creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. Dissertation

Deep Neural Network Training Accelerator Architecture Design: Acceleration of Backward Propagation using Sparsity of Neurons

심층신경망 학습 가속기 구조 설계: 뉴런의 성김을 이용한 역전파 가속

August 2020

Department of Electrical and Computer Engineering
College of Engineering
Seoul National University

Gunhee Lee

Ph.D. Dissertation

Deep Neural Network Training Accelerator Architecture Design: Acceleration of Backward Propagation using Sparsity of Neurons

심층신경망 학습 가속기 구조 설계: 뉴런의 성김을 이용한 역전파 가속

August 2020

Department of Electrical and Computer Engineering
College of Engineering
Seoul National University

Gunhee Lee

Deep Neural Network Training Accelerator Architecture Design: Acceleration of Backward Propagation using Sparsity of Neurons

심층신경망 학습 가속기 구조 설계: 뉴런의 성김을 이용한 역전파 가속

지도교수 이 혁 재
이 논문을 공학박사 학위논문으로 제출함

2020년 7월

서울대학교 대학원

전기 정보 공학부

이 건 희

이건희의 공학박사 학위 논문을 인준함

2020년 7월

위 원 장: 김 태 환 (인)
부위원장: 이 혁 재 (인)
위 원: 유 승 주 (인)
위 원: 류 수 정 (인)
위 원: 이 진 호 (인)

Abstract

Deep neural network has become one of the most important technologies in the various fields in computer science which tried to follow the human sense. In some fields, their performance defeats that of human sense with the help of the deep neural network. Since the fact that general purpose GPU can speed up deep neural network, GPU became the main device used for deep neural network. As the complexity of deep neural network becomes high that deep neural network requires more and more computing resources. However, general-purpose GPU consumes a lot of energy that the needs of specific hardware for deep neural network are rising. And nowadays, the specific hardwares are focusing on inference. With complicated network models, training a model consumes enormous time and energy using conventional devices. So there are increasing needs specific hardwares for DNN training.

The dissertation exploits deep neural network training accelerator architecture. The training process of a deep neural network (DNN) consists of three phases: forward propagation, backward propagation, and weight update. Among these, backward propagation for calculating gradients of activations is the most time consuming phase. The dissertation proposes hardware architectures to accelerate DNN training, focusing on the backward propagation phase. The dissertation makes use of the sparsity of the neurons incurred by ReLU layer or dropout layer to accelerate the backward propagation.

The first part of the dissertation proposes a hardware architecture to accelerate DNN backward propagation for convolutional layer. We assume using rectified linear unit (ReLU), which is the most widely used activation function. Since the output as well as the derivative of ReLU is zero for negative inputs, the gradient for activation is also zero for negative values. Thus, it is not needed to calculate the gradient of input activation if the input activation value is zero. Based on this observation, we design

an efficient DNN accelerating hardware that skips the gradient computations for zero activations. We show the effectiveness of the approach through experiments with our accelerator design.

The second part of the dissertation proposes a hardware architecture for fully connected layer. Similar to ReLU layer, dropoout layer has explicit zero gradient for the dropped activation without gradient computation. Dropout is one of the regularization techniques which can solve the overfitting problem. During the DNN training, the dropout disconnect connections between neurons. Since the error does not propagated through the disconnected connections, we can detect zero gradient becomre computation. Making use of this characteristics, the dissertation proposes a hardware which can accelerate the backward propagation of fully connected layer. Further, the dissertation showed the effectiveness of the approach through simulation.

keywords: Deep Neural Network Training, Sparsity of Neurons, Selective Gradient Computation

student number: 2014-21649

Contents

Abstract **i**

Contents **iii**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**

1.1 Deep Neural Network Training 4

1.2 Convolutional Neural Network 5

1.2.1 Forward propagation 5

1.2.2 Backward propagation 6

1.2.3 Weight update 6

1.3 Rectified Linear Unit 7

1.4 Dropout 8

1.5 Previous Works 9

2 Acceleration of DNN Backward Propagation on CNN layer **12**

2.1 Motivation 12

2.2 Selective Gradient Computation for Zero Activations 17

2.2.1 Baseline Architecture 17

2.2.2 Bit-Vector for Selective Gradient Computation 20

2.2.3 Filter Collector . . . 23
2.2.4 Zero-Gradient Insertion in Write DMA . . . 26
2.3 Overall Architecture . . . 27
2.4 SRAM Buffer . . . 28
2.4.1 Motivation . . . 28
2.4.2 Selective Gradient Computation and SRAM Buffer . . . 29
2.5 Experimental Results . . . 32
2.5.1 Performance Simulator . . . 32
2.5.2 RTL Implementation . . . 33
2.5.3 Performance Improvement . . . 35
2.5.4 Energy Reduction . . . 37
2.5.5 Impacts of SRAM Buffer . . . 39
2.6 Summary . . . 45

3 Acceleration of DNN Backward Propagation on Fully Connected Layer 46
3.1 Motivation . . . 46
3.1.1 Dropout . . . 46
3.1.2 Conventional Dropout Layer Implementations . . . 46
3.1.3 Applications of Dropout . . . 47
3.2 Selective Gradient Computation for Dropped Activations . . . 50
3.2.1 Baseline Architecture . . . 50
3.2.2 Filter Dropper . . . 50
3.3 Overall Architecture . . . 54
3.4 Experimental Results . . . 55
3.4.1 Simulator and Benchmark . . . 55
3.4.2 Results . . . 55
3.5 Summary . . . 56

4 Conclusion 57

Abstract (In Korean) **63**

List of Tables

2.1 The Number of DRAM Accesses for Selective Gradient Computation Comparision on AlexNet 20

2.2 The Number of DRAM Accesses for Selective Gradient Computation Comparision on VGG-16 21

2.3 The Layer of AlexNet 33

2.4 The Layer of VGG-16 34

2.5 Synthesis results of the baseline and the proposed architecture 34

3.1 Simulation results for dropout accelerator architecture 56

List of Figures

1.1 Input-side activation gradient (left) and output-side activation Gradient & Rearranged filters (right) . . . 3
1.2 Ratio of zero output activations for ReLU layers of VGG-16 . . . 3
1.3 Filter rearrangement . . . 7
1.4 ReLU activation function . . . 8

2.1 Sparsity of AlexNet Layers. . . . 14
2.2 Sparsity of VGG-16 layers . . . 15
2.3 Varying sparsity of AlexNet during the training . . . 16
2.4 Varying sparsity of VGG-16 during the training . . . 16
2.5 Selective gradient computation . . . 17
2.6 Overall architecture of *DianNao* . . . 18
2.7 The bit-vector line . . . 22
2.8 Filter collecting scheme . . . 23
2.9 Filter collector flow chart . . . 25
2.10 Zero gradient insertion . . . 26
2.11 Overall architecture and example operation scenario for the proposed architecture. . . . 27
2.12 Data reuse example . . . 29
2.13 Neighboring bit-vector and their selected filters . . . 30
2.14 Dimention of bit-vector line . . . 31

2.15 Filter selecting scheme for *Filter Collector* 31
2.16 Performance improvement on AlexNet 36
2.17 Performance improvement on VGG-16 36
2.18 DRAM energy reduction on AlexNet 38
2.19 DRAM energy reduction on VGG-16 38
2.20 Comparison of the execution cycles for DNN training accelerator on AlexNet without SRAM buffers and with SRAM buffers 40
2.21 Comparison of the DRAM energy for DNN training accelerator on AlexNet without SRAM buffers and with SRAM buffers 40
2.22 Comparison of the execution cycles for DNN training accelerator on VGG-16 without SRAM buffers and with SRAM buffers 41
2.23 Comparison of the DRAM Energy for DNN training accelerator on VGG-16 without SRAM buffers and with SRAM buffers 41
2.24 Performance improvement on AlexNet with SRAM Buffer 43
2.25 Performance improvement on VGG-16 with SRAM Buffer 43
2.26 DRAM energy reduction on AlexNet with SRAM Buffer 44
2.27 DRAM energy reduction on VGG-16 with SRAM Buffer 44

3.1 The recurrent neural network 48
3.2 The unfoleded recurrent neural network 49
3.3 The backward propagation through time for the unfoleded recurrent neural network 49
3.4 *Filter dropper* architecture 50
3.5 Random number genetator generates the same mask for both forward and backward propagation 51
3.6 Overall architecture for the dropout accelerator 55

Chapter 1

Introduction

Deep neural networks (DNNs) are taking an important role in many fields including computer vision [1, 2], speech recognition [3], and natural language processing [4], where artificial intelligence has been considered hard to defeat human senses. As the applications of DNNs are growing, their capacity and scale are also growing. In order to make DNNs work properly, it is typically required to train them with a big dataset to fit the networks to specific applications. Such a training takes days if not weeks or even longer. Therefore, there are high demands for accelerating the DNN training process through the development of new algorithms and/or specialized accelerator architectures. However, there are not many accelerators for DNN training yet, and most of the existing DNN accelerator architectures focus on inference. *DianNao* [5] is one of the representative DNN accelerator architecture. It implements multiply-accumulate (MAC) operations and activation functions in a pipelined manner. Its operation mapping scheme enables highly parallel computations, resulting in large speedup compared to conventional x86 core with SIMD unit. We set this work as the baseline of our idea. Its successor, *DaDianNao* [6] is a multi-node version of *DianNao* that employs eDRAM. It shows significantly high performance and energy efficiency than GPUs for both DNN training and inference.

Skipping unnecessary operations is one of the most effective ways of reducing

the amount of computation. A representative example of unnecessary operations is multiplication with zero, which it typically abundant in DNNs because of many zero-valued activations and/or weights. DNN accelerator architectures that exploits this zero-skipping scheme have been already proposed in previous researches [7, 8, 9, 10]. By skipping multiplications and following accumulations when either of the multiplicand is zero, they can achieve a significant speedup. However, they are mostly focusing only on forward propagations for inference.

The dissertation proposes a DNN accelerator architecture that makes use of zero activation values to skip unnecessary computations in training. We focus on backward propagation (BP) among the three phases of DNN training: forward propagation (FP), backward propagation (BP), and weight update (WU). Our approach exploits the backward propagation characteristics of ReLU layer, which makes backpropagated gradient zero when the corresponding neuron has zero output activation in FP. That means that all the operations for calculating that neuron's activation gradient (as a function of the gradient values that have been backpropagated through the next layer[1]) can be skipped.

Approach of the dissertation is similar with the previous approaches in the sense that zero activations lead to skipping unnecessary operations. However, their ways of exploiting the zero activations are very different. The previous approaches rely on the arithmetic property of zeros in the multiplication operations for inference, whereas our approach relies on the backward propagation property of the ReLU layer. Our approach makes better use of zero activations than the previous ones, because in our approach, a zero activation leads to skipping many operations for calculating the corresponding gradient rather than only a single MAC operation. Note also that our approach can be independently applied with the previous zero-skipping schemes.

[1] By next layer, we mean the layer that takes the output of the current layer in forward propagation

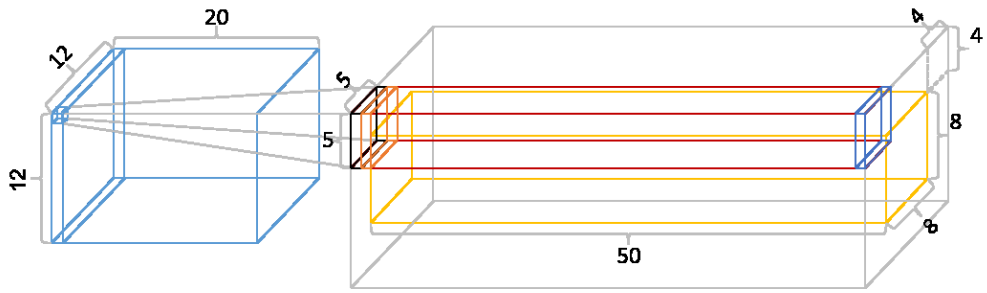

Figure 1.1: Input-side activation gradient (left) and output-side activation Gradient & Rearranged filters (right)

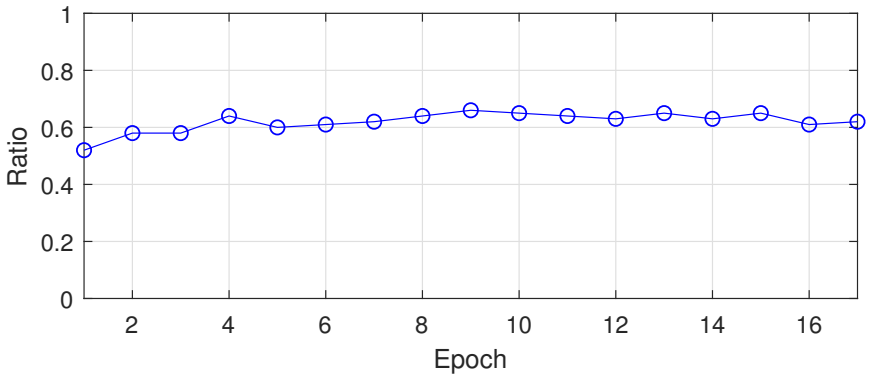

Figure 1.2: Ratio of zero output activations for ReLU layers of VGG-16

1.1 Deep Neural Network Training

In general, training of a DNN for a given task and a training dataset is adjusting the network parameters to maximize the accuracy of the given task. For a classification task, for example, the input from the dataset is propagated through the layers of the network. At the end of the network, the classification result is generated. Comparing the result with the ground truth (label of the input in the case of supervised learning), the error between the classification result and the ground truth is propagated backward to adjust the network parameters (mainly synaptic weights) in the direction of error reduction.

The DNN training process consists of three key phases:

- Forward propagation (FP): In this phase, the input is applied to the network's first layer. From the input, the layer generates the output activation. The output activation becomes an input activation for the next layer. This step is repeated until the propagation reaches the last layer of the network where the final classification output of the network is generated.

- Backward propagation (BP): The classification output generated by the FP phase is compared with the ground truth to calculate the error (or the difference). The error is propagated backward to the previous layer in the form of gradient. The gradient value represents the direction and slope toward the ground truth.

- Weight update (WU): To reduce the error, the weights should be adjusted in the direction to move the output toward the ground truth. To adjust a weight, the gradient of the weight is generated from the gradient of activation so that adjusting the weight in the direction of the weight gradient moves the activation in the direction of the activation gradient. And then the weight is updated with the gradient of the weight.

However, executing the above three phases only for one input may incur overfitting for that specific input. So, to generalize the network, whole input data go through the

DNN training process and this is called an epoch. To train the network well, the epoch is repeated until the training result reaches some criteria. It is the main reason that DNN training takes a lot of time compared to DNN inference

1.2 Convolutional Neural Network

Convolutional Neural Network (CNN) is a kind of DNN that is widely used for image classification. Each convolutional layer has filters, collections of weight elements, to extract meaningful features from input activations. The filters are trained through the training process as explained above with the three phases. Details of the training phases in the convolutional layers are explained in this section

1.2.1 Forward propagation

FP through the convolutional layers takes two input tensors, one for input activations and the other for weights. The formula for FP is given by

$$O(x, y, n) = \sum_{i=0}^{F_x-1} \sum_{j=0}^{F_y-1} \sum_{k=0}^{F_z-1} I(x+i, y+j, k) \times F(i, j, k, n) \tag{1.1}$$

where $I(x, y, z)$ and $O(x, y, n)$ are an input activation and an output activation respectively, and $F(x, y, z, n)$ is a weight of n-th filter. An input activation map is a set of 3D data of size $I_x \times I_y \times I_z$ and a set of weights forms a 3D filters of size $F_x \times F_y \times F_z$ (usually $I_z = F_z$). The filter window slides on the input activation map and their sum of element-wise multiplication results (Hadamard product) becomes an element in the output activation map. Because the filter window slides on the input activation map, the output activation map becomes a 2D data of size $(I_x - F_x + 1) \times (I_y - F_y + 1)$. We consider Fn such filters operating on the same input activation map, generating a final 3D output activation map of size $(I_x - F_x + 1) \times (I_y - F_y + 1) \times F_n$.

1.2.2 Backward propagation

BP through the convolutional layers takes the weight ($F(x, y, n, z)$) and the gradient of activation ($\Delta O(x, y, n)$) from the next layer in order to calculate gradient of activation ($\Delta I(x, y, z)$).in the current layer. BP can be formulated as

$$\Delta I(x, y, z) = \sum_{i=0}^{F_x-1} \sum_{j=0}^{F_y-1} \sum_{k=0}^{F_{\mathbf{n}}-1} \Delta O^{'}(x + i, y + j, k) \times F(\mathbf{i}, \mathbf{j}, z, \mathbf{k}) \qquad (1.2)$$

where $\Delta O^{'}(x, y, n)$ is $\Delta O(x, y, n)$ with *zero-padding*. Unlike FP, the weight's are rearranged to form F_z new 3D filters of size $F_x \times F_y \times F_n$ as shown in Figure 1.3. Thus, BP calculates the gradient map of the input activations from the gradient map of the output activations and the k-th filter forms error propagating paths from the gradient map the output activations (i.e., the gradient map of the input activations of the next layer) to the k-th plane inn the gradient map of the input activations. And for the generated gradient map of the input activations to have the same dimension as the input activation map, the gradient map of the output activations should be padded with $(F_x - 1)$ and $(F_y - 1)$ zeros in the direction of x, and y, respectively. So BP is convolution operations of F_z filters of size $F_x \times F_y \times F_n$ on gradient map of output activations of size $(I_x + F_x - 1) \times (I_y + F_y - 1) \times F_n$ to compute the gradient map of input activations of size $I_x \times I_y \times I_z$.

1.2.3 Weight update

WU in convolutional layers takes the input activation ($I(x, y, z)$) and the gradient of output activation ($\Delta O(x, y, n)$) to calculate the gradient of a weights ($\Delta W(x, y, z, n)$).

$$\Delta W(x, y, z, n) = \sum_{i=0}^{O_x-1} \sum_{j=0}^{O_y-1} \Delta O^{'}(x + i, y + j, n) \times I(i, j, z) \qquad (1.3)$$

However, unlike FP and BP, which perform the convolution operations on 3D data, WU performs the convolution operations on 2D data. Since the input activation map has 3D data of size $I_x \times I_y \times I_z$ and the gradient map of output activations has 3D data

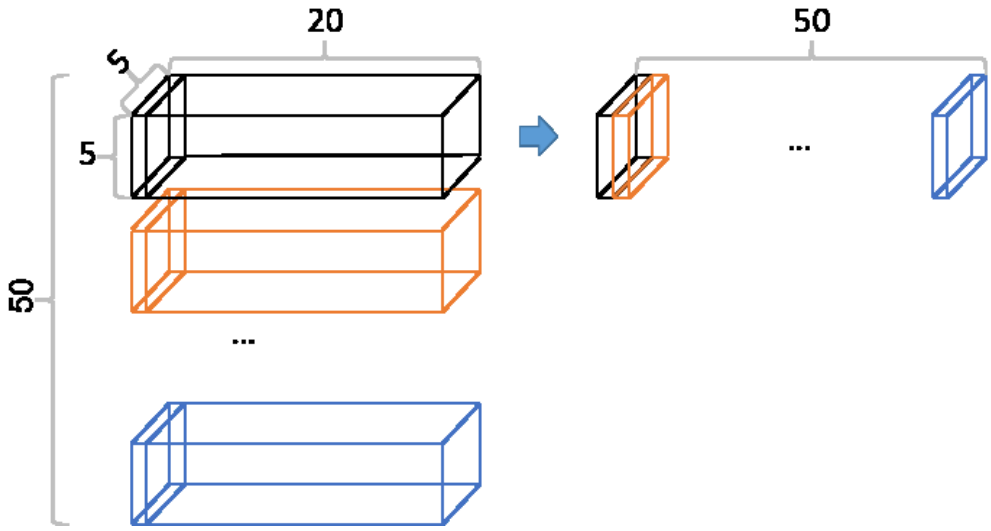

Figure 1.3: Filter rearrangement

of size $O_x \times O_y \times O_n$ which corresponds to $(I_x - F_x + 1) \times (I_y - F_y + 1) \times F_z$, the 2D convolution of $(I_x \times I_y) * (O_x \times O_y)$ is executed $I_z \times O_n$ times. Consider using the number of multiplications as the metric for the computational complexity of the three phases. While FP and WU have the same number of multiplications, BP requires more multiplications due to the $zero - padding$ of the gradient map. However, recent DNNs have relatively small size of F_x and F_y compared to I_x and I_y, and thus the $zero - padding$ does not incur significant computational overhead.

1.3 Rectified Linear Unit

Rectified Linear Unit (ReLU) [11] is a kind of DNN activation functions which is proven to solve vanishing gradient problem efficiently for back-propagation. Thus, it is most widely used in many latest DNN architectures. The ReLU function is defined as below:

$$f(x) = \begin{cases} x & \text{for} \quad x > 0 \\ 0 & \text{for} \quad x \leq 0 \end{cases} \tag{1.4}$$

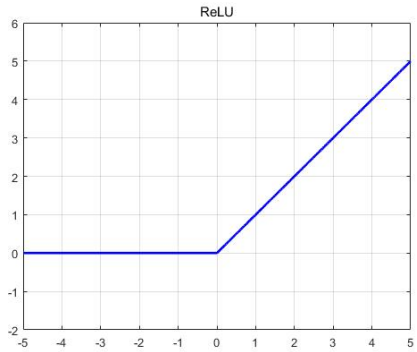

Figure 1.4: ReLU activation function

Fig. 1.4 shows the ReLU activation function. Unlike conventional activation functions such as $sigmoid$ and $tanh$, ReLU outputs zero for negative inputs. Further the reason why ReLU is widely used is that it can prevent gradients from vanishing - since its derivative is one for any positive inputs. Note that in the $sigmoid$ or $tanh$ activation function, the derivative is smaller than 1, and thus, as the backward propagation proceeds with the application of the chain rule, gradient values tend to become very small making $\Delta weight$ also very small. Thus, the weights in the frontend layers are hardly updated. Also, the characteristics of ReLU activation function makes DNN activations sparse because it generates many zero outputs. We can make use of the sparse activation map to optimize the DNN operations.

1.4 Dropout

Dropout [12] is a kind of regularization techniques for DNN training to prevent trained network from overfitting. Usually model combination works best for DNN regularization. However, due to the limited resources, it is inefficient using various models to regularize DNN model. By cutting connections of neurons between two layers the network is found that has similar effect on averaging various models. The cut connection is cut only during the training. The connection is recovered at inference. Procedure for

dropout during the DNN training is shown below.

During the forward propagation, dropout layer picks activations randomly with dropout rate. The connections for picked activations are disconnected. Note that we need to know connections before applying dropout because the connections must be recovered for the DNN inference. For implementation, the values for the picked activations are considered zero.

During the backward propagation, the error cannot be propagated through the disconnected activations. By making the gradients which correspond to the disconnected activations zero, the error is not propagated.

After the DNN trainig, the disconnected connection must be recovered for the DNN inference. To recover the connections, dropout layer must keep the information of these connections.

1.5 Previous Works

DianNao [5] is the first developed DNN accelerator architecture. It focuses on accelerating convolutional layer. It consists of 16 neural functional unit (NFU) and each NFU can perform 16 MAC operations. It can perform 256 MAC operations simultaneously. The results of multiplication is accumulated by adder tree. And it merged convolutional layer and activation function that before the output activation written back to DRAM it goes through the activation function as soon as the output of convolutional layer is generated. The datatype for the architecture is 16-bit fixed point. For DNN inference, 16-bit fixed point shows almost no accuracy drop, *DianNao* adopt it. Since the *DianNao*, various techniques are developed that even 1-bit datatype is feasible for DNN inference. It has SRAM buffers to save the time to access DRAM. Total $34KB$ of SRAM buffer for input activations ($2KB$), filter weights ($16times2KB$), and output activations ($2KB$).

DaDianNao [6] is the first DNN accelerator architecture which is feasible to DNN

training. It concists of 16 *DianNao*s and SRAM buffers are alternated to eDRAM due to the area. Because eDRAM has higher latency than SRAM, it splits eDRAM into 4 banks and interlreaves DRAM accesses to compensate higher latency. Unlike *DianNao*, it adopts 32-bit fixed point as a datapath. For DNN training, same phase is feasible for lower data precisions. However, some phase such as weight update requires high-precision data type with no special techiniques [13, 14, 15, 16], . So *DaDianNao* adopts 32-bit fixed point instread of 16-bit fixed point.

Cambricon-X [10] is further developed version of *DianNao* that can skip unneccessary MAC operation by making use of sparsity of neurons. It adds *Buffer Controller* to check whether the activation is zero or not and get the information of needed neurons for efficient convolutional operation.

Eyeriss [9] is reconfigurable DNN accelerator architecture. It consists of 168 processing elements which has a multiplier and an adder. But it can reconfigure dataflow that can handle data movement between off-chip and on-chip efficient for high throughput and energy efficiency. Further *eyeriss* can skip unnecessary operation invoked by sparse neurons.

TPU [17] is a DNN accelerator developed by Google. It focuses on accelerating general matrix multiplication, which is used for convolutional operation on conventional devices like CPU or GPU. The big difference between *TPU* and conventional devices are *TPU* use systolic array for general matrix multiplication. Matrix multiplication consists of inner products of a row and a column. Systolic array does not need to store immediate values of multiplication. So *TPU* can save energy and execution time for storing and loading the immediate values. However, convolutional operation using general matrix multiplication must waste memory usage due to the im2col and im2row operation. These operations make general matrix multiplication to convolutional operation. Each inner product of a row and a column exactly corresponds to a convolutional operation that the result of matrix multiplication exactly matches the output activation map. But im2col and im2row operation duplicate input activation map that required

memory must be increased upto 6 times.

Chapter 2

Acceleration of DNN Backward Propagation on CNN layer

2.1 Motivation

Rectified linear unit (ReLU) [11] is the most popular DNN activation function. It alleviates the vanishing gradient problem in BP, and thus, it is widely used in many latest DNNs ([1, 2, 18]). The ReLU function is defined as follows:

$$a_{out} = f(a_{in}) = \begin{cases} a_{in}, & a_{in} > 0 \\ 0, & a_{in} \leq 0 \end{cases} \tag{2.1}$$

where a_{out} is output activation and a_{in} is input activation.

The BP formula for ReLU layer can be derived using the chain rule as follows:

$$\frac{dE}{da_{in}} = \frac{dE}{da_{out}} \cdot \frac{da_{out}}{da_{in}} \tag{2.2}$$

where

$$\frac{da_{out}}{da_{in}} = \begin{cases} 1, & a_{in} > 0 \\ 0, & a_{in} \leq 0 \end{cases} \tag{2.3}$$

Here E represents the error. In BP phase, dE/da_{out} is calculated by the backward propagation. We call $\delta_{Rout} = dE/da_{out}$ *output-side activation gradient*, and call $\delta_{Rin} = dE/da_{in}$ *input-side activation gradient*. Based on the formula above, as soon as we observe $a_{in} \leq 0$ (or $a_{out} = 0$) that has been obtained during FP in a ReLU layer, we can immediately set the corresponding $\delta_{in} = 0$ without any BP.

At first glance, it may look like there is little potential of operation skipping, because the operations involved in the BP of a ReLU layer are very simple and there are not so many of them. However, if we take the layer that immediately follows the ReLU layer into account, the potential widely broadens.

To explain this point more clearly, we show the BP for the 2nd convolutional layer of LeNet-5 in Figure 1.1. It shows how the BP for a convolutional layer can be interpreted as a convolution between δ_{out} map and rearranged filters (or BP filters). The BP for the convolutional layer starts with rearranging the filters used in FP as shown in Figure 1.3. It shows how the rearrangement is done for the first BP filter. Generally speaking, n-th BP filter is formed by gathering and mirroring the n-th channel planes of FP filters.

Figure 1.1 show the calculation of the first δ_{Cin} pixel using the δ_{Cout} map and the first BP filter. Zero-padding of 4 pixels in both $x-$ and $y-$ directions on the δ_{Cout} map are necessary to match convolution dimensions. Since this convolutional layer immediately follows a ReLU layer in LeNet-5, the dimensions of the δ_{Cin} map shown in the figure exactly matches those of the output activation map of the ReLU layer.

Now let's assume that the output activation pixel of the ReLU layer that corresponds to the first δ_{Cin} pixel shown in the figure is zero. Then not only the ReLU layer's BP operation, but also the whole convolution operation for the convolutional layer in Figure 1.1 can be skipped. In this specific example, we can skip $5 \times 5 \times 50$ multiplications and $5 \times 5 \times 50 - 1$ accumulations for the single zero activation.

Considering the fact that there are a lot of zero-valued activations at the output of a ReLU layer, this gradient computation skipping scheme has a big potential speeding up

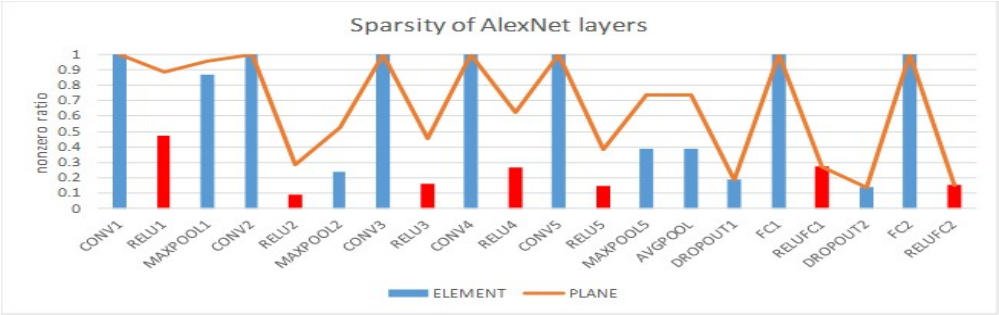

Figure 2.1: Sparsity of AlexNet Layers.

BP phase. For example, Figure 1.2 shows that more than half of the output activations of ReLU layers are zero in VGG-16. The application of batch normalization [19] in recent DNNs further broadens the applicability of our scheme by assuring about half of the output activations to be zero.

To show the ReLU layer makes the network sparse, we analyze the sparsity of AlexNet and VGG-16 during the whole training. Figure 2.1 and 2.2 show sparsity of each layer during the training. x-axis shows the layers consisting each networks and y-axis shows *non-zero* ratio for the output activation of each layer. The blue and red bar shows non-zero ratio for elements of the output activation and the orange line shows non-zero ratio for planes of the output activation. The non-zero ratio for planes will be discussed later in Section 2.4.2. ReLU layer follows by convolutional layer on AlexNet and convolutional layer and batch normalization layer on VGG-16. ReLU layer makes non-positive inputs into zero, output activation of ReLU layer has many zero activations. For AlexNet, non-zero ratio of the output activation of ReLU layer becomes quite small (less then 0.5). For VGG-16, non-zero ratio of the output activation of ReLU layer is about 0.5. This is because batch normalization normalizes mean

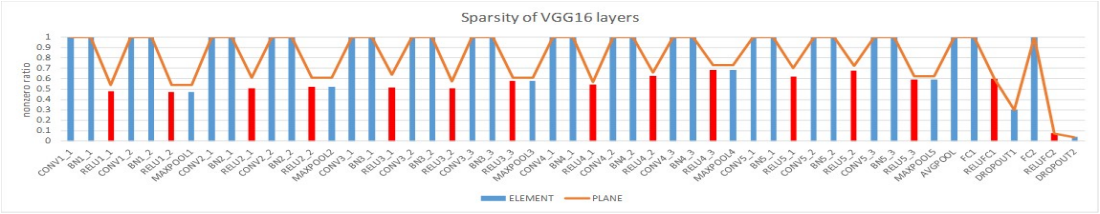

Figure 2.2: Sparsity of VGG-16 layers

of the output activation of convolutional layer to zero that about half of the output activation becomes zero.

To show that the sparsity is being changed during the training, we need to show the non-zero ratio of each network during the whole training. Figure 2.3 and 2.4 show the varying non-zero ratio of each layer during the DNN training. y-axis shows non-zero ratio for the output activation of each layer and x-axis shows layer-wise sparsity changing as the network being trained.

To apply the proposed computation skipping, we need to consider the BP through ReLU layer together with that through the next convolution layer. We can think of this scheme as a *layer fusion* technique applied to BP. One of the layer fusion techniques that is commonly used in DNN FP HW accelerators is the fusion of a ReLU layer with the preceding convolutional or fully connected layer [5]. In this scheme, when an output activation pixel is calculated, before writing it back to DRAM, ReLU operation is applied. This scheme saves the memory bandwidth otherwise spent on writing the output activation and reading it back to perform ReLU operations. When the proposed

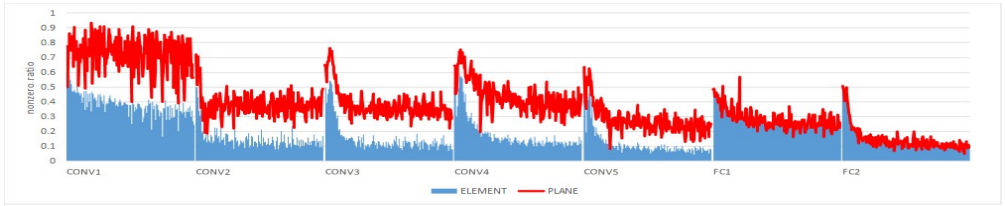

Figure 2.3: Varying sparsity of AlexNet during the training

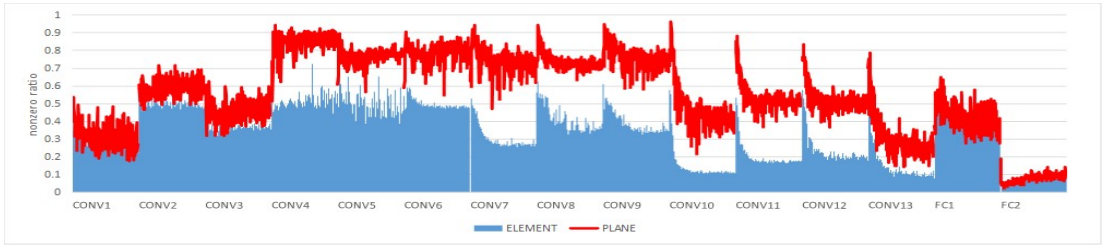

Figure 2.4: Varying sparsity of VGG-16 during the training

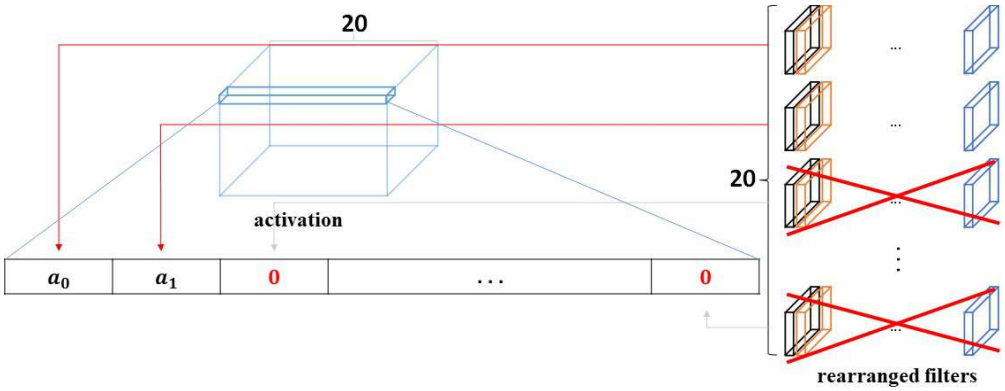

Figure 2.5: Selective gradient computation

scheme is applied to BP HW accelerators, BP for ReLU layer is fused with BP of the next layer, thus saving memory bandwidth.

In addition to that, we can save more memory bandwidth by skipping BP operations through the convolution layer next to the ReLU layer. For example, in Figure 2.5, if the third output activation pixel of the ReLU layer that corresponds to the first δ_{in} pixel is zero, then the corresponding filter weight for the convolution need not be read from the memory, further saving the memory bandwidth.

2.2 Selective Gradient Computation for Zero Activations

2.2.1 Baseline Architecture

Our approach is general enough to be applied to any DNN training accelerators. Among the existing DNN training accelerators, we select *DianNao* [5] as our baseline architecture, because it is simple, yet very powerful. Figure 2.6 shows the overall architecture of *DianNao*. it consists of 16 NFUs, each performing a MAC operation of 16 input activations and 16 weights and a non-linear activation function. NFUs consist of three pipeline states, which are in charge of multiplications, accumulations, and activation functions, respectively. Therefore, it provides its peak throughput of 256 MACs per

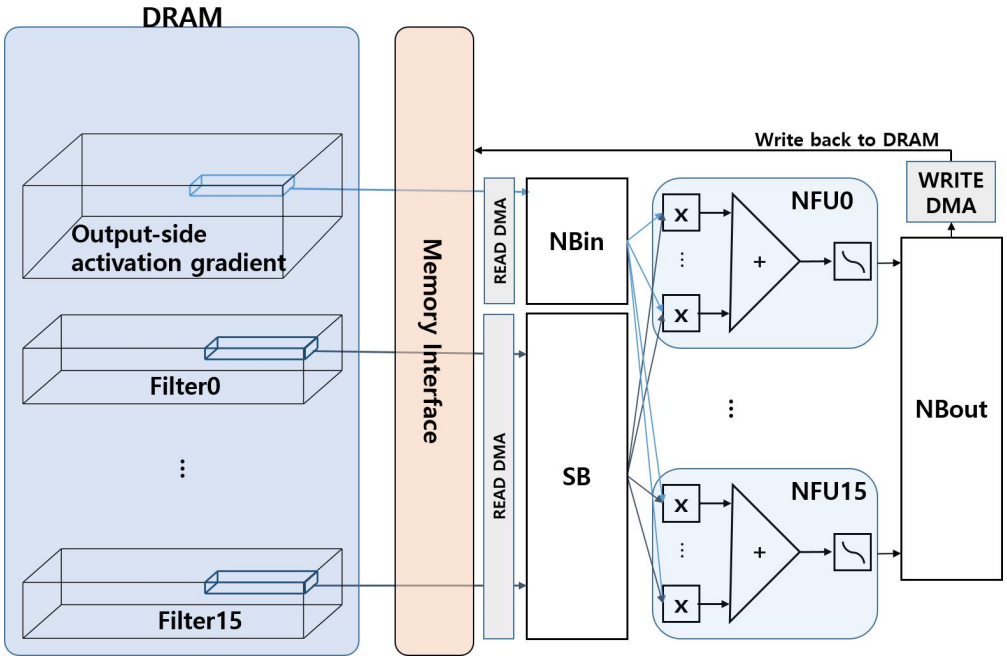

Figure 2.6: Overall architecture of *DianNao*

cycle. The 16 NFUs generate 16 partial sums of the output feature map per cycle. When processing a convolutional layer, the throughput of 16 partial sums per cycle is allocated to the calculating of 16 partial sums of 16 consecutive filters.

For a single filter position on the input feature map, the convolutions for the 16 consecutive filters are completed before performing the convolutions for the next 16 consecutive filters at the same filter position on the input feature map. After completing the convolutions for all the filters, the filter position on the input feature map is shifted in raster-scan order.

DianNao has on-chip buffers for input activations (NBin), output activations (NBout), and filters (SB), each of which has the size of 2KiB, 2KiB, and 32KiB. These buffers are used for data reuse in order to reduce memory bandwidth consumption. However, if the operation order for a convolutional layer described in the preceding paragraph is employed, these buffers are not so useful in terms of data reuse especially when a layer

with many channels is to be processed. Following is the explanation of the reason.

One can consider two sources of data reuse: input activations and filters. The input activation reuse occurs when the convolution operation for a 16 consecutive filters has ended and the next 16 consecutive filters are to be processed. The filter reuse occurs when the convolution operation for a filter position on the input feature map has ended and the filter position is to be shifted in raster-scan order. However, the buffers (NBin and SB) for them should be much bigger than those of *DianNao*. For example, let's consider conv3-512, a convolutional layer of VGG-16, where 512 filters of $3 \times 3 \times 512$ weights are applied on the input feature map with 512 channels. In this case, for the input activation reuse, NBin of $3 \times 3 \times 512$ input activations is required. On the other hand, for the filter reuse, the entire filter weights should be stored in SB. In other words, when 16-bit fixed-point operations are assumed, we need NBin of 9KiB and/or SB of 2MiB, which are much bigger than the amount *DianNao* has.

Since we do not deal with toy examples such as LeNet in this paper, we eliminated the on-chip buffers from the proposed architecture. In some cases, the elimination of the on-chip buffers might make BP through a layer a memory-bound process while it would has been computation-bound with the on-chip buffers. However, the proposed architecture still keeps its advantage in performance and energy consumption over *DianNao* in those cases, because as shown in Section 2.1, our approach reduces the computation as well as memory bandwidth. When BP is computation bound, skipping gradient calculation contributes to the improvement. On the other hand, when BP is memory bound, selective loading of filters provides the contribution.

Finally, *DianNao* employs 16-bit fixed-point datapath for performing only FP whereas we need an architecture that can perform DNN training. The follow-up work of *DianNao*, *DaDianNao*, mentions that simply expanding the width of the datapath to 32 bits enables the architecture to perform DNN training. We validated this statement by performing the training of AlexNet and VGG-16 with 32-bit datapath on ImageNet dataset.

Table 2.1: The Number of DRAM Accesses for Selective Gradient Computation Comparision on AlexNet

		# of DRAM accesses	
Layer	*Omap*	*activation*	*bit-vector*
CONV1	55x55x96	18150	568
CONV2	27x27x256	23328	365
CONV3	13x13x384	8112	127
CONV4	13x13x384	8112	127
CONV5	13x13x256	5408	85

2.2.2 Bit-Vector for Selective Gradient Computation

The operands of convolutional BP are δ_{Cout} map and filters. Note that the output activation of FP is not necessary here. However, as shown in Section 2.1, in order to apply our approach, the information of whether the output activation of the preceding ReLU layer is zero or not is required. If the entire output activation map of the ReLU layer is read for this purpose in BP phase, this might be a significant overhead in terms of performance as well as energy consumption.

Fortunately, the information can be stored in a bit-vector format instead. By storing 0 for zero output activations and 1 for positive output activations, we can reduce required memory bandwidth by $1/32$ (since as mentioned in Section 2.2.1, we employ 32-bit wide datapath). The bit-vector is generated during the FP phase and written to DRAM together with the corresponding output activations. Table 2.1 and 2.2 shows the number of DRAM accesses when we use activation and bit-vector to get the information of the output activation.

Note that even the bit-vector for a layer takes a large memory space, and thus it is burdensome to store the entire bit-vector of a layer in the on-chip buffer during its BP. Therefore, we decided to store the bit-vector in the unit of n bits, which is called a bit-vector line, where n is the number of channels in δ_{Cin} map. Thus, a bit-vector line

Table 2.2: The Number of DRAM Accesses for Selective Gradient Computation Comparision on VGG-16

		# of DRAM accesses	
Layer	*Omap*	*activation*	*bit-vector*
CONV1	224x224x64	200,704	6,272
CONV2	224x224x64	200,704	6,272
CONV3	112x112x128	100,352	3,136
CONV4	112x112x128	100,352	3,136
CONV5	56x56x256	50,176	1,568
CONV6	56x56x256	50,176	1,568
CONV7	56x56x256	50,176	1,568
CONV8	28x28x512	25,088	784
CONV9	28x28x512	25,088	784
CONV10	28x28x512	25,088	784
CONV11	14x14x512	6,272	196
CONV12	14x14x512	6,272	196
CONV13	14x14x512	6,272	196

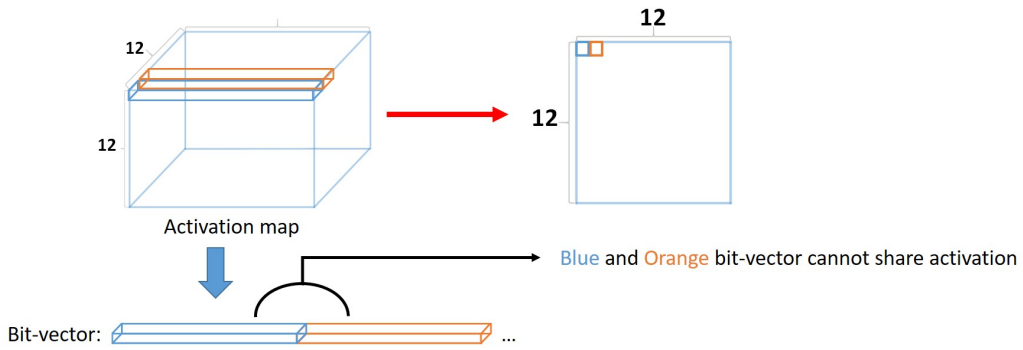

Figure 2.7: The bit-vector line

covers all the corresponding output activations of the ReLU layer for a given (x, y) coordinates. Figure 2.7 shows in detail.

As mentioned in Section 2.2.1, *DianNao* calculates the 16 partial sums for 16 consecutive filters per cycle. This means that in the proposed architecture, we need to gather 16 filters that correspond to the 1s in the bit-vector line and thus should not be skipped. We feed them to NFUs per cycle. Consider the case where the bit-vector line is shorter than the number of channels of δ_{Cin} map. Then there might be the cases where NFUs are underutilized not because there remains no filters not to be skipped, but because the required bitvector has not yet been read from DRAM. This leads to inefficient use of NFUs compared to *DianNao*, which might cause performance degradation. In this sense, the longer the bit-vector is, the better is the efficiency.

On the other hand, the bit-vector line need not be longer than the number of channels of δ_{Cin} map. This is because of *DianNao*'s limitation on the MAC operation scheduling. *DianNao* can perform MAC operations on 16 δ_{Cin} pixels on the same (x, y) coordinates, not on different (x, y) coordinates. The proposed architecture has the same limitation on the MAC operation scheduling. Therefore, even if the proposed architecture employs longer bit-vector line than the number of channels of δ_{Cin} map, the residual bit-vector values cannot be utilized effectively in terms of performance.

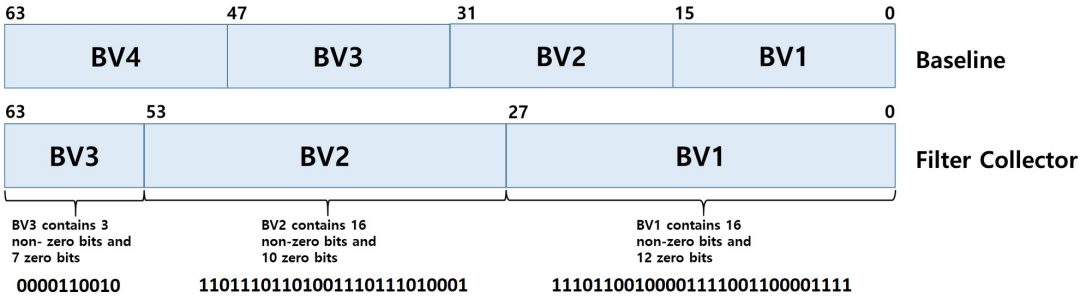

Figure 2.8: Filter collecting scheme

Therefore, we set the upper bound for the length of the bit-vector line to the number of channels of δ_{Cin} map.

2.2.3 Filter Collector

In its simplest form, the proposed architecture could be designed so that BP operation is skipped only when there is 16 consecutive 0s in a bit-vector line. If so, the modification made to *DianNao* for the proposed architecture is minimal, because in this case, filters are skipped and read from DRAM at the granularity of 16 filters, which is the same as that of *DianNao*. We could just modify the addressing logic to consider the skipping of gradient computation. However, when we profiled several DNNs, we found out that the case of 16 consecutive 0s in a bit-vector line is very rare. With this setting, although we could achieve energy saving by power-gating the NFUs with zero bit-vector value, there is absolutely no performance gain, because in this case, the same amount of filters should be read from DRAM, and the computation throughput that corresponds to the 0s in the bit-vector line is wasted.

To solve this problem, we add a module called *filter collector* that can identify non-consecutive filters corresponding to non-zero bit-vector value. For a clear description of the *filter collector* function, Figure 2.9 shows an example of the *filter collector* operation using a bit-vector line. The bit-vector line is from the first convolutional layer of VGG-16 [2], and represented by the blue bars. The triplet in the figure represents the

(x, y, z) coordinates of each bit-vector value on the δ_{Cin} map. As shown in the figure, the length of the bit-vector line is 64, which is the same as the number of channels in the $\delta_i n$ map. Also shown in the figure is that the bitvector line is for calculating the $\delta_i n$ pixels whose (x, y) coordinates are $(0, 0)$.

Starting from the bit position of $(0, 0, 0)$, the *filter collector* first identifies 16 non-skippable filters by finding out the bit positions of 16 non-zero bit-vector values. In the figure, the 16th non-zero bit-vector value is located at $(0, 0, 27)$, and the filter collector sends the identification results to the read DMA module. Here we can derive the fact that there are 12 zeros in the sub bit-vector from $(0, 0, 0)$ to $(0, 0, 27)$, because there are 16 ones in the 28 bits long sub bitvector. For the next step, the filter collector starts from $(0, 0, 28)$ and again identifies the bit positions of 16 non-zero bit-vector values. At this time, the 16th non-zero bit-vector value is located at $(0, 0, 53)$ and this information is sent to the read DMA module. Finally, for the last step for this bit-vector line, the *filter collector* tries to collect 16 non-zero bit-vector value positions from $(0, 0, 54)$, but terminates before reaching the 16 positions, because there are only 3 non-zero bit-vector values before reaching the end of the bit-vector line. As mentioned in Section 2.2.2, instead of moving to the next bit-vector line for more non-zero valued bit-vector positions, it terminates here, and sends the 3 identified non-zero valued bit-vector positions to the read DMA module.

The identified non-zero valued bit-vector positions are used to read corresponding filters from DRAM and they are fed into the appropriate NFUs. By employing the filter collector, we can make efficient use of the NFUs' computation throughput.

In order to fully utilize the peak computation throughput of the NFUs, the filter collector should be able to identify at least one set of 16 non-zero valued bit-vector positions per cycle. We implemented the filter collector in a pipelined manner so that this throughput can be achieved.

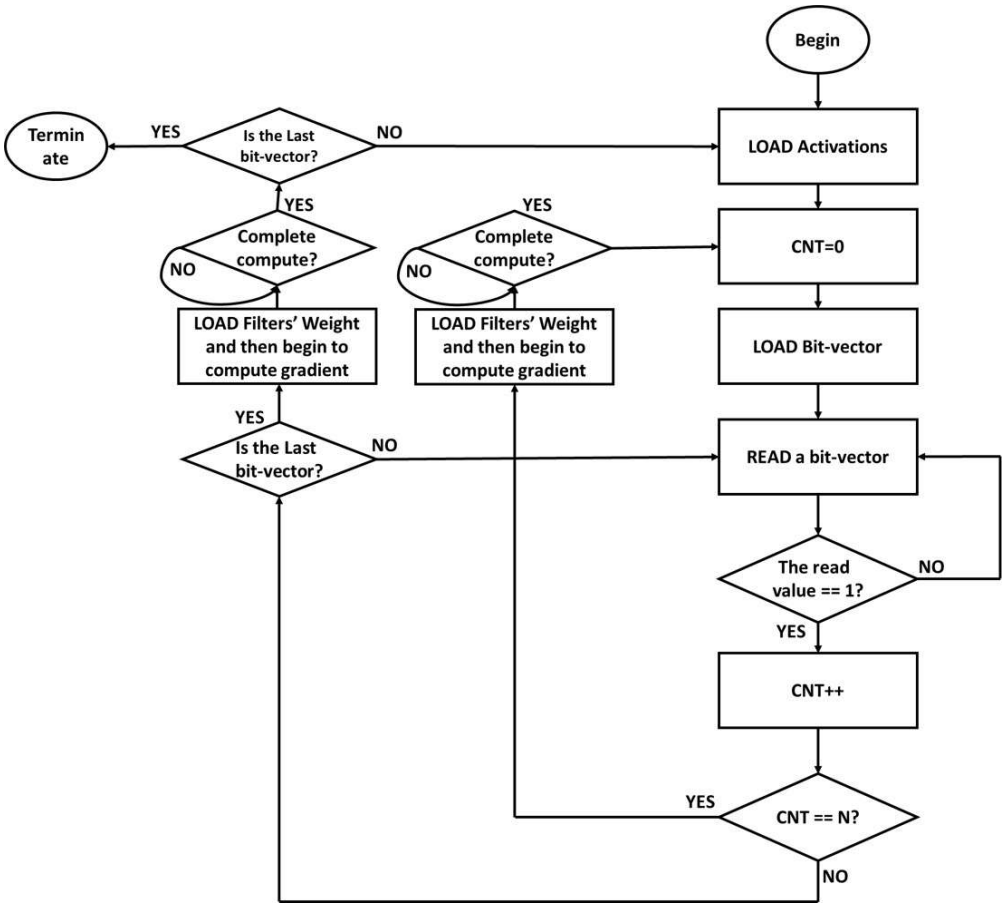

Figure 2.9: Filter collector flow chart

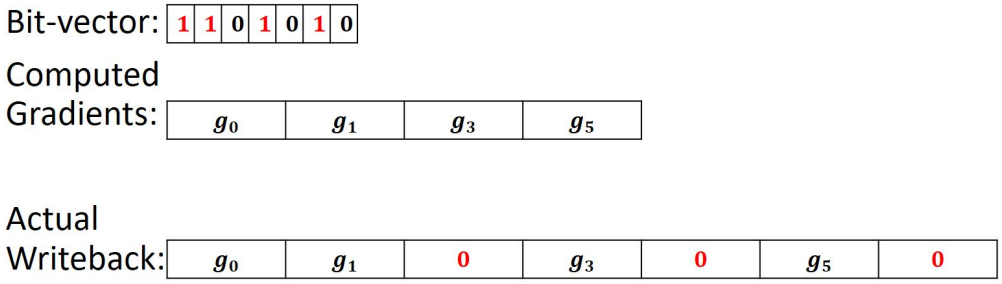

Figure 2.10: Zero gradient insertion

2.2.4 Zero-Gradient Insertion in Write DMA

When a convolution operation for a filter position on input feature map is completed, the results are written back to DRAM by the write DMA module. In *DianNao*, this is a simple process of just writing back the 16 δ_{Cin} pixels to DRAM. However, in the proposed architecture, this is not the case, because the resultant δ_{Cin} pixels are for the collected filters, thus they might not be consecutive in the δ_{Cin} map.

Here it is obvious that the write DMA module should receive the bit-vector that was used for collecting filters. According to the bit-vector, the write DMA module interleaves the resultant δ_{Cin} pixels with zero-gradients and then write them to DRAM to continue BP towards the previous layer. Therefore, the bit-vector used by the filter collector flows through the pipeline to be conveyed to the write DMA.

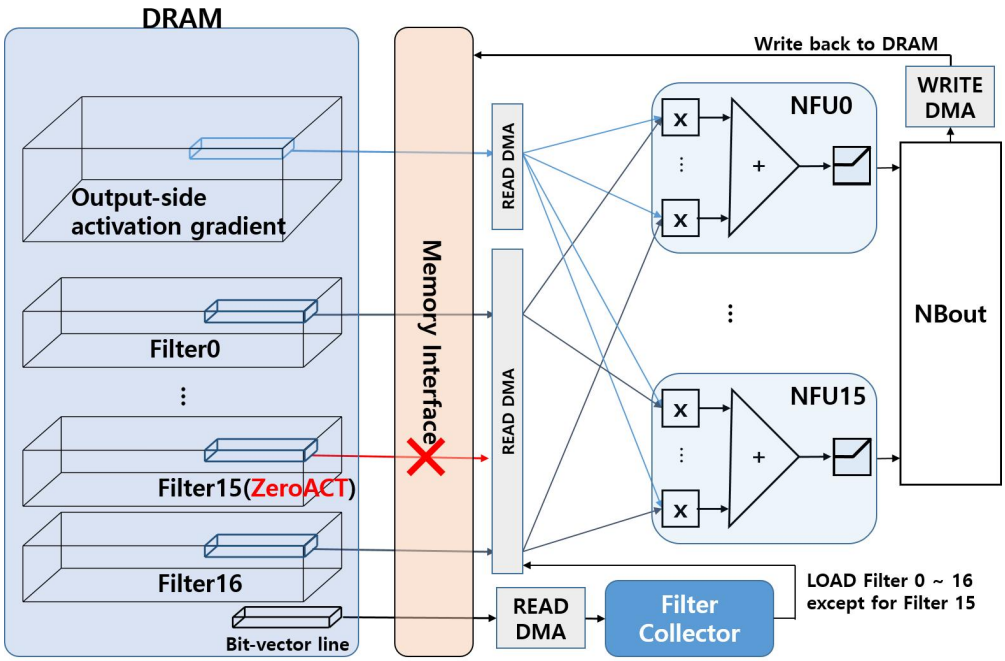

Figure 2.11: Overall architecture and example operation scenario for the proposed architecture.

2.3 Overall Architecture

In this subsection, the overall architecture and an example scenario are presented for better understanding of our scheme. As shown in Figure 2.11 for the proposed architecture, we have added/modified modules in *DianNao* including filter collector, read DMA, and write DMA modules. The filter collector reads the bit-vector line from DRAM and sends the list of filters that should not be skipped to the read DMA module. The read DMA module in *DianNao* is only capable of loading 16 consecutive filters from DRAM, but the read DMA module in the proposed architecture can seamlessly read non-consecutive filters from DRAM to feed the NFUs. The write DMA module in the proposed architecture can interleave the resultant δ_{Cin} pixels with zero gradients to store them to proper memory locations.

In the example scenario of Figure 2.11, the filter collector reads the bit-vector

line from DRAM and processes it for the first 16 nonzero valued bit-vector positions. Those are 0th - 16th bits except the 15th bit. The read DMA reads the corresponding filters and δ_{Cout} pixels from DRAM and feeds them to NFUs. Along with them, the bit-vector used by the filter collector flows through the pipeline. When the convolution operation for a filter position is completed, the write DMA module writes back the resultant δ_{Cin} pixels to DRAM, interleaving them with zero gradients, according to the bitvector that has been delivered. For this specific example, one zero gradient is inserted between the δ_{Cin} pixels generated by NFU14 and NFU15.

2.4 SRAM Buffer

2.4.1 Motivation

The accelerator architecture proposed before does not have buffers for neurons and synapses. Without any buffers, all of the data must be fetched directly from DRAM. The proposed accelerator architecture takes less than 10 cycles for a MAC operation. However, DRAM access takes more than 30 cycles that the DRAM access becomes a bottleneck for performance improvement. Because a DRAM access takes longer than a MAC operation, NFU always waits for neurons and synapses to be read. So reusing neurons and synapses used for MAC operations by fetching them in the SRAM buffers, NFU does not need to waits for them once they are fetched to the SRAM buffers and further it reduces the number of DRAM accesses. Quantifying the number of DRAM accesses ($DRAM_{access}$) for with / without SRAM buffer is shown below.

$$DRAM_{access} = \begin{cases} (F_x \times F_y \times F_n \div D_n)^2 \times F_z \times (I_x \times I_y \times I_z \div D_n), DRAMonly \\ (F_z \times F_y \times F_n \div D_n) \times F_z + (I_x \times I_y \times I_z \div D_n), withSRAMbuffer \end{cases} \tag{2.4}$$

where D_n is the number of elements read by a DRAM access. Here is an example for the reduced number of DRAM accesses. As shown in 2.12, it has $8 \times 8 \times 16$ input

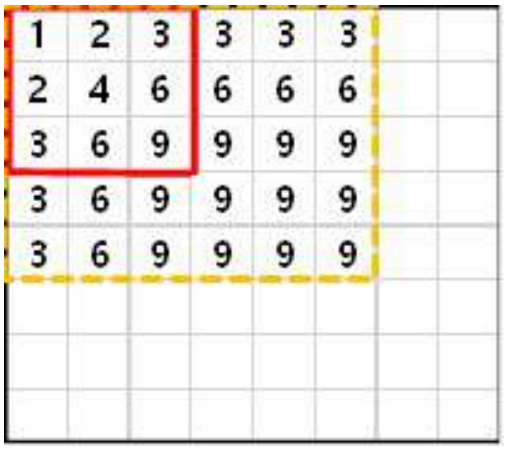

Figure 2.12: Data reuse example

activation map and $3 \times 3 \times 16$ filter weight. If input activations in yellow window (6×5) are fetched to SRAM buffer, once they are fetched, they do not need to be read again from DRAM. The numbers written in the 2.12 indicates how many times the activations are used for MAC operations. Without SRAM buffer, the numbers indicate the number of DRAM accesss for the activations exactly. With SRAM buffer, activations in the yellow window are fetched to SRAM buffer, they are read from DRAM just once.

2.4.2 Selective Gradient Computation and SRAM Buffer

As mentioned in 2.2.3, the order of bits in bit-vector corresponds to the order of the rearranged filters. The values of these bits select the rearranged filters for selective gradient computation. For data reuse using SRAM buffer, output-side activation gradients in a window are fetched to SRAM buffer. And the window determines a window for the input-side activation gradients. The window for the input-side activation gradients

The issue for selective gradient computation with SRAM buffer is neighboring bit-vectors are not coincide that the selected filters for each bit-vector are not same. Here is an example of the issue.

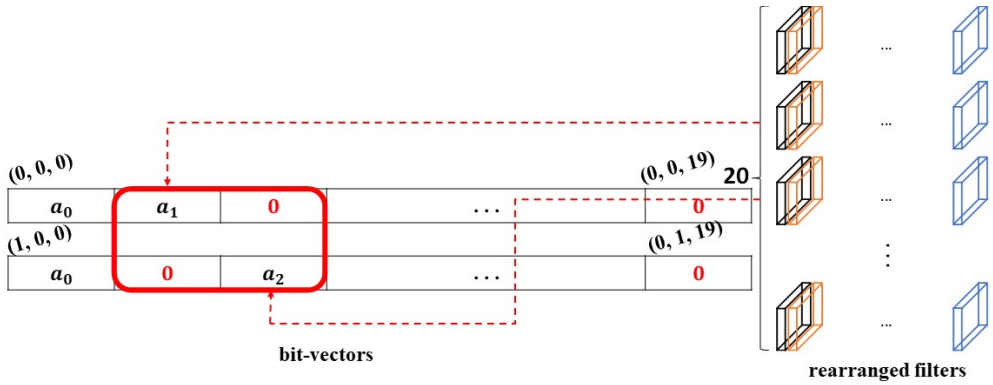

Figure 2.13: Neighboring bit-vector and their selected filters

Among two neighboring bit-vectors, first bit-vector selects 1, 2, ... th rearranged filters. However, second bit-vector does not select 2nd rearranged filter but 3rd rearranged filter. For data reuse of output-side activation gradients and filter weights, if one of rearranged filters of 2nd and 3rd filter is not fetched into SRAM buffer it cannot compute gradients precisely. Select both 2nd and 3rd rearranged filters can solve this issue.

To solve this issue, data mapping for bit-vector loaded to the *filter collector* must be changed and filter selecting scheme also must be changed.

Without SRAM buffer, dimension of the bit-vector line in the *filter collector* is $1\times1\times512$. It selects rearrange filters bit-by-bit by sweeping the bit-vector. But to reuse data with SRAM buffer, the dimension of the bit-vector line must be changed to plane-by-plane way. 2.14 shows how the dimension of the bit-vector line is changed. To keep the capacity of registers in the *filter collector* same and make it possible to reuse data corresponds to the neighboring bit-vector, the dimension is changed to $4 \times 4 \times 32$. The length of the bit-vector line must exceed 16, which is the number of MAC units in the accelerator, to skip selecting unneccesary rearranged filters. Then to reserve enough window size, the window size becomes 4×4.

With SRAM buffer, due to the change of bit-vector line dimension, the granularity

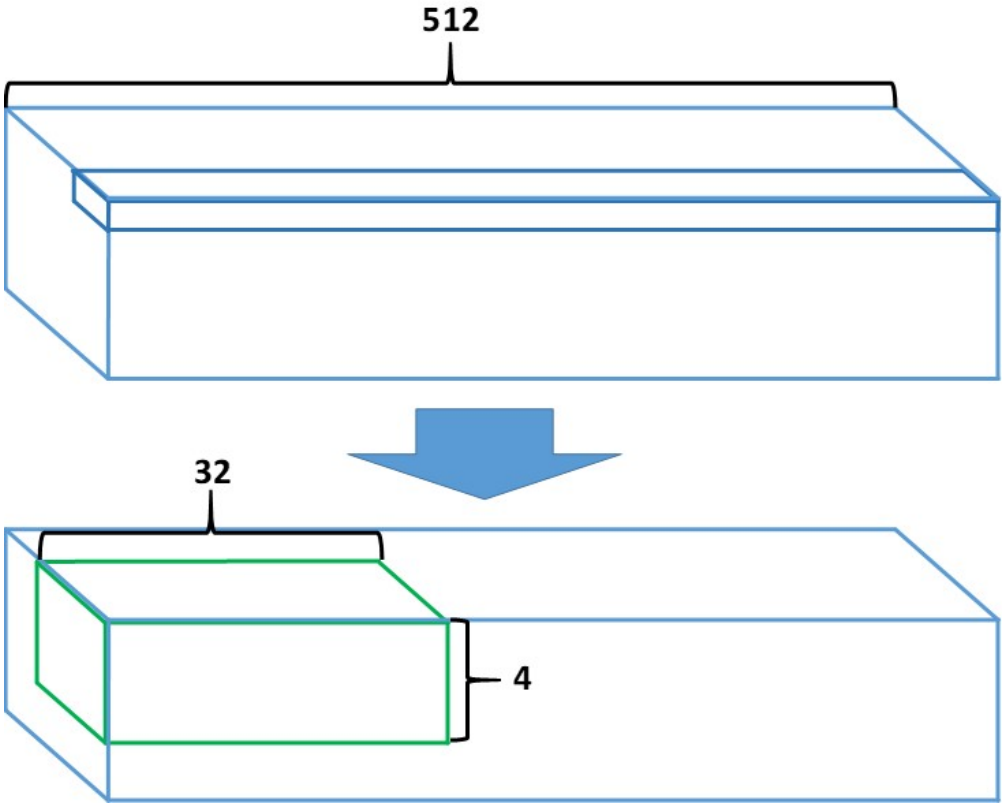

Figure 2.14: Dimention of bit-vector line

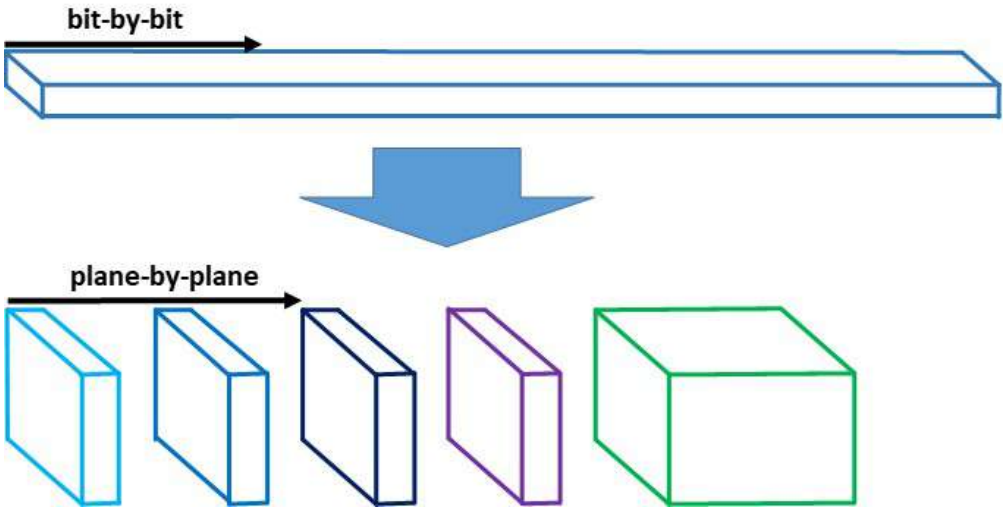

Figure 2.15: Filter selecting scheme for *Filter Collector*

of selecting rerarranged filter is also changed. Instead of bit-by-bit selection, *filter collector* selects rearranged filters by checking plane-by-plane. On a plane, if all of bits are zero, gradient computation using the corresponding rearranged filter is unneccesary that the *filter collector* does not need to select the filter. However eventhrough there are just one bits are non-zero, the corresponding rearranged filter must be selected for gradient computation.

2.5 Experimental Results

2.5.1 Performance Simulator

We build an in-house cycle accurate DNN accelerator simulator for both the baseline architecture (*DianNao*), and the proposed architecture. For the proposed architecture, the simulation model of the *filter collector* is implemented and integrated to the simulator in order to show the effectiveness of our scheme. DRAMSim2 [20] is attached to estimate access latency and energy consumption of DRAM traffic. We use DDR3_micron_32M_8B_x8_sg15 as a DRAM model for DRAMSim2. The simulator measures the execution time of BP in cycles, including DRAM access time. We compare the execution time of the baseline and our work and analyze it using the ratio of zero output activations of ReLU layer to see the effectiveness of our approach. Also we compare the DRAM energy consumption and analyze it using the ratio of zero output activations of ReLU layer.

We use AlexNet [1] and VGG-16 [2] networks on ImageNet dataset to demonstrate our work. Table 2.3 and 2.4 shows the layer configuration of AlexNet and VGg-16, respectively. For the performance simulation, we need to generate bit-vectors from the DNNs. Thus, we perform the whole training of AlexNet and VGG-16 using MatConvNet toolbox [21] on MATLAB. Because the portion of zero output activations of ReLU layer changes as training goes [22], we recorded bit-vectors at each ReLU layer at each epoch. The ratios of zero-activations observed for AlexNet and VGG-16 are

Table 2.3: The Layer of AlexNet

Layer	*Imap*	*Filter*	*# of filters*	*Omap*
CONV1	224x224x3	11x11x3	96	55x55x96
CONV2	55x55x96	5x5x96	256	27x27x256
CONV3	27x27x256	3x3x256	384	13x13x384
CONV4	13x13x384	3x3x384	384	13x13x384
CONV5	13x13x384	3x3x384	256	13x13x256
FC1	13x13x256	13x13x256	4096	1x1x4096
FC2	1x1x4096	1x1x4096	4096	1x1x4096
FC3	1x1x4096	1x1x4096	1000	1x1x1000

0.66 and 0.62, respectively.

2.5.2 RTL Implementation

We implement the synthesizable RTL models of *DianNao* [5] and the proposed architecture. They are functionally verified with constrained random test vectors. Synopsys Design Compiler is used for the syntheses with TSMC 45nm Logic Generic-Superb standard cell library (TCBN45GSBWP). The Synopsys Reference Methodology (RM-gen) is used for proper tool applications. The target operation frequency of both of the architecture is set to 500MHz, which is about half that of the original *DianNao* (0.98GHz). We simply halved the operation frequency because the proposed architecture uses the doubled datapath bitwidth (32-bit fixed-point) compared to that of *DianNao* (16-bit fixed-point). Table 2.5 shows the area and power profile of the two architectures. Power values are found out by using the default toggle rate of 0.1. Note that, as mentioned in Section 2.2.1, we eliminated SRAMs from both *DianNao* and the proposed architecture.

Table 2.4: The Layer of VGG-16

Layer	*Imap*	*Filter*	*# of filters*	*Omap*
CONV1	224x224x3	3x3x3	64	224x224x64
CONV2	224x224x64	3x3x64	64	224x224x64
CONV3	112x112x64	3x3x64	128	112x112x128
CONV4	112x112x128	3x3x128	128	112x112x128
CONV5	56x56x128	3x3x128	256	56x56x256
CONV6	56x56x256	3x3x256	256	56x56x256
CONV7	56x56x256	3x3x256	256	56x56x256
CONV8	28x28x256	3x3x256	512	28x28x512
CONV9	28x28x512	3x3x512	512	28x28x512
CONV10	28x28x512	3x3x512	512	28x28x512
CONV11	14x14x512	3x3x512	512	14x14x512
CONV12	14x14x512	3x3x512	512	14x14x512
CONV13	14x14x512	3x3x512	512	14x14x512
FC1	7x7x512	7x7x512	4096	1x1x4096
FC2	1x1x4096	1x1x4096	4096	1x1x4096
FC3	1x1x4096	1x1x4096	1000	1x1x1000

Table 2.5: Synthesis results of the baseline and the proposed architecture

		DianNao	**Proposed**
Cell area (μm^2)		1,084,915	1,214,180
Power(mW)	**Dynamic**	59.91	90.41
	Leakage	23.58	26.28

2.5.3 Performance Improvement

Figure 2.16 shows performance improvement of convolutional layers on AlexNet. x-axis is separated by each convolutional layer. And on each convolutional layer, the figure shows performance improvement on each epoch, i.e. it shows layer-wise, epoch-wise performance improvement when training AlexNet on ImageNet dataset. Blue bar shows the performance improvement and red line shows the ratio of non-zero activations. During the training of AlexNet, our work can save 68% of the backward propagation execution time. Figure 2.17 shows the result on VGG-16. During its training, our work can save 62% of the backward propagation execution time.

The performance improvement is closely related to the ratio of nonzero activations and it is mainly because the BPs of the target DNNs are memory-bound. The accelerator cannot read all the needed data in a cycle since DRAM can provide only 64 bytes ($64bits \times 8banks$) of data per access. Thus, a DRAM access only returns 16 δ_{Cout} pixels of 16 weights in 32-bit fixed point format. Therefore, to feed all the 16 NFUs, the read DMA module first accesses DRAM 17 times (1 access for 16 δ_{Cout} pixels and 16 accesses for 256 weights). *DianNao* has a sequential memory access pattern when it reads filter weights from DRAM. Although the memory access pattern of our work is not always sequential because it selectively reads non-skippable filters, our work can reduce the number of DRAM accesses so that the performance is improved.

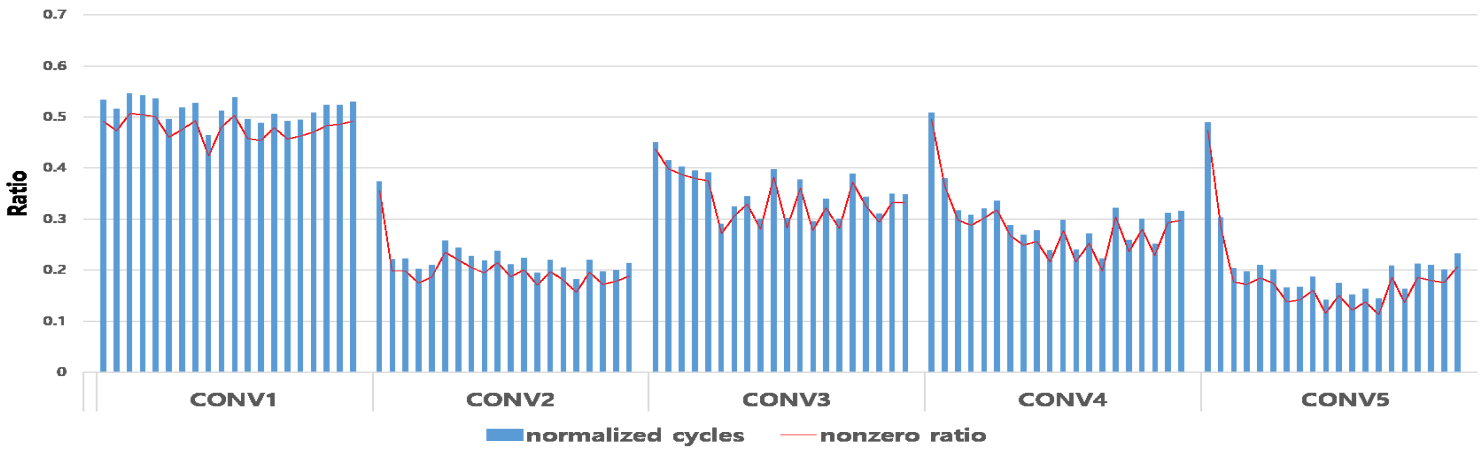

Figure 2.16: Performance improvement on AlexNet

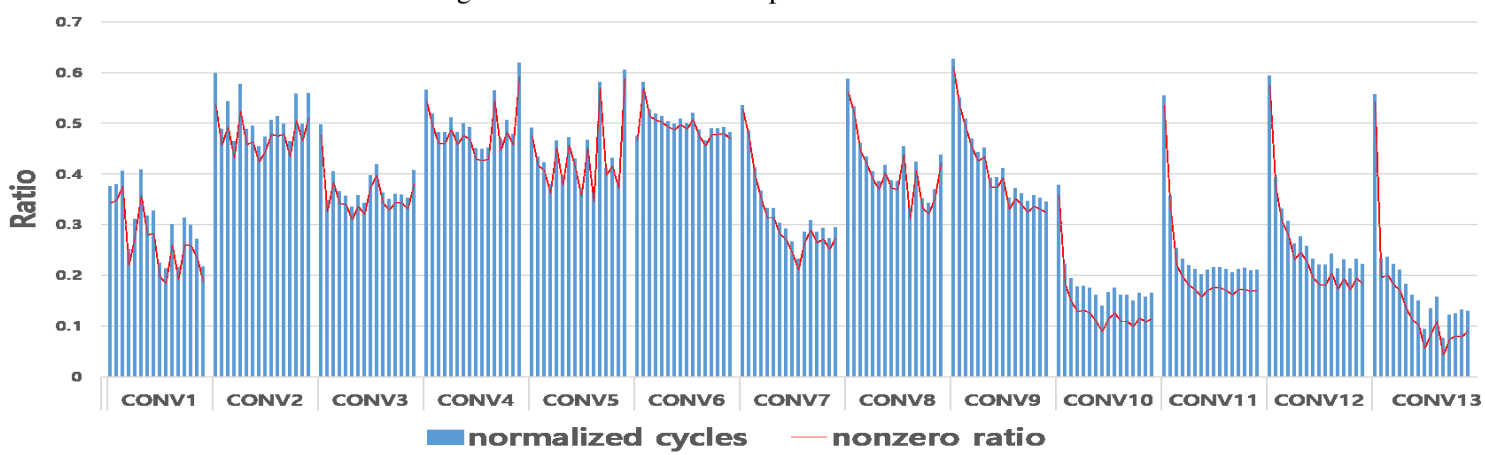

Figure 2.17: Performance improvement on VGG-16

2.5.4 Energy Reduction

Figure 2.18 shows energy reduction results on AlexNet. This figure also shows layer-wise, epoch-wise energy reduction. The energy reduction is very closely related to the ratio of non-zero activations even more closely than performance improvement, because DRAM energy consumption is directly related to the number of DRAM accesses. During the training of AlexNet, our work can save 66% of the backward propagation energy consumption. Figure 2.19 shows the results on VGG-16. During the training, our work can save 62% of the backward propagation energy consumption.

The results do not include logic and buffer energy because the simulator we have implemented focuses on performance simulation. However, we can estimate the logic power consumption based on the synthesis results shown in Table 2.5. This is justified since the number of cycles consumed to compute gradient on an accelerator is deterministic. The only variation is due to DRAM access time. Leakage power is considered constant during the entire operation cycles, and dynamic power is consumed only when the logic is working. Then we can estimate energy reduction on the logic using the following equation.

$$Energy = Power_{leakage} \times Cycles_{total\_simulation} + Power_{dynamic} \times Cycles_{NFU\_simulation} \quad (2.5)$$

According to the estimation based on this, our work can save energy consumption on logic by 53% on AlexNet and 44% on VGG-16.

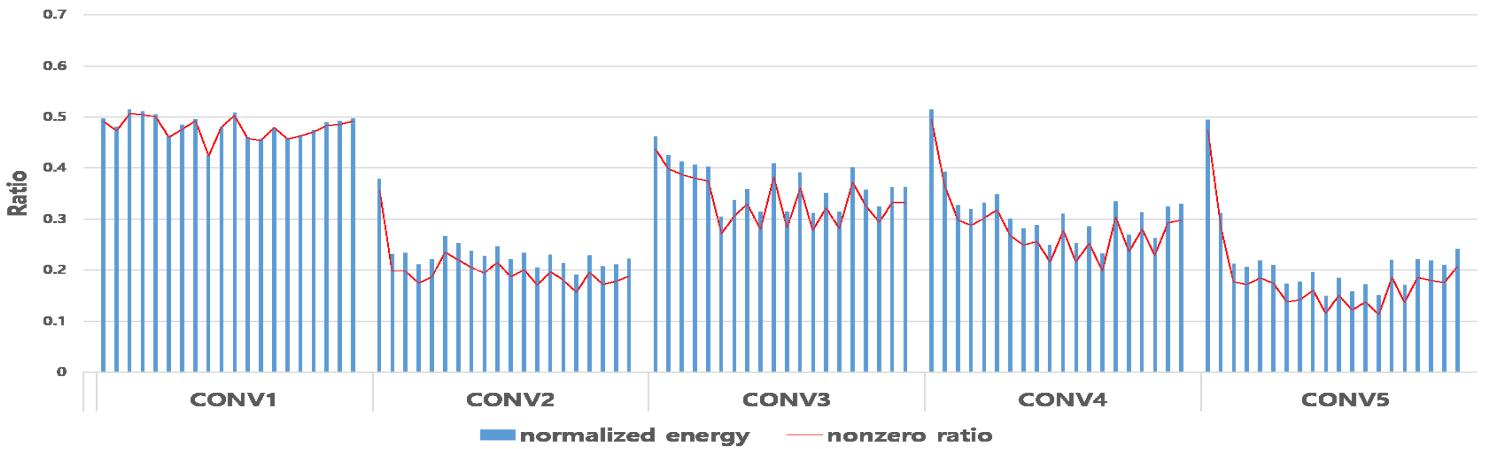

Figure 2.18: DRAM energy reduction on AlexNet

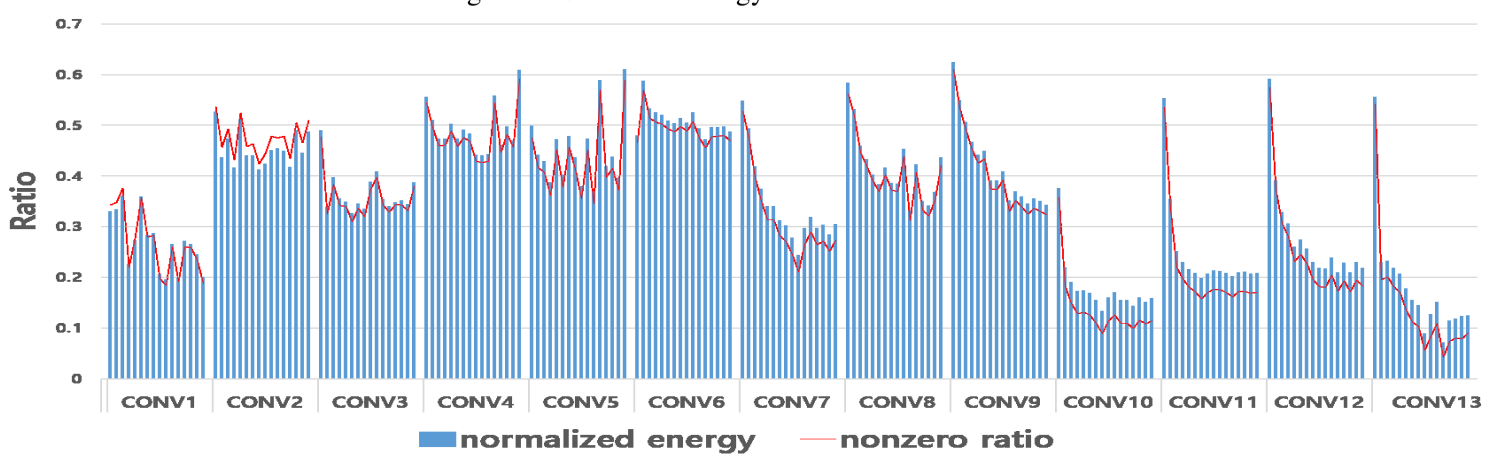

Figure 2.19: DRAM energy reduction on VGG-16

2.5.5 Impacts of SRAM Buffer

Absolute Improvements on Performance and DRAM Energy Consumption

With SRAM buffers for neurons and synapses, we can expect the reduced number of DRAM accesses and it can reduce absolute execution time.

Figure 2.24 shows the results of execution time comparisons for the proposed DNN training accelerator architectures with and without SRAM buffers on AlexNet. The gray bar shows the execution cycles of the accelerator without SRAM buffers and the orange bar shows the execution cycles of the accelerator with SRAM buffers. The yellow line shows speedup when the accelerator use SRAM buffers for neurons and synapses. It achieves $7.35x$ speedup (86.4% reduction) by using SRAM buffers. Figure 2.25 shows the results on VGG-16. It achieves $8.62x$ speedup (88.4% reduction) by using SRAM buffers.

Figure 2.26 shows the results of DRAM energy reduction comparisons for the proposed accelerator architectures with and without SRAM buffers on AlexNet. The gray bar shows the DRAM energy consumption of the accelerator without SRAM buffers and the orange bar shows that of with SRAM buffers. The yellow line shows the relative DRAM energy consumptions with SRAM buffers comared to that of without SRAM buffers. SRAM buffers for neurons and synapses can reduce DRAM energy consumption by 87%. Figure 2.25 shows the results on VGG-16. With SRAM buffers, DRAM energy consumption is reduced by 88.7%.

Using SRAM buffers for neurons and synapses, SRAM buffer can keep upto 32 elements of input feature map and 512 (32×16) elements of filter weight. SRAM buffer can keep filter weight up to 5×5 and keep input feature map upto 32 elements by forming rectangular window. When filter weight sweeps the window, there are no DRAM accesses. By this way, neurons and synapses are reused. Assuming 3×3 filter, each element of input feature map can be reused upto 9 times and it can reduces the number of DRAM accesses $1 \div 9$. Input feature map elements located near the edge of

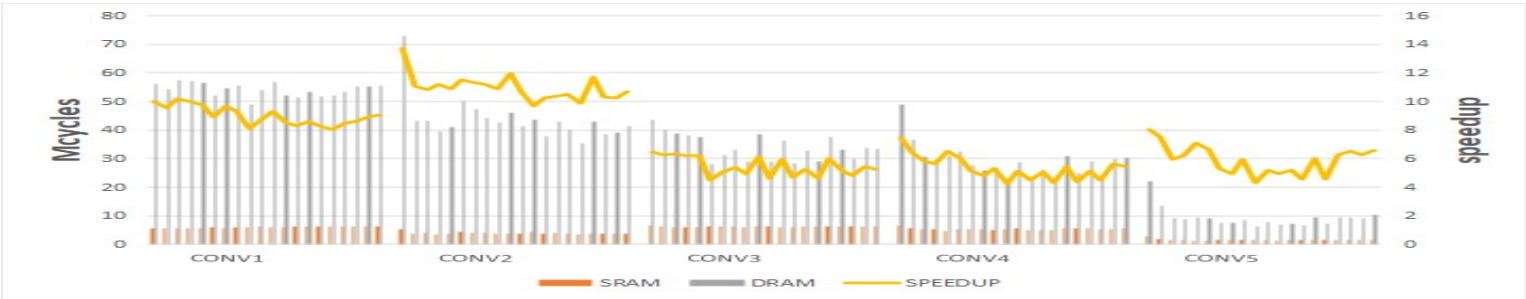

Figure 2.20: Comparison of the execution cycles for DNN training accelerator on AlexNet without SRAM buffers and with SRAM buffers

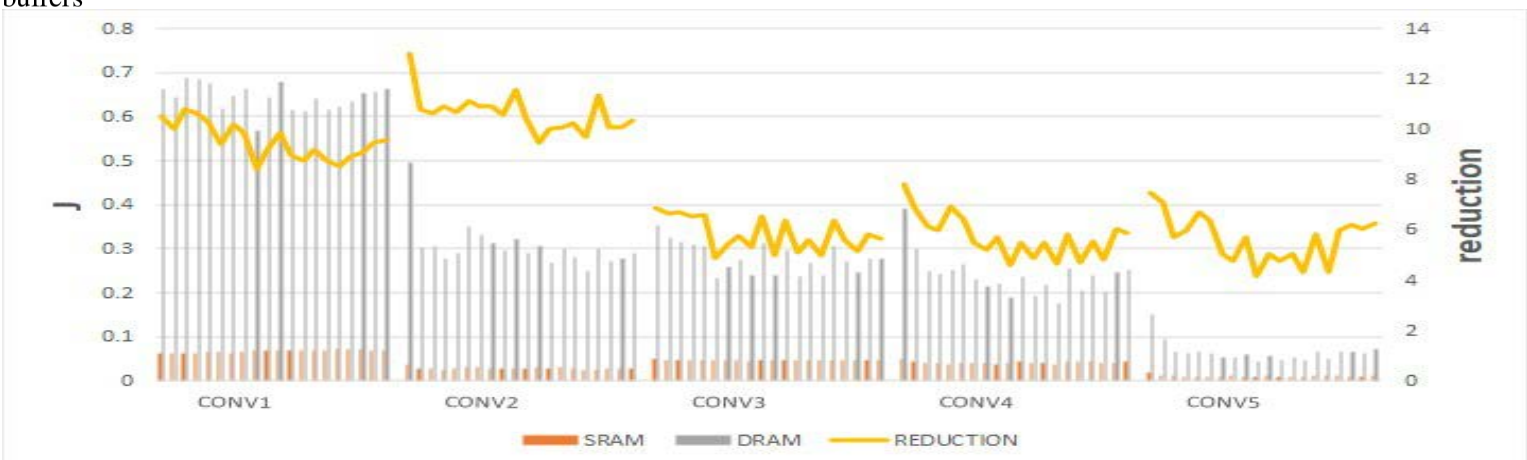

Figure 2.21: Comparison of the DRAM energy for DNN training accelerator on AlexNet without SRAM buffers and with SRAM buffers

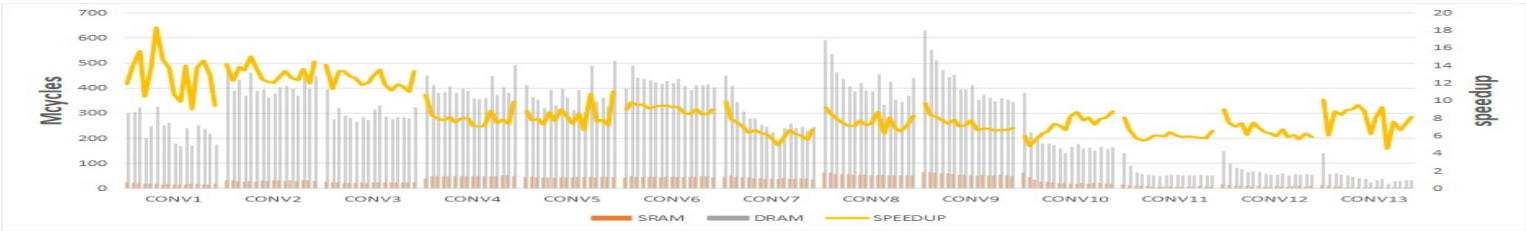

Figure 2.22: Comparison of the execution cycles for DNN training accelerator on VGG-16 without SRAM buffers and with SRAM buffers

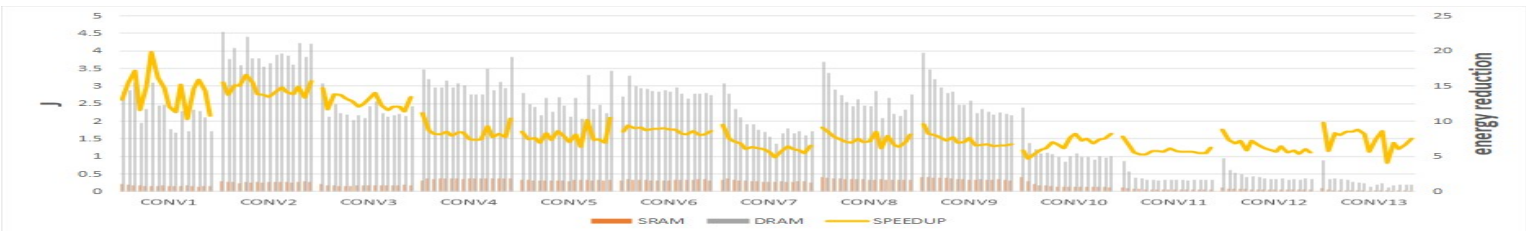

Figure 2.23: Comparison of the DRAM Energy for DNN training accelerator on VGG-16 without SRAM buffers and with SRAM buffers

rectangular has lower chance of reuse due to the characteristics of convolutional operation that theoretical speedup from input feature map reuse is smaller then 9. However, unlike input feature map, filter weight can be reused more than input feture map elements because it sweeps whole input feature map. A filter weight elements can be reused $(I_x - F_x + 1) \times (I_y - F_y + 1)$ times. For AlexNet and VGG-16 network, width of input feature map varies from 224 to 13, 14 that more data reuse is expected from filter weight. However, the size of SRAM buffer constrains unlimited data reuse and the scheme for data reuse affects the number of data reuse.

Improvements on Bit-Vector Plane

To maximize data reuse with SRAM buffers, bit-by-bit bit-vector checking cannot make use of data reuse. So instead of bit-by-bit, we adopts plane-by-plane bit-vector checking methods which can make use of data reuse. But, plane-by-plane is expected to be reduced the zero plane ratio compared to the zero bit ratio that relative improvements compared to the baseline also be expected to be reduced. However, the zero plane ratio is still quite large that plane-by-plane bit-vector checking still shows superior to the baseline with SRAM buffer.

Figure 2.24 shows performance improvement of convolutional layers on AlexNet. x-axis shows convolutional layers in the AlexNet. And the figures in each convolutional layer show epoch-wise performance improvements. The blue bar shows the relative performance improvement when both the baseline and the proposed accelerator architecture have SRAM buffers for neurons and synapses. The blue line shows the ratio of non-zero activations and the red line shows the ratio of non-zero planes. It saves 48.9% of execution time (corresponds to 1.96x speedup). Compared to the result shown in Section 2.5.3, relative performance improvement on plane-by-plane bit-vector checking is lower than bit-by-bit bit-vector checking (68% reduction) due to reduced zero ratio. Figure 2.25 shows the results on VGG-16. It saves 43.3% of execution time (corresponds to 1.76x speedup). Comparing to the result in Section 2.5.3,

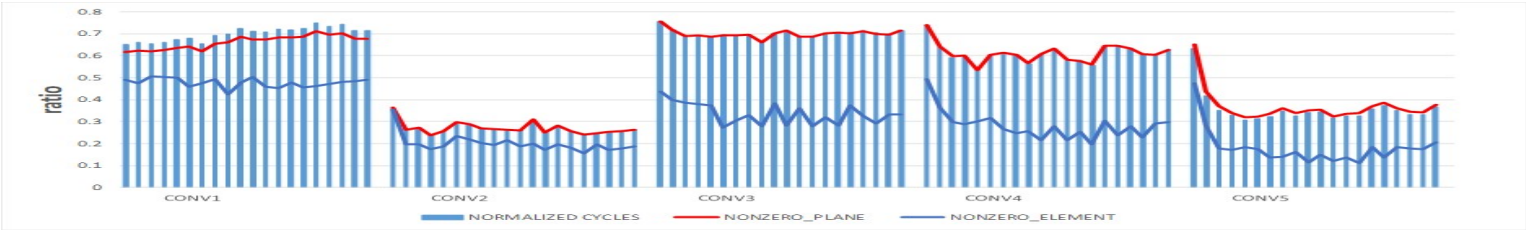

Figure 2.24: Performance improvement on AlexNet with SRAM Buffer

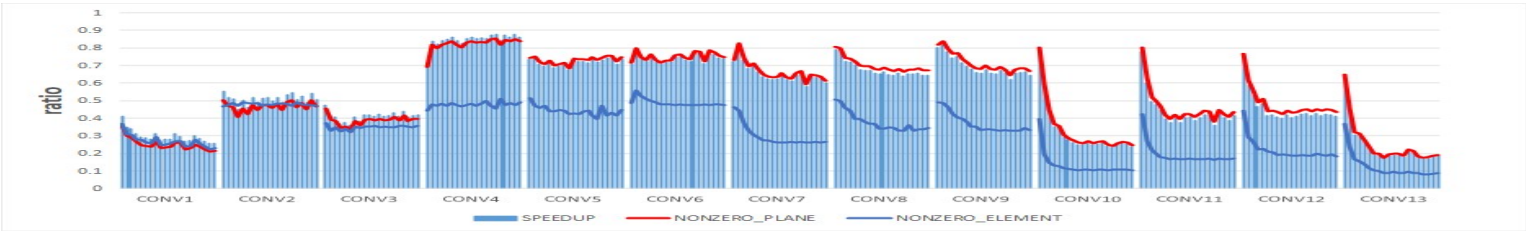

Figure 2.25: Performance improvement on VGG-16 with SRAM Buffer

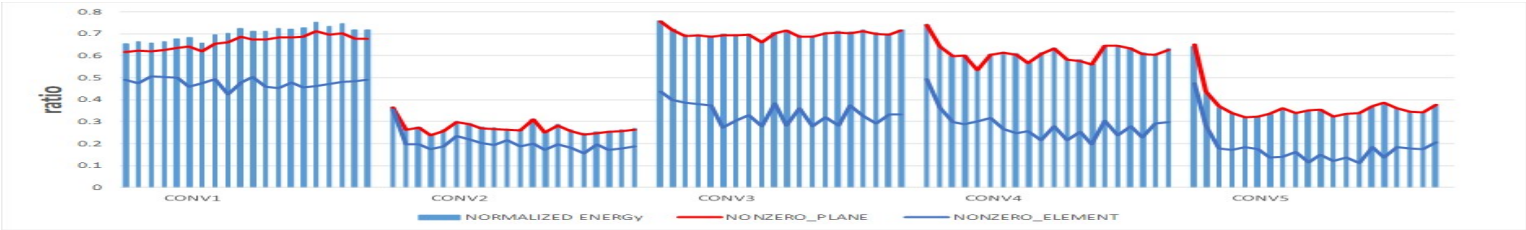

Figure 2.26: DRAM energy reduction on AlexNet with SRAM Buffer

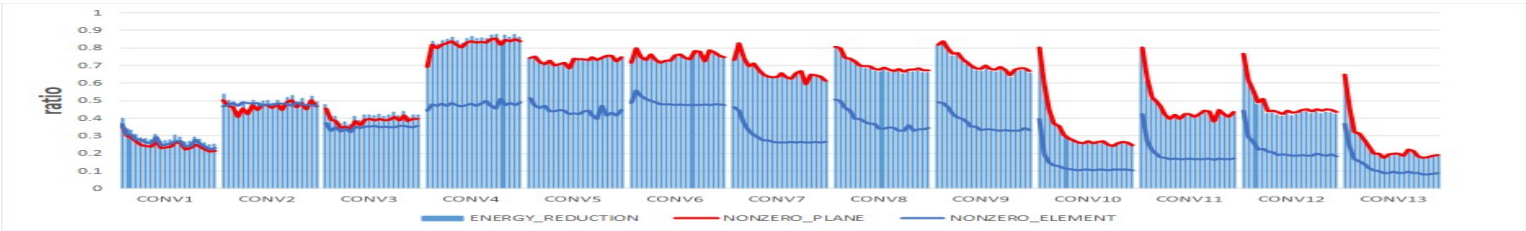

Figure 2.27: DRAM energy reduction on VGG-16 with SRAM Buffer

relative performance improvement is lower (62% reduction).

Plane-by-plane bit-vector checking reduces the relative performance improvement due to the reduced zero-ratio. But ReLU layer still makes about half of the plane sparse that our work can save the execution time of backward propagation during the DNN training.

Figure 2.26 shows DRAM energy reduction results of convolutional layer on AlexNet. Similar to that of performance improvement, the figure shows layer-wise, epoch-wise DRAM energy reduction. The zero-ratio of plane reduces the number of DRAM accesses, it can reduces DRAM energy during the backward propagation of DNN training by 46.2%. Figure 2.27 shows DRAM energy reduction results on VGG-16. It reduces 43.3% of DRAM energy consumption during the backward propagation of DNN training.

2.6 Summary

To the best of our knowledge, our work is the first attempt to make use of the characteristics of ReLU activation function on backward propagation to skip the unnecessary operations. We build a simulator and RTL model of the proposed architecture which exploits the skipping scheme and ported to *DianNao* architecture. We show that the scheme which utilizes the sparsity of δ_{Cin} significantly improves performance and reduces energy consumption. Our work can achieve 3.13x and 2.63x speedup of backward propagation on AlexNet and VGG-16, respectively. And our work can save energy consumed on backward propagation for AlexNet and VGG-16 by 66% and 62 %, respectively. And further use SRAM buffers for neurons and synapse that absolute performance improvement and DRAM energy reduction also shown. With SRAM buffers, our work achieves 1.96x and 1.76x speedup for AlexNet and VGG-16 and 46.2% and 43.3% DRAM energy reduction for AlexNet and VGG-16, respectively.

Chapter 3

Acceleration of DNN Backward Propagation on Fully Connected Layer

3.1 Motivation

3.1.1 Dropout

Dropout [12] is one of the well-known DNN regularization techniques to solve the over-fitting problem for fully connected layer. The dropout layer picks activations randomly and dropped them out during the forward propagation. During the backward propagation, the values of gradients correspond to the dropped activations are determined to zero because the error does not propagated through the dropped activations. Here, the values of gradients are determined before gradient computation that their gradient computation becomes unnecessary computations.

3.1.2 Conventional Dropout Layer Implementations

Conventional implementation of dropout using CPU or GPU consists of two phases. First, for each layer in the forward propagation, dropout layer generates randomly masked activation map. The masked activations are dropped out that their connection to the next layer is disconnected. Second, during the backward propagation, gradi-

ents correspond to the dropped activations become zeros after gradient computations because their connection is disconnected, the error from the next layer does not propagated through them. It is usually implemented as element-wise multiplications of gradients, the dropout masks, and the scale factor in this phase.

Conventional random dropout should solve some implementation issues to achieve both energy efficiency and its accuracy. Since masked activations should be saved, it requires additional memory space and memory acceses as well. Further, conventional implementations compute gradients, and then multiply by masks to drop gradients out rather than skipping the computations since parallelism in the used architectures should be maximized. [23] proposes an idea to accelerate dropout layer using GPGPU. Due to the characteristics of GPU architecture, dropout cannot be applied to element-wise but coarse-grained way to avoid control-flow divergence for better performance efficiency. On an activation map, dropout is applied to its split tile-wise that it incurs accuracy drop by losing randomness.

But my work in this dissertation implements dropout as same as how dropout works, it accelerates dropout layer in the DNN accelerator architecture with no accuracy drop.

3.1.3 Applications of Dropout

For convolutional layer, dropout is applied to regularize the output activations of fully connected layers. The amount of time spending on dropout layer is small because the fully connected layer is found at the late order of network that the number of input and output activation is small compared to the convolutional layer in prior layers. Further, dropout has been replaced to batch normalization [24] for convolutional neural networks. Many latest network adopt batch normalization instead of dropout because batch normalization reduces overall training time for convolutional neural network.

But, dropout is still used for recurrent neural network models. Applying dropout technique as the convolutional neural network does not work on recurrent neural net-

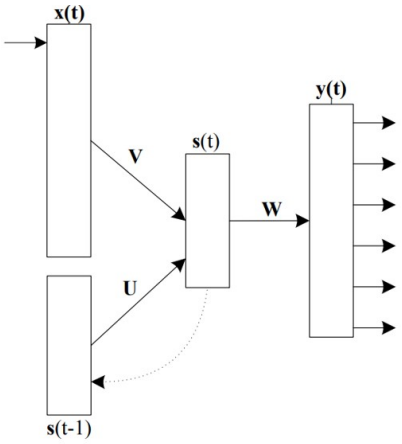

Figure 3.1: The recurrent neural network

work [25]. However, there are many researches which try to apply dropout on recurrent neural networks. There are many works [26, 27, 28] apply dropout to the recurrent neural networks.

For recurrent neural networks, their main operation is fully connected layer. Figure 3.1 shows a recurrent neural network model. If we unfold the model, the model is shown in Figure 3.2. Each black arrow represents matrix-vector multiplication. The matrix-vector multiplication corresponds to the fully connected layer operation.

And during the training of recurrent neural network, it computes gradient using the backward propagation through time (BPTT) algorithm. The red arrows in Figure 3.3 show the error propagation paths for the BPTT algorithm. The operation for each arrow is fully connected layer and the dropout can be applied to all the red arrows. So, applying selective gradient computation for dropout is expected to be used on the recurrent neural network and expected to accelerate BPTT during the training.

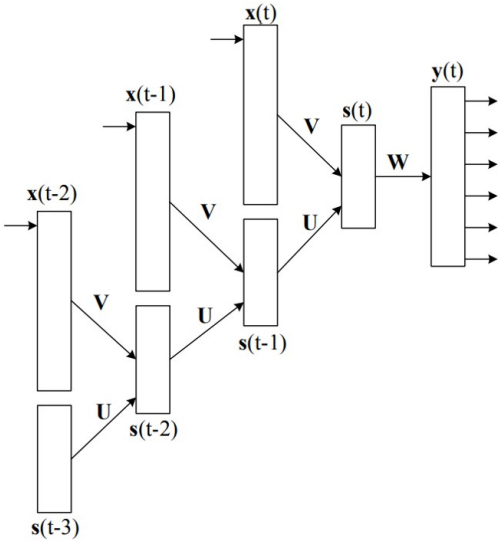

Figure 3.2: The unfoleded recurrent neural network

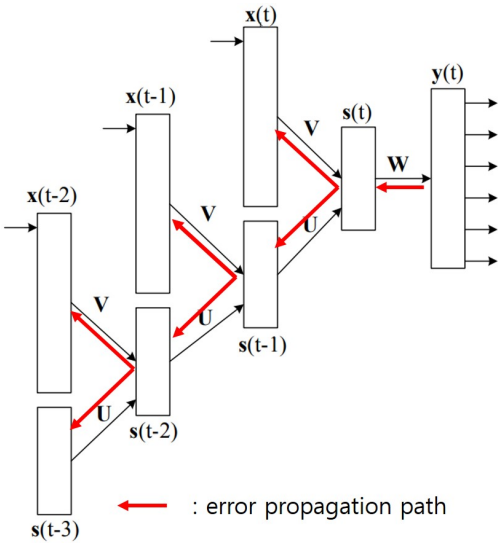

Figure 3.3: The backward propagation through time for the unfoleded recurrent neural network

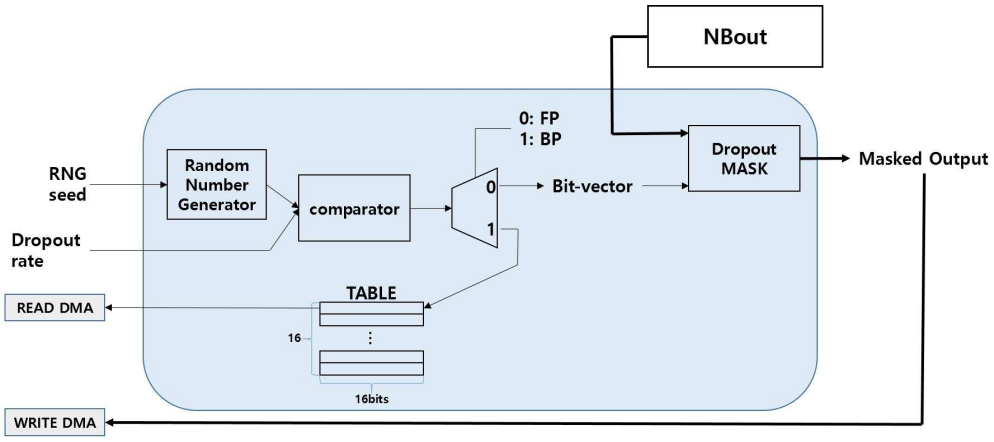

Figure 3.4: *Filter dropper* architecture

3.2 Selective Gradient Computation for Dropped Activations

3.2.1 Baseline Architecture

As same as the previous chapter, *DianNao* is selectd as a baseline architecture. It still does not have SRAM buffers for neurons and synapses. It consists of 16 MAC units and each MAC unit has 16 multipliers and the results of multiplications are accumulated using adder tree. So each MAC unit has 16 multipliers and 15 adders. With 16 MAC units, the baseline architecture can perform 256 MAC operations simultaneously. And the baseline architecture is assumed that the dropout procedure in the backward propagation is done just after the gradient computation. Similar to the merging convolutional layer and ReLU layer, masking gradients before they are written back to DRAM can save the DRAM accesses.

3.2.2 Filter Dropper

On the baseline architecture, a small logic called *filter dropper* is added to compute gradient selectively while keeping utilization of MAC units as high as possible.

3.5 shows the architecture of *filter dropper*.

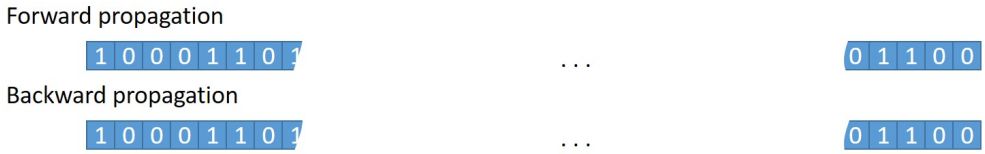

Figure 3.5: Random number genetator generates the same mask for both forward and backward propagation

Filter dropper consists of random number generator, comparator which compares random number generated by RNG and dropout rate, dropout mask, and table to keep unitization of MAC units high.

Random Number Generator

To dropout activations randomly, random number generator (RNG) generates random number for each activation to pick the dropped activations randomly. Because dropout rate is a tuning parameter for dropout that it is neccesary parameter for *filter dropper*. Further, random number generator seed (RNG seed) is also an important parameter for *filter dropper*. Unlike conventional dropout implementations, the proposed architecture does not need to store either dropout mask or masked activations. For a layer, if both the forward propagation and the backward propagation use the same RNG seed, random number generated by RNG is exactly the same that masks for both the forward and backward propagations are proven to be coincide without additional memory usage.

Comparator

Comparator compares random number generated by RNG and dropout rate to determine whether an activation is dropped or not. And the result is expressed as a bit. If the bit is '1', the corresponding activation is not dropped and if the bit is '0', the corresponding activation is a dropped activation.

For the forward propagation, 16 output activations are computed simultaneously by the fully connected layer because the number of MAC units is 16. So 16 bits are gathered to form a bit-vector. Generating a bit-vector takes only 16 cycles. The bit-vector generating cycles can be hidden to fully connected layer computation time. Because fully connected layer computation requires associate all input activations and weights for all output activations, computing fully connected layer is hard to be finished in 16 cycles.

For the backward propagation, a bit generated by comparator determines whether gradient computations for the corresponding activation is skippable or not. Similar to the *filter collector* in 2.2.3, to keep utilization of MAC units high, the bit is collected to table.

Dropout Mask

Dropout mask logic masks output activations using bit-vector. When output activations are about to written back to DRAM, they pass through dropout mask logic to be masked. The masking procedure is simple. Just multiplying output activations, bit-vector, and the scale factor. Then the masked outputs are written back to DRAM.

Table

Table is collecting '1' bits generated by RNG and comparator. When collecting '1' bits, the corresponding index for the activation is recorded into the table. The table can record up to 16 indexed that gradient computations for upto 16 non-dropped activations are done simultaneously to maximize the utilization of the accelerator.

When the table ends filling indexes, DMA requests for the corresponding weights of the recorded index are sent to the DRAM.

The pseudo code for the *filter dropper* is shown in Algorithm 1 and 2.

Algorithm 1: Pseudo code of *filter dropper* for forward propagation

```
1: Input: DropoutRate, RandomNumberSeed
2: Output: MaskedOutput[16]
3: // At the beginning of the computing NBout[16],
4: // Filter Dropper begin to generate bit-vector for masking NBout[16];
5: for i = 0 to 15 do
       Probability = rand() / RANDMAX ;
       if Probability < DropoutRate then
           bitvector[i] = 0;
       else
           bitvector[i] = 1;
6: // When NBout[16] are computed, then mask them using bitvector[16];
7: for i = 0 to 15 do
       if bitvector[i] == 0 then
           MaskedOutput[i] = 0;
       else
           MaskedOutput[i] = NBout[i];
8: // Write MaskedOutput[16] back to the DRAM
9: for i=0 to 15 do
       DRAMWriteRequest(MaskedOutput[i]);
```

Algorithm 2: Pseudo code of *filter dropper* for backward propagation

```
1: Input: DropoutRate, RandomNumberSeed
2: Output: Table[16]
3: TableIndex;
4: NeuronIndex = 0;
5: while NeuronIndex < NumberOfNeurons do
       TableIndex = 0;
6:     // Find 16 non-masked activations
7:     // and then recored their index (NeuronIndex) on the Table[16];
8:     while TableIndex < 16 do
           Probability = rand() / RANDMAX ;
           if Probability < DropoutRate then
               Table[TableIndex] = NeuronIndex; TableIndex++;
           NeuronIndex++;
9:     // Send DRAM read requests for corresponding filters of
       NeuronIndexes on Table;
10:    for i=0 to 15 do
           DRAMReadRequest(Table[i]);
```

3.3 Overall Architecture

Overall architecture is shown in Figure 3.6. *Filter dropper* is added to the baseline architecture. It also does not have input SRAM buffers. Due to the characteristics of the fully connected layer, weights are used exactly once that there are no chance for data reuse.

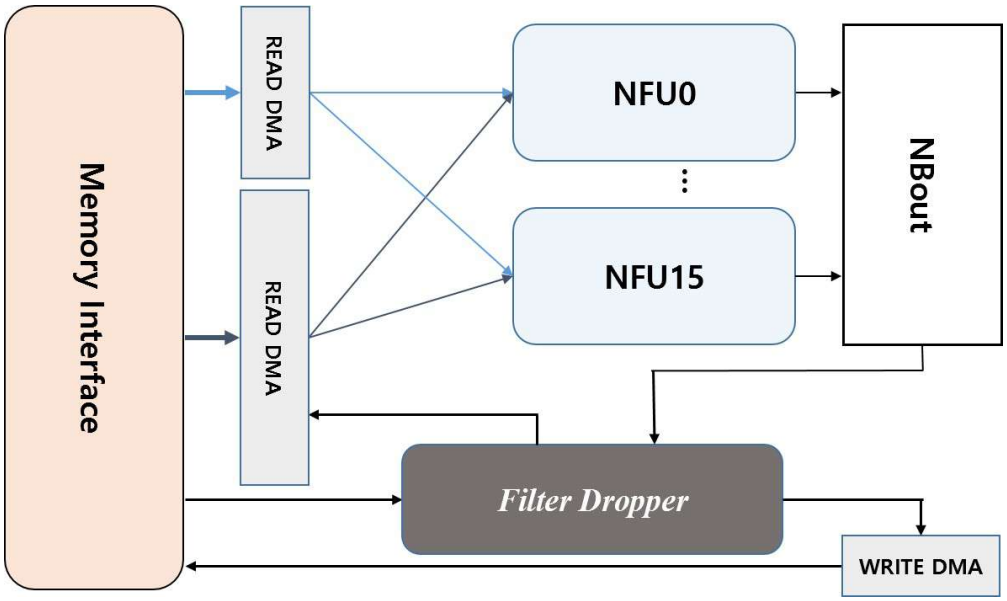

Figure 3.6: Overall architecture for the dropout accelerator

3.4 Experimental Results

3.4.1 Simulator and Benchmark

A cycle accurate DNN accelerator simulators both for the baseline and the proposed architectures are build to show the effective of the proposed approach. DRAMSim2 [20] is attached in the simulator that the simulator can measure the execution time including DRAM access time and the DRAM energy consumption. The experimental results are analyzed in related with the dropout rate.

As a benchmark, MLP network which has 2 hidden layers ($800 \times 800 \times 10$) is used and varying size of fully connected layer with varying dropout rate is used as a micro benchmark.

3.4.2 Results

3.1 shows the results for the effectiveness of the proposed architecture. To compare the results from the baseline and the proposed architecture, the normalized execution

Table 3.1: Simulation results for dropout accelerator architecture

Network Size	*Dropout rate*	*Normalized execution time*	*Normalized DRAM energy*
$800 \times 800 \times 10$ (MLP)	0.5	0.514	0.495
4096×1024 (MLP)	0.3	0.697	0.698
	0.5	0.479	0.499
	0.7	0.309	0.308
4096×4096 (MLP)	0.3	0.398	0.699
	0.5	0.494	0.495
	0.7	0.312	0.312

time and the normalized DRAM energy consumptioin is used. The results show that the performance and the DRAM energy consumption are proportional to the ratio of non-dropped activations. Because reduced DRAM accesses dominates.

3.5 Summary

This part of the dissertation proposes an idae of DNN accelerator architecture which can accelerate the dropout layer. We show both speedup and DRAM energy reduction on the dropout layer by simulation. The improvement is closely related to the dropout rate.

Chapter 4

Conclusion

This dissertation researches hardware architectures which can accelerate deep neural network training making use of the sparsity of neurons. Observing some layers where the sparsity is occured, we designed hardwares to accelerate theses layers during the DNN backward propagation.

First part of the dissertation focuses on convolutional layer. With the ReLU activation function, output neurons become sparse that explicit zero neurons are found. And the characteristics of backward propagation computation make is possible to skip many MAC operations with a zero neuron. The first hardware architecture is designed without SRAM buffer that it takes too long time due to the enoumous DRAM access. So solve this problem, the second hardware includes SRAM buffers for neurons and synapse to reuse these data. It improves absolute execution time and DRAM energy consumptions due to the reduced DRAM accesses. However it is still on a memory-bound that resizing SRAM buffer size is left as a future work.

Second part of the dissertation accelerates fully connected layer during the DNN training. Similar to ReLU layer, dropout layer can make use of sparsity of neurons. The sparsity is determined during the forward propagation that computing gradients for zero neurons are clearly waste of computing resources. Simulation results shows the effectiveness of hardware design. Due to the characteristics of the fully connected

layer, there a few rooms to improve performance and energy efficiency using SRAM buffer, applying SRAM buffer is left as a future work and combining the first part and this part also left as a future works.

Bibliography

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

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

[3] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[4] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[5] 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," *ACM Sigplan Notices*, vol. 49, no. 4, pp. 269–284, 2014.

[6] 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*. IEEE Computer Society, 2014, pp. 609–622.

[7] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 1–13.

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

[9] 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.

[10] 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*. IEEE Press, 2016, p. 20.

[11] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[12] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[13] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, 2015, pp. 3123–3131.

[14] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[15] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Advances in neural information processing systems*, 2018, pp. 5145–5153.

[16] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in neural information processing systems*, 2018, pp. 7675–7684.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1–12.

[18] 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*, 2016, pp. 770–778.

[19] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International Conference on Machine Learning*, 2015, pp. 448–456.

[20] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, 2011.

[21] A. Vedaldi and K. Lenc, "Matconvnet: Convolutional neural networks for matlab," in *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 2015, pp. 689–692.

[22] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 78–91.

[23] Z. Song, R. Wang, D. Ru, Z. Peng, H. Huang, H. Zhao, X. Liang, and L. Jiang, "Approximate random dropout for dnn training acceleration in gpgpu," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 108–113.

[24] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[25] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[26] T. Moon, H. Choi, H. Lee, and I. Song, "Rnndrop: A novel dropout for rnns in asr," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*. IEEE, 2015, pp. 65–70.

[27] S. Semeniuta, A. Severyn, and E. Barth, "Recurrent dropout without memory loss," *arXiv preprint arXiv:1603.05118*, 2016.

[28] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," in *Advances in neural information processing systems*, 2016, pp. 1019–1027.

초 록

심층신경망은 컴퓨터 과학의 다양한 분야 중 인간의 감각을 쫓는 분야에서 가장 중요한 기술이 되어왔다. 몇몇 분야에서는 이미 심층신경망의 도움으로 인간의 감각을 뛰어넘은 분야도 존재한다. GPGPU를 이용한 심층신경망의 가속이 가능해진 이후, GPU는 심층신경망에 있어 가장 주요한 장치로 사용되고 있다. 심층신경망의 복잡도가 높아짐에 따라 연산에 더 많은 컴퓨팅 자원을 요구하고 있다. 그러나 GPGPU는 에너지 소모가 크기에 효율적인 심층신경망 전용 하드웨어 개발에 대한 요구가 증가하고 있다. 현재까지 이러한 전용 하드웨어는 주로 심층신경망 추론에 집중되어 왔다. 복잡한 심층신경망 모델은 학습에 긴 시간이 들고 많은 에너지를 소모한다. 이에 심층신경망 학습을 위한 전용 하드웨어에 대한 요구가 늘어가고 있다.

본 학위논문은 심층신경망 학습 가속기 구조를 탐색하였다. 심층신경망의 학습은 순전파, 역전파, 가중치 갱신 이렇게 세 단계로 이루어져 있다. 이 중 액티베이션의 그래디언트를 구하는 역전파 단계가 가장 시간이 오래 걸리는 단계이다. 본 학위논문에서는 역전파 단계에 중점을 둔 심층신경망 학습을 가속하는 하드웨어 구조를 제안한다. ReLU 레이어 혹은 dropout 레이어로 인해 생긴 뉴론의 성김을 이용하여 심층신경망 학습의 역전파를 가속한다.

학위논문의 첫 부분은 합성곱 신경망의 역전파를 가속하는 심층신경망 학습 하드웨어이다. 가장 많이 쓰이는 활성화 함수인 ReLU를 이용하는 신경망을 가정했다. 음수 입력값에 대한 ReLU 활성화 함수의 도함수가 0이 되어 해당 액티베이션의 그래디언트 또한 0이 된다. 이 경우 그래디언트 값에 대한 계산 없이도 그래디언트 값이 0이 되는 것을 알 수 있기에 해당 그래디언트는 계산하지 않아도 된다. 이러

한 특성을 이용하여 0값인 액티베이션에 대한 그래디언트 계산을 건너 뛸 수 있는 효율적인 심층신경망 가속 하드웨어를 설계했다. 또한 실험을 통해 본 하드웨어의 효율성을 검증했다.

학위논문의 두번째 부분은 완전연결 신경망의 학습을 가속하는 하드웨어 구조 제안이다. ReLU 레이어와 비슷하게 dropout 레이어 또한 그래디언트 계산 없이도 그 결과가 0임을 알 수 있다. Dropout은 심층신경망의 과적합을 해결하는 일반화 기법 중 하나로, 심층신경망 학습 과정 동안에만 무작위로 신경망의 연결을 끊어 놓는다. 신경망이 끊어진 경로로는 역전파 단계에서 에러가 전파되지 않기에 해당 그래디언트 값 또한 0임을 미리 알 수 있다. 이 특성을 이용하여 완전연결 신경망의 역전파를 가속할 수 있는 하드웨어를 설계했다. 또한 시뮬레이션을 통해 본 하드웨어의 효율성을 검증했다.

주요어: 심층신경망 학습, 뉴론의 성김, 선택적 그래디언트 계산
학번: 2014-21649