
Department of CSE
Seoul National University

Junwhan Ahn
Department of EE
Seoul National University

Sungjoo Yoo
Department of CSE
Seoul National University

***Abstract*—It is imperative to accelerate convolutional neural networks (CNNs) due to their ever-widening application areas from server, mobile to IoT devices. Based on the fact that CNNs can be characterized by a significant amount of zero values in both kernel weights and activations, we propose a novel hardware accelerator for CNNs exploiting zero weights and activations. We also report a zero-induced load imbalance problem, which exists in zero-aware parallel CNN hardware architectures, and present a zero-aware kernel allocation as a solution. According to our experiments with a cycle-accurate simulation model, RTL, and layout design of the proposed architecture running two real deep CNNs, pruned AlexNet [1] and VGG-16 [2], our architecture offers 4x/1.8x (AlexNet) and 5.2x/2.1x (VGG-16) speedup compared with state-of-the-art zero-agnostic/zero-activation-aware architectures.**

Keywords—Convolutional neural network, accelerator, zero value, kernel, activation.

I. Introduction

Deep neural networks are ubiquitous in computer vision [1-7], natural language processing [8], and speech recognition [9]. Specifically, convolutional neural networks (CNNs) show an unprecedented level of accuracy for various vision tasks from image classification and segmentation to object detection. The superior accuracy of CNNs is mainly achieved by *deep* convolutional layers. However, as the convolutional layers of CNNs become deeper to achieve higher accuracy (e.g., GoogLeNet [3], ResNet [7], etc.), the portion of convolutions in total execution time has continuously increased, thereby dominating the total execution time of a CNN [10]. Thus, in order to apply CNNs to high performance, real-time or embedded systems, it is imperative to optimize the performance and energy consumption of convolution operations.

In a CNN, each convolutional layer takes three-dimensional data as input (called *input activation*), performs convolution requiring a large number of multiplications between kernel weights and input activations, and applies a non-linear activation function (e.g., rectified linear unit (ReLU) [1]) to the sum of multiplication results, resulting in three-dimensional output data (called *output feature maps* or *channels*). Typically, producing one value in the three-dimensional output requires ~10^3 multiplications and additions. Thus, reducing the amount of multiplications and additions is the key to realize fast convolution.

As a solution to high computation overheads of multiplication and addition, recent researches observed that a significant portion of kernel weights and input activations in a convolutional layer are zero [11, 12] and they can be leveraged to reduce the amount of computation. Table I shows the ratio of zero weights and activations in a representative CNN, AlexNet [1].[1] The abundance of zero weights and input activations is due to the following two reasons. First, pruning techniques [11] increase the portion of zero weights without losing the quality of CNN result. Second,

TABLE I
ZERO WEIGHT/ACTIVATION RATIO OF ALEXNET [1]

Layer	Zero Weight [%]	Zero Activation [%]
conv1	15.7	0
conv2	62.1	50.9
conv3	65.4	76.3
conv4	62.8	61.8
conv5	63.1	59.0

input activation is usually produced by the ReLU function, which returns zero for any negative input value [1].

In this paper, we propose a novel hardware architecture for accelerating convolution operations. It aims at exploiting zero values in both kernel weights and input activations in order to reduce the runtime and energy consumption of convolution. Compared with existing hardware accelerators, which are unaware of zero values [14-16] or utilize only one type of zero values (e.g., zero activation [12, 13]), the proposed architecture can provide higher performance and lower energy consumption. This paper makes the following contributions:

- The proposed architecture is the first hardware accelerator that skips ineffectual computation caused by *both zero weights and activations* to improve the performance and energy consumption of convolutional layers of CNNs.
- We identify *zero-induced load imbalance*, a new problem that prevents us from achieving the full parallelism of convolution in a zero-aware CNN hardware architecture consisting of parallel processing elements. In order to mitigate this problem, we propose a method called *zero-aware kernel allocation*.
- We evaluate our proposed architecture with real deep CNNs, AlexNet [1] and VGG-16 [2], based on the synthesized chip layout of the hardware accelerator as well as our in-house cycle-accurate architecture model.

This paper is organized as follows. Section II reviews related work. Section III explains our basic idea. Section IV describes our proposed architecture. Section V presents the zero-aware kernel allocation method. Section VI reports experimental results and analysis. Section VII concludes the paper.

II. Related Work

Han et al. [11] report that the majority of kernel weights (by up to 66.5% and 66.8% in AlexNet and VGG-16, respectively) of convolutional layers can be pruned. CNNs can be accelerated by skipping ineffectual computations associated with either zero weights or zero activations. In terms of exploiting zero values, previous approaches are classified into zero-agnostic ones [14-16] or partially zero-aware ones [12, 13, 17].

DianNao [14], DaDianNao [15] and ShiDianNao [16] accelerate CNNs utilizing regular data access patterns and computation structures of CNNs. These accelerators focus on accelerating dense models. Thus, they exploit wide SIMD-like architectures

[1] We obtained the ratio of zero activations by applying 100 randomly selected images from ImageNet validation set to the pruned AlexNet.

to achieve high parallelism. However, such a synchronously parallel execution prevents us from utilizing zero data for performance improvement.

Eyeriss [13] is a CNN accelerator exploiting zero activations to save power. It clock-gates computation path and local buffer when zero activation is detected. However, in order to accumulate partial sums from neighbor processing elements (PEs), it performs convolutions in a synchronous manner, and thus, cannot exploit zero values for performance improvement.

Cnvlutin [12] decouples the parallel lanes of DaDianNao [15] into finer-grain groups to utilize zero activation for skipping computations. Thus, it gives runtime reduction proportional to the amount of zero activations. However, as will be explained in Section III, it cannot exploit abundant zero weights for improving performance due to the synchronously parallel execution scheme.

EIE [17] is a hardware accelerator for sparse matrix-vector multiplication (for fully connected layers in a CNN) [11]. It skips computation for zero weights thereby improving performance. However, it is limited only to matrix-vector multiplication (e.g., fully connected layers). Thus, it cannot be utilized for speeding up convolution.

III. Basic Idea of Exploiting Zero Values

We explain our basic idea by comparing a state-of-the-art hardware accelerator called Cnvlutin [12] (Fig. 1a) and our zero weight/activation-aware architecture (Fig. 1b). Fig. 1 shows how the two architectures compute two convolutions, $K_0 * A_0$ and $K_1 * A_0$, in parallel (K: kernel weights, A: activations). We assume that each set of kernel weights is associated with an output feature map.[2] As the result of convolutions, the PE produces the (partial) sums of two output feature maps, P_{00} and P_{01}.

A. Architecture Design

Cnvlutin aims at saving computation involving zero activations. As the example in Fig. 1a shows, the PE utilizes the same activation data (A_0) for computing two parallel convolutions with two different sets of kernel weights. Thus, multiplications and accumulations associated with zero activations can be skipped in those two convolutions in a synchronous manner. The effect of exploiting zero activations is illustrated in Fig. 1a, where the solid region in the rectangle of activation represents the amount of non-zero activations. Cnvlutin skips multiplications and accumulations associated with zero activations (labeled by 'skip'), and thus, its execution time is proportional to the amount of non-zero activations.

However, the design of Cnvlutin misses the opportunity to exploit zero *weights* for further reduction in runtime and energy consumption. This is because the activation is shared by the two convolutions, and thus, computation associated with zero weights can be skipped *only if all the kernel weights are zero at the same cycle*. Observing all-zero kernel weights in a cycle is rare in a typical Cnvlutin configuration, which runs multiple (e.g., 16 in [12]) convolutions in a synchronous manner to provide enough inference performance.

Fig. 1b illustrates our proposed architecture. Instead of synchronously executing multiple convolutions (which prevents us from exploiting *both* zero weights and activations), each PE of

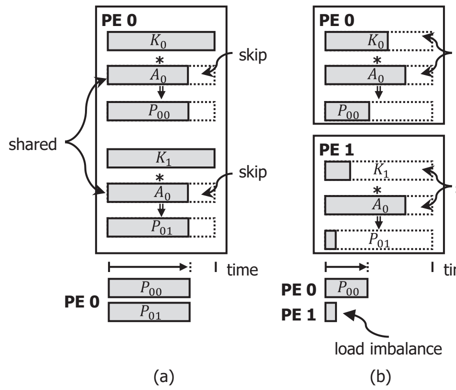

Fig. 1. Simplified operation diagram of (a) Cnvlutin [12] and (b) proposed zero weight/activation-aware architecture.

our architecture performs a single convolution. Providing independent control to each convolution execution allows each PE to individually skip multiplications and accumulations associated with *either zero weights or activations* without being limited by synchronicity. Fig. 1b visualizes the benefit of the proposed approach. As the figure shows, the execution time of our architecture is proportional to the *intersection* of non-zero kernel weights and activations.

B. Zero-Induced Load Imbalance

Exploiting both zero weights and zero activations introduces a new problem to the hardware design, which we call *zero-induced load imbalance*. As shown in Fig. 1b, PE 1 finishes the convolution computation much earlier than PE 0, in which case PE 1 has to wait until PE 0 completes its execution, thereby degrading the efficiency of the proposed architecture. This load imbalance occurs because K_0 has more non-zeros than K_1, and thus, PE 0 has more computation to perform than PE 1 for a single convolution. In Section V, we propose *zero-aware kernel allocation* as a solution to this problem.

IV. Architecture

A. Architecture Overview

Fig. 2 shows the top-level design of our architecture, which consists of on-chip SRAM, a PE array and a ReLU module. A central control module (not shown in the figure for brevity) orchestrates the data transfer among the SRAM, the PE array, and the ReLU module. *Act SRAM* stores activations, output partial sums, and output feature maps (the final result of a convolutional layer), whereas *Weight SRAM* stores kernel weights. The on-chip SRAM and the PE array communicate via a global bus. *ReLU module* computes the ReLU activation function[3] and generates non-zero bit vectors of activations. The overall execution flow of the proposed architecture can be summarized as follows:

1. Given a convolutional layer, we first divide the spatial dimension[4] of activations into *work groups (WGs)*. Each WG is assigned to a set of PEs to produce output feature maps from input activations in the WG (e.g., PEs grouped by a dotted box in Fig. 2 process a WG).

[2] In a convolutional layer, the two-dimensional output called feature map has its own set of kernel weights. Thus, each output feature map is produced by convolution operations with the associated kernel weights and three-dimensional input activations. Note that we use 'channel', 'feature map', and 'activation', interchangeably.

[3] The ReLU module can easily implement other functions, e.g., max pooling, batch normalization, etc., at a small additional cost.

[4] Input activation has three dimensions, width (W), height (H) and channel (C) as shown in Fig. 3. WGs divide the input activation in the spatial dimension, $W \times H$.

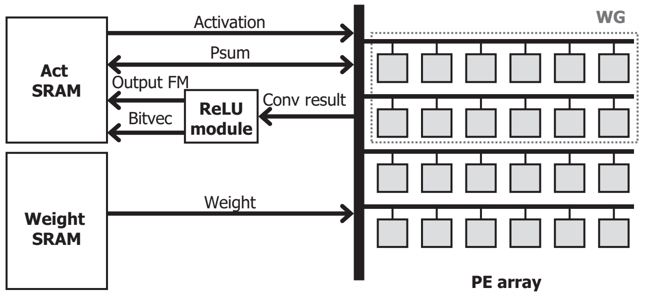

Fig. 2. Top-level architecture of the proposed zero-aware hardware.

2. Activations and weights are broadcast from on-chip SRAM to PEs. All PEs in a WG receive the same activation (e.g., A_0 for PE 0/1 of WG 0 in Fig. 3a), while PEs in different WGs but with the same index receive the same weights (e.g., K_0 for PE 0 of WG 0/1 in Fig. 3a). The exact mechanism will be explained later in this section.
3. Based on the activations and weights assigned to each PE, PEs calculate the partial sums of convolution and accumulate them into the partial sum storage in Act SRAM (arrow annotated with 'Psum' in Fig. 2).
4. After a PE completes the convolution for an output feature map, the convolution result ('Conv result' in Fig. 2) is transferred to the ReLU module to produce the output of the convolutional layer, 'Output FM' (feature map) in Fig. 2, and a non-zero bit vector. Then the final output is stored into Act SRAM.
5. Step 2 to 4 are repeated until all input activations are consumed.
6. The output feature maps are used as the input activations for the next layer in the CNN.

Note that kernel/activation broadcast and local buffer of PE (to be introduced later) are crucial in reducing the traffic between PEs and Weight/Act SRAM. The following describes the details of the above steps based on the example in Fig. 3.

B. *Dataflow and Computation*

As explained previously, a WG is spatial partitioning of activations allocated to a set of PEs. Similarly, distributing the computation in a WG into PEs is done by dividing a WG into multiple subsets (called *sub-WG*) in the vertical (i.e., channel) dimension. Each sub-WG has an associated subset of activations (called *activation tiles*) and weights (called *kernel tiles*). For example, gray boxes in PEs of Fig. 3a represent the current activation and kernel tiles for each PE.

Based on this work distribution model, the proposed architecture computes the entire convolution result of a large convolutional layer by iteratively performing partial convolution with activation/kernel tiles. More specifically,

1. It first computes the partial convolution results for the current sub-WG by sliding over the activation tiles.
2. Then, to start the convolution computation for the next sub-WG, it slides over the next set of kernel tiles and fetches the partial sum from the previous sub-WG.

Kernel Broadcast. At the beginning of a sub-WG, a subset of kernel weights (i.e., *kernel tile*) associated with the current sub-WG is fetched into the local buffer of each PE. Since our architecture determines which kernel tile is stored in the PE based on the PE index, PEs with the same index share the same kernel tile. For example, PE 0 of WG 0/1 stores the same kernel K_0. Therefore, our architecture *broadcasts* kernel tiles to all WGs to reduce the traffic between Weight SRAM and PEs.

In order to exploit zero weights, each kernel tile has its own non-zero bit vector (i.e., a bit vector that stores the information about whether each weight is non-zero or not). The non-zero bit vector of kernel weights is pre-computed at design time and is stored in Weight SRAM together with the associated kernel weights. It is also broadcast to WGs when the kernel weights are broadcast.

Activation Broadcast and Convolution. After the kernel tile broadcast, the first activation tile of the current sub-WG is *broadcast* to PEs in the same WG (① in Fig. 3a) because all PEs in a WG share the same activation (e.g., PEs in WG 0 receive the same activation A_0 in Fig. 3a). In order to exploit zero values in the activation, each activation tile is also paired with a non-zero bit vector, which is computed by the ReLU module when the activation (i.e., the output feature map of the previous layer) is generated by it (shown as 'Bitvec' in Fig. 2).

After both activation and kernel tiles (and their non-zero bit vectors) are received by the PEs, each PE performs convolution and produces the partial sum for its associated output feature map as a result (e.g., a small solid rectangle in $Psum_0$ in Fig. 3a) (②). The result is stored into a hardware structure called *Psum buffer* in the PE. If the Psum buffer is full, the PEs write the partial sums in the buffer to Act SRAM and continue the computation.

After finishing the convolution computation for the current activation tile in the current sub-WG, the next activation tile is broadcast to the PEs. Since a convolution operation slides over the activations, there is an overlap between the next and current activation tiles.[5] Thus, in order to reduce the data traffic between Act SRAM and PEs, our architecture broadcasts only *the difference between the current and the next activation tiles*. This is illustrated in Fig. 3b where only the difference of the activation tile, a_2, is broadcast to the PEs of WG 0.

Moreover, in order to further increase the overlapping between current and next activation tiles, we adopt a *zig-zag order* in visiting the activation tiles in a WG. After the current activation tile is processed, we first move the window horizontally to the right or left and process the next activation tile. Then, when it reaches the horizontal edge of the input activation, we move the window down by a stride[6] and again move horizontally in the opposite direction in a zigzag fashion to choose the next activation tile (① in Fig. 3b).

Processing the Next Sub-WG. After finishing convolution for the current sub-WG, the PEs are configured to process the next sub-WG. For this purpose, they first receive new kernel tiles via broadcast (already explained in 'Kernel Broadcast'). They also receive the partial sums that are associated with the current activation tiles and are from the previous sub-WG. Then, the PEs perform convolution for the new sub-WG by using the same procedure explained above.

Processing the Next Set of Output Feature Maps. After finishing convolution for all sub-WGs in the current WG, the final convolution result is transferred to the ReLU module, whose result (i.e., a set of output feature maps) is stored into Act SRAM.

[5] For instance, 3×3 convolution (stride 1) on two adjacent outputs have 2/3 of input activations in common.

[6] The stride is specified in the convolutional layer. Typically, stride 1 or 2 is utilized.

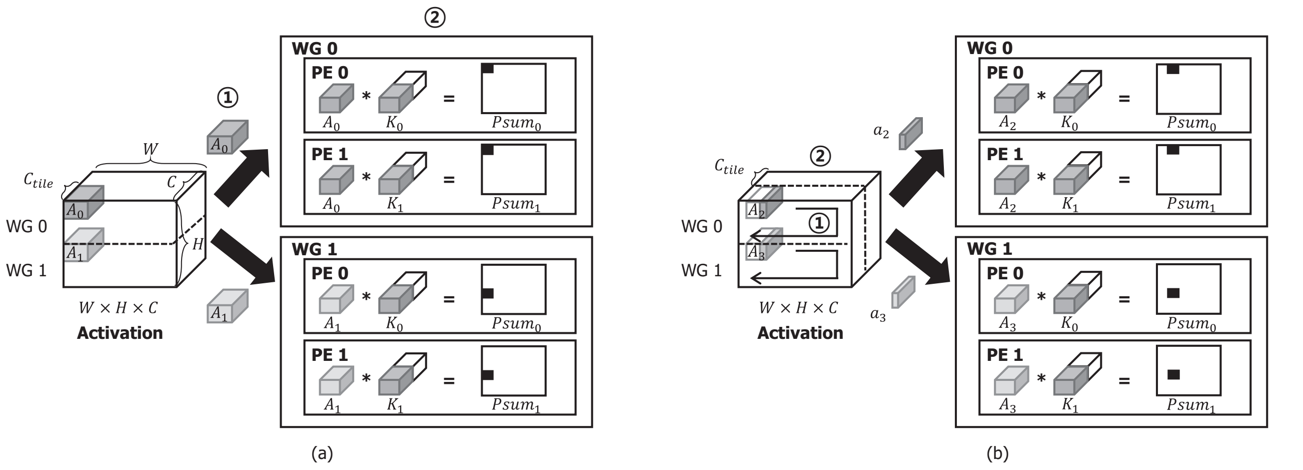

Fig. 3. Broadcast and computation procedures of the proposed architecture. (a) Basic kernel broadcast, activation broadcast, and computation method of the proposed architecture. (b) Activation broadcast and computation order of the proposed architecture.

Then, the PEs start convolution for the next set of output feature maps associated with its WG. This process continues until all output feature maps are processed.

C. Zero-aware PE Architecture

Fig. 4 shows the microarchitecture of a zero-aware PE. Each PE is composed of a fetch controller, data path, and three local buffers (Act buffer, Weight buffer and Psum buffer). The fetch controller receives activations/weights and their associated non-zero bit vectors from Act/Weight SRAM. As illustrated in Fig. 4, it performs a logical AND operation of the two non-zero bit vectors to compute the indices of Act and Weight buffers where both activation and weight are non-zero. Then, it determines which entries to read from the Act and Weight buffers at each cycle ($curr_{addr}$ and $next_{addr}$ in Fig. 4). This allows us to skip the multiplications whose results will be zero (i.e., due to either zero activation or zero weight). In terms of the data path, we implemented two flavors of a zero-aware hardware accelerator by applying different reduced precision schemes: 16-bit fixed point and 5-bit logarithmic quantization (LogQuant) [18].

Fig. 4 shows the data path of a zero-aware PE based on LogQuant. The PE has a 3-stage pipeline (fetch, computation, and write stages) and its data path consists of a shifter and an accumulator. For the current non-zero activation a and weight w (pointed by $curr_{addr}$ in the figure), a is shifted by the amount of $\log_2 w$ since the weight is quantized by a log scale (i.e., $a * w \approx a \ll \log_2 w$). For the 16-bit fixed point implementation (explained in Section VI), the PE has a 4-stage pipeline to meet the clock frequency constraint.

V. Zero-aware Kernel Allocation

Each PE in the WG contains a distinct set of kernel weights, which causes them to have different amounts of zero weights. Thus, in a single WG, some PEs can finish computation later than the others in the zero weight/activation-aware architecture, even though all PEs in the WG perform convolutions with the same activations. We call this type of load imbalance problem *zero-induced load imbalance*. In order to mitigate this problem, we propose allocating kernel weights to PEs in a zero-aware manner. Typically, kernel weights are allocated to PEs based on the kernel index, e.g., kernel 0 is allocated to PE 0 in the WG. Our idea is that, given a convolutional layer, (1) we first sort all

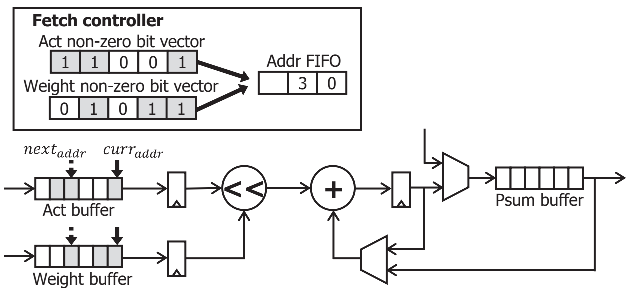

Fig. 4. Microarchitecture of the proposed zero-aware PE.

the sets of kernel weights based on the number of non-zero weights in the sets, and (2) allocate sets of kernel weights to PEs in the sorted order.

Fig. 5 illustrates the problem and our solution in the case of the third convolutional layer (conv3) of pruned AlexNet (see Section VI for more details of pruned AlexNet). For simplicity, we selected, from conv3, a WG covering 384 kernel tiles (size of 3x3x14). We assume that 40 PEs are allocated to the WG. Thus, the WG has 10 ($= \lceil 384/40 \rceil$) sub-WGs. 40 PEs perform convolution on a sub-WG (i.e., for 40 kernel tiles) and proceed to the next sub-WG. This process continues until the convolution of 10th sub-WG (with 24 kernels tiles) is completed.

Fig. 5a shows the ratio of non-zero weights per kernel tile. In the figure, each narrow bar corresponds to a kernel tile. In Fig. 5a, every 40 kernel tiles are allocated to a sub-WG based on the kernel index. As the figure shows, there are significant variations in the ratio of non-zero kernel weights in a single sub-WG. The runtime of a sub-WG is determined by the PE having the longest runtime. Thus, the PE with the largest ratio of non-zero kernel weights (pointed by arrows) determines the runtime of each sub-WG.

Fig. 5b shows the result of the zero-aware kernel allocation. In this case, the kernel tiles are sorted in the ascending order of non-zero weight ratio. Then, the kernel tiles are allocated to sub-WGs in the sorted order. Thus, as the figure shows, each sub-WG has a more uniform distribution of non-zero weight ratio, thereby enabling better load balance among PEs in the same sub-WG. Note that the load imbalance between sub-WGs does

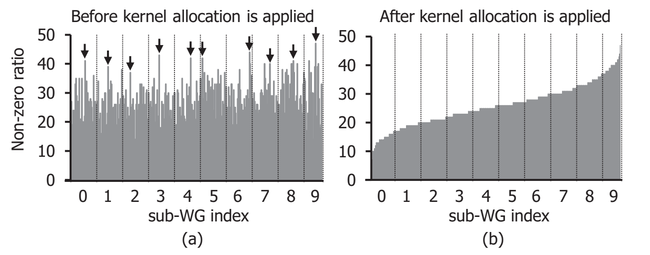

Fig. 5. Non-zero weight ratio of 384 kernel tiles (size of 3×3×14) of conv3 layer in the pruned AlexNet. Dotted vertical lines split sub-WGs. (a) Non-zero weight ratio before zero-aware kernel allocation is applied. (b) Non-zero weight ratio after zero-aware kernel allocation is applied.

not incur performance loss since at any instant only one sub-WG is processed in this example.

In the proposed architecture, the number of PEs allocated to the WG affects (1) zero-aware kernel allocation and (2) the efficiency of broadcast. In our experiments, for each convolutional layer, we determined the number of PEs allocated to the WG by sweeping the number of PEs and finding the one with the largest speedup during design time.

VI. Evaluation

A. Evaluation Methodology

We developed RTL implementations of the baseline (Eyeriss [13]) and two flavors of our architecture, including 16-bit fixed-point based (which we call FixedPoint) and LogQuant [18] based (5-bit activations and 5-bit weights, which we call LogQuant) ones, to measure the area, power, and critical path delay. For fast performance evaluation, we also implemented an in-house cycle-accurate model of our architecture. We used Synopsys Design Compiler under the TSMC 65nm library to synthesize the RTL designs and obtained the chip layout (1.53/1.66/ 1.63mm^2 for the baseline/FixedPoint/LogQuant) using Synopsys Astro. We used PrimeTime PX for power estimation and CACTI v6.0 [19] for modeling SRAM energy/area.

We performed iso-area comparisons (taking both on-chip SRAM and logic area into account) between the baseline (Eyeriss) and our proposed architectures. Table II gives the details of architectural configuration. Our architectures adopt 11×15 and 8×45 PE arrays for FixedPoint and LogQuant, respectively. The bus width of all designs is fixed to 512 bits. Operating clock frequency is 200MHz. Due to control logics and registers for supporting zero-aware computation skipping, FixedPoint and LogQuant have the area overhead of 8.5% and 6.9%, respectively, compared to the baseline.

Comparison between the baseline and our architecture can be complicated when off-chip memory traffic is involved. Thus, in order to focus on comparing the *on-chip* architectures, we selected the size of on-chip SRAM to be large enough to store all the input and output data of a convolutional layer (see Table II). Thus, during the execution of a convolutional layer, there is no access to the off-chip memory. When smaller on-chip SRAM (of the same size for both architectures) is used, both architectures will suffer from the same amount of performance/energy overhead from off-chip memory access.

TABLE II
Architectural Configuration
of Proposed Architecture and Baseline

		Eyeriss [13]	**Proposed (FixedPoint)**	**Proposed (LogQuant [18])**
# PEs		168	165 (11×15)	360 (8×45)
Precision		16-bit fixed	16-bit fixed	5-bit LogQuant
Local buffer (per PE)	Act	12×16b REG	121×16b SRAM	121×5b SRAM
	Weight	225×16b SRAM	121×16b SRAM	121×5b SRAM
	Partial Sum	24 ×16b REG	16×16b REG	16×11b REG
SRAM (AlexNet)	Act	2.03MB	2.06MB	1.22MB
	Weight	1.65MB	1.75MB	634KB
SRAM (VGG-16)	Act	12.25MB	12.63MB	6.13MB
	Weight	28.06MB	29.81MB	8.77MB

TABLE III
The Number of PEs in a WG of Each Layer

		conv1	**conv2**	**conv3**	**conv4**	**conv5**
AlexNet	FixedPoint	33	33	55	11	11
	LogQuant	32	32	40	24	24
VGG-16	FixedPoint	33	33	33	77	77
	LogQuant	48	48	48	88	48

We used the pruned model of AlexNet in [11] and a pruned version of VGG-16, which was obtained by thresholding kernel weights to meet the reported ratio of zero weights in [11].[7] We obtained the logarithm-based quantization of activations and weights for LogQuant by following the procedure described in [18]. To run AlexNet and VGG-16 on the proposed architecture, we allocate PEs to WGs as shown in Table III (also see Section V). For instance, in conv1 in AlexNet, FixedPoint has 5 WGs (=165 PEs / 33 PEs), each of which is allocated 33 PEs.

We use Eyeriss as the zero-agnostic baseline of 16-bit architecture.[8] In order to evaluate the effect of skipping computation associated with zero values, we run our accelerator with four modes: zero-weight-aware mode (WZ), zero-activation-aware mode (AZ), zero-weight- and zero-activation-aware mode (WAZ), and WAZ with the zero-aware kernel allocation method (WAZ+KA). Note that the zero-activation-aware mode (AZ) corresponds to the idea of Cnvlutin [12].

B. Experimental Results

Fig. 6 shows the speedup of the proposed architecture (FixedPoint) over the baseline (16-bit Eyeriss) for the execution of all convolutional layers in AlexNet and VGG-16. Our proposed architecture (WAZ+KA) achieves **4x** and **5.2x** speedup in AlexNet and VGG-16, respectively. The figure also shows the effect of WZ and AZ separately. In both AlexNet and VGG-16, the zero-activation-aware method (AZ) shows higher performance improvement than the zero-weight-aware method (WZ) due to the zero-induced load imbalance problem in WZ. Zero-aware kernel allocation mitigates such a problem and gives a 19.6% additional gain (w.r.t. WAZ) in AlexNet (25.8% in VGG-16). Compared with the zero-activation-aware method (AZ) like Cnvlutin, our architecture (WAZ+KA) gives **1.8x** and **2.1x** speedup in AlexNet and VGG-16, respectively.

Although Eyeriss cannot skip computations associated with zero values (i.e., no performance improvement), it can save energy by clock-gating computation units when activation is zero. Compared with Eyeriss, the proposed architecture (WAZ+KA)

[7] The authors of [11] provided us with the pruned AlexNet. However, their pruned VGG-16 model was not publicly available due to confidentiality.

[8] Eyeriss exploits zero values to reduce off-chip memory traffic by compression and clock-gate computation units for energy reduction.

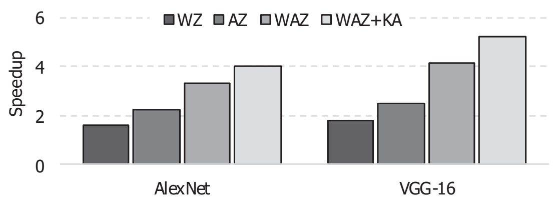

Fig. 6. Speedup of our architecture (FIXEDPOINT) in AlexNet and VGG-16.

achieves higher energy reduction since it exploits both zero weights and zero activations to skip the operation of local buffer and logic. Thus, as Fig. 7 shows, the proposed architecture (WAZ+KA) reduces overall energy by 11.3% (in AlexNet) compared to the baseline. Note that the total energy consumption of WAZ+KA in the figure includes additional SRAM energy consumption (4.1% of SRAM energy) due to non-zero bit vectors.

Fig. 7 shows that the proposed architecture consumes lower SRAM energy than the baseline in VGG-16. There are two reasons. First, our architecture accesses SRAM less frequently (by 5.8%) than the baseline. This is because the proposed architecture allocates more PEs to a WG in VGG-16 (than in AlexNet), especially, for conv4 and conv5 layers, as shown in Table III. This improves the efficiency of broadcast (i.e., the ratio of data reuses), which is proportional to the number of PEs in the WG. Second, VGG-16 requires larger on-chip SRAM than AlexNet (Table II), which makes leakage power consumption of on-chip SRAM more dominant. Since the leakage energy consumption is proportional to the total runtime and the proposed architecture achieves higher speedup in VGG-16 than in AlexNet, it consumes lower SRAM energy in VGG-16.

LOGQUANT (WAZ+KA) achieves **8.3x** / **2.1x** speedup in AlexNet (**9.8x** / **1.9x** speedup in VGG-16) compared to the baseline (Eyeriss) and our FIXEDPOINT (WAZ+KA), respectively. This is because the 5-bit PEs of LOGQUANT replace multipliers with shifters and have narrower bit width, leading to 53.6% smaller area than FIXEDPOINT. As a result, LOGQUANT contains 195 more PEs than the 16-bit one, which enables a higher degree of parallel execution of convolution.

As Fig. 8 shows, LOGQUANT gives a **2.9x** energy reduction over the 16-bit one. This is because (1) smaller on-chip SRAM is used to store narrow input/output data (in 5 bits), (2) lower traffic to on-chip SRAM (in terms of the total amount of accessed bits) due to narrow data, and (3) lower traffic to on-chip SRAM due to higher efficiency of broadcast over PEs, i.e., higher data reuse ratio (= # PEs in a WG).

VII. CONCLUSION

In this paper, we presented a novel hardware accelerator for CNNs, which utilizes both zero weights and zero activations. Existing zero-activation-aware accelerators cannot exploit both types of zero values due to the synchronously parallel execution of convolution. On the other hand, our architecture runs more fine-grained PEs in parallel, but in a less synchronous manner, which enables each PE to individually skip computations associated with zero weight/activation. We also report a load imbalance problem incurred by the different amounts of non-zero kernel weights on zero-aware PEs and propose zero-aware kernel allocation as a solution. Our experiments with two CNNs, AlexNet and VGG-16, show that the proposed architecture achieves **4x** and **5.2x** (**1.8x** and **2.1x**) speedup over the existing zero-agnostic (zero-activation-aware) architecture, respectively.

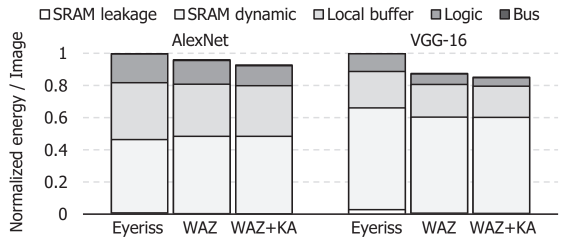

Fig. 7. Energy consumption of the baseline and the proposed architecture.

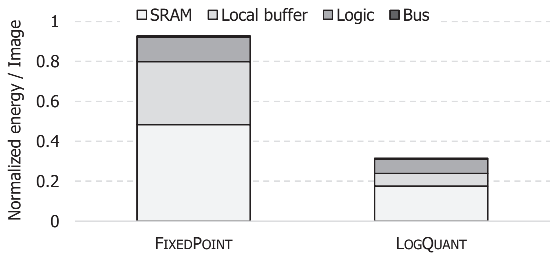

Fig. 8. Energy consumption of FIXEDPOINT and LOGQUANT.

ACKNOWLEDGMENT

This work was supported by Samsung Electronics, SK Telecom, and Research Resettlement Fund for the new faculty of Seoul National University.

REFERENCES

[1] A. Krizhevsky et al., "ImageNet classification with deep convolutional neural networks," in *Proc. NIPS*, 2012.
[2] K. Simonyan et al., "Very deep convolutional networks for large-scale image recognition," arXiv preprint arXiv:1409.1556, 2014.
[3] C. Szegedy et al., "Going deeper with convolutions," in *Proc. CVPR*, 2015.
[4] P. Sermanet et al., "Overfeat: Integrated recognition, localization and detection using convolutional networks," arXiv preprint arXiv:1312.6229, 2013.
[5] R. Girshick et al., "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. CVPR*, 2014.
[6] S. Ren et al., "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. NIPS*, 2015.
[7] K. He et al., "Deep residual learning for image recognition," arXiv preprint arXiv:1512.03385, 2015.
[8] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proc. IEEE*. vol. 86, pp. 2278-2324, 1998.
[9] I. Kipyatkova et al., "Recurrent neural network-based language modeling for an automatic Russian speech recognition system," in *Proc. AINL-ISMW FRUCT*, 2015.
[10] J. Cong et al., "Minimizing computation in convolutional neural networks," in *Proc. ICANN*, 2014.
[11] S. Han et al., "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *Proc. ICLR*, 2016.
[12] J. Albericio et al., "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *Proc. ISCA*, 2016.
[13] Y. H. Chen et al., "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *Proc. ISSCC*, 2016.
[14] T. Chen et al., "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS*, 2014.
[15] Y. Chen et al., "DaDianNao: A machine-learning supercomputer," in *Proc. MICRO*, 2014.
[16] Z. Du et al., "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. ISCA*, 2015.
[17] S. Han et al., "EIE: Efficient inference engine on compressed deep neural network, " in *Proc. ISCA*, 2016.
[18] D. Miyashita et al., "Convolutional neural networks using logarithmic data representation," arXiv preprint arXiv:1603.01025, 2016.
[19] CACTI v6.0, http://www.hpl.hp.com/research/cacti/.