공학박사 학위논문

The Design Method and Application of Convolutional Neural Network Specialized Hardware Accelerator Utilizing Sparsity

컨볼루셔널 뉴럴네트워크의 제로 데이터를 활용한 전용 하드웨어 가속기의 설계 방법론 및 응용

2019 년 8 월

서울대학교 대학원

컴퓨터공학부

김 동 영

Abstract

The Design Method and Application of Convolutional Neural Network Specialized Hardware Accelerator Utilizing Sparsity

Dong Young Kim

Department of Computer Science and Engineering

The Graduate School

Seoul National University

Neural networks show ever-widening usage in various applications. Especially, convolutional neural networks (CNNs) show reliable accuracy on various vision tasks from image classification to segmentation. Even though CNNs offer reliable accuracy, they demand *deep convolutional layers* which have considerable computations. Such deep layers are unsuitable for real-time embedded systems for object recognition and detection (e.g., virtual reality and autonomous car), since they require a large number of computations with restricted hardware resources. Thus, in order to realize real-time recognition systems on embedded systems, it is critical to optimize the convolutional layers.

Typically, each convolutional layer takes three-dimensional data as input, and performs convolution requiring a large number of multiplications and additions (e.g., $\sim 10^3$), resulting in three-dimensional output data. Thus, reducing the amount of multiplications and additions is the key to realize fast convolution as well as real-time recognition systems on embedded devices. As a solution to high computation overheads of convolution, neural network accelerators have been widely studied in recent years.

Based on the fact that CNNs can be characterized by a significant amount of zero values in both kernel weights and activations, we propose Zero-aware Neural Network Accelerator. Unlike previous approaches exploiting sparsity, we aim at utilizing zero values in both activations and weights to reduce runtime and energy consumption of convolution. We also introduce new problem of fine-grained zero-aware parallel hardware architectures for CNN, which we call zero-induced load imbalance problem, and present solutions to mitigate this problem. In order to achieve further energy reduction, we propose several optimization methods for zero-aware architecture such as memory optimization.

We also take notice that the neural network becomes deeper to handle more complicated tasks and to achieve higher accuracy. However, since deeper layers prevent us from implementing real-time embedded applications, future hardware accelerators for neural networks are desirable to support both zero-skipping and very low-precision scheme to meet tight performance requirement while satisfying restricted resource. Applying both zero-skipping and very low-precision approach to current architectures may incur new problems that prevent us from achieving fully optimized hardware accelerators, which however, has not been analyzed quantitatively.

In this dissertation, we propose a novel Zero-aware Neural Network Accelerator for CNNs named *ZeNA*. It aims at exploiting zero values in both kernel weights and input activations thereby reducing the runtime and energy consumption of convolution. The proposed architecture can provide higher performance and lower energy consumption compared with existing hardware accelerators, which are unaware of zero values or utilize only one type of zero values. In addition, we propose various optimization approaches for zero-aware hardware accelerator such as solutions to mitigate zero-induced load imbalance problem and memory energy optimization methods. At the end of the dissertation, we propose quantitative analysis for zero-aware hardware accelerator including impact of data compression on ZeNA and comparison between various ZeNA architectures, which have distinct configurations of bit-width and and zero data ratio.

Keywords: Neural network, neural network optimization, convolutional neural network, low-precision, hardware accelerator architecture

Student Number: 2015-31048

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural networks are ubiquitous in computer vision [1,9–15], natural language processing [16–19], and speech recognition [20,21]. Especially, convolutional neural networks (CNNs) show reliable results on real-time object recognition and detection applications including virtual reality (VR by Oculus [22]), augmented reality (AR by HoloLens [23]), and autonomous car (self driving technology by Tesla [24]). The superior accuracy of CNNs is mainly achieved by deep convolutional layers. However, as the convolutional layers become deeper to achieve higher accuracy (e.g., GoogLeNet [10] and ResNet [14]), the portion of convolutions in total execution time has continuously increased, thereby dominating the total execution time of a CNN [25]. For this reason, CNN based recognition systems perform well on large devices (e.g., GPU server) which have abundant resources but they are unsuitable for mobile devices which are not. Recently, *dedicated hardware accelerator* for neural networks has been widely studied to accelerate [3, 5, 6, 26–29]. Proposed dedicated hardware accelerators utilize characteristics of neural network such as data flow, sparsity and quantization to efficiently execute neural network.

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 $\sim 10^3$ multiplications and additions. Thus, reducing the amount of multiplications and additions is the key to realize fast convolution as well as implementing CNNs on real-time embedded systems. As a solution to high computation overheads of multiplication and addition of the convolution, recent studies mainly utilize *sparsity* and/or *reduced precision* approach to reduce computation complexity of neural networks, thereby achieving better performance and energy efficiency.

Sparsity. Recent studies show that a significant portion of kernel weights and input activations in a convolutional layer are zero [5,30] and they can be leveraged to reduce the amount of computation. Table 1.1 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

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

to the following two reasons. First, pruning techniques [30] increase the portion of zero weights without losing the quality of CNN result. Second, input activation is usually produced by the ReLU function, which returns zero for any negative input value [1].

Han *et al.* [30] 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 [26–28] or partially zero-aware ones [3,5,6, 29].

Chen *et al.* [26], Chen *et al.* [27] and Zidong *et al.* [28] 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 to achieve high parallelism. However, such a synchronously parallel execution prevents us from utilizing zero data for performance improvement.

Chen *et al.* [3] propose a CNN accelerator exploiting zero activations to save power. It applies clock-gating on computation path and local buffer when zero activation is detected. However, in order to accumulate partial sums from neighbor processing elements (PEs), they perform convolutions in a synchronous manner, and thus, cannot exploit zero values for performance improvement.

Albericio *et al.* [5] decouple the parallel lanes of DaDianNao [27] into finer-grain groups to utilize zero activation for skipping computations. Thus, their accelerator gives runtime reduction proportional to the amount of zero activations. However, as will be explained in Chapter 4.1, it cannot exploit abundant zero weights for improving performance due to the synchronously parallel execution scheme.

Han *et. al* [29] propose a hardware accelerator for sparse matrix-vector multiplication (for fully connected layers in a CNN). 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.

As described above, previous neural network specialized hardware accelerators are unable to fully utilize advantages of sparsity. Unlike previous approaches [3,5,6,26–29] we aim at exploiting zero values in both kernel weights and input activations in order to reduce the runtime and energy consumption of convolution.

Reduced precision. By applying reduced precision approach, hardware accelerators can utilize more computation units and buffers within the same area, thereby achieving higher performance and energy efficiency. Binary [31] and ternary [32] weight quantization approaches are proposed for medium-sized neural networks. Moreover, Migacz report that 8-bit quantization is possible without affecting accuracy in deep neural networks [33].

The neural network becomes deeper to handle more complicated tasks and to achieve higher accuracy. Thus, in order to apply emerging deeper neural networks on real-time embedded systems executed with limited hardware resources, future hardware accelerators for neural networks are desirable to support both zero-skipping and very low-precision scheme. Applying both zero-skipping and very low-precision approach to current architectures may incur new problems that prevent us from achieving fully optimized hardware accelerators. However it has not been analyzed quantitatively.

In this dissertation, we focus on optimizing dataflow and computation of convolutions to apply CNNs on high performance and real-time embedded systems. To achieve this goal, we propose a novel Zero-aware Neural Network Accelerator named *ZeNA*. 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 [26–28] or utilize only one type of zero values (e.g., zero activation [3, 5] or zero weight [6, 29]), the proposed architecture can provide higher performance and lower energy consumption. Our contributions are as follows:

- The proposed architecture is the first hardware accelerator that *skips* ineffectual computation caused by *either zero weights or ac-*

tivations 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 methods called *zero-aware kernel allocation* and *dynamic work group allocation.*

- To maximize opportunity of sparsity, we propose several optimization methods for zero-aware architecture including *on-the-fly bit-vector generation* and *memory optimization.*

- We propose quantitative analysis for zero-aware hardware accelerator including *impact of data compression* and *impact of varying bit-width and zero data ratio* on zero-aware hardware architecture.

- We evaluate our proposed architecture with real deep CNNs, AlexNet [1], VGG-16 [9] based on the synthesized chip layout of the hardware accelerator as well as our in-house cycle-accurate architecture model.

This dissertation is organized as follows. Chapter 2 introduces basics of the neural network. Chapter 3 reviews related work. Chapter 4 explains architecture of ZeNA V1 which applies kernel allocation to mitigate load-imbalance problem. Chapter 5 describes architecture of ZeNA V2 which further reduces memory access energy and improves load-imbalance problem using dynamic work group allocation. Chapter 6 describes further analysis for proposed architecture. Chapter 7 concludes the dissertation.

Table 1.1 Zero weight and 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

Chapter 2

Background

2.1 Neural Network

2.1.1 Neuron

Neural Networks is originally inspired by biological neural systems, neuron and synapse. Neuron and synapse are basic computational units of the brain. In biological neural systems, each neuron receives input signals and produces output signals which are transfered to the other neurons. Similarly, in the artificial neural model, the input signal (e.g., input activation, x_0) is multiplied by synaptic strength (e.g., weight, w_0) resulting in a multiplication result (e.g., x_0w_0) which are transferred to other neurons. The basic idea of a computational model for neural systems is that the *weights* are learnable.

In primitive computational model, a neuron adds up every input signal via synapses and transfers the result signal to neighboring neurons if the sum is above a threshold. The number of signals activated from a neuron within the unit time (i.e., firing rate) is modeled as activation which is determined by the activation function. Conventionally, sigmoid function was used as the activation function but recently ReLU has been commonly used for vision tasks.

2.1.2 Linear Classifier

The classifier memorizes training data and classifies test data that will be given in the future. In order to classify a test data set, the linear classifier adopts a score function and a loss function. The first step in designing a linear classifier is to define the score function which maps the input data to score for each class. Assume that image classification task has a training dataset, $x_i \in R^D$, and each of them associates with a target label, y_i. If N samples and K distinct classes exist, the score function is denoted as $f : R^D \to R^K$. The linear classifier predicts which sample is contained in each class.

Commonly, the linear classifier is represented as follows: $f(x_i, W, b) = Wx_i + b$ where image x_i maps to a single column vector of shape $R^{D\times 1}$. The matrix $W \in R^{K\times D}$ (called weight) and vector $b \in R^{K\times 1}$ (called bias) are parameters of the function.

The linear classifier is trained to predict ground truth while performing the training process by minimizing the loss function which represents the difference between ground truth and prediction result. One of the widely used loss function is *softmax* which produces normalized class probabilities as an output. In softmax function, unnormalized log probabilities for each class is represented as follows:

$$L_i = -log(\frac{e^{f_{yi}}}{\sum_j e^{f_j}}) \tag{2.1}$$

where f_j is j-th element of the vector of class scores f and i represents each class.

2.1.3 Back Propagation

Since given training data is fixed, only gradient for the trainable parameters, weight and bias, can be computed by updating them. Assume that a simple multiplication function of two values, $f(x,y) = xy$, exists. Partial derivative for either input is derived as follows:

$$\frac{\partial f}{\partial x} = y, \frac{\partial f}{\partial y} = x \tag{2.2}$$

The derivative on each variable represents sensitivity of the whole expression on its value.

However, since the loss function of a neural network consists of multiple composed functions, it is complicated to derive gradient directly. Exploiting back propagation, the gradient of the complicated function can be computed. We will describe the basic concept of back propagation with a simplified example. Assume there is an expression that involves multiple composed functions such as $f(x,y,z) = (x+y)z$. It can be divided into two sub-expressions: $q = x+y$ and $f = qz$. Derivative of each expression is computed as follows: $\frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q, \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$.

Specifically, in order to update trainable parameters, gradient of f with respect to its inputs (e.g., x, y, z) is necessary. Utilizing the chain rule, gradient of f is derived with derivative of sub expressions as follows: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$.

Back propagation is a local process to calculate gradient of given function which updates trainable parameters. In this process, every gate in a neural network computes both *output value* and *local gradient of its inputs* with respect to its output value. Exploiting the chain rule, the total gradient is computed from and utilized for parameter update.

2.1.4 Basic Neural Network Topology

Neural networks are modeled as an acyclic graph consisting of collections of neurons. Typically, neural networks are organized into distinct layers of neurons without chaotic connections between them. The most common type of layer in a neural network is a fully-connected layer where neurons between two adjacent layers are fully connected. Figure 2.1 and Figure 2.2 illustrate a neural network with stacked fully-connected layers. More specifically, Figure 2.1 illustrates a 2-layer neural network including three inputs, one hidden layer of four neurons and one output layer with two neurons, and Figure 2.2 illustrates a 3-layer neural network including three inputs, two hidden layers of four neurons and one output layer with two neurons. Note that the neurons are connected across layers and each layer is connected via an activation function. Since neurons in the output layer denote target result such as class scores in classification or real-valued number in regression, typically, the output layer does not include an activation function like most layers in the neural network.

As the number of layer in a neural network increases, the capacity of the model rises. Thus, larger neural networks represent more complicated functions. However, as the size of the model grows, it becomes easier to overfit the training data. Overfitting occurs in models with high capacity where the model fits the noise in the training data instead of the original target. In order to mitigate this problem, various regularization methods (e.g., L1/L2 regularization, dropout and input augmentation) can be applied.

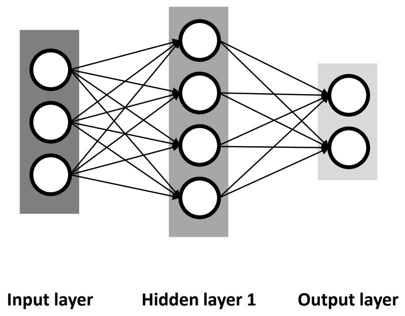


Figure 2.1 A 2-layer neural network including three inputs, one hidden layer of four neurons, and one output layer with two neurons.

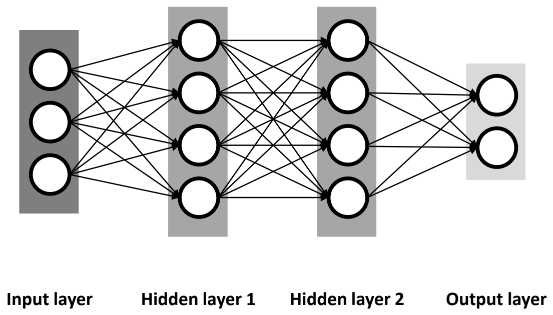


Figure 2.2 A 3-layer neural network including three inputs, two hidden layers of four neurons, and one output layer with two neurons.

2.2 Convolutional Neural Network

2.2.1 Convolutional Neural Network

Convolutional Neural Network (CNN) shows unprecedented accuracy in vision tasks such as imagery and video. Unlike conventional neural network, input data and neurons of each layer are arranged in three dimensions in CNN: *width*, *height* and *channel*. For instance, input size (i.e., $width \times height \times channel$) of representative open image dataset, ImageNet [1], is $128 \times 128 \times 3$. Figure 2.3 illustrates a simplified topology of CNN. The building block of CNN is *convolutional layer*. The convolutional layer receives 3D input data (called input feature map) and generates 3D output data (called output feature map). Instead of connecting neurons between layers in fully-connected manner, neurons in a convolutional layer are connected to a sub-region of the layer ahead.

2.2.2 Layers in Convolutional Neural Network

A CNN includes sequence of several types of layers to generate target result from input data. A basic CNN is constructed by stacking following layers: *convolutional layer*, *nonlinear function layer*, *pooling layer* and *fully-connected layer*. We will describe the basic operation flow of CNN using a simple CNN. Note that the following example assumes one of the general usage of CNN: image classification.

1. **Input**. Input holds the raw pixel values of input image. The width

and height of input data represent spatial dimension of the image, and the channel represents RGB color of the image.

2. **Convolutional layer**. Convolutional layer consists of multiple filters. Each filter is associated with the channel dimension of a output feature map. More precisely, a convolutional layer convolves each filter across the width and height of the input data (i.e., computes dot products between the filter and input data), thereby generating a 2D output feature map. This procedure is repeated on every filter resulting in multiple 2D output feature maps (i.e., 3D output feature map).

3. **Nonlinear function layer**. Nonlinear function layer performs activation function of a neural network. Rectified Linear Unit (ReLU) which performs $f(x) = max(0,x)$ in an elementwise manner is widely used in CNN as a nonlinear function.

4. **Pooling layer**. Pooling layer performs a downsampling operation along the spatial dimensions.

5. **Fully-connected layer**. Fully-connected layer computes scores of each class. Thus, each output neuron of fully-connected layer represents the score of the target class.

The convolutional layer and the fully-connected layer include learnable parameters (e.g., weight and bias) while the nonlinear function and pooling layers do not. We will describe the convolutional layer and the pooling layer in details.

Convolutional layer. The convolutional layer is a basic building block of CNN which performs heavy computations, convolution between 3D input feature map and 4D filters to generate a 3D output feature map. Figure 2.4 illustrates convolutional layer and its computation procedure. A convolutional layer typically consists of multiple 3D filters which includes learnable parameters such as weights and biases. In order to perform regular convolution, at first step, the convolutional layer performs inner product between first block of input feature map (① in Figure 2.4a) and filter, thereby generating a pixel of output feature map (② in Figure 2.4a). Then, the convolutional layer slides the filter over the width (or height) of the input volume to produce the next pixel of output feature map (① in Figure 2.4b), and this process is repeated until the spatial dimension of the output feature map is finished. Note that the same procedure is repeated with the other filters to generate the other 2D output feature maps (i.e., channel dimension out output feature map, ② in Figure 2.4).

Pooling layer. The pooling layer progressively reduces the spatial size of the feature maps, thereby reducing both computations and parameters of CNN. Basically, the pooling layer spatially resizes input feature map using *average* or *max* operation. Assuming that the pooling layer receives input feature map where the size is $W_i \times H_i \times C_i$. In this case, output width W_o, height H_o and channel C_o size after pooling layer are as follows:

- Note that stride is S and pooling size is F.
- $W_o = (W_i - F)/S + 1$
- $H_o = (H_i - F)/S + 1$
- $C_o = C_i$

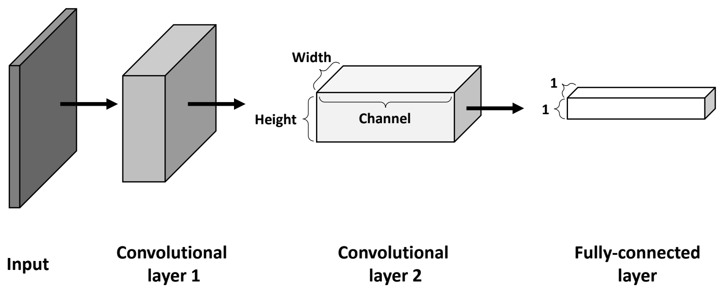


Figure 2.3 Convolutional neural network including two convolutional layers and a fully-connected layer.

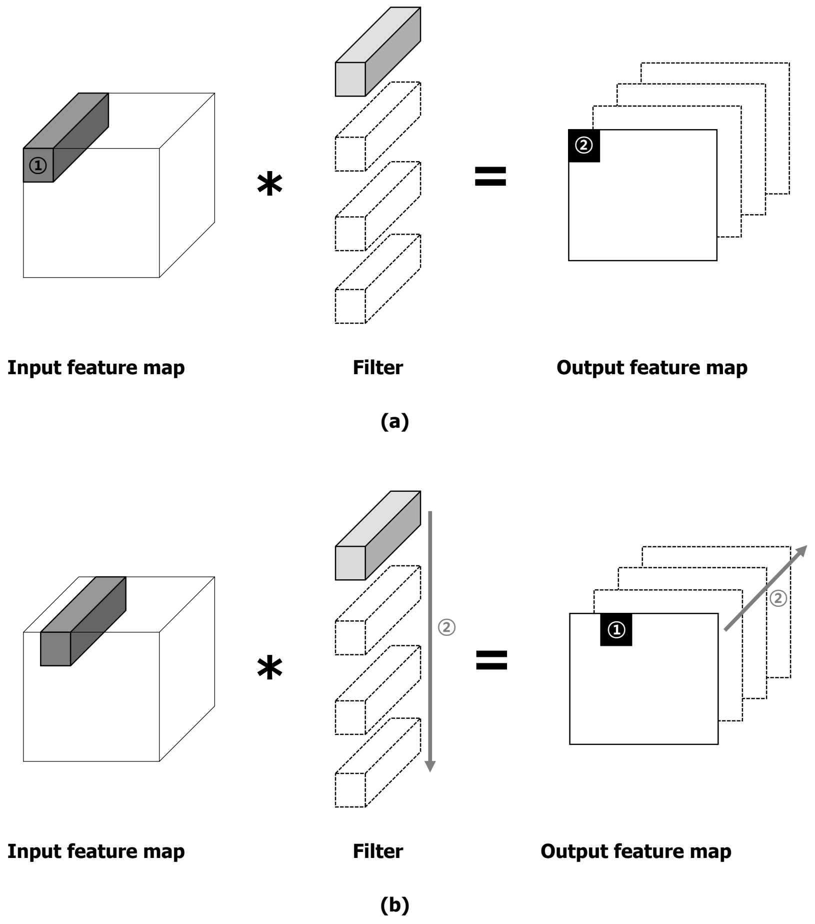


Figure 2.4 Computation procedure of convolutional layer.

Chapter 3

Related Work

Deep neural networks are ubiquitous in various computer tasks due to its superior accuracy compared to the conventional machine learning techniques. Specifically, convolutional neural networks (CNNs) show an unprecedented level of accuracy for various vision tasks from image classification and segmentation to object detection. The exceptional accuracy of CNNs is mainly achieved by deep convolutional layers, which, however, make CNN-based recognition systems require large amounts of computational and memory resources. For this reason, it is challenging to implement CNN-based recognition systems on embedded devices such as cell phones, VR and autonomous car. Two approaches are widely used to accelerate neural networks: *Model optimization* and *dedicated hardware accelerator*.

Model optimization. In algorithm area, model optimization has been widely studied in recent years [7, 31, 34, 34]. Those approaches mainly utilize characteristics of neural network where computations in neural networks are able to be approximated. To optimize a neural network, Han *et al.* [30] utilize *sparsity* where a large number of activations and weights in neural networks can be zero. In addition, various approaches utilizing *quantization* where neural networks can be represented with lower-precision data without losing quality have been proposed [7, 31, 34].

Dedicated hardware accelerator. In hardware area, designing dedicated hardware for neural networks has been widely studied in recent years [3, 5, 6, 26, 27, 29]. They utilize characteristics of neural network such as data flow, sparsity and quantization to design hardware accelerator. Hardware accelerators for dense neural networks which utilize regular data access patterns and computation structures of the neural network have been proposed [26, 27]. Since they focus on accelerating dense models, they exploit wide SIMD-like architectures to achieve high parallelism. However, such a synchronously parallel execution prevents us from utilizing zero data for performance improvement (i.e., zero-agnostic). Exploit abundant zero values in neural network, some accelerators exploit zero activations to save energy and/or enhance performance (i.e., zero-activation-aware) [3, 5], and some others skip memory accesses and computations associated with zero weights thereby accelerate neural networks (i.e., zero-weight-aware) [6, 29].

3.1 Sparsity

3.1.1 Deep Compression

Han *et al.* [30] propose a three stage pipeline named *deep compression* including pruning, quantization and Huffman coding. Deep compression reduces the storage requirement of neural networks while preserving accuracy. Each pipeline stage consists as follows:

Pruning. In order to achieve pruned network, first, they trained network as usual and pruned the small weights below the threshold and retrain the network while preserving sparse connections. Pruned network is stored with compressed sparse row (CSR) or compressed sparse column (CSC) format. According to their experiments, pruned AlexNet and VGG-16 reduce the number of parameters by **9x** and **13x**, respectively.

Quantization. Pruned network is further compressed by network quantization and weight sharing which reduces bit-width of each weight. They reduce the number of weights by sharing the same weights between multiple connections. According to their experiments in pruned AlexNet, they quantize weights to 8-bits (i.e., 256 shared weights) for convolutional layers, and 5-bits (i.e., 32 shared weights) for fully-connected layers without losing accuracy.

Huffman coding. A Huffman code is an optimal prefix code which uses variable-length codewords to encode source data [35]. In Huffman code, commonly emerged stymbols are represented with smaller bits to maximize compression ratio. Applying Huffman code, they reduce extra 20%-30% of network storage.

As a result of deep compression, 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 on the Imagenet dataset, thereby reducing the size of AlexNet and VGG-16 by **35x** and **49x**, respectively.

3.2 Quantization

3.2.1 Logarithmic Quantization

Miyashita *et al.* [7] explore the use of communicating activations, storing weights, and computing the dot-product in the logarithmic domain. Their approach is based on the following two observations: (1) logarithmic encoding allows large dynamic range in fewer bits than fixed-point representation [36], and (2) dot-product which is the key operation of CNNs can be processed with fewer hardware resource after applying logarithmic encoding.

Logarithmic quantization in activation. Transforming activation into log domain replaces multiplication into shift operation as shown in Equation 3.1 where the weight is w and log-domain activation is $\widetilde{x_i} = Quantize(log_2(x_i))$. $Quantize(\cdot)$ quantizes $\cdot$ to an integer and $Bitshift(a,b)$ is the function which shifts a value a by an integer b in fixed-point arithmetic.

$$w^T x \simeq \sum_{i=1}^{n} w_i \times 2^{\widetilde{x_i}} = \sum_{i=1}^{n} Bitshift(w_i, \widetilde{x_i}) \tag{3.1}$$

By applying logarithmic quantization in activation, memory footprint of activation decreases and simple shift operation replaces multiplication which requires expensive digital multipliers.

Logarithmic quantization both in activation and weight. The first method (i.e., logarithmic quantization in activation) can be extended to compute dot products in the log-domain for both activation and weight. After applying logarithmic quantization to both activation and weight, the dot-product is transformed into Equation 3.2 where the log-domain weight is $\widetilde{w}_i = Quantize(log_2(w_i))$ and log-domain activation is $\widetilde{x}_i = Quantize(log_2(x_i))$.

$$w^T x \simeq \sum_{i=1}^{n} 2^{Quantize(log_2(w_i))+Quantize(log_2(x_i))} = \sum_{i=1}^{n} Bitshift(1, \widetilde{w}_i + \widetilde{x}_i) \tag{3.2}$$

On the evaluation with AlexNet and VGG-16, logarithmic quantization with 5-bits shows 1.7% accuracy drop in AlexNet and 0.5% drop in VGG-16 which show higher test accuracy compared to linear 5-bit quantization.

3.2.2 XNOR-Net

Rastegari *et al.* [31] propose Binary weight networks and XNOR networks where weight and/or activation of convolutional layers are binary. They aim at finding the best approximations of the convolutions utilizing binary operations while preserving compatible accuracy. Their binary network requires smaller memory and achieves speedup by replacing costly floating point operations into binary operations.

Binary weight network. Every weight value in Binary weight network is approximated with binary values. In binary approximation, convolution can be approximated by Equation 3.3 where $\bigoplus$ indicates a convolution without multiplication, W is full precision weights, $B \in \{+1, -1\}^{c \times w \times h}$ is binary weights and α is a scaling factor.

$$I * W \approx (I \bigoplus B)\alpha \tag{3.3}$$

XNOR network. In XNOR-Net, both weight and activation values are approximated with binary values. Fully binarization for both weights and activations allows replacing conventional convolution operations into *XNOR* and *bitcount* operations. A convolutional operation can be approximated by Equation 3.4 where K contains the scaling factor for all subtensors in the activation.

$$I * W \approx (sign(I) \circledast sign(W)) \odot K\alpha \tag{3.4}$$

According to their evaluation with various CNNs (e.g., AlexNet, GoogleNet and etc.), Binary weight network requires $\sim$ 32x smaller memory than an equivalent network with single-precision weights, and it achieves $\sim$ 2x speedup by performing convolution only with addition and subtraction. XNOR-Net achieves $\sim$ 58x speedup by performing convolution with XNOR and bitcount operations.

3.2.3 DoReFaNet

Zhou *et al.* [34] propose DoReFa-Net, quantization method for training CNNs which have low-precision weights, activations and gradients. They generalize the method for neural network quantization, and exploit the advantage of quantization both in inference and training. Since operations between low-precision values can be implemented on hardware efficiently, their approach enables to accelerate low-precision neural network inference and training on hardware.

Activation quantization. In order to generate k-bit quantized activation $a_o \in [0,1]$, they performed linear quantization for input activation $a_i \in [0,1]$ as shown in Equation 3.5.

$$a_o = \frac{1}{2^k - 1} round((2^k - 1)a_i) \tag{3.5}$$

Weight quantization. They quantized weights as shown in Equation 3.6 where $quantize_k$ is linear quantization shown in Equation 3.5, w_o is k-bit weights and w_i is fullprecision weights. Since both quantized activations a_o and qauntized weights w_o are k-bit fixed-point integer, convolution between those two values can be efficiently caculated by fixedpoint multiplication and add operations.

$$w_o = 2qunatize_k(\frac{tanh(r_i)}{2max(|tanh(w_i)|)} + \frac{1}{2}) - 1 \tag{3.6}$$

According to their experiments on SVHN and ImageNet datasets, DoReFa-Net achieves comparable accuracy as fullprecision counterparts. AlexNet derived from DoReFa-Net which has 1-bit weights, 4-bit activation, and 32-bit gradient shows 2.9% accuracy drop compared to full-precision network on ImageNet dataset.

3.3 Hardware Accelerator (Zero-agnostic)

3.3.1 DianNao

Chen *et al.* [26] focus on neural network accelerator as well as its control which minimize memory transfers. They propose synthesized design for their accelerator. Their accelerator consists of three main components: storages including an input and output buffers, a neural functional unit which performs computation, and the control logic.

Neural Functional Unit (NFU). Computations of each layer in neural network can be divided into three stages: multiplications between activations and weights, additions of all multiplications, and nonlinear function. To utilize this characteristic, Chen *et al.* designed a 3-stage pipeline for neural network computation where each pipeline stage is executed gradually while generating final output.

Storages. In order to reuse input and output data, unlike general-purpose processor, they adopt scratchpad memory and split them into three parts: an input buffer (NBin), an output buffer (NBout) and a synapse buffer (SB). Each of them has DMA to utilize spatial locality thereby optimizing memory access. All activations and weights are split into chuncks and stored in NBin and SB, respectively. Utilizing NBin and SB as circular buffer, split inputs are reused while executing neural network.

In addition, a dedicated output buffer (i.e., NBout) stores partial sums and reuses them for accumulation, thereby eliminating inefficiency.

3.3.2 DaDianNao

In order to overcome limitations of previous neural network specialized hardware accelerators where only small neural networks can be executed [26, 37, 38], Chen *et al.* [27] propose neural network accelerator consisting of multiple nodes. Each node consists of computational logic, distributed memory and router, and different nodes are tightly interconnected by dedicated mesh.

Distributed memory. They exclude main memory but adopt eDRAM based distributed memory which has sufficient capacity to store weights. To store whole neural networks, multiple nodes share distributed memory and attain adequate capacity to store whole neural network. Providing sufficient eDRAM capacity where combined eDRAM of all nodes holds whole neural network, their architecture saves off-chip DRAM accesses which prevents us from implementing fast and energy efficient neural network accelerator. They store activations close to associated weights thereby minimizing data movement. In addition, since moving weights requires relatively smaller off-chip (i.e., across nodes) bandwidth, they transfer weights rather than activations.

Neural Functional Unit (NFU). NFU consists of four blocks and each block is executed gradually. An adder block contains of 256 parallel adders, a multiplier block consists of 256 parallel multipliers, a max block can perform 16 parallel max operations in parallel, and a transfer block includes two independent sub-blocks performing 16 piecewise linear operations. Its pipelines can be reconfigured for each layer and purpose (e.g., inference or training).

DianNao and DaDianNao [26, 27] focus on accelerating dense models. They exploit wide SIMD-like NFU which performs multiple multiplications, additions and nonlinear operations gradually to generate an output feature map. However, such a synchronously parallel execution prevents us from achieving further improvement utilizing zero data (i.e., zero-agnostic)

3.4 Hardware Accelerator (Zero-activation-aware)

3.4.1 Eyeriss

Chen *et al.* [3] propose a neural network accelerator which adopts spatial architecture using an array of 168 processing elements (PEs). Since memory consumes a large portion of energy while running neural network, they adopt hierarchical memory architecture to maximize low-cost data movement while reducing high-cost data movement. In order to maximize advantages from hierarchical memory architecture, they also adopt a new dataflow for CNN, named row stationary (RS), which reconfigures the spatial architecture to map the computation of a given layer efficiently.

Processing Element (PE) and data gating. Each PE consists of three local buffers and the datapath. Each local buffer stores an input feature map, weights and partial sums, respectively, and they are used at the I/O of each PE. Input feature maps, weights, and partial sums are reused internally thereby reducing high-cost external data movement. In order to exploit zeros in the input feature map, data gating logic is implemented in the data path. For instance, if a zero input feature map value is detected, the gating logic disables to read both the input feature map and the associated weight thereby preventing the datapath from switching. It saves energy while performing computations.

Row Stationary (RS) dataflow. In addition, Chen *et al.* propose row stationary data flow to minimize high-cost data access (e.g., DRAM access) by maximally reusing data from low-cost data transfer such as data movement between computation logic and scratch pad memory within PE, and inter PE communication. In order to take advantage of the reusability of neural network computation, the main controller utilizes multicasts which broadcasts data from high-cost memory to PEs, and point-to-point data delivery which occurs between multiple PEs:

Multicast

- Rows of weights are reused across PEs horizontally.
- Rows of input feature maps are reused across PEs diagonally.

Point-to-point data delivery

- Rows of partial sums are accumulated cross PEs vertically.

3.4.2 Cnvlutin

Albericio *et al.* [5] propose hardware accelerator for neural network which eliminates ineffectual computations associated with zero activations thereby improving both performance and energy consumption. They utilize hierarchical parallel units where groups of lanes utilize shared activations and skip ineffectual computations. Unlike DaDianNao [27] which utilize wide SIMD lanes to take advantage of the regular access pattern of neural network computation, Albericio *et al.* [5] decouple these lanes into finer-grain groups thereby enabling skipping over the zero activations.

Architecture for zero skipping. DaDianNao units [27] consists of two parts: (1) front-end SIMD lanes including the activation buffer, weight buffer and multipliers, and (2) back-end adder trees and partial sum buffer. Albericio *et al.* [5] divide front-end SIMD lanes into 16 subunits, and then, they further divide each subunits into 16 datapath which generate multiplication result for their own output feature map. In other words, a SIMD lane which includes 16 subunits generates 16 partial sums at once, and each subunit generates 16 multiplication results for each partial sum utilizing the same activation but different weights. To accelerate neural network computation, they skip ineffectual multiplications associated with zero activations within each subunit.

Eyeriss [3] utilizes zero activations to reduce energy consumption and Cnvlutin [5] improves both runtime and energy efficiency by exploiting zero activations. However, they only take advantage of zero activations (i.e., zero-activation-aware) and do not exploit abundant zero weights for improving performance due to the synchronously parallel execution scheme.

3.5 Hardware Accelerator (Zero-weight-aware)

3.5.1 Cambricon-X

Zhang *et al.* [6] propose a neural network accelerator which can efficiently cope with sparse network named *Cambricon-X*. It consists of multiple Processing Elements (PEs) coupled with a Buffer Controller (BC) which exploits the sparsity of the neural network. More precisely, Indexing Module (IM) in BC selects only non-zero weights from centralized weight buffers, and then transfers such weights to PEs. After receiving only non-zero weights, PEs perform computation with locally stored compressed activations.

Overall architecture. Cambricon-X is composed of a Control Processor (CP), a Buffer Controller (BC), two buffers (NBin and NBout), a Direct Memory Access module (DMA) and a Computation Unit (CU) including multiple Processing Elements (PEs). The BC selects necessary weights (i.e., non-zero weights) for each PE and transfers them to PEs for skipping computation associated with zero weights. IM in the BC performs key role in zero-skipping. CU is designed for accelerate vector multiplication-addition operation utilizing multiple PEs, and PE is further divided into multiple multipliers for parallel execution.

Buffer Controller (BC) and Indexing Module (IM). The BC includes IM which is utilized to index non-zero weights. By indexing only non-zero weights and transferring them to PEs, Cambricon-X skips computation associated with zero weights (i.e., zero-weight-aware). In order to reduce overhead due to IM, they centralize IM on BC instead of adopting it on each PE, which, however, prevent us from utilizing zero activations for further improvement.

Chapter 4

Zero-aware Neural Network Accelerator (ZeNA) V1

4.1 Idea Overview

We explain our basic idea to exploit sparsity by comparing state-of-the-art hardware accelerators, Cnvlutin [5] and Cambricon-X [6], where each of them take advantage of utilizing sparsity in activations and weights, respectively. Figure 4.1 and Figure 4.2 illustrate how previous architectures which are aware of only one type of zero value (e.g., zero activations or zero weights) compute two convolutions, $K_0 * A_0$ and $K_1 * A_0$, in parallel, where K and A denote kernel weights and activations, respectively. Assume that each set of kernel weights is associated with an output feature map. The PE produces the partial sums of output feature maps, P_{00} and P_{10}, as the result of convolutions.

Figure 4.1 illustrates Cnvlutin which aims at saving computation involving zero activations. Parallel lanes in the same PE utilize the same activation data (A_0) stored in shared memory to perform two parallel convolutions with two different sets of kernel weights, K_0 and K_1. Thus, computations associated with zero activations can be skipped in those two convolutions in a synchronous manner. Simplified mechanism for exploiting zero activations is illustrated in Figure 4.1 where the solid region in the rectangle of activation A_0 represents the amount of non-zero activations. Cnvlutin skips multiplications and accumulations associated with zero activations represented by the empty region in the rectangle of activation. Thus, its execution time is proportional to the amount of non-zero activations. However, the design of Cnvlutin misses the opportunity to utilize zero weights for further reduction in runtime and energy consumption. As illustrated in *total runtime* in Figure 4.1, Cnvlutin cannot utilize zero weights represented by patterned regions in the rectangle of kernel weights, K_0 and K_1, to reduce runtime. This is because different lanes within the same PE share activations to perform two convolutions, and thus, computation associated with zero weights can be skipped only if all the kernel weights are zero during the same cycle. However, observing all-zero kernel weights in a cycle is rare in a typical Cnvlutin configuration which runs multiple (e.g., 16 in [5]) convolutions in a synchronous manner to provide enough performance.

Figure 4.2 illustrates Cambricon-X which utilizes zero weights to save computation. Each PE receives a set of non-zero weights (solid region in rectangular K_0 and K_1) together with associated activations, and then PEs execute two convolutions in parallel. Each PE consists of multiplication-and-adder-tree which computes set of computations synchronously. Figure 4.2 shows how Cambricon-X exploits zero weights to save runtime. Each PE of Cambricon-X skips computations associated with zero weights represented by the empty regions in the rectangle of kernel weights, K_0 and K_1. Due to synchronous execution of set of PEs (PE 0 and PE 1 in Figure 4.2), execution time of Cambricon-X is proportional to the amount of non-zero weights of straggler PE (i.e., PE who finishes its execution late). In addition, since each PE computes multiple multiplications and additions synchronously to generate an output feature map (denoted as *SIMD-like* in Figure 4.2), Cambricon-X is unable to exploit the opportunity of zero activations represented by the patterned regions in the rectangle of kernel weights, K_0 and K_1 for speedup.

Figure 4.3 illustrates our proposed architecture. Instead of synchronously executing multiple convolutions (or computations) which prevents us from exploiting both zero weights and activations, each PE of our architecture performs a signle computation for a single convolution at each cycle. Providing independent control to each convolution execution allows each PE to individually skip computations associated with either zero weights or activations without being limited by synchronization. Figure 4.3 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, and thus our proposed architecture achieves further runtime reduction compared to accelerators which exploit only one type of zero values. In Chapter 4.2, we will describe the architecture of our proposed design in detail.

Exploiting zero values to skip computations introduces a new problem to the hardware design, which we call *zero-induced load imbalance*. As shown in Figure 4.3, PE 1 finishes the convolution computation much earlier than PE 0. However, 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 Chapter 4.3 we will propose *zero-aware kernel allocation* as a solution to mitigate this problem.

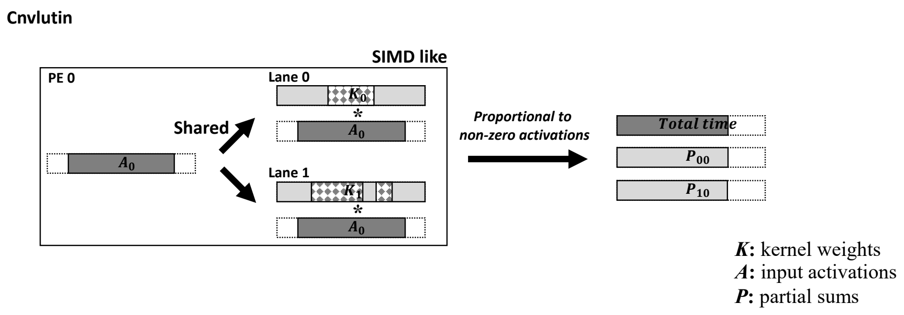


Figure 4.1 Simplified operation diagram of Cnvlutin [5].

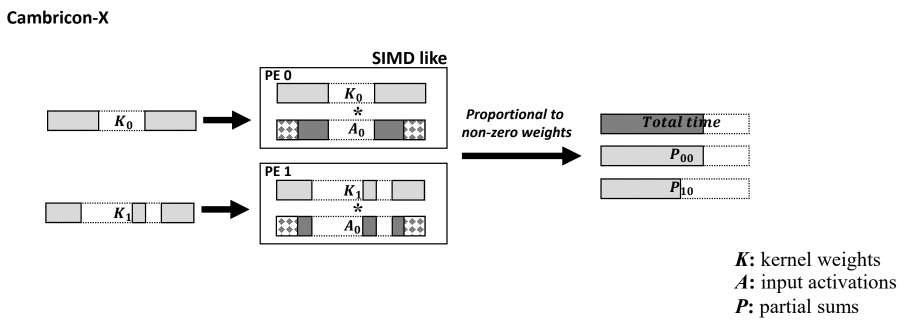


Figure 4.2 Simplified operation diagram of Cambricon-X [6].

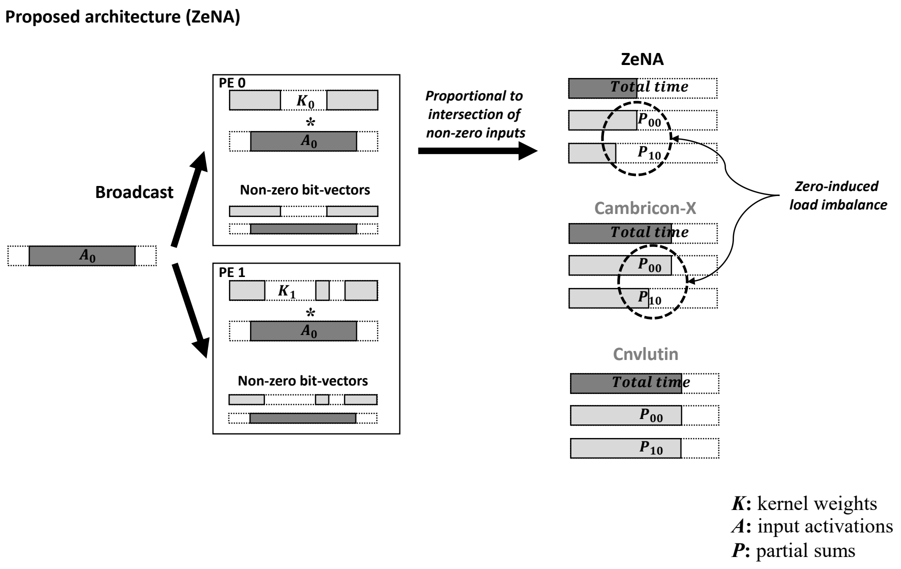


Figure 4.3 Simplified operation diagram of ZeNA V1 [2].

4.2 Architecture

4.2.1 Architecture Overview

Figure 4.4 illustrates the top-level architecture of ZeNA V1 [2] consisting of *on-chip SRAM*, *PE array* and *ReLU module*. *Inter-cluster control module* (not shown in the figure for brevity) orchestrates the data transfer between the SRAM, the PE array and the ReLU module where every component communicates via *global bus*. On-chip SRAM is further divided into *Act SRAM* which stores activations, output partial sums and output feature maps (i.e., the final result of a convolutional layer), and *Weight SRAM* which stores kernel weights. In order to skip computation associated with either zero weights or activations, we utilize non-zero bit-vector which stores the information about whether each weight/activation is zero or not. Act SRAM and Weight SRAM store data together with non-zero bit-vectors. ReLU module performs the non-convolutional operations such as nonlinear function, normalization and maximum operations, and generates non-zero bit-vectors of activations.

4.2.2 Work Distribution

Figure 4.5 illustrates how we distribute data to each PE. *Work Group* (WG) is a spatial partition of activations (① in Figure 4.5) which is allocated to a set of PEs called *PE group* (e.g., dotted box in Figure 4.4). Similarly, a WG is further divided into multiple subsets called *sub-WG* in the vertical (i.e., channel) dimension (② in Figure 4.5). Each sub-WG consists of associated subset of activations called *activation tiles* and weights called *kernel tiles* (③ in Figure 4.5).

4.2.3 Dataflow and Computation

ZeNA performs partial convolution between kernel and activation tiles iteratively to compute output feature maps. ZeNA first computes partial sums for the current sub-WG by sliding over the tiles (① in Figure 4.6). Utilizing the next set of activation tiles, kernel tiles and associated partial sums computed previously, ZeNA generates new partial sums for the current sub-WG (② in Figure 4.6). Above procedures are repeated until a complete output feature map of assigned sub-WG is generated (③ in Figure 4.6).

Figure 4.7 and Figure 4.8 illustrate how WGs and sub-WGs are mapped to each PE together with the execution flow of the proposed architecture. The overall execution flow of the proposed architecture can be summarized as follows:

1. Given a convolutional layer, we first divide the spatial dimension of activations into WGs, and then, each WG is further divided into sub-WGs. The WG including several sub-WGs is assigned to group of PEs named PE group to perform convolution. (e.g., PE group 0/1 surrounded by dotted box in Figure 4.7 process WG 0/1).

2. Activation and kernel tiles in a sub-WG are broadcast from on-chip SRAM to PEs. All PEs assigned the same WG receive the same activation (e.g., every PE in PE group 0 receives the same activation tile, A_0, in Figure 4.8), while PEs in different PE groups but with the same index receive the same weights (e.g., PE 0 of PE group 0 and PE group 1 receive the same kernel tile, K_0, in Figure 4.7).

3. Utilizing assigned activation and kernel tiles, each PE calculates the partial sums of convolution and accumulate them in Act SRAM.

4. After a PE completes the sub-WG whose result is a set of output feature maps, the convolution result (*Conv result* in Figure 4.4) is transferred to the ReLU module. The ReLU module generates output of the convolutional layer (*Output feature map* in Figure 4.4) as well as a non-zero bit-vector, and stores them in Act SRAM.

5. Step 2 to 4 are repeated until all activations tiles in a sub-WG are consumed.

6. Step 2 to 5 are repeated until all sub-WGs in a WG are consumed to complete a convolutional layer.

7. The output feature maps are used as the input activations for the next layer in the CNN.

The following describes the details of the above steps based on the example in Figure 4.7 and Figure 4.8.

Kernel broadcast. At the beginning of a sub-WG, each PE fetches kernel tiles associated with the current sub-WG into the local buffer (named *Weight buffer*) of each PE. In order to reduce the bandwidth requirements of Weight SRAM, ZeNA determines which kernel tile is stored in the PE based on the PE index (of different PE groups). At the beginning of Step 2 shown above, ZeNA fetches kernel tiles from Weight SRAM and broadcasts them to the PEs with the same index. As shown in Figure 4.7 which shows the example of kernel tile broadcast, PE 0 of PE group 0 and PE group 1 where each of them is assigned WG 0 and WG 1, respectively, stores the same kernel, K_0. Broadcasting kernel tile to all PE groups, our architecture reduces the memory traffic between Weight SRAM and PE array. In order to take advantage of skipping computation associated with zero weights at the PE side, 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). Since our proposed architecture aims to accelerate inference, trained model is fixed before execution. Thus, we pre-compute non-zero bit-vectors for kernel tiles at design time and store them in Weight SRAM together with the associated kernel tiles. They are also broadcast to PEs when the kernel tiles are broadcast.

Activation broadcast and partial convolution. Activations are broadcast after kernel broadcast. Figure 4.8 shows an example of activation broadcast as well as procedures of executing convolution in proposed architecture. Taking advantage of our work distribution model where all the PEs in the same PE group perform convolution for the same activations, activation tiles of the current sub-WG are broadcast to PEs in the same PE group to reduce memory traffic between Act SRAM and PEs. For instance, PEs in PE group 0 receive the same activation A_0 in Figure 4.8. In order to exploit zero values in the activation, each activation tile is also paired and broadcast with a non-zero bit-vector. Since the output feature maps of the previous layer are utilized as input feature maps of the next layer in CNN, we generate activation non-zero bit-vectors for the next layer while storing output feature maps of the current layer. ReLU module generates non-zero bit-vector when activation (i.e., the output feature map of the previous layer) is generated by it (shown as *Bitvec* in Figure 4.4).

After both activation and kernel tiles (as well as 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. As shown in Figure 4.8, PE 0 in PE group 0 performs convolution with A_0 and K_0, and then, produces partial sum result for its associated output feature map, Y_{00}. Partial sum result is stored into a local buffer in the PE called *Psum buffer* which is initially filled during activation broadcast with the previous partial sums associated with the current activation tile. Psum buffer is flushed back to Act SRAM during the idle period of the bus to avoid bus contention between activation broadcast and partial sum flush. In addition, to optimize partial sum flush, we prioritize PE groups with the largest difference between the number of broadcast activation tiles and the number of flushed partial sums.

After finishing the convolution for the current activation tile in the current sub-WG, the next activation tile is broadcast to the PEs. There is an overlap between the next and current activation tiles (① in Figure 4.8) because convolution operations slides over the activations. ZeNA *reuses the activation* by broadcasting only the difference between the current and the next activation tiles, thereby reducing the data traffic between Act SRAM and PEs. This is illustrated in Figure 4.8 where only the difference of the activation tile, patterned region in ①, is broadcast to the PEs of PE group 0.

In addition, ZeNA broadcasts activation tiles in the WG in zig-zag order, thereby increasing overlap between the current and the next activation tiles. We first move the window horizontally by a stride to the right or left and process the next activation tile after the current activation tile is processed. When it reaches the horizontal edge of the input activation, we move the window down by a stride and again move horizontally in the opposite direction in a zigzag fashion to choose the next activation tile (② in Figure 4.8).

After finishing convolution for all activation and kernel tiles in the current sub-WG, the final convolution result is transferred to the ReLU module. ReLU module performs non-convolutional operations such as nonlinear function, normalization and maximum operations whose result (i.e., a set of output feature maps) is stored into Act SRAM. Note that non-zero bit-vectors of activations are also generated and stored together with the result.

Executing next sub-WG. After finishing convolution for the current sub-WG, the PEs load data to execute the next sub-WG and perform convolution for the next set of output feature maps associated with the assigned WG. First, the PEs receive new kernel tiles via broadcast as explained in *Kernel broadcast*. Then, following the same procedure explained in *Activation broadcast and partial convolution*, PEs perform convolution for the new sub-WG. When all sub-WGs in the current WG are processed, the PEs start convolution for the next WG. This process continues until all WGs are processed.

4.2.4 Zero-aware Processing Element Architecture

Figure 4.9 and Figure 4.10 show the microarchitecture of a zero-aware PE. Each PE consists of a *fetch controller*, *data path*, and three local buffers including *Act buffer*, *Weight buffer* and *Psum buffer*. The fetch controller is the key to skipping computations associated with zero values. It receives activations and weights as well as associated non-zero bit-vectors from Act SRAM and Weight SRAM. As illustrated in Figure 4.9, it performs a *logical AND operation* of the two non-zero bit-vectors to generate 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. This allows us to skip the multiplications whose results will be zero due to either zero activation or zero weight. In Figure 4.9, *curridx* and *nextidx* denotes entries to be loaded to the current and the next cycle, respectively. Note that the '1' in non-zero bit-vectors represents associated activations or weights that are non-zero.

Implementing two types of zero-aware PEs, we briefly analyzed the impact of quantization on zero-aware hardware accelerator. Note that detailed analysis of correlation between zero-aware architecture and reduced precision will be described in Chapter 6. In terms of the data path, we implemented two flavors of a zero-aware hardware accelerator by applying different low precision methods: 16-bit fixed-point (Figure 4.9) and 5-bit logarithmic quantization (LogQuant [7]) (Figure 4.10).

In order to meet the clock frequency constraint, the PE based on 16-bit fixed-point implementation consists of 4-stage pipeline including fetch, computation, and write stages (Figure 4.9). However, PE based on LogQuant implementation adopts a 3-stage pipeline where data path consists of a shifter and an accumulator. Figure 4.10 illustrates zero-aware PE where data path aims to compute neural networks applied logarithmic quantization [7]. In order to perform convolution, the PE loads non-zero activation, a, and weight, w (pointed by *curraddr* in Figure 4.10). Then, it shifts activation, a, by the amount of weight, w, to perform multiplication, since the weight is quantized by a log scale (i.e., $a \times w = a \ll w$).

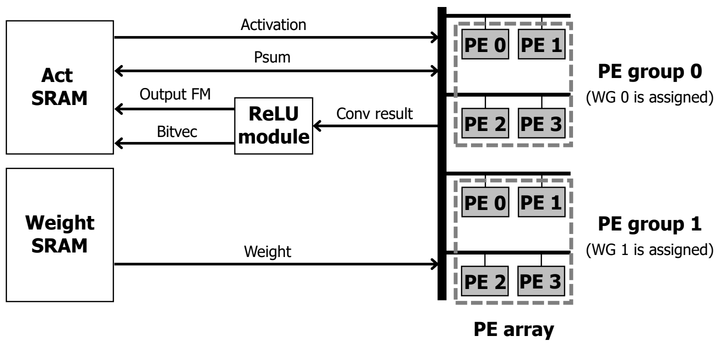


Figure 4.4 Top-level architecture of the proposed ZeNA V1 where *output FM* denotes output feature map, *Bitvec* denotes non-zero bit-vector, and *Conv result* represents convolutional result before applying nonlinear function.

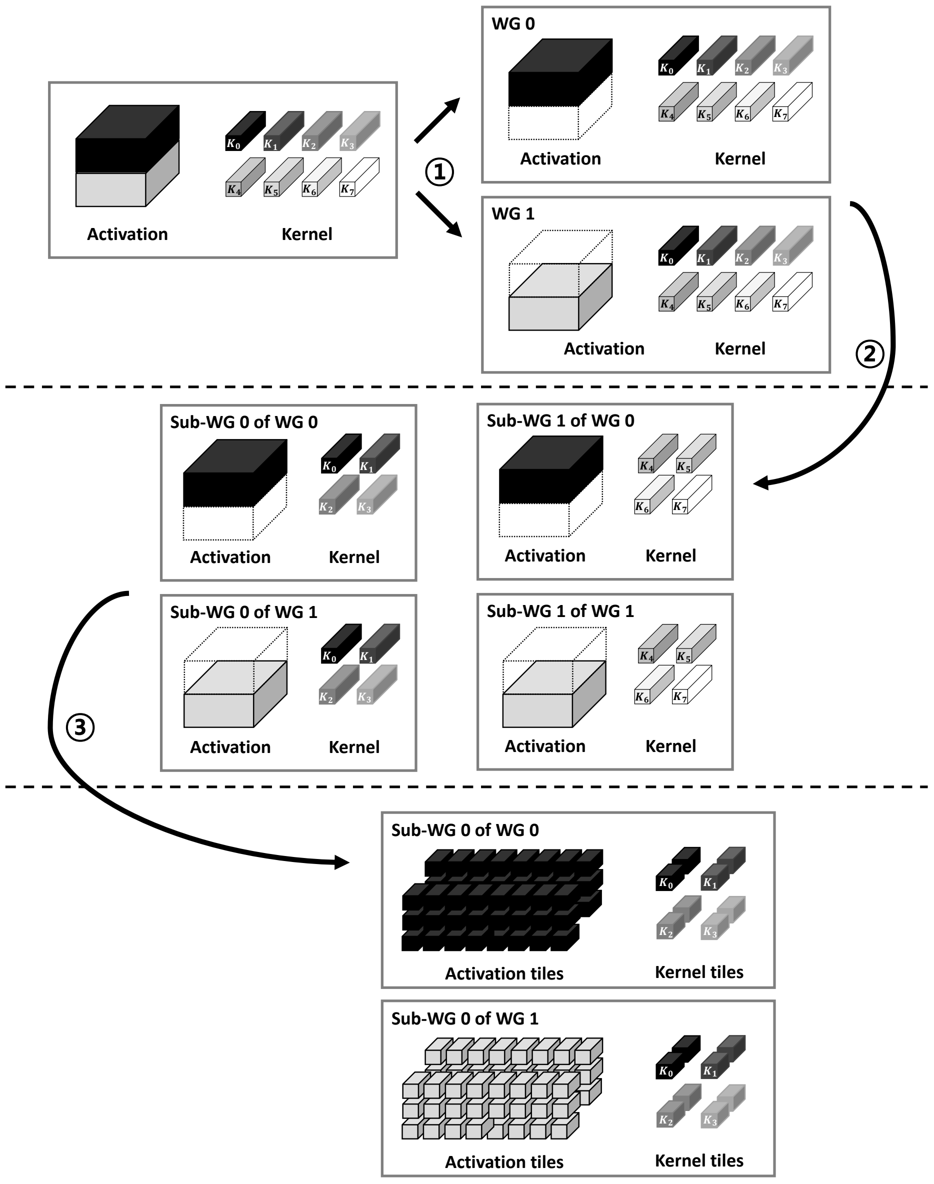


Figure 4.5 Structure of Work Group (WG), sub-WG and activation/kernel tile.

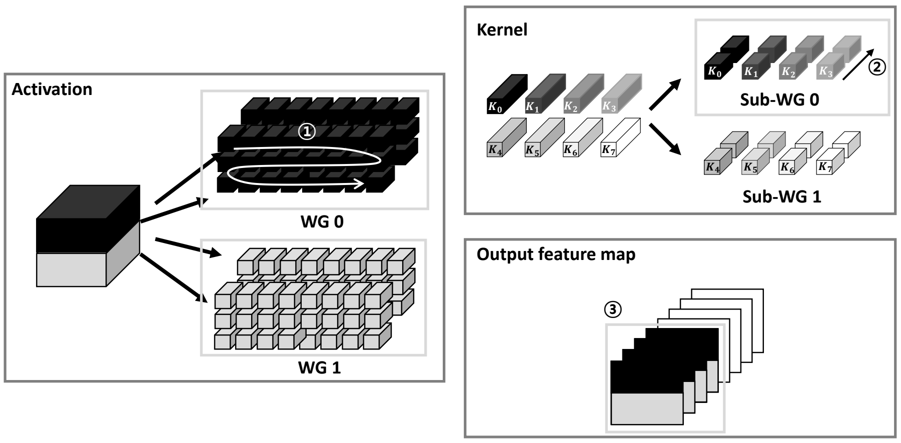


Figure 4.6 Computation procedures of proposed architecture.

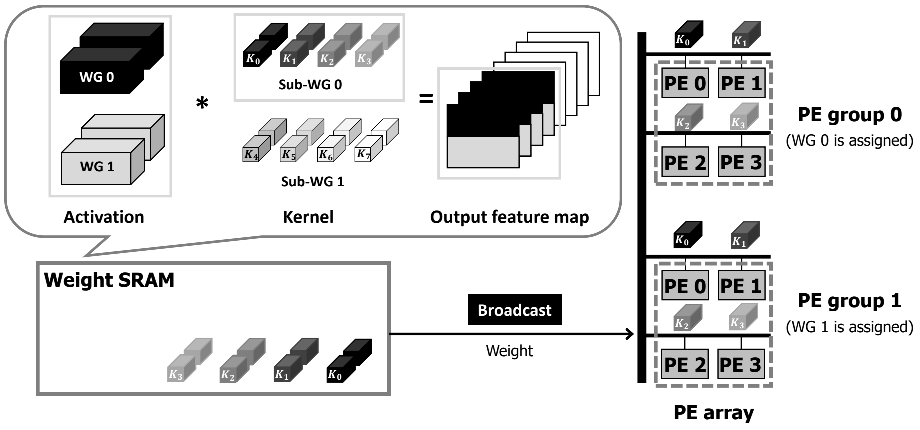


Figure 4.7 Kernel broadcast procedure. PEs which have the same index but in the different PE group receive the same kernel tile via broadcast.

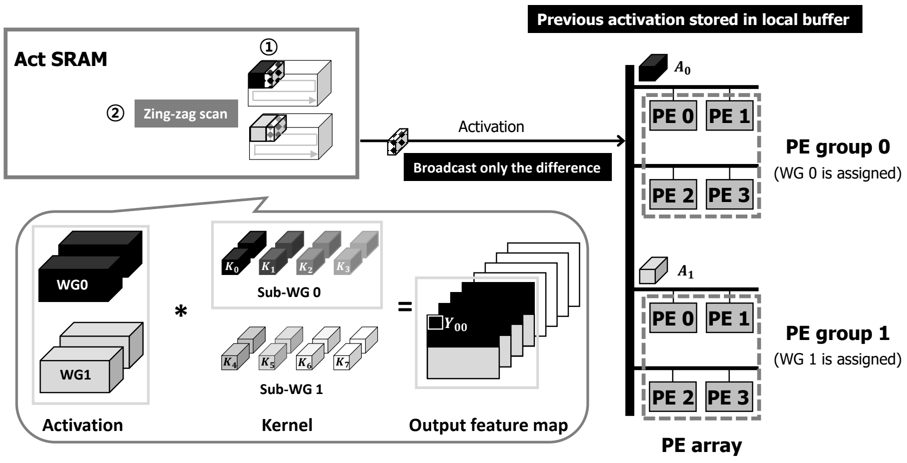


Figure 4.8 Activation broadcast procedure where PEs in the same PE group receives the same activation tile via broadcast. Since previous activation tile is stored in the local buffer of PE, ZeNA broadcasts only the difference to perform convolution with next activation tile while sliding the window.

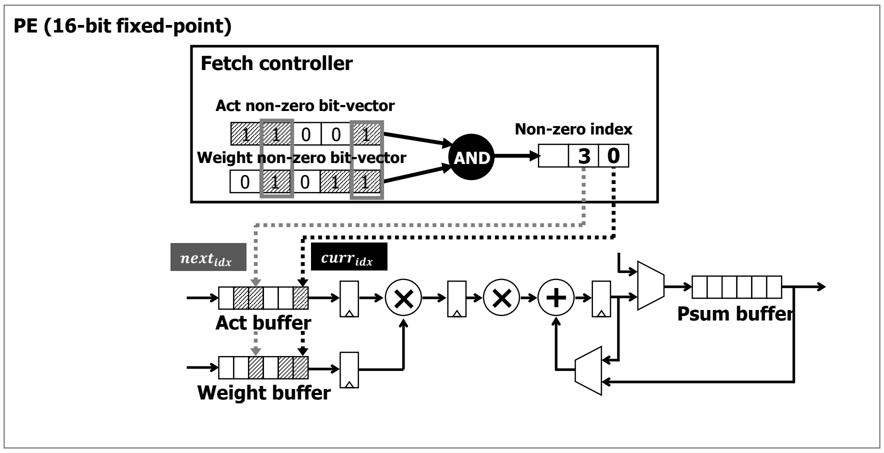


Figure 4.9 Microarchitecture of zero-aware PE of ZeNA V1 (16-bit fixed-point).

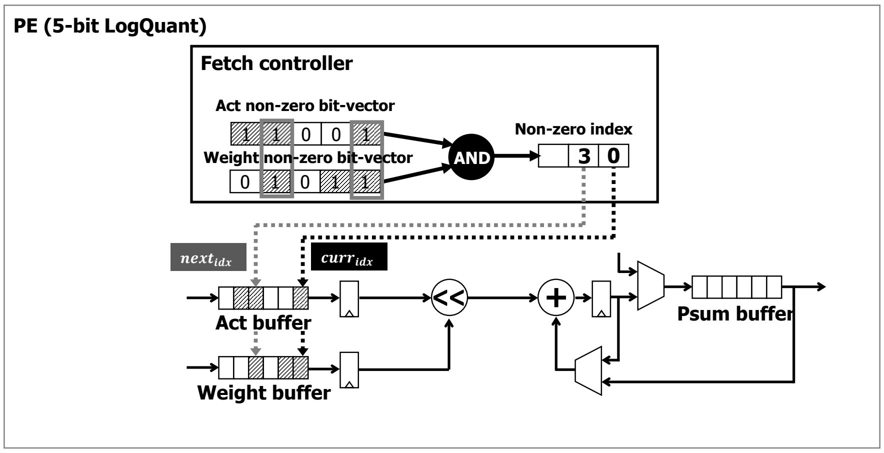


Figure 4.10 Microarchitecture of zero-aware PE of ZeNA V1 (5-bit LoqQuant [7]).

4.3 Kernel Allocation

4.3.1 Intra-WG Load Imbalance

In order to support independent control for PEs while maximizing data reuse, ZeNA performs convolutions with two levels. First, ZeNA spatially divides input activations into WGs which are further divided into sub-WGs and assigns them to a set of PEs called PE group. Until finishing processing a WG, each PE in the PE group computes an output feature map from its own kernel weights and the activations in the sub-WG iteratively. However, since each PE in the same PE group is assigned a distinct set of kernel tiles with varying proportion of zero weights, they have different amounts of effective computations where both activation and weights are non-zero. Thus, some PEs can finish computation later than others in the zero-aware architecture while executing a single sub-WG, even though all PEs in the PE group perform convolutions with the same activations. We call this uneven work distribution across PEs, *intra-WG load imbalance*.

Figure 4.3 illustrates an example of intra-WG load imbalance where PE 0 finishes execution much later than PE 1, and thus, other PEs in the same PE gorup (i.e., PE 1) have to wait until the straggler PE complete its execution. This load imbalance degrades the efficiency of the zero-aware architecture. The main reason for intra-WG load imbalance is that 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.

4.3.2 Kernel Allocation

As shown in Figure 4.7, typically, kernel weights are allocated to PEs based on the kernel index (e.g., K_0 is allocated to PE 0 in the PE group). In order to mitigate intra-WG load imbalance problem, we propose *zero-aware kerenl allocation* which allocates kernel weights to PEs in a zero-aware manner. In this method, given a convolutional layer we apply the following procedures:

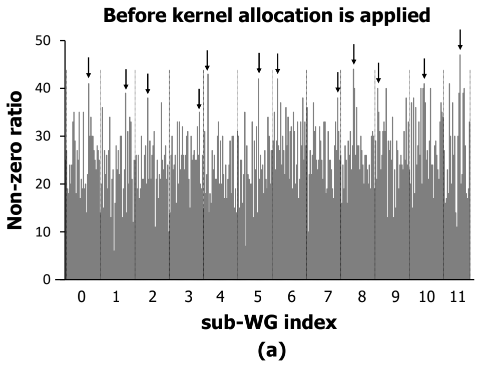
1. We first sort all the sets of kernel tiles in the increasing order of the number of non-zero weights in the sets.

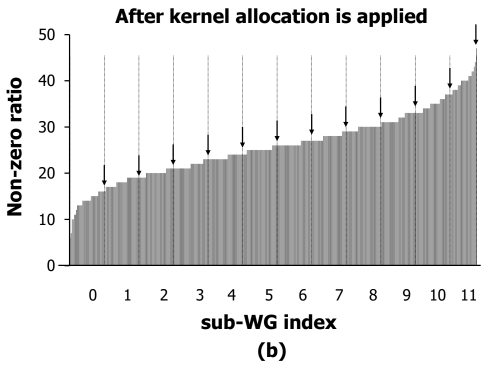
2. Then, we allocate sets of kernel tiles to sub-WG in the sorted order.

Figure 4.11 illustrates the intra-WG load imbalance problem and our proposed solution with a simplified example of a WG containing 384 kernel tiles (size of $3 \times 3 \times 14$) from the third convolutional layer (conv3) of the pruned AlexNet. Assuming that 33 PEs are allocated to the WG (i.e., PE group contains 33 PEs), the WG is divided into 12 ($= \lceil 384/33 \rceil$) sub-WGs. As described in Chapter 4.1, in this case, 33 PEs in the same PE group perform convolution on a sub-WG consisting of a set of 33 kernel tiles and proceed to the next sub-WG. This process continues until the convolution of 12rd sub-WG (with 21 kernels tiles) is completed.

Figure 4.11a shows the non-zero weight ratio per kernel tiles in each sub-WG. Each narrow bar corresponds to the non-zero weight ratio of each kernel tile, which is proportional to the execution time of the associated PE. Note that typically a set of kernel tiles is assigned to a sub-WG based on the kernel index. As shown in the Figure 4.11a, there is a significant variance in the non-zero weight ratio of kernel tiles in a single sub-WG. Since a PE group proceeds to the next sub-WG only after all PEs finish convolution for the current sub-WG, the runtime of each sub-WG is determined by the straggler PE which finishes computation last (kernel tile which has the largest number of non-zero weights at each sub-WG is pointed by arrow in Figure 4.11). Thus, conventional kernel allocation policy which assigns kernels without considering non-zero weight ratio suffers from severe load imbalance, thereby degrading performance of the zero-aware architecture.

Figure 4.11b illustrates the result of proposed zero-aware kernel allocation. Unlike conventional kernel allocation policy, kernel tiles are sorted in the ascending order of non-zero weight ratio and allocated to sub-WGs. Each sub-WG contains more uniformly distributed kernel tiles where non-zero weight ratio is balanced. As shown in the Figure 4.11c which shows magnified graph for sub-WG 0, zero-aware kernel allocation policy distributes non-zero weights more uniformly inside each sub-WG, thereby acheiving better load balance across PEs in the same sub-WG. It enables further improvement in both runtime and energy consumption.





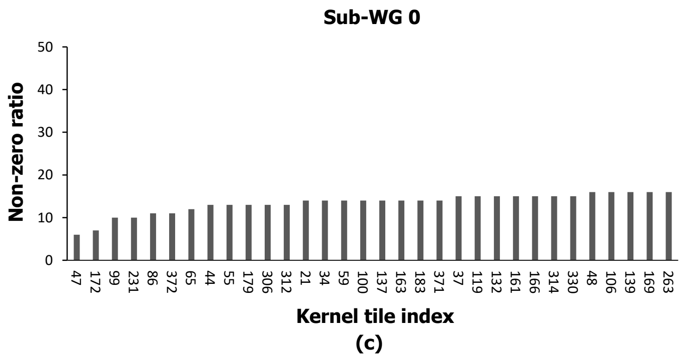


Figure 4.11 (a) Non-zero weight ratio before kernel allocation is applied, (b) non-zero weight ratio after kernel allocation is applied, and (c) non-zero weight ratio of each kernel tile in sub-WG 0.

4.4 Evaluation

4.4.1 Evaluation Methodology

We developed RTL implementations of the baseline (Eyeriss [3]) and two flavors of our architecture, including 16-bit fixed-point based (which we call FIXEDPOINT) and LogQuant [7] 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 [39] 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 ZeNA V1. Table 4.1 gives the details of our 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 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 4.1). 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.

We used a pruned model of AlexNet in [30] and a pruned version of VGG-16, which was obtained by thresholding kernel weights to meet the reported ratio of zero weights in [30]. We obtained the logarithm-based quantization of activations and weights for LogQuant by following the procedure described in [7]. To run AlexNet and VGG-16 on the proposed architecture, we allocated PEs to WGs as shown in Table 4.2. For instance, in conv1 in AlexNet, FixedPoint has 5 WGs (=165 PEs / 33 PEs), each of which is allocated 33 PEs. We used Eyeriss as the zero-agnostic baseline of 16-bit architecture. In order to evaluate the effect of skipping computation associated with zero values, we ran our accelerator with four modes:

- Zero-weight-aware mode (**WZ**).
- Zero-activation-aware mode (**AZ**). Note that AZ corresponds to the idea of Cnvlutin [5].
- Zero-weight- and zero-activation-aware mode (**WAZ**).

- WAZ with the zero-aware kernel allocation method (**WAZ+KA**). Note that WAZ+KA is proposed in ZeNA V1.

4.4.2 Performance

Figure 4.12 shows the speedup of ZeNA V1 (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 over 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.

4.4.3 Energy

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, ZeNA V1 (**WAZ+KA**) achieves higher energy reduction since it exploits both zero weights and zero activations to skip the operation of

Table 4.1 Architecture configuration of proposed architecture (ZeNA V1 [2]) and baseline (Eyeriss [3])

		Eyeriss [3]	ZeNA V1 (FIXEDPOINT)	ZeNA V1 (LOGQUANT [7])
# 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 × 16 b 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 4.2 The number of PEs in a PE group 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

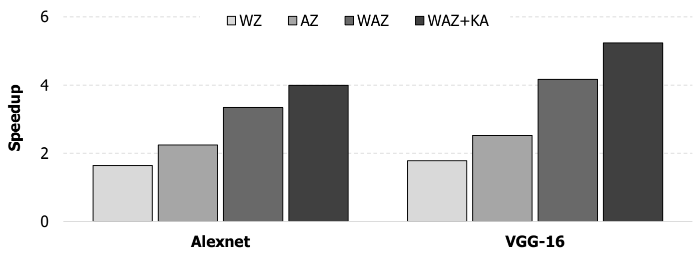


Figure 4.12 Speedup of proposed architecture (FIXEDPOINT) in AlexNet and VGG-16.

the local buffer and logic. Thus, as Figure 4.13 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. Figure 4.13 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 4.2. This improves the efficiency of broadcast (i.e., the ratio of data reuses), which is proportional to the number of PEs in a WG. Second, VGG-16 requires larger on-chip SRAM than AlexNet (Table 4.1), 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 Figure 4.14 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).

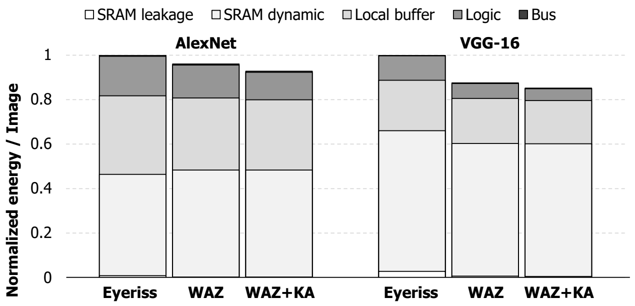


Figure 4.13 Energy consumption of the baseline and the proposed architecture.

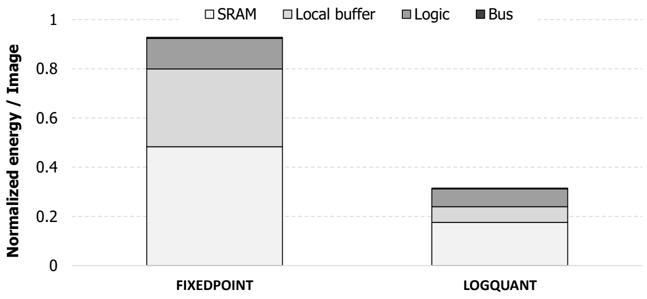


Figure 4.14 Energy consumption of FIXEDPOINT and LOGQUANT.

Chapter 5

Zero-aware Neural Network Accelerator (ZeNA) V2

5.1 Idea Overview

5.1.1 Intra-/Inter-WG Load Imbalance

ZeNA parallelizes convolution operations in two levels to enable independent control of each PE while reducing data movement. Input activations are devided into WGs and they are assigned to a set of PEs called a PE group. The WG is further divided into sub-WGs, and each PE in the PE group generates an output feature map from its own kernel weights and activations in the sub-WG. Under this parallelization model, we observe the uneven work distribution which we call intra-WG load imbalance and propose a solution to mitigate this problem as described in Chapter 4.3.

However, we observe other types of uneven work distribution which prevent us from fully utilizing tbe potential of zero-aware hardware architecture. Different WGs can exhibit different execution cycles, which we call *inter-WG load imbalance*. This is because each WG takes a distinct set of activations as input, and thus, the intersection of non-zero kernel weights and activations can vary from one WG to another. Such a difference makes some WGs finish earlier than others.

Figure 5.1 illustrates how ZeNA performs four convolutions, $K_0 * A_0$, $K_1 * A_0$, $K_0 * A_1$ and $K_1 * A_1$ in parallel with two PE groups, where K and A denotes kernel weights and activations, respectively. Since each PE in ZeNA is assigned a kernel tile associated with an output feature map, four PEs in the two PE groups produce partial sums of four output feature maps, P_{00}, P_{01}, P_{10} and P_{11} as the result of convolutions. As shown in Figure 5.1, PE group 1 finishes earlier than PE group 0. In this case, PE group 1 has to wait unitl PE group 0 completes its execution, thereby degrading the efficiency of the proposed architecture. In Chapter 5.3 we will propose *dynamic WG allocation* as a solution to mitigate this problem.

5.1.2 Appropriate Memory Architecture for Embedded Systems

While 40MB of on-chip main memory (i.e., Act SRAM and Weight SRAM) is feasible in large-scale server architectures, it might not be favorable for embedded applications. Since our architecture aims to accelerate CNN which is largely used in real-time embedded systems for vision tasks such as VR, AR and autonomous car, we reduce on-chip

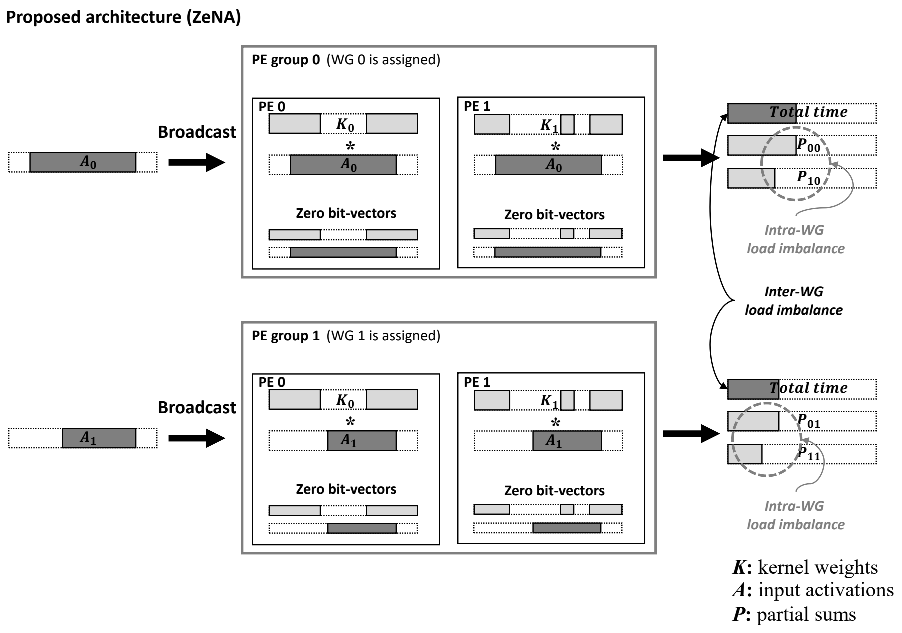


Figure 5.1 Simplified operation diagram of ZeNA which suffers from intra-/inter-WG load imbalance.

SRAM of ZeNA V1 to fit on embedded devices. We revise the memory configuration in ZeNA V1 so that the on-chip SRAM is sized to be just enough to store all current/next activations in the current WGs for double buffering in off-chip memory accesses and the rest of the data is brought from external DRAM. This results in shrinking on-chip SRAM size down to 407KB (AlexNet) and 4.82MB (VGG-16) which is feasible in embedded applications. When external DRAM is taken into account, ZeNA is expected to achieve even higher energy efficiency compared to the results. This is because the idle energy consumption of DRAM is proportional to the total runtime and ZeNA shows shorter runtime compared to the baseline (Eyeriss).

5.1.3 Memory Optimization

According to our experiment result shown in Chapter 4.4, ZeNA shows 4-5x speedup compared to the baseline and reduces energy consumption in computation logics dramatically. However, since a large portion of energy is consumed in memory including on-chip SRAM (e.g., Act SRAM and Weight SRAM) and local buffers (e.g., Act buffer, Weight buffer and Psum buffer in the PE), ZeNA V1 gives us only 11.3-18% energy reduction. In order to reduce energy consumption in on-chip SRAM, we shrink on-chip memory size and adopt external memory (i.e., DRAM) as described in Chapter 5.1.2. In addition, we propose clock gating to reduce energy consumed by local buffers in PE.

5.2 Architecture

Recently, due to the increasing use of high resolution images and videos on embedded systems such as VR, AR and autonomous car, computation requirement of mobile recognition systems has been increasing. To cope with such applications which contains a tremendous amount of computations, we also propose multi-cluster architecture for ZeNA V2 [4].

5.2.1 Architecture Overview

Each ZeNA cluster consists of a *PE array*, on-chip SRAM for activations (i.e., *Act SRAM*) and weights (i.e., *Weight SRAM*), and *ReLU* and *Bitvec modules*. Figure 5.2 illustrates the architecture of ZeNA V2 consisting of multiple ZeNA clusters and a global memory. Inter-cluster (or intra-cluster) control modules (not shown for brevity) orchestrate the data transfer between global (or local) memory and ZeNA clusters (or PEs) via a global (or intra-cluster) bus. The intra-cluster bus is connected to the PE rows via row controllers. Each row controller has a small buffer (242-byte) named *Row buffer* and receives data and row ID from the intra-cluster bus. Then, it broadcasts data to PEs in the row after ID matching. In a ZeNA cluster, Act SRAM stores input activations and output partial sums/feature maps, whereas Weight SRAM stores kernel weights. ReLU module performs the ReLU activation function, normalization, and maximum operations.

In order to skip computation associated with zero weights and activations, each PE receives a zero bit-vector (i.e., a bit-vector that stores the information about whether each weight/activation is zero or not). Utilzing non-zero bit-vectors, ZeNA skips ineffectual computation associated with either zero activations or zero weights. Since neural networks go deeper to be applied to more complex applications, future hardware acceleratsors for neural networks may require to utilize both sparsity and very low-precision methods. However, the size of non-zero bit-vector (or non-zero indices) is determined by the number of input data, not by the bit-width. Thus, as the bit-width gets reduced, the overhead of control path tends to remain the same, thereby possibly occupying a signification portion of the total cost in very low-precision.

In order to reduce on-chip SRAM energy due to non-zero bit-vectors, Bitvec module of ZeNA V2 reads activations and weights from the on-chip SRAM and generates non-zero bit-vectors from them unlike ZeNA V1. *On-the-fly non-zero bit-vector generation* can save on-chip SRAM resource otherwise required for non-zero bit-vector storage.

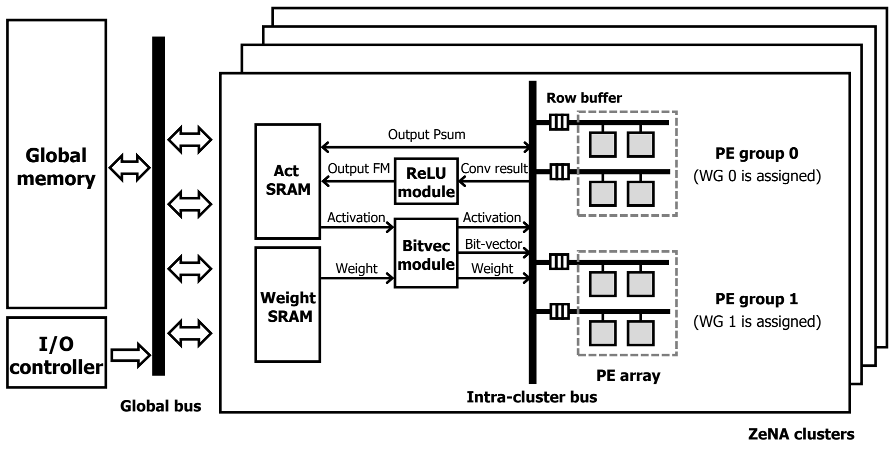


Figure 5.2 Top-level architecture of the proposed ZeNA V2 where *output FM* denotes output feature map, *Bit-vector* denotes non-zero bit-vector generated by Bitvec module, and *Conv result* represents convolutional result before applying nonlinear function.

5.3 Dynamic WG Allocation

5.3.1 Inter-WG Load Imbalance

Since each WG performs convolution with its own activation tiles, the execution time of a WG can vary across different WGs, which we call *inter-WG load imbalance*. It occurs due to the following two reasons:

1. Different WGs have different amounts of zero activations.

2. ZeNA divides the spatial dimension of activations into WGs, and thus, the number of activation tiles in each WG can be different depending on the configuration of the convolutional layer and that of the PE array.

For instance, if 14 activation tiles are grouped into four WGs, the first two WGs have four activation tiles, while the last two have only three tiles (i.e., 4+4+3+3=14). Mitigating inter-WG load imbalance enhances performance gain from zero-aware architecture.

5.3.2 Dynamic WG Allocation

In order to address the inter-WG load imbalance, we propose *dynamic WG allocation*, which is conceptually similar to work-stealing queues [40]. Intra-cluster control module of ZeNA V2 adopts down counters, each of which is associated with a WG, and stores the number of remaining activation tiles to execute. When a WG completes its computation for the current sub-WG and there are other running WGs, a leader WG whose counter value is zero steals an activation tile that was originally assigned to a straggler WG (i.e., WG with the biggest counter value) and produces an output activation for it. This can be efficiently implemented because all WGs share the same kernel tiles, and thus, the leader WG can process the newly assigned activations with its current kernel tiles without fetching any additional kernel tiles. In dynamic WG allocation, memory port contention may occur if multiple leader WGs try to fetch the same input activations from memory at the same time. To address this problem, we adopt a strategy that *re-assigns only one activation tile at a time* and restricts activation tile reassignment to happen only during bus idle time. Through this, our architecture can avoid memory port contention between leader WGs as well as bus contention between normal activation tile transfer and irregular transfer triggered by dynamic WG allocation.

5.4 Memory Optimization

5.4.1 Clock Gating

ZeNA adopts three-level hierarchical memory architecture as follows: (1) On-chip main memory named *on-chip SRAM* including *Act SRAM* and *Weight SRAM*, (2) Local register of PE row named *Row buffer*, and (3) Local registers of each PE including *Act buffer*, *Weight buffer* and *Psum buffer*. In order to reduce runtime and energy consumption utilizing zero activations and weights, each PE stores data into their local registers (i.e., Act buffer and Weight buffer). Adopting local registers in each PE, ZeNA maximizes data reuse and replaces high cost on-chip SRAM and DRAM accesses into local register accesses. As a result, ZeNA consumes a large portion of energy in local registers among total energy consumption as described in Chapter 4.4.3. Thus, optimizing local registers of the PE is the key to implementing efficient memory architecture for ZeNA. Figure 5.3 illustrates the type of on-chip memory accesses that occurs in ZeNA. Local registers of ZeNA has following four types of accesses:

- **Act/Weight buffer** to **Computation unit** access.
- **Row buffer** to **Act/Weight buffer** access.
- **Computation unit** to **Psum buffer** access.
- **Psum buffer** to **Row buffer** access.

Since ZeNA only loads non-zero inputs to perform computations, memory accesses from Act/Weight buffer to Computation unit (dotted arrow in Figure 5.3) can be reduced, thereby reducing energy consumption. However, the other types of memory access have not been utilized to optimize energy consumption in ZeNA V1. Utilizing clock gating shceme in local registers, ZeNA V2 aims to further reduce energy consumption in local registers.

Taking advatage of the fact that input data (e.g., activation and weight) is coupled with non-zero bit-vector in ZeNA, we utilize non-zero bit-vector as the enable signal of clock gating, thereby avoiding ineffectual local register accesses. Figure 5.4 illustrates exmaple of clock gating utilizing non-zero bit-vector as the enable signal where three consecutive operations occur at time t_0, t_1 and t_2. Input data a is fed to local register at t_0, and after, input data b comes in at t_1. In this case, non-zero bit-vector is '1', and thus, local register is enabled. However, when the input data is zero at next time step, t_2, the local register is disabled to prevent unnecessary data toggling, thereby reducing energy consumption.

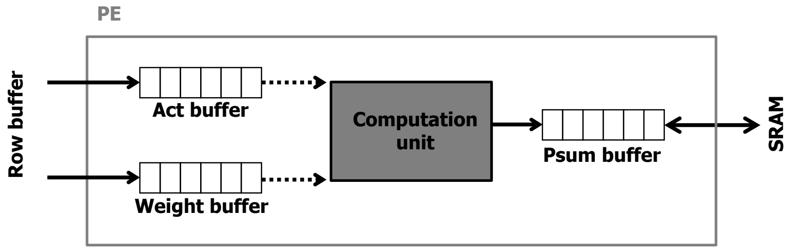


Figure 5.3 Four types of local register access in ZeNA. Each line denotes local register access where memory accesses represented by the dotted line are reduced exploiting benefit of zero-aware architecture in ZeNA V1.

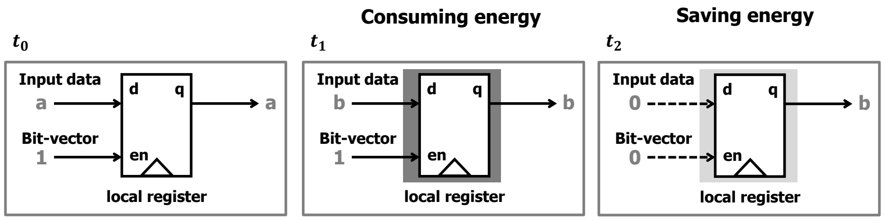


Figure 5.4 Clock gating utilizing non-zero bit-vector as the enable signal. Three consecutive operations occur at time t_0, t_1 and t_2.

5.5 Evaluation

5.5.1 Evaluation Methodology

We implemented RTL designs of the baseline (Eyeriss [3]) and two types of our accelerator, including 16-bit fixed-point based (called FIXEDPOINT) and LogQuant [7]based (5-bit activations and 5-bit weights, which we call LOGQUANT) ones, to measure the area, energy consumption, and speed. We also implemented an in-house cycle-accurate model of our accelerators for fast performance evaluation. We used Synopsys Design Compiler under the TSMC 65nm library to synthesize the RTL designs and obtained the chip layout (1.53/1.69/1.65mm^2 for the logic circuit of the baseline/FIXEDPOINT/LOGQUANT) using Synopsys Astro. We used PrimeTime PX for power estima-tion and CACTI v6.0 [11] for SRAM energy and area. We used 8Gb dual-channel mobile LPDDR3 as off-chip DRAM and modeled its access energy with Micron's DRAM Power Calculator [41]. In our experiment, peak off-chip bandwidth of FIXED-POINT and LOGQUANT is 1.98GB/s and 0.8GB/s in AlexNet (1.5GB/s and 1.64GB/s in VGG-16), respectively, which are well below the maximum bandwidth of off-chip DRAM (i.e., 12GB/s). Therefore, off-chip memory bandwidth is not a performance bottleneck in our configuration.

Table 5.1 shows the details of architectural configuration for a single ZeNA cluster equipped with 11×15 and 8×45 PE arrays for FIXED-POINT and LOGQUANT, respectively. On-chip SRAM is sized to be just enough to store all current/next activations in the current WGs for double buffering in off-chip memory accesses. By applying on-the-fly non-zero bit-vector generation, ZeNA V2 uses 3.6%/5.0% (FIXEDPOINT) and 7.5%/12.8% (LOGQUANT) smaller SRAM in AlexNet/VGG-16. We set the operating frequency to 200MHz to meet the timing constraint. The bus width of all designs is fixed to 512 bits. Compared to the baseline, FIXEDPOINT and LOGQUANT have additional logic and buffers for zero-aware computation skipping (10.5% and 8.3% area overhead in FIXEDPOINT/ LOGQUANT). We used pruned AlexNet in [30] and pruned VGG-16, which was obtained by thresholding kernel weights to meet the reported ratio of zero weights in [30]. In order to obtain logarithm-based quantization of activations and weights for LOGQUANT, we followed the procedure described in [7]. We used Eyeriss as the zero-agnostic baseline for fixed-point architectures. In order to evaluate the advantages of zero-skipping computation, we ran our accelerator with five modes:

- Zero-weight-aware mode (**WZ**)
- Zero-activation-aware mode (**AZ**, corresponding to Cnvlutin [5]).
- Zero-weight- and zero-activation-aware mode (**WAZ**).
- WAZ with the zero-aware kernel allocation (**KA**, corresponding to ZeNA V1)

- KA with dynamic WG allocation and Bitvec module (**ZeNA**, corresponding to ZeNA V2).

For each convolutional layer, we found the number of WGs (each having the same number of rows in the PE arrray) giving the largest speedup through exhaustive search at design time. We allocated PEs to WGs at the granularity of a row of the PE array. Thus, the number of possible PE allocations is small, i.e., the number of rows. Table 5.2 shows PE allocation to run AlexNet and VGG-16 (with batch size of 1) on ZeNA.

5.5.2 Performance

Figure 5.5 shows the speedup of ZeNA over the baseline (fixed-point Eyeriss) for the execution of all convolutional layers in AlexNet and VGG-16. FIXEDPOINT (ZeNA) shows 4.4x/5.6x speedup in AlexNet/VGG-16 over the baseline. Compared to the zero-activation-aware method (**AZ**, corresponding to Cnvlutin), **ZeNA** achieves 2x/2.4x speedup in AlexNet/ VGG-16. Reducing inter-WG load imbalance, ZeNA V2 (**ZeNA**) achieves 10.1%/9.2% additional improvement in AlexNet/VGG-16, compared to ZeNA V1 (**KA**). LOGQUANT (ZeNA) achieves 9x/2x speedup in AlexNet (10.5x/1.9x speedup in VGG-16) compared to the baseline (Eyeriss) and FIXEDPOINT (ZeNA), respectively, since LOGQUANT contains 195 more PEs than FIXEDPOINT due to its lower cost from narrow bit width and use of shifters instead of multipliers.

Zig-zag scan reduces up to 14% (FIXEDPOINT) and 11.9% (LOGQUANT)

Table 5.1 Architecture configuration of proposed architecture (ZeNA V2 [4]) and baseline (Eyeriss [3])

		Eyeriss [3]	**ZeNA V2 (FIXEDPOINT)**	**ZeNA V2 (LOGQUANT [7])**
# 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 × 16 b SRAM	121 × 16b SRAM	121 × 5b SRAM
	Partial sum	24 × 16b REG	16 × 16b REG	16 × 11b REG
SRAM (AlexNet)	Act	391KB	391KB	180KB
	Weight	16KB	16KB	5KB
SRAM (VGG-16)	Act	4.8MB	4.8MB	2.51MB
	Weight	22KB	22KB	7KB

Table 5.2 The number of PEs in a PE group 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

of input activation reads from on-chip SRAM in AlexNet (5.5% and 6.2% in VGG-16), respectively. Zig-zag scan gives a higher reduction in on-chip SRAM traffic for smaller spatial dimension of an input feature map.

Figure 5.6 shows the scalability of FIXEDPOINT (ZeNA) in AlexNet. Batch size represents the number of input images. For each convolutional layer, all ZeNA clusters repeatedly process convolution for the layer input of one image at a time until all images in the batch are processed for the layer. In large batch sizes (e.g., 16), ZeNA shows good scalability with speedup proportional to the number of ZeNA clusters. On the other hand, smaller batches degrade the scalability because the maximum degree of parallelism in each layer is proportional to the batch size, i.e., smaller batches can keep fewer PEs busy in parallel. The batch size of modern CNN workloads is quite large (e.g., 512), in which case the large-scale configuration of ZeNA clusters is promising.

5.5.3 Energy

Figure 5.7 compares the energy consumption of various architectures. Eyeriss cannot improve performance by skipping zero values in convolution but saves energy by applying clock-gating to computation units when activations are zero. ZeNA V2 (**ZeNA**) achieves higher energy reduction compared to Eyeriss (FIXEDPOINT: 25.4%/25.2% reduction in AlexNet/VGG-16, LOGQUANT: 77.4%/84.5% reduction in AlexNet/VGG-16) because of the following two reasons. First, ZeNA skips the opera-

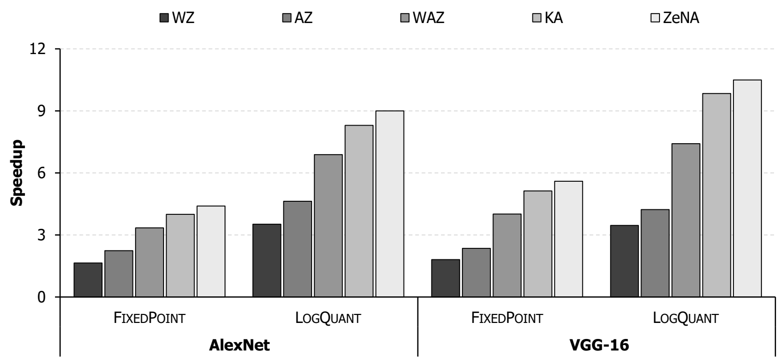


Figure 5.5 Speedup of proposed architecture in AlexNet and VGG-16.

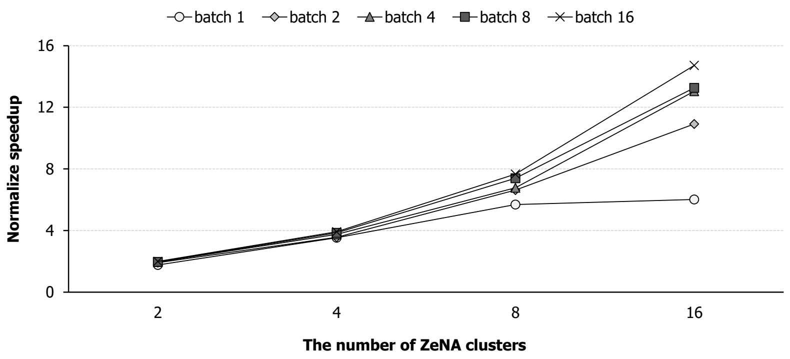


Figure 5.6 Speedup of multiple ZeNA clusters over the single cluster in AlexNet.

tion of the local buffer and logic associated with zero weights and/or zero activations. Second, shorter execution time of ZeNA compared to the baseline contributes to reducing the static energy consumption, particularly, the idle energy consumption of DRAM. Figure 5.7 shows that FIXEDPOINT (ZeNA) and LOGQUANT (ZeNA) give 1.4% and 2% additional energy reduction (w.r.t. **KA**) in AlexNet (2.2% and 7.5% in VGG-16), respectively. This further energy reduction is mainly achieved by bit-vector generation mechanisms. In order to skip ineffectual operations associated with zero values, both **KA** and **ZeNA** require zero bit-vectors. Unlike on-the-fly bit-vector generation of ZeNA V2 (**ZeNA**), ZeNA V1 (**KA**) stores pre-computed zero bit-vectors in on-chip SRAM, thereby consuming additional resource and energy. LOGQUANT achieves larger gain than FIXEDPOINT from this because zero bit-vectors occupy a relatively larger portion of on-chip SRAM under narrower bit width. Thus, we expect that on-the-fly zero bit-vector generation will become more useful in future designs with more aggressive reduced precision, e.g., 1–4 bits.

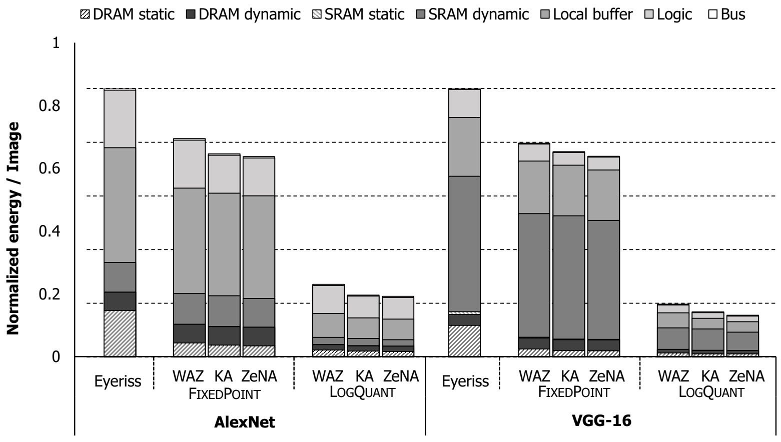


Figure 5.7 Energy consumption of the ZeNA V2 in AlexNet and VGG-16.

Chapter 6

Further Analysis

6.1 Impact of Data Compression on ZeNA

In order to reduce external memory access energy, exploiting abundant zero values in both activation and weights, ZeNA V1 and ZeNA V2 architecture trasfer data between on-chip SRAM and external memory in compressed format. ZeNA stores data on on-chip SRAM in decompressed format for simplicity. However, storing data on on-chip SRAM in decompressed format may prohibit further energy reduction utilizing data compression. To analyze the impact of compression on fine-grained zero-aware architecture such as ZeNA, we implemented two types of designs: (1) Applying compression/decompression between external DRAM and on-chip SRAM, which is the original ZeNA design, and (2) applying compression/decompression occurs between on-chip SRAM and PE array.

6.1.1 Data Compression Methodology

We utilize non-zero bit-vector which coupled with associated data (e.g., activation and weight) as metadata for compression and decompression. Since compression on ZeNA aims to compress abundant duplicated zero values, we only apply compression on activation and weight excluding partial sums due to lack of zero values. Figure 6.1 shows original ZeNA V2 architecture which adopts the *Compression/decompression module* between external memory (e.g., DRAM) and on-chip SRAM. Note that ZeNA V1 contains the compression/decompression module at the same position. Figure 6.2 illustrates modified ZeNA architecture which adopts the compression/decompression module between on-chip SRAM and PE array. Since we utilize non-zero bit-vector as metadata for compression and decompression, *on-the-fly bit-vector generation* is excluded in architecture which adopts the compression/decompression module between on-chip SRAM and PE array.

Figure 6.3 illustrates how data is stored in on-chip SRAM after applying compression. Input activations and weight (as well as non-zero bit-vector coupled with associated input) are stored as a $1 \times 1 \times n$ tensor where n denotes channel dimension. Each SRAM entry includes multiple tensors in compressed format after applying the compression/decompression module between on-chip SRAM and PE array (*on-chip SRAM* in Figure 6.3). Each entry includes *metadata*, *non-zero bit-vector* and *data*

(*SRAM entry* in Figure 6.3). Details are described as follows:

- **Metadata** stores the number of tensors and non-zero data, and it is utilized while performing compression and decompression.
- **Non-zero bit-vector** is utilized to decompress data as well as to skip computations associated with zero values in ZeNA architecture.
- **Data** represents packed input activations or weights.

6.1.2 Evaluation and Analysis

To evaluate the impact of storing data on on-chip SRAM in compressed format, we compare the following four designs:

- **Eyeriss**. Baseline design which only exploits zero activations for saving energy consumption.
- **ZeNA V1**. Proposed zero-aware hardware accelerator described in Chapter 4 where extra optimization methods are not applied.
- **ZeNA V2**. Proposed zero-aware hardware accelerator described in Chapter 5 where clock gating is applied on local buffers including *Row buffer*, *Act buffer*, *Weight buffer* and *Psum buffer*.
- **ZeNA + Comp**. ZeNA adopting the compression/decompression module between on-chip SRAM and PE array described in Chapter 6.1.1.

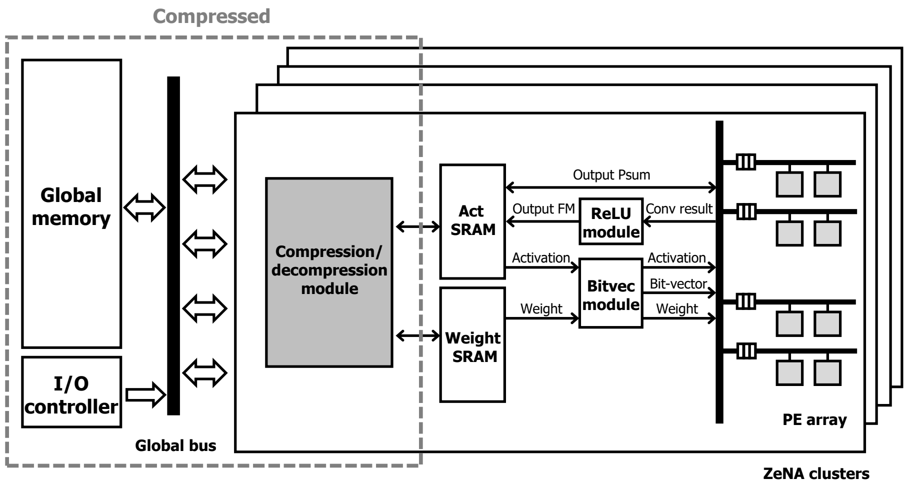


Figure 6.1 Origianl ZeNA architecture which compresses data transferred between DRAM and on-chip SRAM.

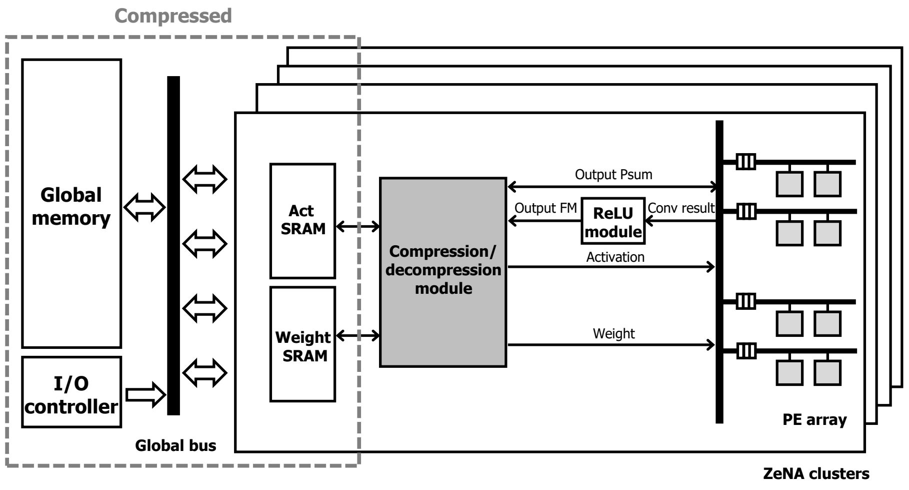


Figure 6.2 On-chip SRAM energy optimized ZeNA architecture which compresses data transfered between on-chip SRAM and PE array.

Figure 6.4 shows energy consumption comparison between *Eyeriss*, *ZeNA V1*, *ZeNA V2* and *ZeNA + Comp*. *ZeNA + Comp* gives 12.8% and 4.5% energy reduction in on-chip SRAM (3.4% and 2.5% in total energy consumption) while executing AlexNet and VGG-16, respectively, compared to *ZeNA V2*. The energy reduction of data compression is below our expectation due to the following two reasons:

- Performing compression and decompression in the middle of execution consumes extra energy. It incurs 2.1% and 1.5% additional energy consumption in AlexNet and VGG-16, respectively.
- Compression is applied on activations and weights except partial sums.

We take advantage of ZeNA architecture where data is always coupled with associated non-zero bit-vector. Proposed compression method generates compressed data by packing only non-zero data and decompresses it using non-zero bit-vector. Thus, we only applied compression on activations and weights where zero data is abundant.

Figure 6.5 shows magnified energy breakdown of on-chip SRAM shown in Figure 6.4. Due to the lack of duplicated data (e.g., zero data), partial sum is difficult to take advantage of compression, and thus, we exclude partial sum compression. However, partial sum movement dominates energy consumed on on-chip SRAM in ZeNA architecture. As shown in Figure 6.5, energy consumed by activation movement is reduced 45.8% and 39.8% in AlexNet and VGG-16, respectively. In addition, energy consumed by weight movement also is reduced 55.1% and 40% in AlexNet and VGG-16, respectively. However, at the same time, that of partial sum remains, and thus, total on-chip SRAM energy is reduced only 12.8% and 4.5% in AlexNet and VGG-16, respectively.

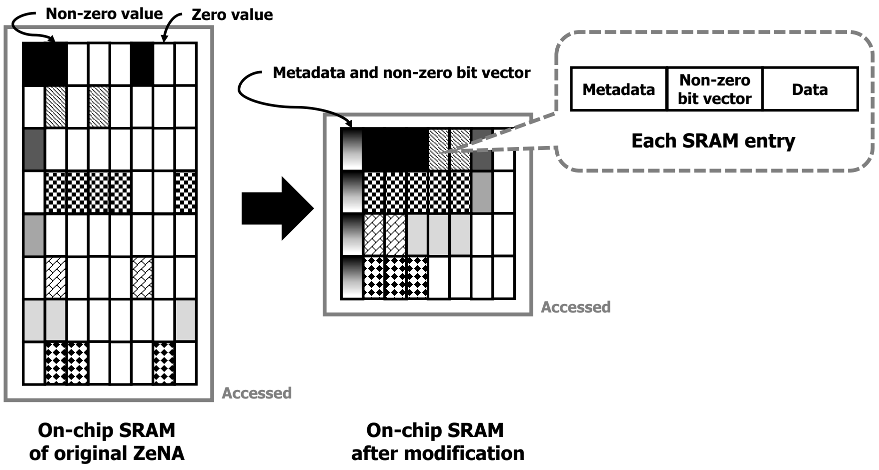
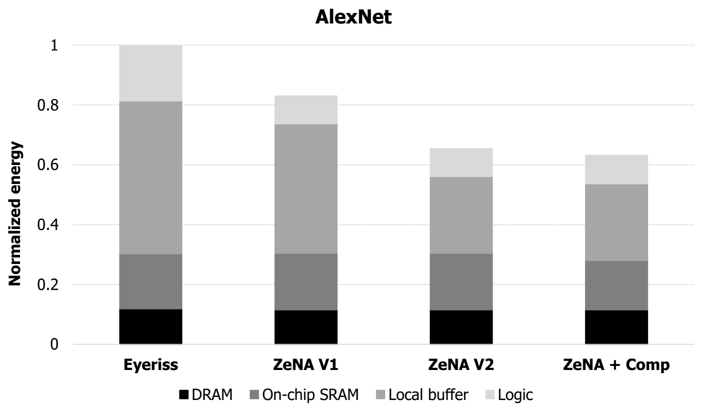

Non-zero value
Zero value
Metadata and non-zero bit vector
Metadata
Non-zero bit vector
Data
Each SRAM entry
Accessed
Accessed
On-chip SRAM of original ZeNA
On-chip SRAM after modification

Figure 6.3 Data which should be accessed after adopting Compression/decompression module between on-chip SRAM and PE array together with structure of each data entry.



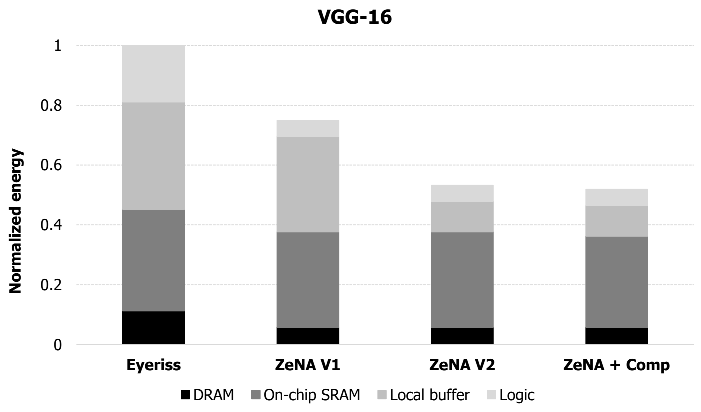
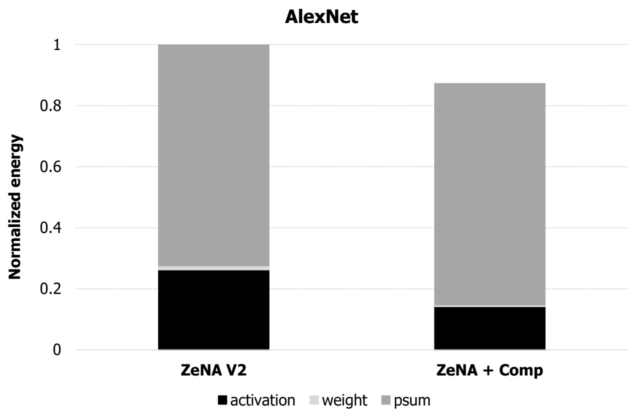


Figure 6.4 Energy breakdown after applying compression between on-chip SRAM and PE array.



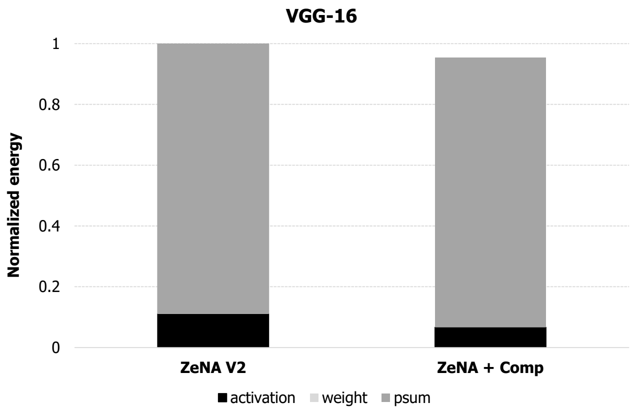


Figure 6.5 Magnified energy breakdown of on-chip SRAM. Activation, weight and psum denote the source of on-chip SRAM energy consumption.

6.2 Zero-aware Hardware Accelerator with Very Low-precision

The neural network becomes deeper to handle more complicated tasks and to achieve higher accuracy. Thus, in order to apply emerging deeper neural networks to real-time mobile and/or data center applications, future hardware accelerators for neural networks are desirable to support both zero-skipping and very low-precision scheme. Applying both zero-skipping and very low-precision approach to current architectures may incur new problems that prevent us from achieving fully optimized hardware accelerators, which however, have not been analyzed quantitatively. We propose quantitative analysis for neural network accelerators which support both fine-grained zero-skipping (e.g., ZeNA) and very low-precision.

Low-precision can offer better area/energy efficiency by allowing us to perform more computations (with more compute units) with smaller energy consumption (with smaller compute and memory units) on the same silicon area. However, such benefits are limited only to the data path. The control path does not scale with the bit-width of the data path. For instance, the size of zero bit-vector (or zero indices) is determined by the number of input data, not by the bit-width. Thus, as the bit-width gets reduced, the overhead of control path tends to remain the same, thereby possibly occupying a signification portion of total cost in very low-precision. In order to analyze the impact of low-precision approach in ZeNA, we both implement FPGA and ASIC based design and propose analysis result.

6.2.1 Prototyping

Architecture Overview

Figure 6.6 shows an overall flow of FPGA prototpying from high-level description to FPGA implementation. The neural network is designed in a neural network framework like Caffe [42] where data is processed in the form of *solver*, *net* and *layer*. Each of them performs the following functions:

- **Solver** leads the training process by solving net parameters on training data.
- **Net** consists of layers of which adjacent layers are connected to each other.
- **Layer** defines computations which input undergoes (e.g., convolution, inner product, and softmax) and produces output which is delivered to the next layer.

The FPGA device includes *PCIe core*, *Direct Memory Access (DMA)* core, *device memory* and *ZeNA* implementation. The PCIe core is designed to support PCIe specifications, a high-speed serial expansion bus, and additional functions (e.g., packet decoding, error handling, and etc.). The DMA core mainly performs memory copy between host memory (not shown in the figure for brevity but every data in host-side application is stored in host memory) and device memory. The device memory is far smaller than the host memory but is sufficiently large (e.g., a few megabytes) for running a small hardware accelerator on FPGA. The data in the device memory might be the computation result of ZeNA or the duplicate of data stored in the host memory.

The neural network framework first divides input data stored in the host memory into data chunks that are suitable for operations on ZeNA. Then, the framework drives the DMA core to move the data chunks to the device memory. After ZeNA finishes its computation, the framework drives the FPGA device again to gather the result back to the host memory, and merges it with partial results if necessary. The process is repeated until the neural network framework completes its desired work.

In our implementation, Caffe [42] is chosen due to its extensibility as a neural network framework for the whole process including training (i.e., generating kernel weights), handling the input data (i.e., input image) and performing layer functionality. We implemented a resource manager and custom Caffe layers with a set of API functions which interact with the FPGA device. Table 6.1 shows details of the environment setup for the proposed implementation.

Table 6.1 Environment setup

Host		**FPGA**	
CPU	Intel i7 7700	**Board**	Alpha Data ADM-PCIE-KU3
Memory	DDR4-2400 32 GB	**Tools**	Vivado 2016.3
Storage	SSD 1 TB	**Driver**	adb3-driver-linux-1.4.18 (1.4.19b5)
OS	CentOS 7 (Ubuntu 16.04)	**SDK**	admpcieku3_sdk-2.0.0
Kernel	3.10 (4.13)		

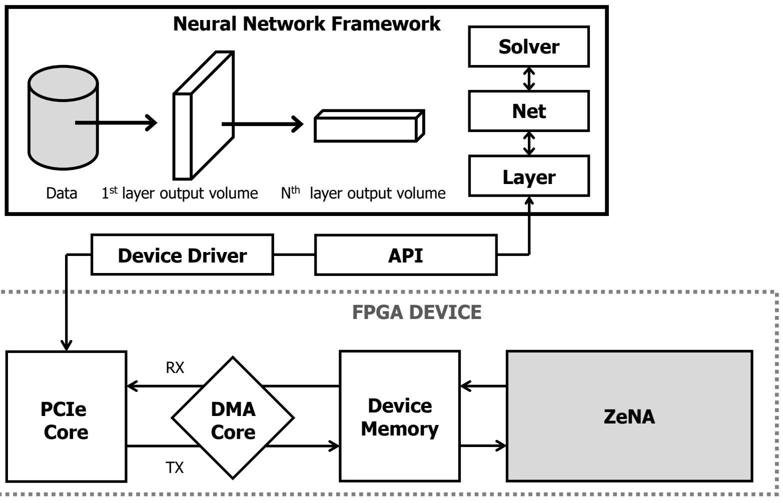


Figure 6.6 Top-level architecture of ZeNA implemented on FPGA.

PCIE and DMA

Communication between the host and FPGA is implemented with PCIe-based DMA. DMA engines enable transferring data chunks from the host memory (i.e., DRAM of PC in our implementation) to the device memory (i.e., BRAM of FPGA in our implementation), and vice versa. In order to simplify implementation, and to achieve high performance, we built our system based on commercial PCIe hard IPs and drivers. We modified reference designs of *Alpha Data* which includes a fully functional PCIe bridge, a DMA engine, and a corresponding API library [43]. By avoiding high level synthesis tools, we designed our system to gain tight control over low level signals.

Figure 6.7 shows a block diagram of the system architecture for the communication between host and FPGA. PCIe endpoint translates the PCIe transaction layer to internal memory mapped ports. A direct slave (DS) port forwards a set of host initiated PCIe transactions, up to 256-bit at once. In contrast, DMA ports generate burst transactions to support high data transfer rates. Note that burst transaction includes only 1 header (12 or 16 bytes) per 128 bytes. Since the DMA port is more efficient than the DS port in both speed and energy, we designed the system architecture in a way that a small number of control signals are transferred through the DS port while most data are transferred by DMA engines at high throughput.

DMA engines transfer data chunks from the host memory into the device memory, and then, ZeNA performs following executions until it finishes partial convolutions associated with data currently stored in the device memory:

1. Reading the chunks of data from the device memory (i.e., BRAM).
2. Performing partial convolution executions.
3. Writing the partial result back to the device memory.

After ZeNA finishes its executions, DMA engines transfer the result back to the host memory. In our experiment, a single DMA engine supports about 4GB/s in half duplex transactions, four DMA engines support up to about 6GB/s in half duplex and about 8GB/s in full duplex transactions.

ZeNA Implementation

We implemented a single ZeNA cluster on FPGA. On-chip SRAMs of ZeNA (i.e., *Act SRAM* and *Weight SRAM*) are implemented with the device memory (shown in Figure 6.6) of FPGA called block random access memory (BRAM). Local buffers are also implemented with remaining BRAMs (e.g., *Act buffer* and *Weight buffer*) or piles of flip-flops (e.g., *Psum buffer*). Computation units in PE could be implemented with a specialized hardware called the digital signal processor (DSP) in FPGA. However, since the DSP generally supports operations with fixed large bit-width, e.g., 16-bit, merely using it in a low-precision computing is not area-efficient. Thus, we implemented PE computation units using logic fabric and performed sensitivity analysis without utilizing DSP slices.

Neural Network Framework

We extend Caffe [42] to include functionality of managing and running the FPGA device. In order to make Caffe import necessary API functions, the contents of Alpha Data SDK [43] are bundled up into a shared object. We implemented the FPGA resource manager with singleton pattern in the top-level object. Thus, only one manager object interacts with the FPGA device while the object can be used from Caffe layers. Custom Caffe layers utilize input data produced from the previous layer, and generate chunks of data (as shown in Chapter 4.2) for running partial convolutions in FPGA. Partial convolutions are performed iteratively until convolution executions for each layer are completed. In order to run PCIe-based DMA, custom Caffe layers utilize a series of C/C++ API functions given by the Alpha Data. For example, the *read/write* access for the DS port is accomplished by a custom *mmap* function which utilizes a simple pointer as an input. In addition, the *write* of a DMA engine works as a custom *memcpy* function which has three inputs: a source pointer in host virtual address space, a destination address in device address space, and a transfer size in bytes. The *read* of a DMA engine works in a similar way.

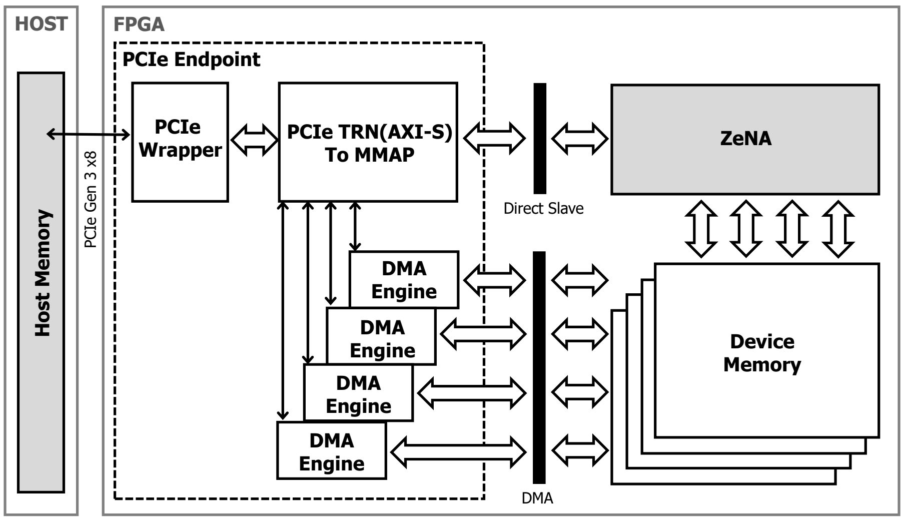


Figure 6.7 PCIe and DMA engines for communication between host and FPGA.

Table 6.2 Hardware configuration of ZeNA

# PEs	256 (8 × 32)	**Bus width**	512 bit	**SRAM**	512 kB
Precision		**16-bit**	**8-bit**	**4-bit**	**2-bit**
Row buffer (per row)	**Act/weight**	12 × 128 b	12 × 64 b	12 × 32 b	12 × 16 b
	Partial sum	4 × 128 b	4 × 64 b	4 × 32 b	4 × 16 b
Local buffer (per PE)	**Act/weight**	12 × 128 b	12 × 64 b	12 × 32 b	12 × 16 b
	Partial sum	2 × 32 b	2 × 16 b	2 × 8 b	2 × 4 b

Pseudocode 1 shows a simplified version of a forward function in the custom Caffe layer. The execution can be summarized as follows:

1. **Crop data and allocate temporal resources.** When starting the execution of each layer, it processes the input data (i.e., *hostData*), and pre-determined configurations for the FPGA (i.e., *fpgaConfig*) including size of device memory, PE array configuration, and etc. *hostData* contains input activations and kernel weights which were delivered from external data (e.g., kernel weight and input activations of first layer) or generated by the previous layer (e.g., input activations excepting first layer). As a result of *CropLayer*, input data are converted into chunks of data which fit on the device memory of FPGA. Then, the application allocates temporal resources such as additional memory used by the DMA (line 5). In order to prevent coherency problem, the system locks the memory region within *DmaBuffer* before running the DMA (line 6). *MapWindow* maps the memory region that can be referenced by the address space of a DS port (line 7).

2. **DMA (host to FPGA).** At this step, a chunk of data for partial convolution is transferred from the host memory to the device memory. The data to be transferred are located in the host memory indicated by *DmaBuffer* pointer (line 12). Then, *WriteDMALockedEx* is executed to transfer the data (line 13).

3. **Run FPGA.** Once data are transferred, the layer uses memory space for storing trigger signal (called *RUNCODE*) to activate the FPGA (line 16). If *RUNCODE* is asserted via a DS port, ZeNA starts to run its routines to execute partial convolutions. After finishing its routine, ZeNA changes the *RUNCODE* to zero to indicate that it finished its routine. In *while* loop, the application detects whether *RUNCODE* is changed or not (line 17).

4. **DMA (FPGA to host).** The application runs *ReadDMALockedEx* to gather the results stored in device memory through the DMA (line 20) and stores partial result in *fpgaData*. Steps 2 to 4 are repeated until all partial convolutions are done.

5. **Deallocate temporal resources and generate the input of the next layer.** After finishing all the partial convolutions, the layer deallocates temporal resources and gathers the partial convolution results to generate output feature maps (i.e., input activations for next layer). Steps 1 to 5 are repeated to compute every layer of a given neural network.

6.2.2 Evaluation and Analysis

Figure 6.8 shows the resource usage of the FPGA in various levels of bit precision (*16-bit*, *8-bit*, *4-bit*, and *2-bit*). We reported the resource usage in terms of flip-flop (FF), look-up table (LUT), and block random access memory (BRAM). Resource usage of each implementation is normalized by that of *16-bit* implementation, and each bar shows a resource breakdown of *controller*, *computation unit*, and *buffer*.

FF usage varies significantly across different functional blocks. In the controllers including intra-cluster controller, row controller and zero-skip controller, the flip-flops are deployed for pipeline stages and control signals. Their resource usage does not change when bit-width gets reduced. Similarly, the size of the bit-vector used by the zero-skip controller is not influenced by the reduced precision. In *2-bit* implementation, the bit-vector requires 50% additional capacity compared to the input data (each element is 2-bit data). In contrast, the number of flip-flops for computation unit and buffers can be reduced in low-precision. FF usage of computation unit in *2-bit* implementation shows 80% reduction compared to *16-bit* implementation. At the same time, FF occupation of controllers increases from 40% in *16-bit* to 76.3% in *2-bit* implementation.

LUT usage gets shrunk in both computation unit and most controllers in low-precision. Note that the zero-skip controller is an exception because its combinational logic remains unchanged. Compared to *16-bit* implementation, *2-bit* implementation shows 90.4% and 39.4% reduction in LUT usage in computation unit and controller, respectively. The amount of reduction in a computation unit is larger than that of a controller. Thus, the LUT occupation of controllers increases from 24.4% up to 67.1%. In addition, BRAM usage is almost linearly proportional to bit-width since most of the buffers (Psum buffer is an exception) are implemented with BRAM.

As bit-width gets reduced, controllers occupy most of the FF and LUT resource usage, thereby preventing us from achieving optimized zero-skipping neural network accelerator in low-precision. In order to further investigate the impact of controller overhead in zero-skip architecture, we employed the ASIC approach because comparing areas among different primitives, LUT, FF and BRAM, in FPGA is limited due to the resource diversity and implementation characteristics (e.g., LUT for logic gates). Thus, we used Synopsys Design Compiler under 65nm library to synthesize the RTL design into ASIC design and used CACTI v6.0 [39] for SRAM cost.

Figure 6.9 shows the area of PE array in various bit-widths. The area of each implementation is normalized by the *16-bit* implementation. The figure shows that *2-bit* implementation achieves 84.6% area reduction (90.4% in computation unit, 87.4% in buffer and 39.4% in controller) compared to *16-bit* implementation. Again, the reduction ratio of the controllers is the smallest, and thus, its occupancy in total area increases from 7.4% to 29% while the occupancy of the computation unit decreases from 22.8% to 14.2%. Note that the relative size between controllers and the computation unit is reversed (size of controller is two times larger than computation unit) in *2-bit* implementation.

When fine-grained zero-skipping and very low-precision are both applied, the area cost of a controller becomes comparable to or larger than that of a computation unit. In order to mitigate this problem of increasing control overhead in low-precision, the zero-skipping operation could be applied in a more coarse-grained manner. However, the coarse-grained zero-skipping can hurt the benefits of zero-skipping by reducing the chance of exploiting zero values.

Analyzing performance between the hardware implemented with different reduced precision under various networks which have different zero-ratio can be complicated when software and FPGA communication overhead is involved. Moreover, the neural network, which heavily applies reduced precision methods (i.e., 4-bit or less) and various levels of sparsity, is not available yet. Thus, in order to focus on on-chip architecture and analyze performance trend of zero-skipping architecture under various bit-widths and sparsity levels, we implemented an in-house cycle-accurate model and estimated performance. We generated synthesized layers which have the same configuration with five convolutional layers from AlexNet [1] (e.g., conv1 to conv5). Then, we randomly generated activations and weights based on the zero data ratio in each case.

Figs. 6.10 and 6.11 show the speedup of *2-bit* and *16-bit* implementation, respectively, under the various sparsity settings, compared to the performance of the zero-agnostic accelerator. We varied zero ratio of weight from 0.4 to 0.8 corresponding to *WZ(0.4)* to *WZ(0.8)* in Figs. 6.10 and 6.11, and then, in each case, we varied zero ratio of activation from 0.1 to 1 (x-axis).

Figure 6.10 shows speedup of *16-bit* implementation. Even though the zero ratio of activation is changed, the speedup of *WZ(0.8)* is saturated at **4.5x**. When the activation zero ratio is 0.1, it shows higher speedup as weight zero ratio goes higher. However, after the activation zero ratio passes 0.7, speedup is saturated at **4.5x**. In other words, the zero-skipping architecture can not fully exploit the advantage of zero data if the network has very high sparsity. This is due to the communication contention between on-chip memory and PE array. PE array can process more data as it finishes its computation earlier, however, communication bandwidth (e.g., intra-cluster bus bandwidth of ZeNA) is limited. Thus, PE array gets stall waiting for new input data, which prevents further speedup.

Figure 6.11 shows the speedup of *2-bit* implementation. Compared with *16-bit* implementation, speedup of *WZ(0.8)* is not saturated regardless of the activation zero ratio. The speedup of every case is saturated at **18.7x** when the activation zero ratio is 1 (speedup of *WZ(0.8)* is stuck when activation zero ratio is 0.9). Until this point, the speedup continues to increase as either of the two zero ratios increases. Since we utilized the same bus width in every bit-width, the communication bandwidth of *2-bit* implementation is effectively 8 times higher than that of *16-bit* implementation. Thus, the effectively larger bus width mitigates the speedup problem of *16-bit* implementation due to communication contention.

Figure 6.12 shows operations-per-second (OPS)/area of each implementation while running convolutional layers of prunned AlexNet [30]. Each PE of ZeNA should contain its own controller to enable fine-grained zero-skipping. The size of a controller does not decrease linearly as the bit-width of input data is reduced. Thus, *2-bit* implementation shows **6.7x** higher OPS/area compared to that of *16-bit* implementation, not $\approx$**8x**.

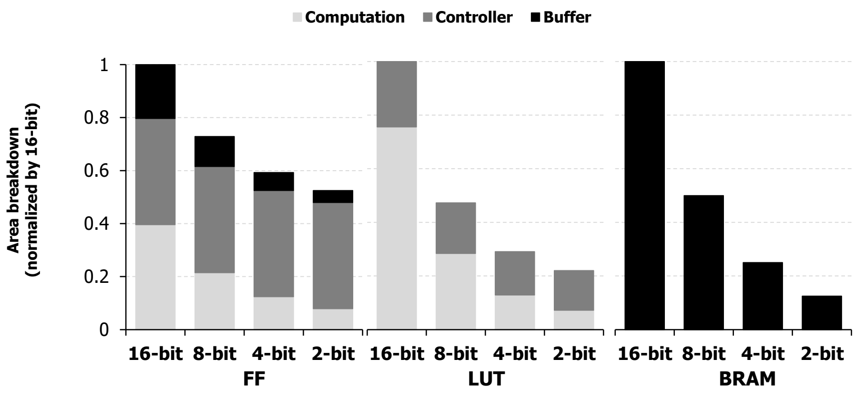


Figure 6.8 FPGA resource usage of PE array in different bit-widths. FPGA resource usage is divided into flip-flop (FF), look up table (LUT) and block SRAM (BRAM).

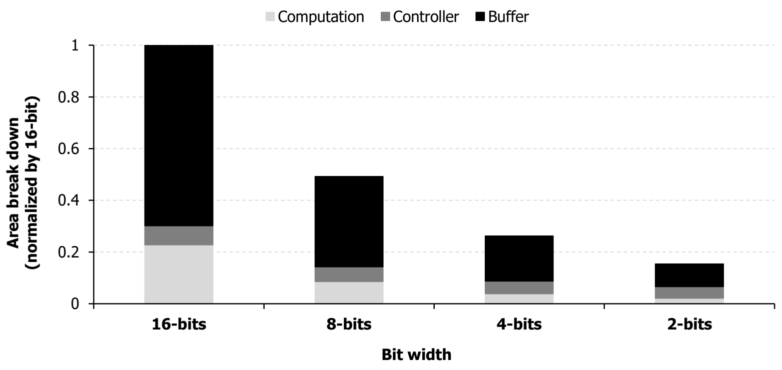


Figure 6.9 Area break down of PE array in different bit-widths.

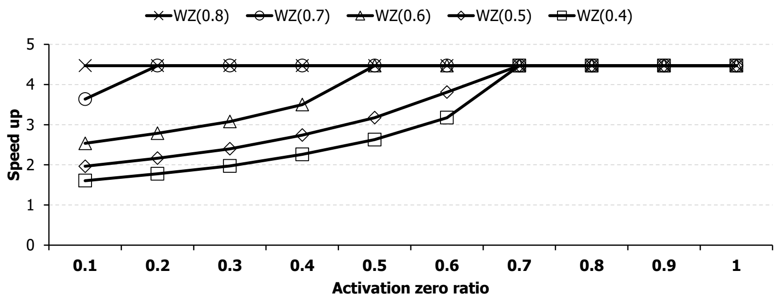


Figure 6.10 Speedup of *16-bit* implementation in AlexNet with various sparsity where weight sparsity range is 0.4 to 0.8 and activation sparsity range is 0.1 to 1.

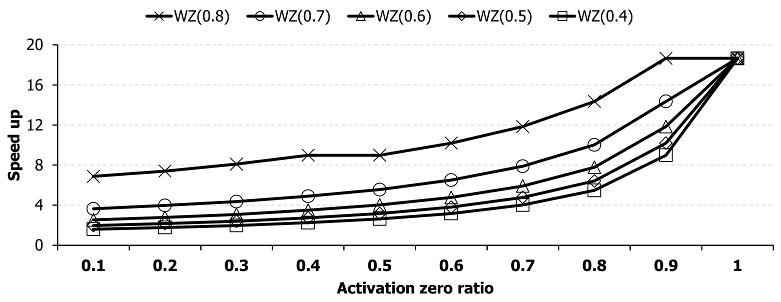


Figure 6.11 Speedup of *2-bit* implementation in AlexNet with various sparsity where weight sparsity range is 0.4 to 0.8 and activation sparsity range is 0.1 to 1.

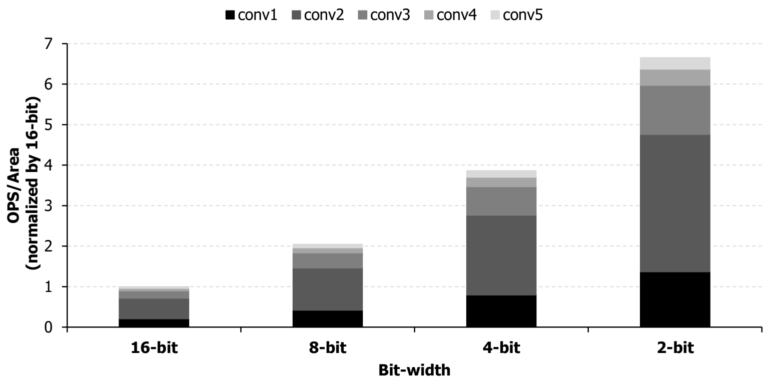


Figure 6.12 OPS/area in AlexNet with various bit-width (2-bit to 8-bit).

Pseudocode 1 Forward function in the custom Caffe layer

```
// Crop data
CropLayer(fpgaData, hostData, layerId);

// Allocate temporal resources
void* DmaBuffer = malloc(DMASIZE);
Lock(DmaBuffer, DMASIZE);
MapWindow(FpgaControl, MAPSIZE);

// Main for loop
for (i = 0; i < fpgaData.size(); i++) {
    // DMA (host to FPGA)
    memcpy(DmaBuffer, fpgaData[i], DMASIZE);
    WriteDMALockedEx(DMASIZE, DEVICEADDR);

    // Run FPGA
    *FpgaControl = RUNCODE;
    while (*FpgaControl == RUNCODE);

    // DMA (FPGA to host)
    ReadDMALockedEx(DMASIZE, DEVICEADDR);
    memcpy(hostData[i], DmaBuffer, DMASIZE);
}

// Deallocate temporl resources
UnmapWindow(FpgaControl);
Unlock(DmaBuffer);
free(DmaBuffer);

// Generate input to next layer
GenNextLayer(fpgaData, hostData, layerId);
```

6.3 Comparison between Fine-grained Zero-aware Hardware Accelerators

Parashar *et. al* [8] studied activation/weight zero-aware hardware accelerator for convolutional neural network (called SCNN) in parallel with proposed ZeNA [2,4]. We will compare two different fine-grained zero-aware hardware accelerators in this section.

6.3.1 SCNN

SCNN employs a dataflow that enables maintaining the sparse weights and activations in a compressed encoding, thereby reducing unnecessary data transfers and storage requirements. The key component for zero-skipping on SCNN is a processing element (PE) with multiplier array. In order to accelerate CNN skipping multiplication associated with either zero activation or weight, the SCNN only delivers non-zero weights and activations to the multiplier array which performs the *Cartesian product*. Moreover, SCNN adopts *input stationary* dataflow which holds an input activation stationary at the computation units and multiplies with all the kernel weights.

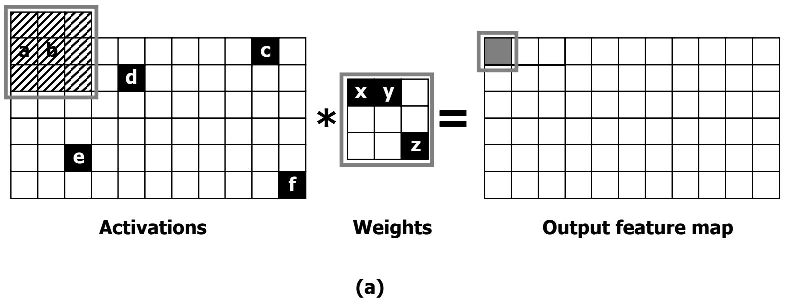
SCNN should accumulate the partial products generated by multiplier array at last due to perform convolution with small chunk of data. However, since SCNN performs Cartesian products between only non-zero activations and weights, it should track the output coordinate associated with each multiplication result and accumulate it on a proper buffer.

Figure 6.13 illustrates a simplified operation diagram of SCNN which only convolves between non-zero activations and weigths (colored in black with alphabats) with stride one and results in the output feature map. Since non-zero intersections between activations and weights is roughly 20-50% per layer, sliding window based convolution may be wasteful due to useless multiplications (patterned region in Figure 6.13a).

The basic idea of SCNN is that all non-zero activations (a, b, c, d, e and f in Figure 6.13) should be multiplied by all non-zero weights (x, y and z in Figure 6.13) at some point in time. SCNN applies the Cartesian product (i.e., all-to-all) based convolution as shown in Figure 6.13b. Multiplier array in PE (*PE frontend* in Figure 6.13b) performs the Cartesian product between non-zero activations and non-zero weights. The output coordinate of each multiplication is traced and utilized to accumulate it on buffers via a scatter network (PE backend in Figure 6.13b).

6.3.2 Evaluation Analysis

We evaluated **throughput/area** to compare two different fine-grained zero-aware hardware accelerators: SCNN [8] and proposed ZeNA V2 [4]. Since the original ZeNA contains a smaller number of multipliers (e.g., 165) compared to SCNN (e.g., 1024), to focus on comparing architecture itself, we implemented two flavors of ZeNA architecture by applying different number of multipliers as follows:



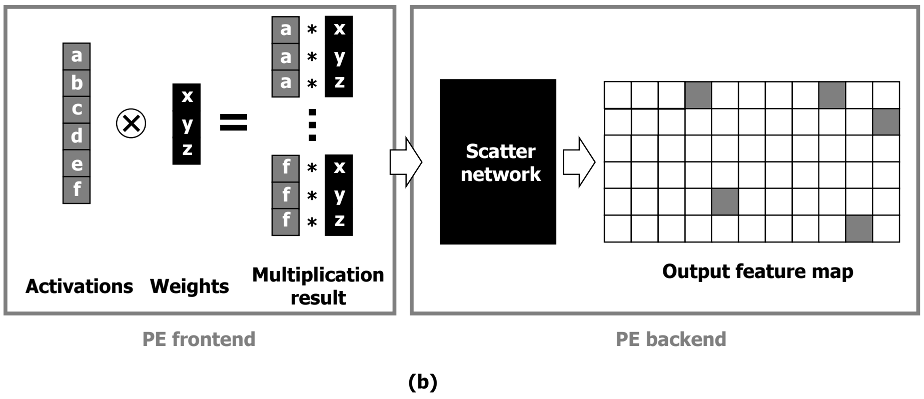


Figure 6.13 Simplified operation diagram of SCNN [8].

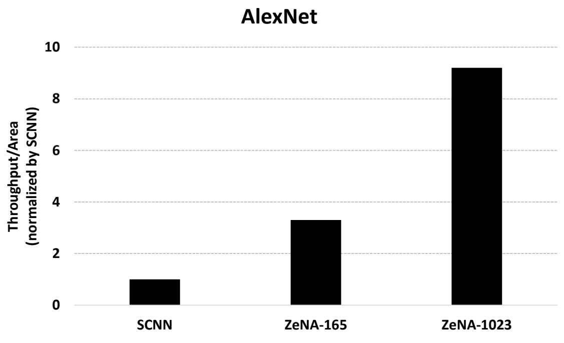
- **ZeNA-165**. Original ZeNA V2. PE array includes 165 (11×15) PEs.
- **ZeNA-1023**. ZeNA V2 where PE array includes 1023 (11×93) PEs.

Note that PE in ZeNA contains only one multiplier each.

For the performance evaluation, we implemented an in-house behavior model of SCNN and our architecture. We assumed both architectures operate on the same clock frequency and estimated throughput. Since SCNN and ZeNA implemented with different standard cell library, we scaled area for comparison [44]. We used the pruned model of AlexNet in [30] and a pruned version of VGG-16, which was obtained by thresholding kernel weights to meet the reported ratio of zero weights in [30]. To run AlexNet and VGG-16 on the *ZeNA-165*, we allocated PEs to WGs as shown in Table 5.2. Note that *ZeNA-1023* allocates **6x** PEs shown in Table 5.2 to WGs.

Figure 6.14 shows throughput/area comparison between SCNN and ZeNA. In AlexNet, *ZeNA-165* and *ZeNA-1023* achieves **3.3x** and **9.2x** improvement in throughput/area compared to SCNN. However, in VGG-16, *ZeNA-165* shows **0.2x** throughput/area compared to SCNN. This is because ZeNA utilizes sufficient on-chip SRAM to store current and next WG, thereby simplifying memory control and optimizing energy consumption. Since *ZeNA-165* has on-chip SRAM which occupies a large portion of area (85.7% and 98.6% in AlexNet and VGG-16, respectively), computing power (i.e., the number of multipliers) within the same area is degraded. Adopting more PEs while maintaining other hardware configurations improves throughput/area. On-chip SRAM area of *ZeNA-1023* occupies smaller portion of area compared to *ZeNA-165* (49.5% and 92% in AlexNet and VGG-16, respectively). As a result, *ZeNA-1023* achieves **9.2x** and **1.2x** throughput/area on AlexNet and VGG, respectively, compared to SCNN.

Recently, CNNs which adopt depth-wise convolutional layers are widely used for vision tasks due to exceptional accuracy. Since the most on-chip memory area of ZeNA is utilized to store partial sums and depth-wise convolutional layers can be computed without storing partial sums, ZeNA requires smaller on-chip memory to execute CNN consisting of depth-wise convolutional layers. Thus, we expect that ZeNA shows better throughput/area compared to SCNN in CNNs consisting of depth-wise convolutional layers.



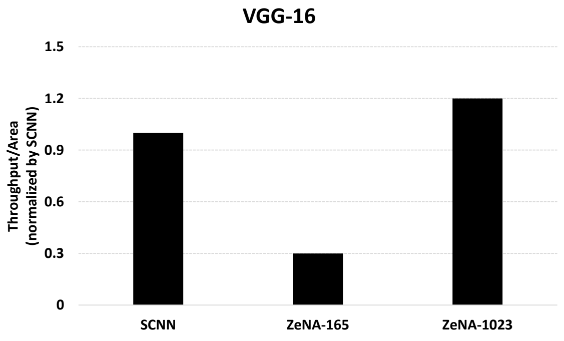


Figure 6.14 Throughput/area comparison between SCNN and ZeNA.

Chapter 7

Conculsion

In this dissertation, we propose a novel Zero-aware Neural Network Accelerator for CNNs named *ZeNA*. Unlike previous studies utilizing sparsity, we exploit both zero activations and weights to reduce runtime and energy consumption of convolution. We propose two versions of ZeNA architecture: *ZeNA V1* and *ZeNA V2*. In addition, we introduce two different types of zero-induced load imbalance problems, *intra-/inter-WG load imbalance*, which prevent us from achieving full opportunity of zero-aware parallel hardware architectures for CNN.

ZeNA V1 applies *zero-aware kernel allocation* as the solution for mitigating intra-WG load imbalance. We first sorted all the sets of kernel tiles in the increasing order of the number of non-zero weights in the sets, and then, allocated sets of kernel tiles to sub-WG in the sorted order. Applying zero-aware kernel allocation, we uniformly distributed kernel tiles thereby improving runtime and energy efficiency. According to our experiments, ZeNA V1 achieves 4x and 5.2x speedup while achieving 11.3% and 18% energy reduction in AlexNet and VGG-16, respectively, compared to baseline (Eyeriss).

We propose further optimized ZeNA architecture, ZeNA V2 which adopts multi-cluster architecture. ZeNA V2 applies *dynamic WG allocation*, which is conceptually similar to work-stealing queues, as the solution for mitigating inter-WG load imbalance. In addition, to achieve further energy reduction on on-chip SRAM and local buffer, we applied *on-the-fly bit-vector generation* and *clock gating*. According to our experiments, ZeNA V2 achieves 4.4x/5.6x speedup and reduces energy consumption by 25.4%/25.2% in AlexNet/VGG-16 over the baseline (Eyeriss).

We also perform quantitative analysis for zero-aware hardware architecture. We analyzed the impact of data compression on ZeNA and compared various ZeNA architectures, which have distinct configurations of bit-width and and zero data ratio, to verify the impact of reduced precision on zero-aware hardware architecture. According to our analysis, zero-aware architecture which does not apply output stationary data flow might be unsuitable for storing data on on-chip SRAM in compressed format. We also report overhead of fine-grained zero-aware hardware architecture such as ZeNA after applying very-low precision.

Recently, neural network specialized hardware accelerators have been applied on products such as cell phones. Qualcomm (e.g., Snapdragon 855) and Apple (e.g., A11) provide a neural processing unit (NPU) and its APIs. Huawei (e.g., Kirin 970) provides a NPU and an API which are restricted to specific applications. Samsung Galaxy S10 adopts a NPU which is utilized for internal functions.

We expect that competition in the NPU market is going to be more intense in the coming years. In this era, we expect two types of NPU approaches will exist. The first is a NPU for general purpose. Utilizing APIs, application developers can exploit the NPU for accelerating their own tasks. In order to achieve this goal, the NPU for general purpose should concentrate on supporting and accelerating various neural networks. Moreover, these NPUs should support good programability to be utilized. The other way is utilizing the NPU for accelerating specific task which cannot be controlled by users (i.e., internal functions). Those NPUs can be utilized for accelerating specific tasks such as face recognition (for unlocking the phone) and enhancing camera resolution using deep learning. For this purpose, the NPU should be more optimized for target neural network in manner of runtime and power consumption. Since various types of neural network specialized hardware accelerators will be required in the competitive era of NPUs, more researches and analysis for various NPU architectures are necessary.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[2] D. Kim, J. Ahn, and S. Yoo, “A novel zero weight/activation-aware hardware architecture of convolutional neural network,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 1462–1467, IEEE, 2017.

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

[4] D. Kim, J. Ahn, and S. Yoo, “Zena: Zero-aware neural network accelerator,” *IEEE Design & Test*, vol. 35, no. 1, pp. 39–46, 2018.

[5] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-neuron-free deep neural net-

work computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[6] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 20, IEEE Press, 2016.

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

[8] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 27–40, IEEE, 2017.

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

[10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[11] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, “Overfeat: Integrated recognition, localization and detection using convolutional networks,” *arXiv preprint arXiv:1312.6229*, 2013.

[12] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580–587, 2014.

[13] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Advances in neural information processing systems*, pp. 91–99, 2015.

[14] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[15] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated residual transformations for deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1492–1500, 2017.

[16] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[17] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[18] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, “Character-aware neural language models,” in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

[20] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, “Recurrent neural network based language model,” in *Eleventh annual conference of the international speech communication association*, 2010.

[21] T. Sainath and C. Parada, “Convolutional neural networks for small-footprint keyword spotting,” 2015.

[22] “Oculus vr.” `https://www.oculus.com`. Accessed: 2019-03-18.

[23] “Hololens ar.” `https://www.microsoft.com/en-us/hololens`. Accessed: 2019-03-18.

[24] “Tesla self driving.” https://www.tesla.com. Accessed: 2019-03-18.

[25] J. Cong and B. Xiao, “Minimizing computation in convolutional neural networks,” in *International conference on artificial neural networks*, pp. 281–290, Springer, 2014.

[26] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *ACM Sigplan Notices*, vol. 49, pp. 269–284, ACM, 2014.

[27] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, *et al.*, “Dadiannao: A machine-learning supercomputer,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[28] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “Shidiannao: Shifting vision processing closer to the sensor,” in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[29] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: efficient inference engine on compressed deep neural

network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, IEEE, 2016.

[30] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *ICLR*, 2016.

[31] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[32] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," *arXiv preprint arXiv:1612.01064*, 2016.

[33] S. Migacz, "Nvidia 8-bit inference with tensorrt," *GPU TechnologyConference*, 2017.

[34] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[35] J. Van Leeuwen, "On the construction of huffman trees.," in *ICALP*, pp. 382–410, 1976.

[36] M. Gautschi, M. Schaffner, F. K. Gürkaynak, and L. Benini, "4.6 a 65nm cmos 6.4-to-29.2 pj/flop@ 0.8 v shared logarithmic floating

point unit for acceleration of nonlinear function kernels in a tightly coupled processor cluster," in *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 82–83, IEEE, 2016.

[37] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ACM SIGARCH Computer Architecture News*, vol. 40, pp. 356–367, IEEE Computer Society, 2012.

[38] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE Computer Society, 2012.

[39] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories, HPL-2009-85*, 2009.

[40] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," *ACM Sigplan Notices*, vol. 33, no. 5, pp. 212–223, 1998.

[41] "[12] micron dram calculator." http://www.micron.com/support/tools-and-utilities/power-calc/. Accessed: 2019-04-14.

[42] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *ICME*, 2014.

[43] “Reference design – rd-ku3.” https://www.alpha-data.com/dcp/rd.php?product=rd-ku3. Accessed: 2018-06-16.

[44] A. Stillmaker, Z. Xiao, and B. Baas, “Toward more accurate scaling estimates of cmos circuits from 180 nm to 22 nm,” *VLSI Computation Lab, ECE Department, University of California, Davis, Tech. Rep. ECE-VCL-2011-4*, vol. 4, 2011.

국문초록

뉴럴네트워크는 다양한 어플리케이션에서 널리 쓰이고 있다. 특히, Convolutional Neural Network (CNN)은 영상 인식 및 분류 시스템에서 괄목할만한 정확도를 보이고 있다. 비록 CNN이 다양한 어플리케이션에서 높은 정확도를 보이고 있지만, 이러한 정확도를 얻기 위해서는 많은 convolutional layer가 필수적이다. 하지만 이러한 network는 많은 계산량을 요구하기 때문에 제한된 하드웨어 자원을 가지고 있는 VR혹은 자율주행 자동차 등의 물체 인식 및 감지를 위한 실시간 임베디드 시스템에 CNN을 적용하는 것은 기술적으로 넘어야 할 장벽이 존재한다. 그렇기 때문에 물체 인식 및 감지를 위한 실시간 임베디드 시스템에 CNN을 적용하기 위해 convolutional layer 최적화는 매우 중요하다.

일반적으로 convolutional layer는 하나의 결과값을 계산하기 위해 3차원 데이터를 입력으로 받아 수천개의 곱셈 및 덧셈을 수행하고, 3차원 출력을 생성하기 위해 이러한 과정을 반복한다. 따라서 convolutional layer를 수행하는데는 매우 많은 양의 연산이 필요하며, 곱셈 및 덧셈의 양을 줄여 빠른 convolution을 구현함으로써 물체 인식 및 감지를 위한 실시간 임베디드 시스템에 CNN을 적용하는것이 가능 할 수 있다. 뉴럴 네트워크 가속을 위해 최근 몇년간 전용 하드웨어 가속기에 대한 연구가 활발히 진행되고 있다. 제안된 뉴럴 네트워크 전용 하드웨어 가속기들은 주로 뉴럴 네트워크에 제로 데이터가 많다는 특성 (spar-

sity)과 양자화 적용 후에도 정확도 손실이 적다는 특성 (low-precision) 을 활용하여 뉴럴네트워크를 최적화한다.

CNN의 입력 activation및 weight에 많은 제로 데이터가 있다는 사실을 활용하여 우리는 제로 데이터를 활용한 뉴럴네트워크 전용 하드웨어 가속기를 제안한다. 기존 뉴럴네트워크 전용 하드웨어 가속기와 달리 우리는 activation 및 weight의 제로 데이터를 모두 활용하여 가속기의 수행시간 및 에너지 소모량을 감축한다. 또한 제로 데이터를 활용한 전용 하드웨어 가속기의 병렬성을 증가시킬 때 발생하는 새로운 문제인 zero-induced load imbalance를 소개하고 이를 완화하기 위한 방안을 제시한다. 본 논문은 제로 데이터를 활용한 아키텍처에서 추가적인 에너지 소모 감소를 위한 다양한 최적화 기법 또한 제안한다.

우리는 뉴럴네트워크가 더 복잡한 어플리케이션을 처리하고 높은 정확도를 얻기 위해 깊어지고 있는 현상에 주목한다. 뉴럴 네트워크는 향후 layer수가 증가할수록 계산량이 기하급수적으로 증가하여 최적화 없이 하드웨어 자원이 제한된 실시간 임베디드 시스템에 적용하기 어려워질 것으로 예상된다. 따라서 우리는 미래 뉴럴 네트워크 전용 하드웨어 가속기가 sparsity와 low-precision 모두 적용하는 등 극도의 최적화를 진행해야 제한된 하드웨어 자원 상에서 원하는 성능을 달성할 수 있을 것으로 예상한다. 이러한 최적화 방법을 현존하는 뉴럴 네트워크 가속기 구조에 그대로 적용할 경우 기대 성능을 얻을 수 없도록 하는 새로운 문제가 발생할 수 있지만 아직 이에 대한 분석이 부족하다.

본 논문에서 우리는 제로 데이터를 활용한 뉴럴네트워크 전용 하드웨어 가속기 (ZeNA)를 제안한다. ZeNA는 CNN의 activtion과 weight

모두의 제로 데이터를 활용하여 뉴럴 네트워크의 수행 시간은 물론 소모 전력 또한 감축한다. 제안된 전용 하드웨어 가속기는 제로 데이터를 전혀 활용하지 못하거나, activation과 weight의 제로 데이터 중 한 종류의 제로 데이터만 활용하는 기존의 뉴럴 네트워크 전용 하드웨어 가속기 대비 더 높은 성능과 낮은 전력 소모량을 보인다. 또한 우리는 zero-induced load imbalance의 해결법, 메모리 전력 최적화 기법 등 제로 데이터를 활용한 전용 하드웨어 가속기를 위한 다양한 최적화 기법 및 sparsity와 low-precison을 모두 활용하는 미래의 뉴럴 네트워크 전용 가속기에 대한 분석 또한 제시한다.

주요어: 뉴럴네트워크, 뉴럴네트워크 최적화, 컨볼루션 뉴럴네트워크, 양자화, 하드웨어 가속기 구조

학번: 2015-31048