Master’s Thesis

Minimizing Required Data in a Neural Network Accelerator Using Double-Stage Weight Sharing

Taekoan Yoo

Department of Electrical and Computer Engineering

The Graduate School

Sungkyunkwan University

Master’s Thesis

Minimizing Required Data in a Neural Network Accelerator Using Double-Stage Weight Sharing

Taekoan Yoo

Department of Electrical and Computer Engineering

The Graduate School

Sungkyunkwan University

Minimizing Required Data in a Neural Network Accelerator Using Double-Stage Weight Sharing

Taekoan Yoo

A Master's Thesis Submitted to the Department of
Electrical and Computer Engineering
and the Graduate School of Sungkyunkwan University
in partial fulfillment of the requirements
for the degree of Master of Science in Engineering

October 2019

Approved by

Jong Tae Kim

Major Advisor

This certifies that the master's thesis
of Taekoan Yoo is approved.

심사위원장 signature

Committee Chair:

심사위원 signature

Committee Member:

지도교수 signature

Major Advisor:

The Graduate School
Sungkyunkwan University
December 2019

Contents

List of Tables

List of Figures

Abstract

Minimizing Required Data in a Neural Network Accelerator Using Double-Stage Weight Sharing

Recently, the size of deep neural networks has increased, and the use of fully connected layers has been increasing. However, fully connected layers require many parameters and cannot reuse weight like convolutional layers do, so they require significant amounts of on-chip storage (SRAM).

Moreover, the demand for practical hardware that can efficiently process fully connected layers in an embedded device has been increasing for edge computing; however, in various papers, the size and the power dissipation of neural network accelerators are dominated by SRAM. To minimize the data required for efficiently utilizing SRAM, a technique called double-stage weight sharing is proposed in this paper. Through this technique, the required weight data was minimized, and the accuracy was compensated to be comparable to previous model compression techniques. This paper also presents a custom accelerator optimized for the minimal data that results from this process.

For a 32 nm cell library, the area of our design was reduced by 50.23% and the performance is expected to be 60% higher than with a representative accelerator employing the typical weight sharing technique. Consequently, the performance-area efficiency has been improved by 3.21 times with highly reduced weight information.

Keywords: quantization, hashing trick, weight sharing, neural network accelerator

1. Introduction

Recently, Deep Neural Networks have been adopted for computer vision, speech recognition, natural language processing (NLP), and various other domains. However, a neural network with high accuracy requires from 1 million to 10 billion parameters or weights, and the demand for fully connected (FC) layers, which cannot be reused (unlike convolutional (Conv) layers), has been increasing [43]. Thus, resource constraints and the workload involved have become bottlenecks to better performance.

In particular, embedded devices with strict constraints require high accuracy for various applications such as NLP or advanced driver assistance systems (ADAS). To overcome this limitation, a special hardware neural network accelerator (NNA) was introduced to accelerate the inference time when working with an application processor (AP) [1−9]. NNAs are a kind of vector processor of which the multiply−accumulate (MAC) operation is optimized for processing neural networks, and NNAs can be divided into two types. One type is implemented for Conv layers and the other is implemented for FC layers.

With the emergence of NNA, various methods for improved efficiency in implementation and optimization have been proposed. For example, to execute a Conv layer, memory utilization is relatively efficient because of the reuse of parameters, but the number of operations is too large, so various methods for efficiently processing the heavy workload have been proposed [6–9]. On the other hand, when it comes to an FC layer, there are too many parameters that cannot be reused, so methods for resolving memory (DRAM) bandwidth problems or model compression have been proposed [1]. As the application of convolutional neural networks (CNNs) has increased, there has been a great

deal of state-of-the-art research for accelerating a Conv layer to improve performance efficiency. However, there are not many studies about accelerating FC layers, which are commonly used for popular applications such as NLP [43].

To utilize memory efficiently in an embedded device, many model compression techniques have been created to compress a neural network to a small size. These techniques can be more efficiently applied to FC layers not only because FC layers require much data in spite of relatively small computation, but also because model compression techniques are experimentally effective for FC layers. For example, the pruning technique [10], which is used to convert the original matrix to a sparse matrix by disconnecting the weight connection, is more effective for FC layers (Conv layer: ~70%, FC layer: ~90% [1]). Moreover, the quantization technique, which is used to convert original parameters to approximate values while giving up only a small amount of accuracy, is also more effective for FC layers.

To utilize fully the compressed data in an NNA, it is necessary to implement hardware depending on how the data has been made and which techniques have been used. For example, EIE [1] tried to put all parameters into the on-chip memory (SRAM) to eliminate DRAM access and thereby resolve the DRAM bandwidth problem. Thus, to minimize the required memory size, EIE utilized data made with techniques introduced in a deep compression [11] paper. The paper first employed a pruning technique and expressed sparse data with an interleaved compressed sparse column (CSC) format. In addition, the paper employed a non-linear quantization technique that produces shared parameters. To process data, EIE used suitable units such as pointer SRAM and a "sparse matrix read unit" that fully utilized the sparse matrix format. EIE also used a look-up table (LUT) for shared weights.

However, because an FC layer requires using a great deal data and cannot be reused, most of the accelerators use SRAM of large size. Moreover, most of the power dissipation and area result from using SRAM. For example, 93% of the area and 59% of the power are consumed by the SRAM in EIE [1], and the on-chip buffer occupies up to 70% of the chip area in TETRIS [12] even though (unlike EIE) it has DRAM. That is, this increases the cost and decreases the computational density. Because SRAM is used inefficiently in basic computing units called processing elements (PEs), the chip area greatly increases according to the number of PEs. Furthermore, because of the SRAM chip area, the performance-area efficiency has not been well considered.

In this paper (as with EIE [1]), the focus is on putting all data for an FC layer into on-chip SRAM without DRAM access. However, to use memory efficiently and consider the performance-area efficiency, in this work, parameters are compressed using double-stage weight sharing with a hashing trick [13] and index mapping, which are model compression techniques that do not degrade accuracy and performance.

The rest of this paper is organized as follows. In Chapter 2, related work is introduced according to the techniques employed. In Chapter 3, we show how we trained a neural network with double-stage weight sharing and why we used the hashing trick in the first stage. In Chapter 4, design exploration to find suitable architecture is introduced and the experimentation and evaluation related to our model is shown in Chapter 5. Finally, we present conclusions from the work reported in this paper in Chapter 6.

2. Related Work

In this chapter, the work related to that in this paper is introduced starting with large categories then discussing exact techniques. First, in Chapter 2.1, we divided original accelerator optimization techniques into three types [14]: algorithm, datapath, and model optimization. In Chapter 2.2, representative model compression techniques for model optimization is provided according to six categories: sparsity regularizers and pruning, quantization, knowledge distillation, low rank factorization, structural matrices, and compact convolutional filters. In Chapter 2.3, the quantization techniques employed in this study, are classified into two key methods: representation and training methodology. Finally, in Chapter 2.4, accelerators employing the parameter sharing quantization technique is described.

2.1 NNA Optimization

A neural network accelerator is a vector processor optimized for neural network computation, and there are three ways of optimizing NNAs. The following is about how state-of-the-art NNAs were optimized depending on the algorithms, data paths, and neural network models involved.

Before beginning the explanation, we should know that the basic components of an NNA include a control unit, memory control unit, and computational unit (CU) [15]. The CU controls all the other units in the middle, and the memory control unit preprocesses data from external memory (DRAM) or efficiently distributes and temporarily stores data. Last, the CU is the most basic

computing unit and consists of a bunch of processing elements (PEs). The following explanation will contain references to the previously described units.

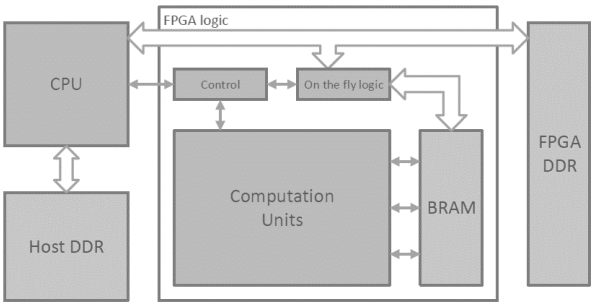

Fig. 2-1. Block graph of a typical FPGA based NNA [15]

A. Algorithmic Optimization

Algorithmic optimization is a process for applying algorithms appropriate for reducing the workload or vectorizing implementations for higher performance. For example, the general matrix multiplications (GEMM) transformation [16] is used for the data aligned along the batches or on another basis: the aligning method for images is called Im2Col. Fig. 2-2 shows GEMM operation.

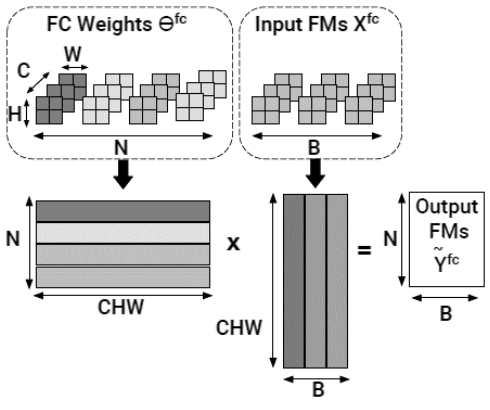

Fig. 2-2. GEMM operation [14]

Moreover, the Winograd transformation [17] can be applied when the stride of convolution is 1 and the filter size is small enough. The transformation converts the original matrix with A, B, and C (the transform matrices). To reduce the arithmetic complexity, a fast Fourier transform (FFT) can also be used in cases requiring a larger filter size.

B. Datapath Optimization

Datapath optimization varies the architecture of the CU or relocates a group of PEs in the CU to reuse data more efficiently or to distribute data to the PEs. For example, if an accelerator does not use data expressed in the sparse matrix format, Conv layer parameters can be executed with a 2D pipeline architecture [4,18]. Otherwise, accelerators can employ a single input multiple data (SIMD) structure to execute data in parallel [1−3].

A SIMD structure is implemented depending on which method was used to make the data. For example, the data tiling technique is used when an accelerator computes a group of parameters simultaneously according to SRAM size, and the loop unrolling technique is used to divide a long loop into small loops to be executed in parallel. In addition, the roofline model [19] helps find an optimized model for maximizing resource utilization and minimizing memory access by analyzing the computation and memory bounds.

Last, there are a number of techniques for reusing parameters to use memory more efficiently: weight stationary (WS), output stationary (OS), no local reuse (NLR), and row stationary (RS). Among these techniques, the row stationary technique is known as the best optimized technique for reusing Conv layer parameters.

C. Model Optimization

Model optimization is a technique that basically changes the neural network model to a more optimized one by training with a new training method. The most representative example is the network compression technique, including pruning [10] and quantization. Network compression is used to reduce resource or computation complexity by compressing the original neural network model. Details of this process will be covered in Chapter 2.2.

Name	EIE [1] (ISCA'16)	Cambricon-X [3] (MICRO'16)	Cnvlutin [20] (ISCA'16)	Eyeriss [4] (ISCA'16)	SCNN [2] (ISCA'17)	TPU [21] (ISCA'17)	Sparsecore [22] (sysml'18)
Target	FC	FC/Conv	Conv	Conv	Conv	FC/Conv	Conv
Dataflow	Inner Product (A+W)	Inner Product (W)	Vector Scalar + Reduction (A)	Row stationary	Cartesian Product + Input stationary (A+W)		Vector Scalar (A+W)
PE type	SIMD	SIMD	SIMD	Systolic	Systolic	Systolic	SIMD
Sparse	Interleaved CCS format	Step indexing	ZFNAF format	X	Interleaved CCS format	X	CCS format
Quantized	Weight Sharing	Low bit (16-bit)	Low bit (16-bit)	X	Low bit (16-bit)	X	X

A: Activation, W: weight

Table 2-1. Comparison of recent accelerators

Various accelerators have been optimized for neural networks using the previous three optimization techniques. They are implemented using a field programmable gate array (FPGA) or application-specific integrated circuit (ASIC). Table 2-1 compares the various accelerators by dataflow and model compression techniques: Sparse and Quantized.

2.2 Neural Network Model Compression

Embedded devices such as mobile devices have limited resources (power, memory, and area), so a variety of model compression techniques, which reduce workload or data without accuracy degradation, have been proposed for them [23]. Below are descriptions of six typical model-compression categories.

A. Sparsity Regularizers and Pruning

Pruning [10] converts the original weight matrix into a sparse matrix to reduce the data required by trimming redundant connections that do not contain information. This reduces network complexity and avoids the network overfitting problem. For example, a network was trained using deep compression [11] along with pruning. EIE [1] presented a novel NNA to utilize the compressed data in sparse format: interleaved compressed sparse column (CSC) or interleaved compressed sparse row (CSR).

B. Quantization

Quantization reduces the weight parameter size using approximate values.

Quantization is the core technology of this paper, so it will be explained in detail in Chapter 2.3. Quantization techniques are broadly divided into two groups: weight sharing (parameter sharing) and low bit-width quantization.

C. Knowledge Distillation

Knowledge distillation, also called the teacher-student mechanism, evolved from ensemble learning. The teacher model and student models are both trained with the same dataset, but the student model is an ensemble of teacher models so that the teacher models give direction. Fig. 2-3 shows how knowledge distillation operates between the two models.

In a typical paper about using this technique [24], it was suggested to distill parameters that are used only for training, to improve accuracy and reduce the network size.

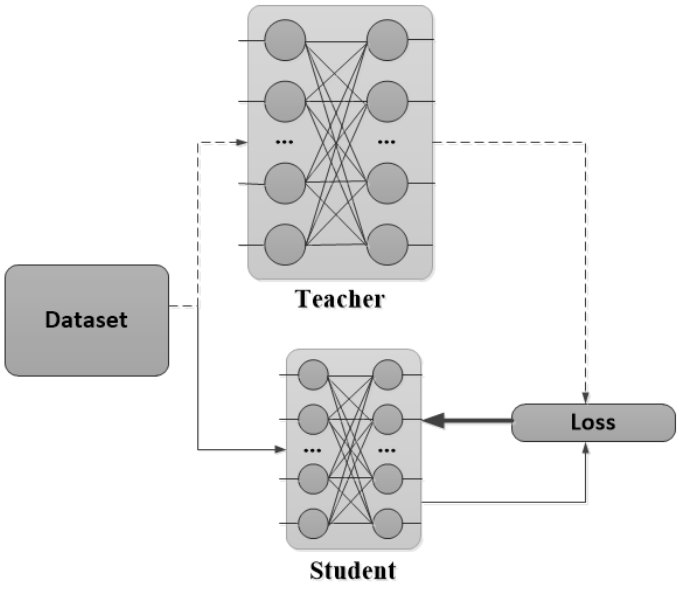

Fig. 2-3. Teacher-Student model

D. Tensor Decomposition

A Conv layer requires relatively too many operations, so it is necessary to reduce computation complexity. Tensor decomposition divides original matrix multiplication into several smaller matrices so as to reduce computation and improve performance.

Low rank regularization [25] is a newly proposed technique used to convert the original matrix into new matrices with fewer ranks using batch normalization to transform hidden neurons. Training with this algorithm from the beginning not only improved performance but also improved accuracy.

E. Structural Matrices

An FC layer has too many parameters (unlike a Conv layer) and it does not reuse parameters. Thus, a new technique is required to use memory more efficiently. Structural matrices could convert the original matrix into another matrix that has repetitive values using non-linear transformation. With this approach, the amount of data required could be reduced.

For example, circulant projection [26] used a circulant matrix to improve memory utilization without degradation of accuracy.

2.3 Quantization

As we mentioned, among model compression techniques, quantization has been suggested for representing exact weight values with approximate values. For quantization, two questions must be answered to replace the original matrix.

- How many values will we use to express the matrix?
- How precisely will we express a value?

Depending on the answers to the previous questions, quantization techniques are divided into two groups: low bit-width quantization (fixed point quantization) [27] and weight sharing (parameter sharing) [11][13][28][29]. That is, low bit-width quantization uses approximated, low-precision values in the neural network. Low bit-width quantization mostly uses fixed points, so it is also called fixed point quantization.

With the parameter sharing technique, the same weight is shared within the centroids array. It makes a difference whether centroids are shared by index, made by k-means clustering, or shared by hashing. Two methods can be used simultaneously, so we can use low bit-width while we use parameter sharing. Fig. 2-4 shows the difference between parameter sharing and low-bit width training, and the reason why they differ depending on the two answers above.

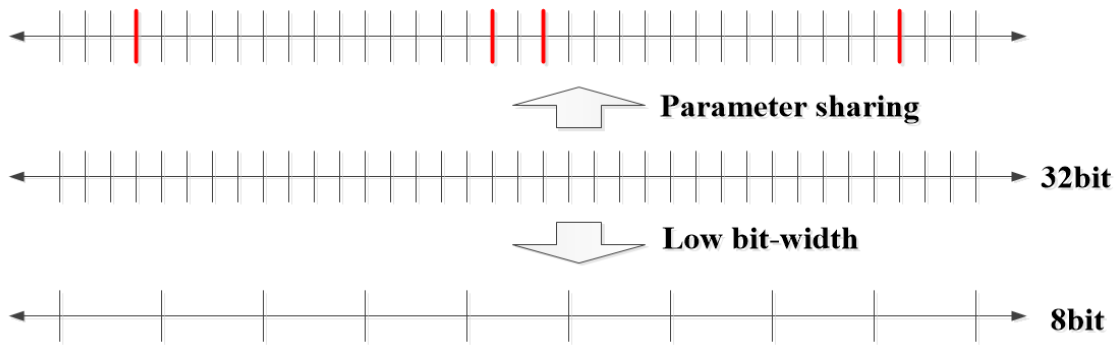

Fig. 2-4. Parameter sharing and low bit-width quantization

We can also divide the training techniques into two groups depending on the training methodology, and we will refer to these groups as pretrained network

and from-scratch training. The pretrained network approach involves skill training in which the first network training is done with full precision and then the pretrained network is quantized. From-scratch training involved skill training that uses the quantization of the from-scratch method, but without the pretrained network.

This chapter will show four quantization techniques depending on two classification standards with various examples. Table 2-2 shows the various quantization techniques.

Name	Deep Compression [11] (ICLR'16)	Hashing Trick [13] (ICML'16)	DoReFa-Net [35] (arxiv'16)	XNOR-Net[34] (ECCV'16)	Fixed Point Quantization [27] (ICML'16)
Target	FC/Conv	FC	FC/Conv		
Post/scratch	post	scratch	scratch	scratch	post
Sparse	CCS format	x	x	x	x
Quantization	Weight Sharing	Weight Sharing	Low bit-width	Low bit-width	Low bit-width

Table 2-2. Quantization Techniques

A. Low bit-width Training with Pretrained Network Quantization

One example of a technique that changes a pretrained 32-bit precision network into a low bit-width network is fixed point optimization [30]. The technique used in this paper is the most intuitive method. The width of bits in each layer is selected to minimize the L2 error.

Another example, ternary residual networks [31] changes pretrained parameters into one of three values: +1, 0, or −1. Two techniques are used at the same time. First, they changed the pretrained network using fine−grained quantization (FGQ), and applied "residual edge" to parameters with large quantization error, using the sum of differences between the original matrix and the quantized matrix.

B. Low bit−width Training with From−scratch Training

This technique is what most people first think of when they consider quantization. Typical examples include binarized neural networks (BNN) [32], quantized neural networks (QNN) [33], XNOR−net [34], and DoReFa−Net [35]. BNN expresses all networks using 1−bit values, and QNN did so with 2−bit values. XNOR−Net was proposed to lessen accuracy loss using 1−bit values. DoReFa−Net used low bit−width not only for weights and activations, but also for gradients.

Furthermore, "numerical precision" [36] made 16−bit fixed points by stochastic rounding and the study results showed that the loss of accuracy with their model was close to zero. They also implemented NNA as well.

C. Parameter Sharing with Pretrained Network Quantization

This technique is used to train pretrained 32−bit networks by parameter sharing. Mostly pretrained networks were firstly processed with k−means clustering to make shared centroids and thereby reduce quantization error with post−training.

The most representative choice, deep compression [11], uses an index for parameter sharing. As mentioned, a pretrained 32-bit matrix were clustered with k-means clustering method to make shared centroids. Then, they were post-trained several times using custom back propagation. Fig. 2-5 shows how parameter sharing can be implemented using an index matrix and centroid vectors with real weights.

Fig. 2-5. Deep Compression [11]

Similarly, there are other methods that change the original matrix using fixed-point quantization [27] or vector quantization [29]. Fixed-point quantization determines the parameters shared by two variables: bit-width and step size. They have the following relation.

$$\text{Range} \approx 2^{bit-width} * \text{Stepsize}$$

For example, Fig. 2-6 shows examples of bit-width and step size. To express four values, the bit-width is $\log_2 4 = 2$ and step size is the distance between the values (that is, 0.996). The pattern of the pretrained data determines the two variables, and they can share weights determined using the previous two variables.

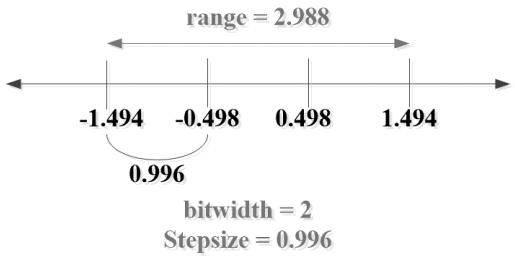

Fig. 2-6. Example of bit-width and step size

D. Parameter Sharing with From-scratch Training

Training with quantization from-scratch is not that intuitive, so we need an elaborate training methodology to compensate for this ambiguity.

The hashing trick [13] shares centroids using a hash function (as in Fig. 2-7). The authors trained a network from-scratch with random sharing, and revised the centroids by custom back propagation. Because of the random sharing attribute, this technique requires elaborate training skill to achieve accuracy. The hashing trick technique is one of the double-stage weight sharing techniques employed in this paper.

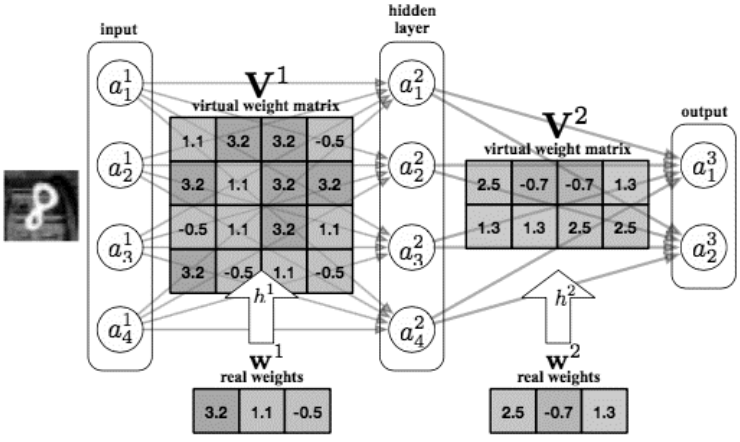

Fig. 2-7. Hashing Trick Example [13]

2.4 NNAs Employing Weight Sharing

In cases of an NNA employing weight sharing, EIE [1] is representative, as was mentioned in Chapter 1. The biggest features of weight sharing for an accelerator are a large number of look-up operations and a large look up table (LUT) [37].

For example, a new framework was proposed [38] in which clustered pretrained data was created considering the required accuracy and limitations of the platform and it even synthesized adequate NNA. Moreover, with PQ-CNN [39] a new integrated-design method was proposed by which to consider both software and hardware. This way, it could reduce entries of centroids so that computation complexity was decreased. Last, the Q-CNN framework [40] used a newly proposed way of quantization that improved memory efficiency and minimized estimation error. The sub-codebook used in this paper was made with weight sharing and potential output was preprocessed.

Furthermore, BHNN [4] did not specifically propose a structure for NNA, but they proposed the "blocked hashing strategy" to overcome the disadvantage of random access to memory in the hashing trick.

3. Training Methodology

In this chapter, we describe how we trained a neural network using weight sharing. First, in Chapter 3.1, we explain why the hashing trick [13] was employed for minimizing the amount of data required. In Chapter 3.2, a neural network model using double-stage weight sharing to minimize the amount of

data required is introduced, and we present how we trained our model in three steps.

3.1 Data Conversion

As can be seen in Table 2-1, there are various NNAs optimized with the model compression technique. This paper employed a weight sharing technique for quantization, and we used it in two stages, one of which was the hashing trick [13]. The latter was used to compress the weight parameters and improve performance-area efficiency when we put all of the parameters into the on-chip SRAM as with EIE. In this chapter we show the reason why we employed the hashing trick instead of other model compression techniques and why we thought the hashing trick is the technique that requires the least information. Fig. 3-1 shows how the data format changes when we apply different model compression skills.

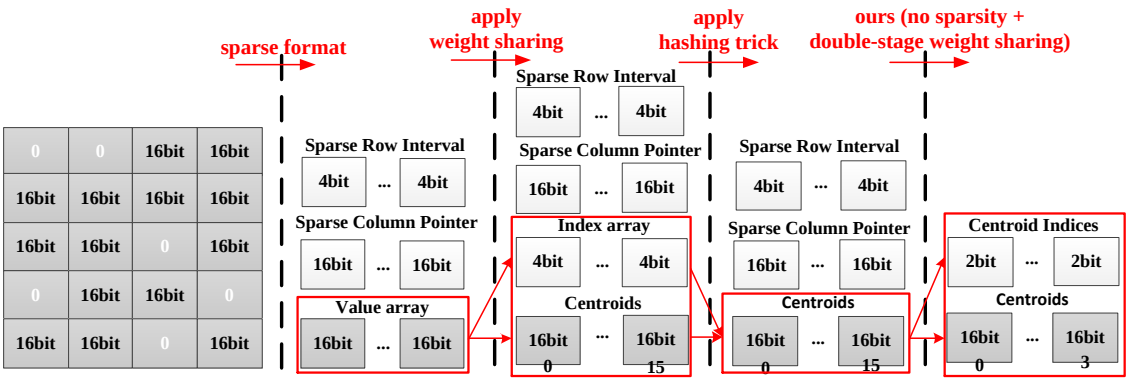

Fig. 3-1. Data format (sparse matrix > apply CSC format > apply weight sharing > apply hashing trick > our format)

A. Sparse Format

The first picture in Fig. 3-1 is the original sparse matrix. When the sparse format is applied, the matrix can be changed into three arrays as in the second picture in Fig. 3-1. The given format is compressed sparse column (CSC), which was aligned along the columns. The value array is non-zero values. The sparse column pointer is for pointing to the index of the value array where a column is changed. The "sparse row interval" is the number of zeros before the value. When it comes to the interleaved CSC format used in the EIE [1], if the number of zeros overflows the given bit-width, another zero is put into the value array.

For example, assume that the original 16x16 matrix has 16-bit precision and that the sparsity of the matrix is 50% (which is known as maximally 90%). The original matrix has

$$16 * 16 * (16\text{-bit}) = 4{,}096 \text{ bits}$$

as required data. Moreover, when we apply the sparse format to the original matrix,

$$16 * 16 * (\text{sparsity } 0.5) * (4\text{-bit}) + 16* (16\text{-bit})$$
$$+ 16 * 16 * (\text{sparsity } 0.5) * (16\text{-bit}) = 2{,}816 \text{ bits}$$

is the data required for the sparse format. This way, the sparse matrix can be represented with fewer parameters.

B. Shared Weight Format

When we apply a typical weight sharing technique, the situation shown in the second picture in Fig. 3-1 can be changed into that in the third picture. For

example, deep compression [11] used typical weight sharing, and it changes the value array in the second picture into centroids and the index array for the indices of centroids. By using typical weight sharing, these two arrays are smaller than the original value array.

For example, when we have the same condition we assumed before and 16 centroids for weight sharing (see EIE [1]), 2,816 bits of data can be reduced to

$$16 * 16 * (\text{sparsity } 0.5) * (4\text{-bit}) + 16 * (16\text{-bit})$$
$$+ 16 * 16 * (\text{sparsity } 0.5) * (4\text{-bit}) + 16 * (16\text{-bit}) = 1{,}536 \text{ bits}$$

As you can see, with centroids and index array, the data required can be compressed.

C. Hashing Trick

With the hashing trick [13], the indices of the centroids are not needed anymore because we use the hash function for weight sharing. Thus, as shown in the fourth picture in Fig. 3-1, the index array can be removed.

For example, when we assume the same condition we assumed before, the data can be reduced as indicated below.

$$16 * 16 * (\text{sparsity } 0.5) * (4\text{-bit}) + 16 * (16\text{-bit})$$
$$+ 16*16*(\text{sparsity } 0.5) * (4\text{-bit}) + 16 * (16\text{-bit}) = 1024 \text{ bits}$$

We can conclude that the hashing trick requires the least weight data. Furthermore, this shows the potential for saving bandwidth when we take parameters from DRAM, and we can reduce the size of the SRAM when we put all the parameters into the on-chip SRAM.

D. Double-stage weight Sharing

There were two possible problems with using the hashing trick. First, for the hashing trick, there are too many centroids for comparable accuracy. This is because, when we use random mapping, the mapping is fixed and just randomly mixed, so it is hard to express the original matrix with shared weight when the number of centroids is small. Thus, we propose double-stage weight sharing, as illustrated in the last picture in Fig. 3-1.

Second, to minimize the required data, we explored various designs to employ different designs according to sparsity and parallelism. We found that employing sparsity of the weight matrix cannot optimally minimize the data required in an NNA. Consequently, we decided to give up sparsity information and compensate performance with parallelism. Therefore, we removed the "sparse row interval" and "sparse column pointer" as you can see in Fig. 3-1. This will be addressed in Chapter 4 in detail.

Of course, the previous numerical explanation is different depending on how many centroids and indices of the centroids we need for comparable accuracy, but we can still intuitively understand that the required data was minimized as in Fig. 3-1.

3.2 Double-Stage Weight Sharing for an Accelerator

There are training requirements for our accelerator. First, we introduce the factors we needed to take care of when applying the hashing trick [13] in an NNA. Then, we talk about how we trained the neural network that was used as the accelerator.

A. Hashing Trick for Hardware

There are several issues to resolve before employing the original hashing trick [13]: the random access problem, the bulky hash function, and too many centroids. The first two issues are addressed in this chapter and the other is addressed in the next chapter.

First, because the original hashing trick used a large number of centroids when it was parallelized, there was a problem with random access to memory. By dividing this memory into several blocks, the number of which corresponds to the number of PEs, this problem was rectified, and this was proven in BHNN [41]. Fig. 3-2 shows how blocked hashing was applied in the weight matrix, and how matrices in our neural network were divided (as follows).

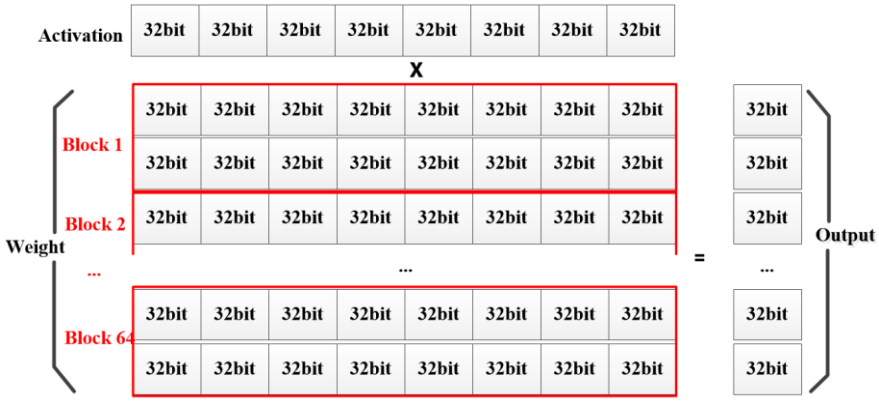

Fig. 3-2. Blocked hashing for efficient memory

Second, the original hashing trick used XXhash, but it contained too many multiplications. Moreover, each PE contained 16 hash units, so the area could not be optimized. We tried to find an alternative random hash function that could randomly map two 16-bit position values to a 10-bit centroid index. Among

the features of the hash function, confusion is less important, because our objective is to uniformly map indices to the centroid index (diffusion). Moreover, the cryptographic attribute seems redundant to us. This is addressed in Chapter 5 as well.

B. Accuracy Compensation: Double-Stage Weight Sharing

The most important feature we need in the hashing trick is the random mapping feature. However, depending on the position, the indices of centroids are statically shared, so that it cannot represent a feature of the matrix with a small number of centroids. Because of this problem, the original hashing trick has too many centroids for comparable accuracy: 98,000 centroids for a 784 x 1,000 matrix.

To overcome this problem, we proposed double-stage weight sharing with indices. We first scattered shared weight using the hashing trick with enough centroids, and we clustered them as in the index shown in Fig. 3-3. Interestingly, we found that this data representation can achieve comparable accuracy only when using a small number of real centroids. Consequently, with mapping from hashed output to the index of real centroids, we were able to reduce the amount of data required and compensate for any loss of accuracy.

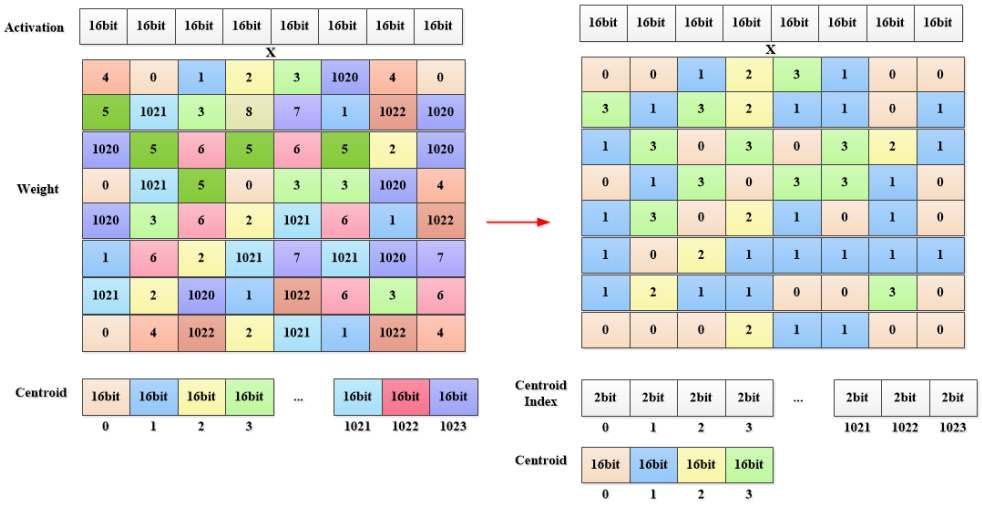

Fig. 3-3. Double-stage weight sharing

C. Training Methodology

We trained our neural network using double-stage weight sharing, including the hashing trick in three steps: pre-training, training, and post-training. As we mentioned in Chapter 1, our target was the FC layer and we trained the MNIST dataset using only FC layers. The neural network consisted of 1,000 hidden layers, so the synapse matrices consisted of 784×1,000 and 1,000×10 weight, that is, almost a million parameters.

The first step is pre-training with a normal neural network to initialize the neural network in the second step. Once the neural network is trained, it can be reused for the next step in the training.

The second step is fundamental training for the neural network with the hashing trick. We first made an index matrix, which contains the centroid indices, with a hash function. When indices in the matrix are the same, they were

initialized with a mean value of themselves. Moreover, during training, gradients are also used with a mean value for the same indices and applied equally applied. Thus, the gradient for index i and j of a matrix of which the hashed output is K, is as follows in l layer

$$\frac{dL^l}{dW_{h(i,j)=K}} = \sum \frac{dL^l}{W_{h(i,j)=K}} \; / \; num(W_{h(i,j)=K})$$

The last step is for using the index. The neural network is initialized with the result network of the second step and k−means clustering was done for the array of centroids to improve accuracy and reduce the number of centroids. After clustering was applied, we had a new index matrix for training and several hashed outputs, which had different indices before, were now clustered and had the same index. The method for applying gradients is the same as with the second step.

D. Post−Training with Pruning

This chapter is only for Chapter 5 to show the effectiveness of pruning for double−stage weight sharing. After shared weights were scattered with the hashing trick in the second step, we applied K−means clustering to cluster a lot of centroids, and we masked N% of the smallest centroids, when N was the sparsity of weight. The ratio of sparsity was steadily raised until we reached the planned sparsity, and we did train with pruning for five times.

However, even though we gave N% sparsity, sometimes we could not actually make an N% sparse matrix. This is because sometimes N% of the matrix values cannot share weight; however, through experimentation we found that our neural network can get close to the target.

4. Design Exploration

We have explained how we made the data by training with double-stage weight sharing. In this chapter, we show how we selected the design of the data accelerator. First, in Chapter 4.1, we review the EIE structure [1] because we experimented with our design considering EIE as a baseline. This is because, as you can see in Table 2-1, EIE is a typical NNA employing weight sharing quantization. In Chapter 4.2, we show the factors we considered before we explored various designs and used a post-multiplication method to overcome certain problems. In Chapter 4.3, we show various models according to certain factors, and show how the final model was chosen. Finally, in Chapter 4.4, we explain our final model in detail.

4.1 Baseline NNA for EIE

For sparse matrix vector multiplication (SPMV), NNA has many basic computing units or PEs. How engineers design the PEs is the main feature of an NNA, so we needed to design elaborate PEs. In this chapter, we first reveal the design of PEs in a typical accelerator EIE [1], which employed weight sharing, and then we show the simplest way to implement the hashing trick with it.

A. PE in EIE

The PE in a representative accelerator is illustrated in Fig. 4-1. A leading non-zero detection (LNZD) node is used to employ the sparsity of input

activations. According to the picture, PE has 5 kinds of storage: pointer SRAM, sparse matrix SRAM, LUT, output registers, and read/write SRAM. Pointer SRAM is for the sparse column pointer in the third picture of Fig. 3-1, and it has 32 KB in each PE of the EIE. In addition, sparse matrix SRAM is for the sparse row interval and index array, both of which have 4-bit precision (shown in the third picture of Fig. 3-1), and they have 128 KB. The LUT and output registers are implemented using a register file. Because it was assumed that there were 16 centroids, the LUT has 256-bit registers. Furthermore, the read/write SRAM is for computing operations of which the output requires more registers for output registers (it has 2 KB). In all, the EIE has 162 KB SRAM for 1 PE.

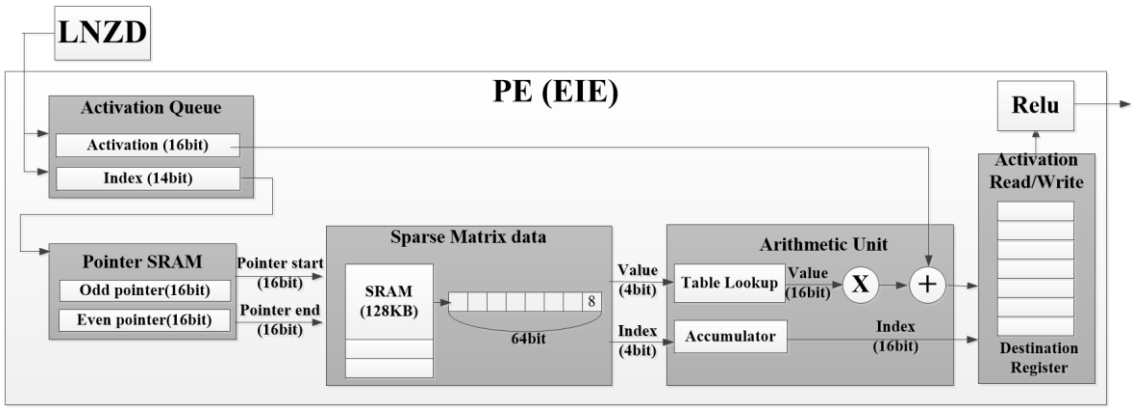

Fig. 4-1. Structure of a PE in EIE [1]

B. Simplest Way to Apply the Hashing Trick

As illustrated in Fig. 3-1, the simplest way to apply the hashing trick is just removing the index array for centroids. When we apply it to the original EIE [1], as illustrated in Fig. 4-2, the hashing accelerator has a new hash unit and it

must be located before the accumulation unit for the row index. Moreover, the size of the original Sparse Matrix SRAM was reduced by 64 KB because the index array could be removed. Now, a PE requires 98 KB SRAM and it is now 63.1% of the original size in the EIE. Considering the pipeline structure, area, and power, it is expected that the reduction will occur without performance degradation. When we implemented this design, there was a 29.8% area reduction, so if the hash unit can fit the frequency of the original EIE, the performance-area should be improved by 42%.

However, a hashing trick with 16 centroids in each PE cannot satisfy comparable accuracy and the hashing trick cannot employ such a large sparsity of weight because they are randomly shared. Thus, this structure cannot improve performance-area efficiency and cannot be applied for our objective. This will be further explained in Chapter 4.2.

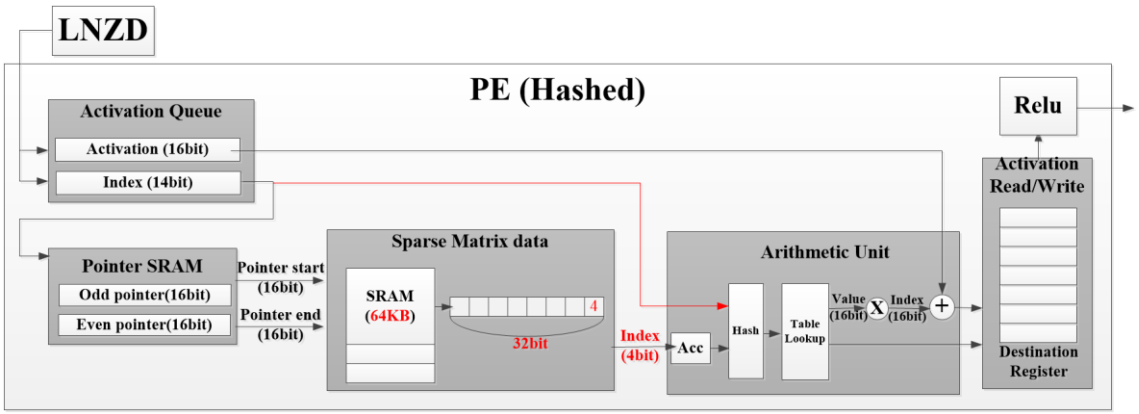

Fig. 4-2. Simplest Design for the Hashing Trick

4.2 Design Consideration

We found out that the simplest design shows the potential for 42%

improvement of performance-area (relative to EIE) by applying the hashing trick; however, it could not be applied for our objective. In this chapter, we will show the two factors that we took into account to find the optimal design: sparsity and parallelization. Moreover, there is a problem with parallelization when using the weight-sharing technique, so we will show how we solved this problem.

A. Key Factors: Sparsity and Parallelism

The representative NNA of EIE [1] employed both sparsity of weight and activation without parallelization, because the PE could not be parallelized because of the large amount of memory required. However, as we addressed data conversion in Chapter 3, we had to consider not employing sparsity to minimize required data and instead, to compensate the performance with parallelization. The feasibility of employing the sparsity of weight or activation was determined by which way we chose to parallelize the data, so we explored various designs regarding these two factors. In this chapter, we show what the two factors are and they are called PARALLEL and SKIP.

- Intra-PE parallelization (PARALLEL):

  Using many PEs (Inter-PE parallelization) to improve performance is easy, but it requires more area and power to provide high performance. Thus, we tried to parallelize computations in a PE. If we could compute in parallel for the same input datum like a SIMD, performance could be improved.

  In Fig. 4-3, there are two examples of ways we might parallelize data.

In the first picture parallelization was along the activations and in the second parallelization was along the outputs. According to the way we parallelized, the number of outputs we could get in one cycle would vary.

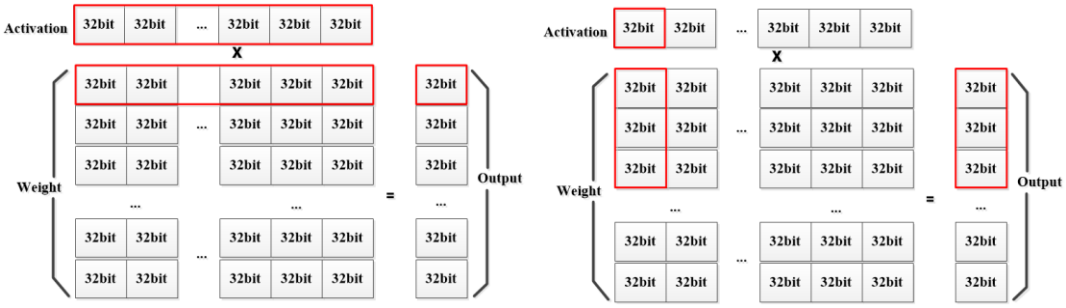

Fig. 4-3. Two possible ways to achieve parallelization

However, there is a problem with parallelization for weight sharing, which we call the LUT problem. As we mentioned in Chapter 2.4, the main problem of NNA for parameter sharing is the many look-up operations and consequently, the big LUT. Especially regarding parallelism, the number of LUT increases as in Fig. 4-4, so the size of memory increases. To solve this problem, we proposed the post-multiplication method. This will be discussed in the next chapter.

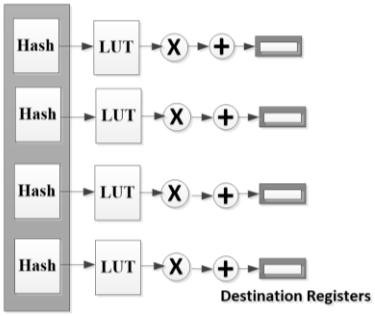

Fig. 4-4. LUT problem for weight sharing

- Employ sparsity (SKIP):

  We needed to consider not employing sparsity for minimizing the required data, but the best option would be if we could employ both sparsity of activations and weights like EIE. However, when we parallelize data, sometimes it is determined that we cannot employ sparsity. For example, in the first picture in Fig. 4-3, it is hard to employ both sparsity of activation and weight at the same time, in parallel. When sparsity is employed, the CSC format or CSR format is determined by the parallelization selected, because the difference between them is in how they are aligned.

B. Post-Multiplication

To overcome the LUT problem, we proposed using the post-multiplication method. As illustrated in the first picture of Fig. 4-5, we accumulated all the activations according to the output of a hashed unit, which is the centroid index. Next, we multiplied the accumulated array with the weights of centroids just one time. This way, our accelerator required only one LUT for the centroids and did not need as many LUTs as required for intra-PE parallelization.

Moreover, this method can only be applied when the number of outputs is small. For example, when the number of outputs is large (second picture of Fig. 4-3), the number of output registers piles up and interconnections increase.

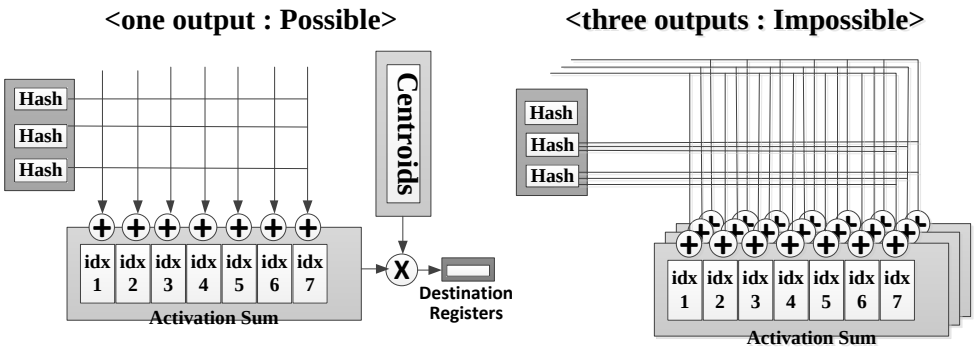

Fig. 4-5. Post-Multiplication: Possible and Impossible Cases

4.3 Design Exploration

With the key factors we addressed in Chapter 4.2, we considered various design models and determined the final design. Table 4-1 shows five possible designs for hashing the NNA. Design (i) does not employ sparsity and parallelization as in Fig. 4-2, we explored it as the simplest design. In this chapter, we will review other designs to show why we chose design(iv) as the final model.

	SIMD (PARALLEL)		Skip target (SKIP)		Post-multiplication
	A	W	A	W	
(i)	Serial	Serial	Skip	Skip (CSC)	X
(ii)	Serial	Parallel (↕)	Skip	Skip (CSC)	X
(iii)			Skip	X	X
(iv)	Parallel (↔)	Parallel (↔)	Skip	X	O
(v)			X	Skip (CSR)	O

A: Activation, W: Weight

Table 4-1. Possible designs with sparsity and parallelization

A. Designs (ii) and (iii)

Design (ii) and (iii) are parallelized along the outputs like the second picture in Fig. 4−3, so the number of outputs is more than one. Because they have more than one output, the post−multiplication method could not be applied and in design (iii) it was hard to internally parallelize for weight sharing because of the LUT problem even though the sparsity of weights could not be employed.

On the other hand, design (ii) employed both sparsity of weight and activation, so the output location would be random. However, when it was parallelized along the outputs, the register for the outputs could not be shared, so this design was impossible. Consequently, both of these designs were not suitable for minimizing the amount of data required.

B. Designs (iv) and (v)

Designs (iv) and (v) were parallelized along the activations like the first picture in Fig. 4−3. It is impossible to employ both sparsity of weight and activation in parallel in hardware, so in design (iv) the sparsity of activation was employed and in design (v) the sparsity of weight was employed. Both designs have one output register, so post−multiplication could be employed.

However, because design (v) has information about the sparsity of weight, it has same SRAM size as before, but the performance will deteriorate, because it does not employ the sparsity of activation. Thus, this design is useless.

On the other hand, design (iv) employed only sparsity of activation, so the information for the sparsity of weight is not needed and requires minimal data. Moreover, it has one output and post−multiplication can be applied, so

performance will be improved depending on the parallel factor. The area will be determined by how much area is taken from the post-multiplication and registers for the second weight sharing, which have the real centroid indices.

The best design choice for the hashing trick was design (iv) and the biggest advantage is that it can minimize the data required and has the potential for minimizing the SRAM size.

4.4 Details of the NNA Architecture

The architecture of the proposed NNA is like that of the general NNA in Fig. 4-6. It has an input buffer, CCU, CU, and LNZD. In this chapter we discuss these four units in detail.

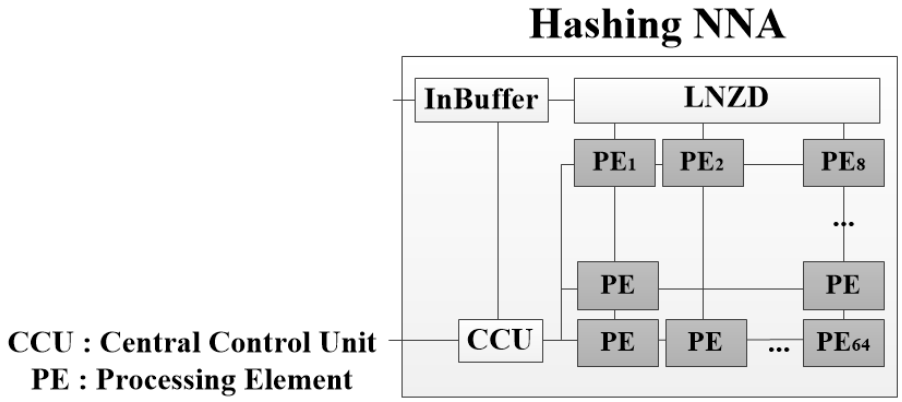

Fig. 4-6. Overview of our NNA

A. Input Buffer

The input buffer is for saving input activation. Because we parallelized along the input activations, the DRAM bandwidth must be adequate to distribute the

activations for the many PEs.

For example, Snapdragon 820, 835, 845, 850 uses a 64-bit dual channel at 1,866 MHz, so we assumed that the memory bandwidth is 29.8 GB/s. With the bandwidth, it can provide 298 bits in one cycle of an 800 MHz system, so our NNA can afford to get 16 activations per cycle, which means 256 bits per cycle. Moreover, the input buffer requires that it be implemented with two frequencies: a core frequency and the interface frequency.

B. Leading Non-Zero Detection (LNZD)

LNZD is used for detecting leading non-zero values in activations. This is for utilizing the sparsity of activation just like with EIE [1]. The difference is that one PE of EIE computes 64 outputs, so even though it has LNZD, it cannot fully utilize the sparsity of activations. However, in our design, because one PE computes one output, the sparsity of activations can be fully utilized.

When it comes to the experiment in Chapter 5, when we computed the number of cycles to compare performance, we assumed that EIE could fully utilize the sparsity of activations.

C. Central Control Unit (CCU)

The CCU controls all units in the middle. Hashing the NNA works according to two modes (as with EIE): I/O mode and computing mode. However, in I/O mode, the CCU directs PEs to load only centroid weights, not activations. The CCU manages the size of the matrices as well, and, in computing mode, it distributes activations from the input buffer. It also distributes information about

rows to the PEs to let each PE know which row of the matrix it needs to compute. When each row of computation is done, the CCU directs PEs to flush their registers to prepare for computing the next row.

D. Computing Unit (CU)

The CU consists of many PEs used computing each of the rows for vector–matrix multiplication. PE has 5 pipeline stages: 2 stages for hashing, hashed output for the centroid index, activation accumulation, multiplication with centroids, and accumulation followed by a Relu operation. According to the pipeline stages, details of the PE architecture is illustrated in Fig. 4–7. Dotted lines indicate each pipeline stage and the other lines show the data flow.

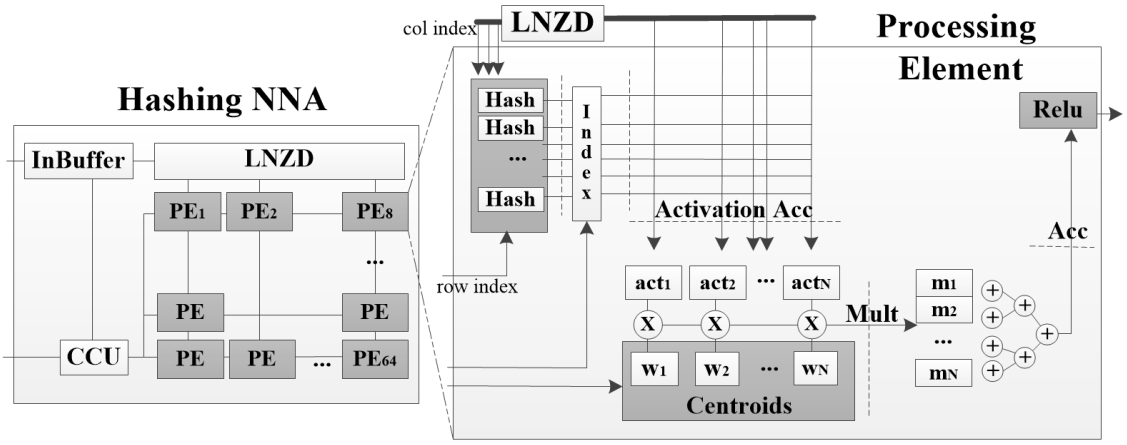

Fig. 4–7. Structure of a PE in our NNA

5. Experiment and Evaluation

In this chapter, we present the results of the experiments. In Chapter 5.1, we

show how we selected the design factors for our accelerator and show the quantitative results for such as performance, area, and power of our design in Chapter 5.2.

When it comes to training, we used an Intel Core i7−8700K which has 6 cores and one GeForce GTX 1080 Ti GPU. Each step took about 5 minutes for training with momentum, learning rate decay, and weight decay. When it comes to workload, the neural network consisted of only fully connected layers and it had one hidden layer with 1,000 neurons. Therefore, there were two kinds of matrices for the MNIST dataset: 784×1,000 and 1,000×10.

Furthermore, we implemented the register transfer level of our design using Verilog. It was synthesized with a design compiler (DC) using 32 nm CMOS technology and the chip circuit area was estimated with a Synopsys IC Compiler (ICC) at the floor planning stage.

5.1 Design Factors

In this chapter, we show how we selected the factors for our accelerator, including parallelism factors and the number of centroids.

A. Inter PE parallel factor

As the number of PEs increases, performance increases. However, power use and area will increase at the same time. However, Fig. 5−1 shows how the area−performance ratio changes according to the number of PEs. As you can see, when the number of PEs increases, because the number of cycles decreases, performance monotonously increases. Moreover, the performance−

area ratio increases, but it seems to become saturated around 64 PEs, so we decided to use 64 Pes as with EIE [1], to compare with it and to choose the number for which the performance-area almost converged.

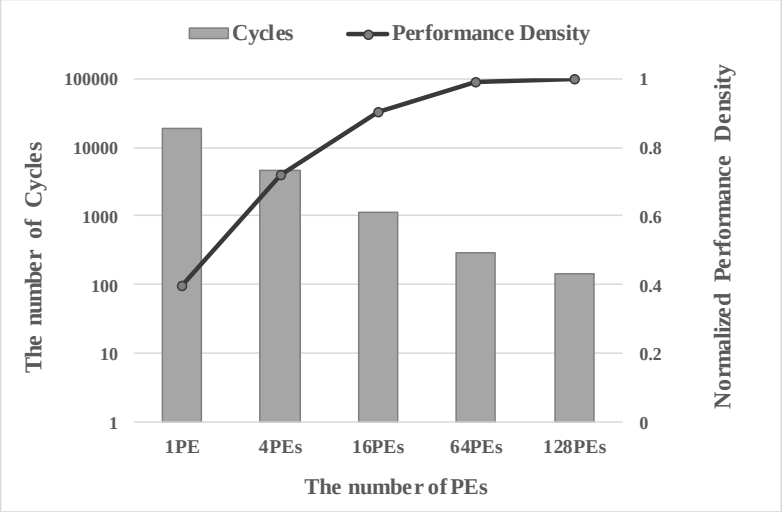

Fig. 5-1. Result of Inter-PE Parallelism

B. Intra PE parallel factor

As we mentioned in Chapter 4.4, we parallelized our design along the input activations, so this was determined according to the bandwidth of the input buffer. Providing DRAM bandwidth optimized like in the Snapdragon series, our design could compute 16 activations in one cycle.

With 16 intra-PE parallelization, the performance of our design was comparable to that of EIE [1].

C. Number of Centroids

As the number of centroids increases, each PE requires more interconnections and registers, but accuracy also increases because there are more values to express. Thus, we needed to determine the number of centroids to guarantee accuracy and to minimize the circuit size.

Fig. 5-2 presents the accuracy and the required memory depending on the number of centroids. Black and white bars in the pictures are about accuracy, and orange bars are about the data required for a fully trained neural network with 64 PEs.

The darkest black bar shows the original accuracy of MNIST with fully connected layers. As you can see in Fig. 5-2, this is 98.30%. The second darkest black bars are the result accuracy of the second-step training. The number of centroids means the number of indices of centroids. For example, 1,024/250 means that the neural network was trained with 1,024 centroids for the first layer and 250 centroids for the second layer in the second step. 1,531/250 used the same number of centroids with the original hashing trick and its accuracy was 95.82%. Furthermore, 1,024/250 centroids had accuracy of 90.01%, the 512/250 centroids had 83.49%, and the 256/250 centroids had 32.46%.

The brightest black graph shows the accuracy result of the final training with hashing and index sharing, so it has the data required. For example, a neural network trained with 1,024 centroids in the second step can be trained with from 1 to 64 real centroids in the final step. Interestingly, the accuracy from training only with the hashing trick was improved when it was post-trained with the index, because it had more expressiveness.

We assumed that minimal accuracy should satisfy 90% and chose a neural network with minimal data. Finally, we chose 1,024/250 centroids for the

second step training and 4 real centroids for the last step training. These had 90.54% accuracy and required 20 KB for 64 PEs.

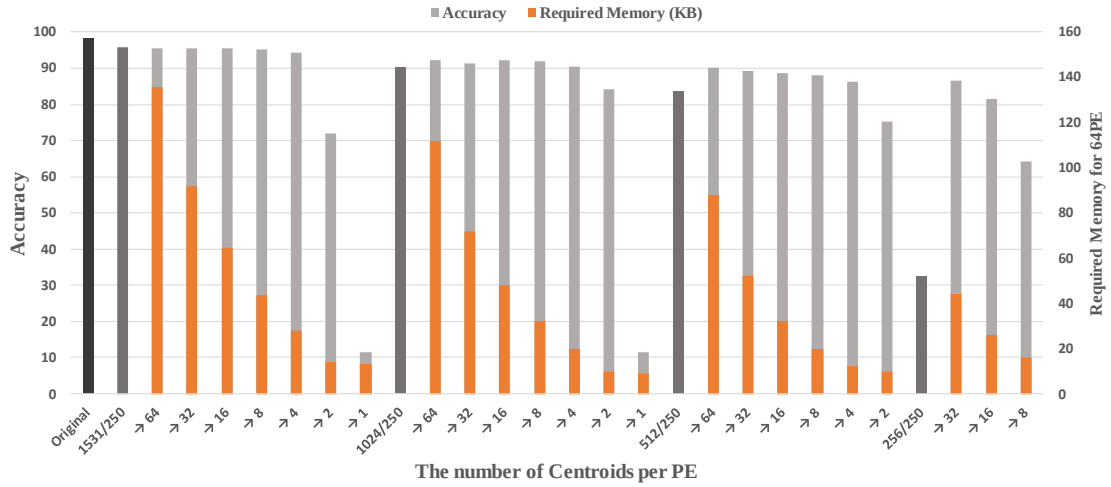

Fig. 5-2. Accuracy and Memory Depending on the Number of Centroids

5.2 Quantitative Results

A. Performance

We computed performance by counting the number of cycles for the same workload. The workload was

$$\text{input} \times (\text{input} \times \text{output})$$

of SPMV computation and we assumed that activation had 70% sparsity and weight had 90% sparsity.

Because EIE [1] was parallelized along the outputs and each PE can compute 64 rows, the EIE can compute all outputs at one time. However, it computes one column in 8 cycles because it is not optimized when each PE has less than 8

non−zero weights. However, we proved that EIE can employ the sparsity of weights depending on the number of non−zero weights by implementation. This way, EIE requires the following cycles for the workload:

$$(\boldsymbol{input} * \mathbf{0.3}) * ((\boldsymbol{output}/\mathbf{64}\boldsymbol{PE}) * \mathbf{0.1}) = \boldsymbol{input} * \boldsymbol{output} * \mathbf{4.69} * \mathbf{10^{-4}} \text{ cycles}$$

On the other hand, our design is parallelized along the input activations, so 16 activations can be computed at one time. The PEs in our design cannot compute all outputs at one time, so we needed to compute 1,000/64 PE times. For this reason, our design requires the following cycles.

$$((\boldsymbol{input} * \mathbf{0.3})/\mathbf{16}) * (\boldsymbol{output}/\mathbf{64}\,\boldsymbol{PE}) = \boldsymbol{input} * \boldsymbol{output} * \mathbf{2.93} * \mathbf{10^{-4}} \text{ cycles}$$

Consequently, our accelerator requires 37.5% fewer cycles for the same workload, so we can say 60% of performance was improved.

B. Data Required for the Accelerator

For the same workload, we figured out how many data we needed for one PE. For a 784×1,000 matrix in MNIST, our model requires (1,024×2 bits =) 2,048 bits for the indices of real weights, and (16 bits ×4 =) 64 bits for the real weight data. Thus, one PE in our model required 2,112 bits for inference.

On the other hand, EIE requires (784×16 bits) for the sparse column pointer, (8 bits ×1,000×0.1 / 64 PE) for the sparse row interval and the index array, and (16×16 bits) for the real weight data. Thus, one PE of the EIE requires 12,812 bits for inference.

Our design requires 83.51% less data, and this means that, in I/O time, our design takes less time to load the required data.

C. Performance−Area Efficiency

As we mentioned, the area was estimated using the ICC at the floor planning stage. The result is organized in Table 5-1. Our design has three kinds of memory: input buffer, centroid index array, and real centroids. The input buffer is 16 kB to store input activations and the centroid index array contains the index of centroids to get the real centroids. As we addressed in a previous chapter, each PE requires a 264 B register file.

In Table 5-1, the performance ratio and the required data were just as we addressed in the previous chapters. The on-chip memory and final model data for EIE [1] were references from the original paper, but the logic gate count came from our implementation of EIE. Our design has 50.23% less area and has smaller SRAM size without performance degradation. However, most of the cost of our design comes from the logic circuit, so it should be optimized. We will talk about this in the next chapter.

			EIE [1]	Hashed
Cycle ratio for same workload			468.75 u	292.97 u
Required data for MNIST			800 KB	16.5 KB
On-Chip Memory	Register size		2 KB	16 KB
	SRAM	Size	10,368 KB	16 KB
		Area	38.07 mm2	0.76 mm2
Final model (64PE)	Logic gate count		(2,497,763)	13,692,262
	Area	Logic	2.76 mm2	19.56 mm2
		Total	40.83 mm2	20.32 mm2

Table 5-1. Comparison of Area and SRAM

Consequently, with 60% improvement of performance and 50.23% reduction of area, the performance-area efficiency of our design is 3.21 times greater than the original EIE.

D. Hash Function Evaluation

After we synthesized our design, we analyzed the number of gates to see what proportion was occupied by each unit in our model. Table 5-2 shows the proportions of gates used for each unit before and after optimizing hash function.

Design Stack	Hashed						
	CCU	64 PE					
		16 Hash	Act_sum			Reg	
			16 CE	63 ACC	4 Mult	Comb	Regs
Before	0.01%	41.89%	0.7%	1.6%	1.1%	53.55%	
After	0.01%	11.43%	1.07%	2.39%	1.68%	78.61%	

Table 5-2. Gate count of our design

Table 5-2 shows that XXhash, which was originally used, takes 41.89% of the gates in our design. Also, the design using this hash function has 30.02mm^2 of area. That is, as was addressed in Chapter 3.2, XXhash is not optimized for hardware, because it has 10 multiply operations for one hash unit. In fact, BHNN [41] used a pseudo-hash generator only with XOR operation and proved that it does not incur performance degradation.

Required feature for hash function consists of the following two attributes. Firstly, for efficient hardware design, it should process hash function in parallel for the time limit and it should have a minimal number of the complex arithmetic units such as multipliers. Secondly, for accuracy, the hash function requires proper uniformity (diffusion) and randomness (confusion).

Consequently, we optimized the hash function for area and power only with two multipliers for one hash unit, and the area can be reduced to 20.32 mm^2 as showed in Table 5-1.

E. Expected Power

Because we did not employ sparsity of the matrix weights yet, there can be a question about the power of our NNA. Considering Amdahl's law, we can approximate how much power our accelerator will have. We compared our design with EIE [1] regarding the power proportion of each component.

Because 59.15% of the power use of EIE occurred in memory, we can expect almost all of this power to be reduced according to Amdahl's law. However, 20.38% of the power use of EIE was incurred in the register and combinational components, and we have even more of these kinds of components. After calculations, we expect that our design will have 22.36% more power usage than EIE does.

However, we have the potential to reduce the proportion of combinational circuits by employing the sparsity. By pruning parts of the centroids, we could employ sparsity of weights to reduce the power use in our design. In fact, with 1,024/250 centroids in the second step and 8 real centroids in the last step, we could get 87.75% accuracy with a degradation of only 4.05% when we prune 25% of the centroids. Moreover, with 4 real centroids, we could get 86.97% accuracy with degradation of only 3.57% when we prune 25% of the centroids. Thus, pruning will help us reduce the power usage of our accelerator.

6. Conclusions

To efficiently utilize SRAM for a neural network accelerator, we proposed a novel weight sharing technique that we call double-stage weight sharing. With this technique, we could minimize the data required with accuracy comparable to a system without this weight sharing technique. Furthermore, we explored various designs considering sparsity and parallelization and determined the optimal design for using the minimized data.

The performance-area efficiency of our accelerator is 3.21 times higher than a representative neural network accelerator that employed the original weight sharing technique. Moreover, there are optimization points remaining with the potential to improve our accelerator, so our model has great potential.

Except for trivial optimization points of our current model, we have three future studies in mind for a more useful accelerator. First, our model experiment only included FC layers, so we need to figure out if our accelerator can compute Conv layers as well. Second, our experiments included processing with MNIST datasets and a small neural network with relatively small workload (only 1 M parameters). However, neural network in various domains have as many as 1 G parameters, so we need to prove that our accelerator can be used to compute neural networks with many more parameters. For this, we need to first check to see if double-stage weight sharing can provide comparable accuracy with the number of centroids used in larger systems.

Last, as addressed in Chapter 5, we need to measure how much power our NNA design uses and figure out how to optimize our design to reduce power dissipation by employing the sparsity of weights.

References

[1] Han, Song, et al. "EIE: efficient inference engine on compressed deep neural network." 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA). IEEE, 2016.

[2] Parashar, Angshuman, et al. "Scnn: An accelerator for compressed-sparse convolutional neural networks." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[3] Zhang, Shijin, et al. "Cambricon-x: An accelerator for sparse neural networks." The 49th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Press, 2016.

[4] Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks." ACM SIGARCH Computer Architecture News. Vol. 44. No. 3. IEEE Press, 2016.

[5] Sze, Vivienne, et al. "Efficient processing of deep neural networks: A tutorial and survey." Proceedings of the IEEE 105.12 (2017): 2295-2329.

[6] Chen, Tianshi, et al. "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning." ACM Sigplan Notices. Vol. 49. No. 4. ACM, 2014.

[7] Chen, Yunji, et al. "Dadiannao: A machine-learning supercomputer." Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014.

[8] Du, Zidong, et al. "ShiDianNao: Shifting vision processing closer to the sensor." ACM SIGARCH Computer Architecture News. Vol. 43. No. 3. ACM, 2015.

[9] Qiu, Jiantao, et al. "Going deeper with embedded fpga platform for

convolutional neural network." Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2016.
[10] Han, Song, et al. "Learning both weights and connections for efficient neural network." Advances in neural information processing systems. 2015.
[11] Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." arXiv preprint arXiv:1510.00149 (2015).
[12] Gao, Mingyu, et al. "Tetris: Scalable and efficient neural network acceleration with 3d memory." ACM SIGARCH Computer Architecture News. Vol. 45. No. 1. ACM, 2017.
[13] Chen, Wenlin, et al. "Compressing neural networks with the hashing trick." International Conference on Machine Learning. 2015.
[14] Abdelouahab, Kamel, et al. "Accelerating CNN inference on FPGAs: A Survey." arXiv preprint arXiv:1806.01683 (2018).
[15] Guo, Kaiyuan, et al. "A Survey of FPGA-Based Neural Network Accelerator." arXiv preprint arXiv:1712.08934 (2017).
[16] Intel FPGA. The Intel FPGA SDK for Open Computing Language (OpenCL), 2016.
[17] Winograd, Shmuel. Arithmetic complexity of computations. Vol. 33. Siam, 1980.
[18] Farabet, Clément, et al. "Cnp: An fpga-based processor for convolutional networks." 2009 International Conference on Field Programmable Logic and Applications. IEEE, 2009.
[19] Zhang, Chen, et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks." Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2015.

[20] Albericio, Jorge, et al. "Cnvlutin: Ineffectual-neuron-free deep neural network computing." ACM SIGARCH Computer Architecture News 44.3 (2016): 1-13.
[21] Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit." 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2017.
[22] Chole, Sharad, Ramteja Tadishetti, and Sree Reddy. "SparseCore: An Accelerator for Structurally Sparse CNNs."
[23] Cheng, Yu, et al. "A survey of model compression and acceleration for deep neural networks." arXiv preprint arXiv:1710.09282 (2017).
[24] Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. "Distilling the knowledge in a neural network." arXiv preprint arXiv:1503.02531 (2015).
[25] Tai, Cheng, et al. "Convolutional neural networks with low-rank regularization." arXiv preprint arXiv:1511.06067 (2015).
[26] Cheng, Yu, et al. "An exploration of parameter redundancy in deep networks with circulant projections." Proceedings of the IEEE International Conference on Computer Vision. 2015.
[27] Yang, Bo, et al. "Quantization and training of object detection networks with low-precision weights and activations." Journal of Electronic Imaging 27.1 (2018): 013020.
[28] Lin, Darryl, Sachin Talathi, and Sreekanth Annapureddy. "Fixed point quantization of deep convolutional networks." International Conference on Machine Learning. 2016.
[29] Gong, Yunchao, et al. "Compressing deep convolutional networks using vector quantization." arXiv preprint arXiv:1412.6115 (2014).
[30] Anwar, Sajid, Kyuyeon Hwang, and Wonyong Sung. "Fixed point

optimization of deep convolutional neural networks for object recognition." 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2015.
[31] Kundu, Abhisek, et al. "Ternary residual networks." arXiv preprint arXiv:1707.04679 (2017).
[32] Courbariaux, Matthieu, et al. "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or−1." arXiv preprint arXiv:1602.02830 (2016).
[33] Hubara, Itay, et al. "Quantized neural networks: Training neural networks with low precision weights and activations." The Journal of Machine Learning Research 18.1 (2017): 6869−6898.
[34] Rastegari, Mohammad, et al. "Xnor−net: Imagenet classification using binary convolutional neural networks." European Conference on Computer Vision. Springer, Cham, 2016.
[35] Zhou, Shuchang, et al. "Dorefa−net: Training low bitwidth convolutional neural networks with low bitwidth gradients." arXiv preprint arXiv:1606.06160 (2016).
[36] Gupta, Suyog, et al. "Deep learning with limited numerical precision." International Conference on Machine Learning. 2015.
[37] Wang, Erwei, et al. "Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going." arXiv preprint arXiv:1901.06955 (2019).
[38] Samragh, Mohammad, Mohammad Ghasemzadeh, and Farinaz Koushanfar. "Customizing neural networks for efficient fpga implementation." 2017 IEEE 25th Annual International Symposium on Field−Programmable Custom Computing Machines (FCCM). IEEE, 2017.

[39] Zhang, Jialiang, and Jing Li. "PQ-CNN: Accelerating Product Quantized Convolutional Neural Network on FPGA." 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018.
[40] Wu, Jiaxiang, et al. "Quantized convolutional neural networks for mobile devices." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2016.
[41] Zhu, Jingyang, Zhiliang Qian, and Chi-Ying Tsui. "BHNN: A memory-efficient accelerator for compressing deep neural networks with blocked hashing techniques." 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2017.
[42] Collet, Yann. "xxHash--Extremely Fast Hash Algorithm." (2016).
[43] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems. 2017.

논문요약

Minimizing Required Data in a Neural Network Accelerator Using Double-Stage Weight Sharing

성균관대학교
전자전기컴퓨터공학과
유 태 관

최근 딥뉴럴 네트워크의 깊이와 크기가 깊어지는 한편 자연어 처리와 같은 분야의 Fully Connected 레이어의 활용이 늘어났다. 하지만 Fully Connected 층은 많은 수의 파라미터를 연산할 뿐 아니라, Convolutional 층과 달리 파라미터의 재사용이 적어 많은 온 칩 메모리인 SRAM을 요구한다. 엣지 디바이스에서의 활용을 위해 효율적으로 네트워크를 처리할 수 있는 뉴럴 네트워크 가속기에 대한 요구 또한 늘어나고 있지만, 기존 뉴럴 네트워크 가속기의 경우 많은 면적과 파워가 파라미터를 저장하기 위한 SRAM에서 발생하는 것을 확인했다.

이에 본 논문은 Double-Stage 파라미터 공유 기법을 활용해 요구되는 데이터를 최소한으로 줄이고 모든 데이터를 SRAM에서 처리하더라도 면적 대비 성능이 효율적인 가속기를 제안한다. 데이터 포맷을 통해 먼저 필요한 데이터를 최소한으로 줄이기 위해 해싱 트릭과 Double-Stage 파라미터 공유를 활용한 경우의 데이터 포맷을 살펴보았으며, 직접 구현 및 학습한 결과를 보였다.

또한 실제 이를 활용한 최적의 가속기를 제안하기 위해 다양한 디자인을 희소성과 병렬화 두가지의 요소를 기반으로 살펴보았으며, 가속기 기본 유닛의 병렬화나 개수 등을 결정하기 위한 실험을 진행했다.

텐서플로우를 활용해 직접 원하는 네트워크와 학습 방법을 구현하였으며, 본 기술을 적용시 공유되는 중앙 값의 개수에 따른 정확도를 찾고 이에 따른 가속기를 Verilog를 활용해 직접 구현했다. 3 nm CMOS 기술로 Design Compiler를 통해 합성 후 Floor Planning 단계에서 Synopsys IC Compiler를 통해 면적을 측정한 결과 기존의 파라미터 공유 기법을 활용한 대표적인 논문인 EIE에 비해 50.23%의 면적이 줄고 60%의 성능이 향상해 면적 대비 성능이 3.21배 향상되었다.

주제어: 양자화 기법, 해싱 트릭, 파라미터 공유 기법, 뉴럴 네트워크 가속기

Master's Thesis

Minimizing Required Data in a Neural Network Accelerator Using Double-Stage Weight Sharing

2020

Taekoan Yoo