



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](#)

Ph.D. DISSERTATION

Memory Layout and Computing  
Techniques for High Performance  
Neural Networks

고성능 인공 신경망을 위한  
메모리 레이아웃 및 컴퓨팅 기법

BY

BYUNGMIN AHN

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL AND  
COMPUTER ENGINEERING  
COLLEGE OF ENGINEERING  
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Memory Layout and Computing  
Techniques for High Performance  
Neural Networks

고성능 인공 신경망을 위한  
메모리 레이아웃 및 컴퓨팅 기법

BY

BYUNGMIN AHN

FEBRUARY 2021

DEPARTMENT OF ELECTRICAL AND  
COMPUTER ENGINEERING  
COLLEGE OF ENGINEERING  
SEOUL NATIONAL UNIVERSITY

# Memory Layout and Computing Techniques for High Performance Neural Networks

고성능 인공 신경망을 위한  
메모리 레이아웃 및 컴퓨팅 기법

지도교수 김 태 환  
이 논문을 공학박사 학위논문으로 제출함

2020년 12월

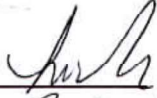
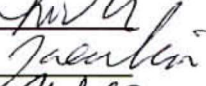
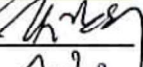

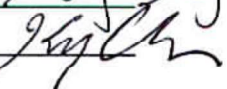
서울대학교 대학원

전기정보공학부

안 병 민

안병민의 공학박사 학위 논문을 인준함

2021년 1월

위 원 장:	이 혁재	
부위원장:	김 태 환	
위 원:	김 간우	
위 원:	성 원용	
위 원:	최 귀형	



# Abstract

Although the demand for exploiting neural networks is steadily increasing, there are many design challenges since deep neural networks (DNNs) entail excessive memory and computation cost. This dissertation studies a number of new techniques for effectively processing DNN inference operations.

Firstly, we attempt to overcome that the maximal computation speedup is bounded by the total number of non-zero bits of the weights. Precisely, this work, based on the signed-digit encoding, (1) proposes *a transformation technique* which converts the two's complement representation of every weight into a set of signed-digit representations of *the minimal number of essential bits*, (2) formulates the problem of selecting signed-digit representations of weights that *maximize the parallelism of bit-level multiplication* on the weights into *a multi-objective shortest path problem* to achieve a maximal digit-index by digit-index (i.e. column-wise) compression for the weights and solves it efficiently using an approximation algorithm, and (3) proposes *a supporting novel acceleration architecture (DWP)* with no additional inclusion of non-trivial hardware. In addition, we (4) propose *a variant of DWP* to support bit-level parallel multiplication with the capability of *predicting a tight worst-case latency of the parallel processing*. Through experiments on several representative models using the ImageNet dataset, it is shown that our proposed approach is able to reduce the number of essential bits by 69% on AlexNet, 74% on VGG-16, and 68% on ResNet-152, by which our accelerator is able to reduce the inference computation time by up to  $3.57\times$  over the conventional bit-level weight pruning.

Secondly, a new algorithm for extracting common kernels and convolutions to maximally eliminate the redundant operations among the convolutions in binary- and ternary-weight convolutional neural networks is presented. Specifically, we propose (1) *a new algorithm of common kernel extraction* to overcome the local and limited

exploration of common kernel candidates by the existing method, and subsequently apply (2) *a new concept of common convolution extraction* to maximally eliminate the redundancy in the convolution operations. In addition, our algorithm is able to (3) *tune in minimizing the number of resulting kernels* for convolutions, thereby saving the total memory access latency for kernels. Experimental results on ternary-weight VGG-16 demonstrate that our convolution optimization algorithm is very effective, reducing the total number of operations for all convolutions by 25.8-26.3%, thereby reducing the total number of execution cycles on hardware platform by 22.4% while using 2.7-3.8% fewer kernels over that of the convolution utilizing the common kernels extracted by the state-of-the-art algorithm.

Finally, we propose solutions for DNNs with “unfitted compression” to maintain the accuracy, in which all distinct weights of the compressed DNNs could not be entirely contained in on-chip memory. Precisely, given an access sequence of weights, (1) the first problem is to arrange the weights in off-chip memory, so that the number of memory accesses to the off-chip memory (equivalently the energy consumed by the accesses) be minimized, and (2) the second problem is to devise a strategy of selecting a weight block in on-chip memory for replacement when a block miss occurs, with the objective of minimizing the total energy consumed by the off-chip memory accesses and the overhead of scanning indexes for block replacement. Through experiments with the model of compressed AlexNet, it is shown that our solutions are able to reduce the total energy consumption of the off-chip memory accesses including the scanning overhead by 34.2% on average over the use of unoptimized memory layout and LRU replacement scheme.

**Keywords:** Deep neural networks, Bit-level weight pruning, Signed-digit representation, Common kernel extraction, Common convolution extraction, Unfitted compression

**Student number:** 2015-20943

# Contents

<b>Abstract</b>	<b>i</b>
<b>Contents</b>	<b>iii</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Deep Neural Networks and Its Challenges . . . . .	1
1.2 Redundant Weight Elimination Methods in DNN . . . . .	4
1.3 Redundant Representation Elimination Methods in DNN . . . . .	8
1.4 Contributions of This Dissertation . . . . .	12
<b>2 Bit-level Weight Pruning Techniques for High-Performance Neural Networks</b>	<b>17</b>
2.1 Preliminary . . . . .	17
2.1.1 Bit-level Weight Pruning in Binary Representation . . . . .	17
2.1.2 Bit-level Weight Pruning in Signed-digit Representation . . . . .	19
2.1.3 CSD Representation Conversion . . . . .	21
2.2 Motivations . . . . .	23
2.2.1 Inefficiency in Two's Complement Representation . . . . .	23
2.2.2 Inability to Exploit Signed-digit Representation . . . . .	25

2.3	Signed-digit Representation-based Deeper Weight Pruning . . . . .	28
2.3.1	Generating Signed-digit Representations . . . . .	28
2.3.2	Selecting Signed-digit Representations for Maximal Parallelism	30
2.3.3	Extension to the Low-precision Weights . . . . .	32
2.4	Supporting Hardware Architecture . . . . .	33
2.4.1	Technique for Using a Single Bit to Encode Ternary Value . .	33
2.4.2	Structure of Supporting Architecture . . . . .	35
2.4.3	Memory Analysis . . . . .	37
2.4.4	Full Utilization of Accumulation Adders . . . . .	38
2.4.5	Modification for Hybrid Approach . . . . .	38
2.5	Bit-level Intra-weight Pruning . . . . .	41
2.5.1	Signed-digit Representation Conversion . . . . .	41
2.5.2	Encoding Technique . . . . .	41
2.5.3	Supporting Hardware Architecture . . . . .	42
2.6	Experimental Results . . . . .	44
2.6.1	Essential Bits . . . . .	44
2.6.2	Memory Usage . . . . .	46
2.6.3	Performance . . . . .	46
2.6.4	Area . . . . .	50
2.6.5	Energy Efficiency . . . . .	56

### **3 Convolution Computation Techniques for High-Performance Neural Networks** **59**

3.1	Motivations . . . . .	59
3.1.1	Limited Space Exploration for Common Kernels . . . . .	59
3.1.2	Inability to Exploit Common Expressions of Convolution Values	61
3.2	The Proposed Algorithm . . . . .	63
3.2.1	Common Kernel Extraction . . . . .	63
3.2.2	Common Convolution Extraction . . . . .	67

3.2.3	Memory Access Minimization . . . . .	69
3.3	Hardware Implementation . . . . .	70
3.4	Experimental Results . . . . .	72
3.4.1	Experimental Setup . . . . .	72
3.4.2	Assessing Effectiveness of ConvOpt-op and ConvOpt-mem	72
3.4.3	Measuring Performance through Hardware Implementation .	78
3.4.4	Running Time of ConvOpt . . . . .	78
<b>4</b>	<b>Memory Layout and Block Replacement Techniques for High-Performance Neural Networks</b>	<b>81</b>
4.1	Motivation . . . . .	81
4.2	Algorithms for Off-chip Memory Access Optimization for DNNs with Unfitted Compression . . . . .	84
4.2.1	Algorithm for Off-chip Memory Layout . . . . .	84
4.2.2	Algorithm for On-chip Memory Block Replacement . . . . .	86
4.2.3	Exploitation of Parallel Computing . . . . .	91
4.3	Experimental Results . . . . .	94
4.3.1	Experimental Setup . . . . .	94
4.3.2	Assessing the Effectiveness of Mem-layout . . . . .	94
4.3.3	Assessing the Effectiveness of MIN-k Combined with Mem- layout . . . . .	97
<b>5</b>	<b>Conclusions</b>	<b>101</b>
5.1	Bit-level Weight Pruning Techniques for High-Performance Neural Networks . . . . .	101
5.2	Convolution Computation Techniques for High-Performance Neural Networks . . . . .	102
5.3	Memory Layout and Block Replacement Techniques for High-Performance Neural Networks . . . . .	102



# List of Tables

1.1	Energy table for 45nm process [15] . . . . .	6
1.2	Fraction of zero-valued weights [42] . . . . .	6
2.1	The normalized number of essential bits for two’s complement, sign-magnitude, and our signed-digit representations. . . . .	45
2.2	Area comparison for $B = 16$ and $k = 16$ . . . . .	53
2.3	Area breakdown of DWP for $B = 16$ and $k = 16$ ( $mm^2$ ) . . . . .	54
2.4	Area breakdown of DWP-intra for $B = 16$ and $k = 16$ ( $mm^2$ ) . . . . .	55
3.1	Statistics of the number of partial kernels in ternary-weight CNN models.	64
3.2	Comparison of the <i>total number of operations</i> produced by No-sharing (the convolution without exploitation of common kernels), Local-sharing (the existing work in [64] of common kernel sharing), and ConvOpt-op & ConvOpt-mem performing in ② to ④ in Fig. 3.5 for the convolutional layers in VGG-16. . . . .	74
3.3	Comparison of the <i>total number of kernels</i> used by No-sharing (the convolution without exploitation of common kernels), Local-sharing (the existing work in [64] of common kernel sharing), and ConvOpt-op & ConvOpt-mem performing convolutions in ② in Fig. 3.5 for the convolutional layers in VGG-16. . . . .	75

3.4	Comparison of the <i>total number of execution cycles</i> on the hardware platform in Fig. 3.5 when using the kernels produced by Local-sharing [64], ConvOpt-op, and ConvOpt-mem for image inferencing with VGG-16. . . . .	77
3.5	Comparison of <i>running time</i> (hour:min:sec) spent by Local-sharing [64] and ConvOpt-op for VGG-16. . . . .	80
4.1	Statistics of the number of weights before and after the compression. Note: column <i>before</i> represents the total number of weight accesses while column <i>after</i> represents the number of distinct weights. . . . .	93
4.2	The numbers of off-chip weight accesses before and after applying our Mem-layout under various conventional schemes of block replacement, assuming $ B  = 4$ . . . . .	95
4.3	Energy reduction by our MIN-k and Mem-layout over the conventional approach. . . . .	98



# List of Figures

1.1	Top-5 classification error rates and the number of parameters for each model of ILSVRC. Since the introduction of DNN models, the top-5 error rates have significantly decreased. . . . .	2
1.2	An example of extracting a common kernel and sharing its convolution to reduce the redundancy in convolution computation. . . . .	10
2.1	Conventional bit-level pruning [42] for weights in binary representation. (a) 6 weights of an 8-bit fixed-point representation before pruning. For performing <i>Eq.2.2</i> in parallel on each bit column $b$ , 6 execution cycles are required. (b) 4 weights after the application of bit-level pruning. By reducing the number of weights from 6 to 4, 2 execution cycles are saved. . . . .	18
2.2	Bit-level prunings for weights in signed-digit representation. . . . .	20
2.3	The reductions of the amount of essential bits for negative weights of 8 bits by converting them from two's complement representation to sign-magnitude representation. . . . .	24
2.4	Comparison of word/bit-level value distributions of the weights in AlexNet and VGG-16 trained with ImageNet dataset when using two's complement and sign-magnitude representations. . . . .	26
2.5	A stepwise generation of all possible signed-digit representations for a weight value of -55 in sign-magnitude representation. . . . .	29

2.6	An illustration of full flow (from left to right) of selecting signed-digit representations for maximal parallelism of bit-level multiplication. The left of the graph illustrates the signed-digit representation generation process performed in Sec. 2.3.1. The graph in the middle shows our graph-based formulation of selecting signed-digit representations, and the arc cost setting is shown at the right of the graph. The alignment of essential bits for maximal parallelism taken from the selected signed-digit representations is placed at the right of the illustration.	31
2.7	(a) An initial weight alignment in signed-digit representation. (b) Column-wise compressed alignment for initial weight alignment. (c) Column-wise realignment obeying the <i>ternary ordering rule</i> for the compressed form. (d) Encoding ternary values of signed-digit representations in memory with the flag values, and the decoding implication. . . . .	34
2.8	(a) Our proposed hardware architecture with 16 parallel lanes, each lane processing $k'$ condensed weights, one weight at a time. 16 processing elements (PEs) are operating concurrently. (b) Microarchitecture of splitter for processing signed-digit representation. (c) Implementation for decoding a flag and memory bit. (d) Final adder tree for summing all bit-level partial-sums. . . . .	36
2.9	Effectiveness of using two accumulation adders for the essential bits in the least significant bit (LSB) position. . . . .	39
2.10	Modified final adder tree to support hybrid approach. . . . .	40

2.11 (a) An architecture including conceptual structure of shift-and-add (SAA) units. Each SAA unit accepts  $B$  pairs of input activation and condensed weight and has  $B$  shifting units and an adder tree. (b) A structure of shifting unit. The decoded signals from the decoding logic which is similar to Fig. 2.7(c) determine whether the input activation is to be used as it is, or converted to a negative number, or left as 0. Then, the activation is shifted to the left by  $idx_i$ . . . . . 43

2.12 Comparison of memory usage by Tetris [42] and our DWP when bit-size  $B = 8$  and 16, and pruning stride  $k = 8, 16$  and 32.  $M_w$  includes the bits for weight encoding (Tetris and DWP) and flag (DWP) and  $M_{idx}$  includes the bits for activation selection in the splitters (Tetris and DWP). . . . . 47

2.13 Comparison of memory usage by uncompressed case, DWP, and DWP-intra for several models when bit-size  $B = 8$  and 16, and pruning stride  $k = 8$  and 16. For a fair comparison, DWP with  $(B, k) = (16, 8)$  and  $(8, 16)$  has compared with DWP-intra with  $(B, k) = (8, 16)$  and  $(16, 8)$ , respectively. . . . . 48

2.14 Comparison of the averaged number of clock cycles in performing a set of weights with  $B = 8$  and 16, and pruning stride  $k = 8$  and 16. . . . . 49

2.15 Comparison of the averaged number of clock cycles for various model sparsity in performing a set of weights with  $B = 8$  and pruning stride  $k = 8$  and 16. . . . . 51

2.16 Comparison of the averaged number of clock cycles for uncompressed case, Tetris [42], and DWP in performing a set of weights with low precision and pruning stride  $k = 16$ . . . . . 52

2.17 Comparison of the average of EDP with  $B = 16$  and  $k = 16$ . (The lower, the better.) . . . . . 57

3.1	Examples showing the limitation of the common kernel extraction by [64] . . . . .	60
3.2	Exploiting a common expression of convolution values . . . . .	62
3.3	An example showing the common kernel extraction of ConvOpt. For simplicity, we do not consider opposite kernels in this example. (a) (Step 1.1) Generating all partial kernels. (b) (Step 1.2) Calculating the operation saving costs. (c) (Step 1.3) Iteratively selecting common kernels and update $\mathcal{S}$ . . . . .	66
3.4	An example showing the common convolution extraction of ConvOpt.	68
3.5	A hardware architecture supporting ConvOpt . . . . .	71
3.6	A breakdown of operation counts used by No-sharing, Local-sharing [64], ConvOpt-op, and ConvOpt-mem for the convolutional layers in VGG-16. Blue, green, and orange bars stand for the operation counts in convolution on common or original kernels (②), calculating common expressions of intermediate convolution results (④), and accumulation to produce final convolution outputs (③), respectively. . . . .	76
3.7	A breakdown of the execution cycles on the hardware platform in Fig. 3.5. Blue, green, and orange pies stand for the execution cycles in convolution on common kernels (②), calculating common expressions of intermediate convolution results (④), and accumulation to produce final convolution outputs (③), respectively. . . . .	79
4.1	An example of DNN with unfitted compression to illustrate our motivation of reducing the number of off-chip memory accesses in Fig. 4.2. (a) MAC (multiply-and-accumulate) operations in a layer of DNN and index sequence for accessing the weights, assuming DNN compression with weight sharing results in a total number of 8 weights. (b) The configuration of off-chip (containing 8 weights) and on-chip memories (containing up to 4 weights). The block size for access is 2. . . . .	82

4.2	Four cases of handling memory layout and block replacement for the DNN in Fig. 4.1. (a) Off-chip memory accesses resulting from the use of unoptimized memory layout and LRU replacement scheme. (b) Off-chip memory accesses resulting from the use of our optimized memory layout and LRU replacement scheme. (c) Off-chip memory accesses resulting from the use of unoptimized memory layout and our optimized replacement scheme. (d) Off-chip memory accesses resulting from the use of our optimized memory layout and our optimized replacement scheme. . . . .	83
4.3	An example of Mem-layout. (a) An access graph $G$ for the index sequence in Fig. 4.1. (b) Select nodes $n_1$ and $n_4$ whose edge has the largest $eSize(\cdot)$ value. (c) Update $G$ by merging $n_1$ and $n_4$ , then repeat this process by selecting $\{n_1, n_4\}$ and $n_7$ . (d) Merge $\{n_1, n_4, n_7\}$ and $n_3$ . (e) The iteration stops since there is no pair of nodes to merge that meets $nSize(\cdot) + nSize(\cdot) \leq  B  = 4$ . . . . .	87
4.4	Flow of selecting a block for replacement by MIN-k. . . . .	90
4.5	An example of calculating one layer with two PEs. Each PE includes on-chip memory and MAC operation unit as shown in Fig. 4.1(b). The red line represents the calculation performed at PE1 and the blue line represents the calculation performed at PE2. Each PE has their own weight index sequence. . . . .	92
4.6	Changes in the number of off-chip memory accesses by Mem-layout for the conventional replacement schemes under various sizes, in terms of the number of blocks, of on-chip memory (The size of a unit block is 4 in terms of the number of weights.) . . . . .	96
4.7	Curves showing the trade-off between $E_{off}$ and $E_{scan}$ as the value of scanning distance parameter $k$ changes. . . . .	99



# Chapter 1

## Introduction

### 1.1 Deep Neural Networks and Its Challenges

Deep neural networks (DNNs) have been inspired by biological neural networks. In biological neural networks, impulses from dendrites are carried toward the cell body, and new impulses are transmitted to another cell body through the axon when the stimulus collected in the cell body exceeds the threshold. In DNNs, imitating this idea, each layer is composed of nodes corresponding to the cell bodies and edges toward the next layer's nodes, and a quite large number of layers are arranged repeatedly.

With the introduction of DNNs, remarkable progress and improvement have been made in various fields. In particular, since the advent of AlexNet [1] DNN in ImageNet Large Scale Visual Recognition Competition (ILSVRC; the most representative competition for image classification) in 2012, several outstanding DNNs (e.g., VGGNet [2], GoogLeNet [3], ResNet [4], Xception [5], SENet [6], GPipe [7]) have significantly improved the recognition accuracy.

Fig. 1.1 shows top-5 classification error rates for each model of ILSVRC. For example, VGGNet [2] in 2014 achieved an error rate of 7.3%. ResNet [4] in 2015 achieved an error rate of 3.5%, which has shown that neural network models can recognize images with less misclassification than humans. In addition, long short-term

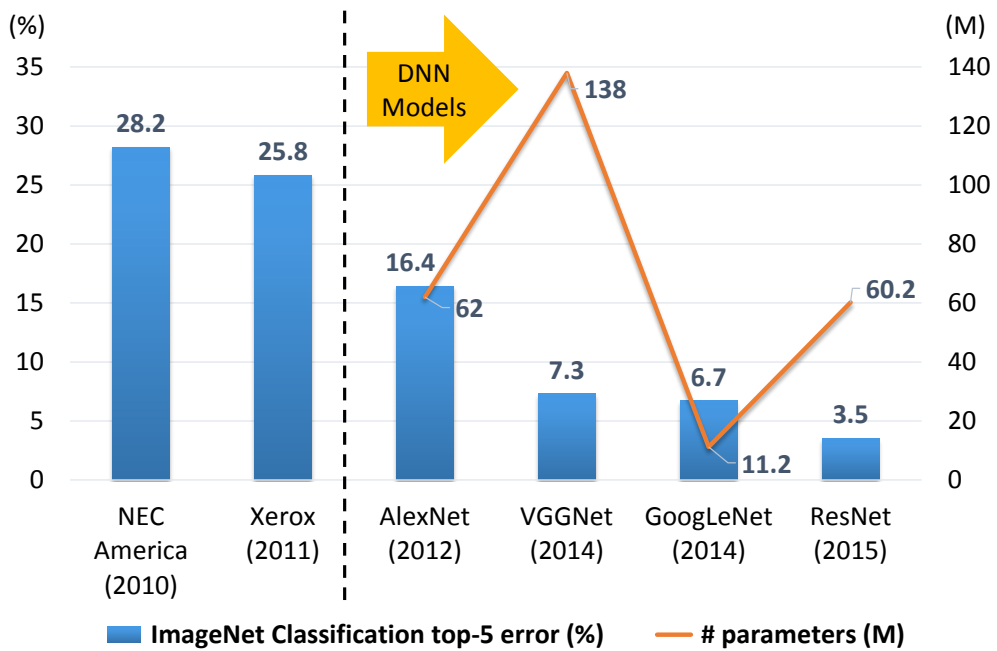


Figure 1.1: Top-5 classification error rates and the number of parameters for each model of ILSVRC. Since the introduction of DNN models, the top-5 error rates have significantly decreased.



memory (LSTM) [8] and gated recurrent unit (GRU) [9], which are variants of DNNs, have made outstanding progress on natural language processing (NLP). Besides, various models based on DNN show good performance on the real-time object detection [10, 11, 12] and semantic segmentation [13, 14].

According to that DNN proves its efficiency in various fields and computing technology greatly advances, DNN has been proposed as a solution to many problems. Early DNNs have aimed at improving application performance, which motivates to design deeper and wider networks to achieve the goal. However, its demand for excessive memory space makes it difficult to efficiently perform DNN inference. As shown in Fig. 1.1, which also shows the number of parameters for each model of ILSVRC, a huge amount of parameters are still needed to build models. For example, AlexNet in Caffe model representation needs nearly 240MB memory to store all the parameter values, and VGG-16 in Caffe model needs 500MB memory [16]. Since most devices have a limited on-chip memory space, the whole parameter values cannot be stored into the on-chip memory, which means many memory read invocations are required to access the parameter values from off-chip memory to on-chip memory. The penalty caused by these off-chip memory accesses is significant. For example, by referring to the energy consumption numbers of memory operations in a 45nm CMOS process in Table 1.1, fetching a 32-bit data from off-chip LPDDR2 DRAM consumes 640pJ, which is 128 times more than that of fetching the data from on-chip SRAM.

Furthermore, its intensive computation requirement hinders the installation of DNNs. In particular, in the convolutional neural networks (CNNs), the convolutional operations between input activations and filter weights performed in successive layers occupy almost all of the DNN operations. According to [17], more than 92% of processing time is consumed in the multiply-and-accumulate (MAC) operations between input activations and filter weights in the convolutional layer.

In order to overcome these problems, various studies have been conducted such as reducing the size of DNN models or proposing techniques for efficiently performing

DNN computation.

## 1.2 Redundant Weight Elimination Methods in DNN

In earlier researches, it has been aimed only at improving the computational speed-up through parallelization of the MAC operations between input activations and weights [17, 18]. Representative examples of high performance and low energy architectures are those in [19, 20], developed with emerging non-volatile memories such as resistive RAM. However, they did not efficiently utilize the hardware resources in that they overlooked the resource waste on MAC operations with zero-valued weights or zero-bits in weight representation that never contribute to the quality of the computation outcomes.

In this section, we review a number of representative techniques that eliminate redundant parameters in DNN model or redundant operations in DNN inference computation. To improve computation efficiency, many studies have attempted various approaches. Thus, numerous studies have focused on making memory and computationally expensive DNN inference energy-efficient and fast in various hardware devices such as CPU, GPU, FPGA, and ASIC.

A low-rank approximation method has been proposed to eliminate redundancy for the weight of DNNs. For a weight matrix  $W$  having a size of  $m \times n$ ,  $W$  can be expressed as a product of two full-rank matrices  $U$  and  $V$  having a size of  $m \times r$  and  $r \times n$ , respectively. The number of parameters  $mn$  of the weight matrix  $W$  is replaced by  $(mr + rn)$ , so it decreases if  $(mr + rn) < mn$  is satisfied. Jaderberg, Vedaldi, and Zisserman [21] achieves a  $4.5\times$  speedup while approximating a full-rank filter bank as combinations of a rank-1 filter basis. Denton *et al.* [22] achieve  $2\text{-}2.5\times$  speedup with negligible classification performance drop by applying a low-rank approximation to the first layers of CNN. Taking one step further, there have been several studies that have achieved performance improvement through tensor decomposition by consider-

ing kernels as 3D tensors for a single convolutional layer of CNN [23, 24, 25, 26].

Various acceleration architectures have taken into account attenuating the computation waste and reducing the DNN model size, such as *weight pruning* [27, 28, 29, 30, 31, 32, 33], which eliminates some of the weights to exploit its sparsity, and *weight sharing* or *quantization* [34, 35, 36, 37, 38, 39], which shares a small number of representative weights. Chen *et al.* [40] propose a DNN compression model called HASHNET, which expresses all parameter values (also referred to as weights) on the edges (i.e., synapses) between neuron nodes in every layer of DNN with a limited number of distinct values through *weight sharing*: initially, they are given a hash function  $h(i, j)$  and a number  $K$  such that  $h(i, j)$  returns an integer value in  $[0, K - 1]$  for the edge weight between neurons  $i$  and  $j$ . Thus, the hash function leads to weight sharing; then, they perform DNN training under the weight sharing constrained by the hash function; finally, the trained  $K$  values will be stored in on-chip memory. DNN then will use the hash function to access weights during the inference stage. One fundamental drawback of HASHNET is that since the training is constrained by the predetermined hash function, the accuracy loss is inevitable.

On the other hand, rather than employing hash function, Han, Mao, and Dally [41] proposed a three-step compression model, called DEEPCOMPRESSION: (1) for an initially trained uncompressed DNN, prune edges with nearly zero-weight; (2) cluster the remaining unpruned weights in a way that weights with a small difference should be grouped to the same cluster, from which a representative value is assigned to each cluster; (3) apply Huffman encoding to the edge indexes based on the occurrence frequency of accessing the cluster values. Note that DEEPCOMPRESSION would have less accuracy loss than HASHNET since the weight clustering is performed with no hash function constraint. However, in this process, a time-consuming repetitive retraining process is inevitable and an accuracy loss still occurs due to the compressed weights. Also, it requires ‘index’ storage space that is used to link from each edge index to an address of weight memory.

Table 1.1: Energy table for 45nm process [15]

<b>Operation</b>	<b>Energy (pJ)</b>
32-bit int ADD	0.1
32-bit float ADD	0.9
32-bit int MULT	3.1
32-bit float MULT	3.7
32-bit SRAM access	5.0
32-bit DRAM access	640

Table 1.2: Fraction of zero-valued weights [42]

<b>Models</b>	<b>Zero weights</b>
AlexNet	0.093%
GoogLeNet	0.050%
VGG-16	0.156%
VGG-19	0.182%
NiN	0.193%
<b>GeoMean</b>	<b>0.135%</b>

Meanwhile, studies for accelerating the DNN inference process by eliminating unnecessary operations while maintaining model accuracy by not changing any values of the weights that have already been trained have been simultaneously conducted. These can be classified into two methods: *word-level* weight pruning and *bit-level* weight pruning. Word-level weight pruning skips the weights with zero values from the operation to reduce inefficient operations caused by zero-valued weights with no accuracy loss [43, 44, 45]. However, as shown in Table 1.2, the ratio with the weight value of zero is only 0.135% of the total weight on average [42] for the typical DNN models, so the performance improvement is not significant. Bit-level weight pruning<sup>1</sup> is a method of avoiding zero-bits in the value representation of weights in the MAC operations of inference computation. The work in [42] called Tetris has proposed to extract non-zero bits in a bit column after stacking multiple weights together with split-and-accumulate (SAC) calculation-based accelerator architecture suitable for bit column-wise condensed weights. Overall, it achieves  $1.3\times$  to  $1.5\times$  the inference computation performance improvement compared to DaDianNao [17], although it has  $1.13\times$  more area.

In addition, prior studies have widely conducted bit-serial processing similar to bit-level pruning. Stripes [46] has extended the DaDianNao [17] accelerator by proposing a method to sequentially process each bit of input activation with pre-specified per layer precisions. Pragmatic (PRA) [47] has taken one step further by removing the zero bits that are still present in the input activations in addition to the zero prefix and suffix bits, thereby achieving 10% more energy efficiency and  $1.4\times$  speedup. However, since each input activation varies depending on the model input, additional hardware logic is required to perform bit-level pruning for the input activation. Loom [48] has proposed an accelerator architecture that supports bit-serial processing approach for not only convolutional layers but also fully-connected layers. For the long short-term

---

<sup>1</sup>In a strict sense, it means to “zero-bit elimination”. However, we use, in this dissertation, the general term “bit-level weight pruning” to be consistent with the terms used in the prior works.

memory (LSTM), BLINK [49] has reduced circuit area and power consumption by 38.7% and 38.4%, respectively, without loss of accuracy by making use of bit-sparse representation.

Also, several studies have tried to reduce the DNN model parameters by changing the existing convolution method differently. SqueezeNet [50] introduces the Fire module composed of  $1\times 1$  and  $3\times 3$  filters to reduce a model size while maintaining the AlexNet-level accuracy. MobileNetV2 [51] uses depth-wise separable convolution rather than the standard convolution in order to build a very small DNN model for mobile devices. ShuffleNet [52] utilizes point-wise group convolution and channel shuffle.

### **1.3 Redundant Representation Elimination Methods in DNN**

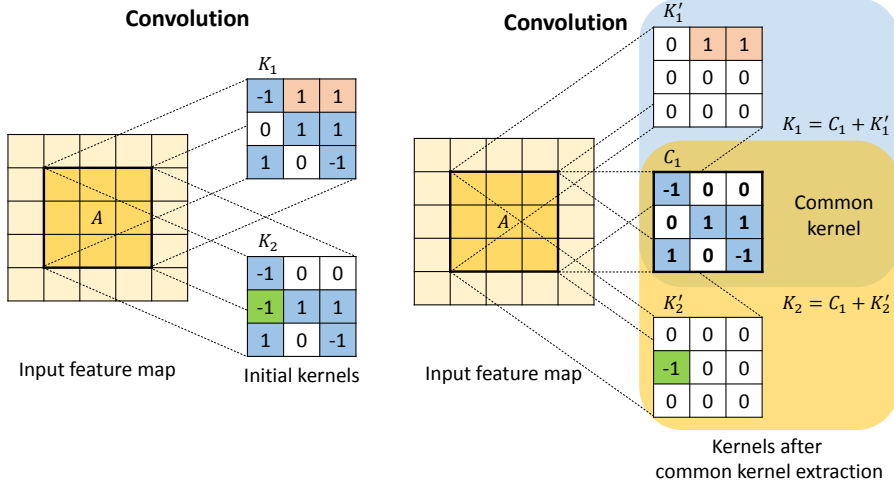
One noticeable research direction is to reduce the size of bit-width of DNNs' weights or activations to speed up the computation as well as to save the power consumption on the DNN inference while maintaining the loss of accuracy minimally. Most of the arithmetic operations of DNNs were based on the 32-bit floating-point representation for high accuracy. However, sufficiently good performance is achieved even if the numerical precision is reduced to 32-bit, 16-bit, and 10-bit fixed-point representation [53, 54, 55]. Further memory space reduction is possible without loss of accuracy through 8-bit quantization [41].

Recently, it has been proposed to express DNN weights or input activations extremely as binary. BinaryConnect [56] uses weights limited to two values (-1 and +1) to train a neural network. XNOR-Net [57] binarizes both weights and inputs to convolutional layers. In addition, DoReFa-Net [58] trains CNNs using low bit-width parameter gradients. Furthermore, Bitwise Neural Network [59] not only binarizes the input, output, weight, and bias terms but also replaces floating-point or fixed-point operations with fast and simple logical operations. On the other side, ternary DNNs

[60, 61, 62, 63], in which every weight value is constrained to be one of three numbers (-1, 0, and +1), are proposed to increase the accuracy of image classification over the binary DNNs at the expense of the compression rate. Consequently, though a little accuracy loss occurs, all multiplication operations between the input activations and kernel weights for convolution in a layer can be replaced by addition and subtraction operations, thereby significantly improving the hardware efficiency. In addition, the work in [64] shows that by transforming the original kernels in a convolutional layer to produce overlapping sparser kernels (we call it *common kernels*) in the binary- or ternary-weight CNNs and convolving the input feature map with the transformed kernels on behalf of the original kernels, it is possible to significantly reduce the total number of addition/subtraction operations required for all convolutions on the layer.

Fig. 1.2 shows an illustrative example of the common kernel extraction and utilization in convolution computation. Fig. 1.2(a) shows two convolutions of the original kernels  $K_1$  and  $K_2$  without utilizing common kernels, resulting in 13 total number of operations. On the other hand, Fig. 1.2(b) uses common kernel  $C_1$  of  $K_1$  and  $K_2$ . Thus, the convolutions  $A \cdot K_1$  and  $A \cdot K_2$  can be evaluated by computing  $A \cdot C_1 + A \cdot K'_1$  and  $A \cdot C_1 + A \cdot K'_2$  where  $K'_1$  and  $K'_2$  indicate the  $K_1$  and  $K_2$  with no  $C_1$ , satisfying the element-wise additions  $K_1 = C_1 + K'_1$  and  $K_2 = C_1 + K'_2$ , respectively. Since  $A \cdot C_1$  takes 5 operations and the resulting convolution value can be shared when computing the convolutions  $A \cdot C_1 + A \cdot K'_1$  and  $A \cdot C_1 + A \cdot K'_2$ , the total number of operations for the CNN convolutions is  $5 + (2+1) + (1+1) = 10$ , which results in eliminating 3 redundant addition and subtraction operations in Fig. 1.2(a).

YodaNN [65] is the firstly proposed binary-weight CNN accelerator. By avoiding the expensive multiplication operations in convolutions and reducing I/O bandwidth and storage for weight values, it significantly saves the energy consumption while speeding up the convolution process. However, it does not take into account the exploitation of common kernels potentially present like the one in Fig. 1.2(b) to remove the redundancy in the binary-weight convolution computation. The work in [66] de-



$$\begin{array}{r}
 A \cdot K_1 \rightarrow 7 \text{ OPs} \\
 +) A \cdot K_2 \rightarrow 6 \text{ OPs} \\
 \hline
 \mathbf{13 \text{ OPs}}
 \end{array}$$

$$\begin{array}{r}
 A \cdot C_1 \rightarrow 5 \text{ OPs} \dots \textcircled{1} \\
 \textcircled{1} + A \cdot K_1' \rightarrow 3 \text{ OPs} \\
 +) \textcircled{1} + A \cdot K_2' \rightarrow 2 \text{ OPs} \\
 \hline
 \mathbf{10 \text{ OPs}}
 \end{array}$$

(a) The number of addition/subtraction operations in convolutions  $A \cdot K_1$  and  $A \cdot K_2$  is  $7 + 6 = 13$  where  $K_1$  and  $K_2$  are original kernels.

(b) By extracting common kernel  $C_1$  from  $K_1$  and  $K_2$ , the number of operations in convolutions  $A \cdot C_1$ ,  $A \cdot K_1'$ , and  $A \cdot K_2'$  including two addition operations of the resulting values of the three convolutions is  $5 + (2+1) + (1+1) = 10$ .

Figure 1.2: An example of extracting a common kernel and sharing its convolution to reduce the redundancy in convolution computation.



composes every original binary-weight kernel into exactly two kernels, called base kernel and filtered kernel. Every position in the base kernel contains the value of -1, and the filtered kernel is the one formed by filtering out the positions containing the value of -1 in the original kernel. Thus, the convolution result on the base kernel can be shared among all convolutions on the original kernels. The limitations of the kernel decomposition method are that it is not applicable to ternary-weight CNNs and it is infeasible to exploit multiple base kernels.

On the other side, the work in [64] overcomes the limitations in [66] by extracting multiple common kernels. More precisely, for a set  $\mathcal{S}$  of kernels, initially consisting of all original kernels, (1) it extracts the largest common kernel (i.e., the kernel containing the largest number of non-zero common values) for every pair of kernels in  $\mathcal{S}$  and selects the common kernel  $C$  that maximizes the sharing (i.e., overlapping) among all kernels in  $\mathcal{S}$ ; (2) it then updates  $\mathcal{S}$  by including  $C$  and replacing every kernel in  $\mathcal{S}$  that embeds  $C$  by its filtered kernel with respect to  $C$ , as shown in Fig. 1.2(b); it repeats (1) and (2) until there is no common kernel with at least two non-zero values. Besides  $C$  in ternary-weight CNNs, its opposite ( $+ \leftrightarrow -$ ) kernel should also be considered for the maximal common kernel candidates. This method entails three critical limitations:

- **Limited search space:** Since it considers the maximal common kernels between *every pair* of kernels in  $\mathcal{S}$ , it severely limits the search space for the exploration of common kernels.

- **Lack of exploiting common expressions of convolutions:** It does not take into account the possibility of sharing the evaluation results of common (addition/subtraction) expressions of the convolution values of common kernels on deriving the final outputs corresponding to the convolution values of original kernels. (We refer such common expressions to as *common convolutions*.)

- **Lack of controlling or minimizing the number of kernels:** Since all kernels to be used in the convolution process should be stored in an external memory, it is highly important to control or minimize the total number of resulting kernels. Nevertheless,

it does not address controlling/minimizing the number of kernels during the common kernel extraction.

## 1.4 Contributions of This Dissertation

In this dissertation, we present several methodologies to overcome excessive memory and computation requirement problems for high performance neural network accelerator.

In Chapter 2, we address a new *bit-level* weight pruning which we call DWP (deep weight pruning). The inherent limitation of the existing accelerator supporting bit-level pruning without accuracy loss is that the speedup is limited. Precisely, for a DNN with  $N$  total number of bits in the representation of all weights, the maximal speedup cannot be more than  $\frac{n_e}{N}$  where  $n_e$  is the total number of *essential* bits<sup>2</sup> in the value representation of all weights. For example, +62 can be expressed as  $(00111110)_2$  in two's complement representation. Assuming that one bit per cycle is processed, 8 cycles are required whereas 5 cycles are needed when 0-bits are skipped. In this case, the maximum achievable speedup is  $1.6\times$ , and this value could not be changed unless the value of the weight is changed. So far, it is challenging to push down the amount of  $n_e$ , though it is an important factor that critically determines the inference performance.

The contribute of this work can be summarized as follows:

1. We propose a *transformation technique* to increase the degree of freedom for each weight representation using a signed-digit encoding, by which we transform the fixed-point representation of every weight into a set of signed-digit representations of *minimal number of essential (i.e., non-zero) bits*.<sup>3</sup>

---

<sup>2</sup>In binary representation, *essential* bits refer to all bits other than 0-bits (i.e., 1) while in signed-digit representation, *essential* bits indicate -1 as well as 1.

<sup>3</sup>In this dissertation, we consistently use the term 'bit' rather than 'digit' to indicate '-1' as well as 0 and 1 in signed-digit representation unless confusion occurs.

2. We formulate the problem of selecting the signed-digit representation of weights with the objective of *maximal digit-serial parallelism* among the weights into a *multi-objective shortest path problem* and solved it effectively using an approximation algorithm.
3. We propose *a novel hardware architecture* which is able to exploit the bit-level pruning fully and effectively on the signed-digit representation of weights.
4. We also propose DWP-intra (a variant of DWP) which is suitable for bit-level parallelism exploiting signed-digit representations with the capability of predicting a tight worst-case latency of the parallel processing. Our signed-digit transformation technique is able to reduce not only the number of essential bits in the same bit column on a set of weights stacked vertically but also the number of essential bits for each weight. (Note that both techniques cannot be applied at the same time.) Thus, it is also possible to perform weight-wise (i.e. horizontal) condensation by changing the compressing direction of the column-wise (i.e. vertical) condensation in DWP. We call the parallel processing based on the horizontal condensation DWP-intra (deep intra-weight pruning).
5. Through experiments, it is shown that our proposed signed-digit based weight pruning is able to reduce the number of essential bits by 69% on AlexNet, 74% on VGG-16, and 68% on ResNet-152, thereby DWP and DWP-intra accelerating the inference time up to  $2.22\times$  and  $3.57\times$  over the conventional bit-level weight pruning, respectively. Further, our supporting architecture has achieved an energy-delay product (EDP) improvement of up to  $1.42\times$  with a slight area overhead over the state-of-the-art architecture supporting bit-level weight pruning.

In Chapter 3, we propose a new algorithm called ConvOpt for common kernel and common convolution extraction to fully and effectively eliminate the redundancy

in the convolution computations in binary- and ternary-weight DNNs as well as to take into account controlling/minimizing the number of resulting kernels.

The contributions of this work can be summarized as:

1. We propose *a new algorithm of common kernel extraction* to overcome the local and limited exploration of common kernel candidates by the existing method.
2. We subsequently apply *a new concept of common convolution extraction* to maximally eliminate the redundancy in the convolution operations.
3. We develop an algorithm, which is able to *tune in minimizing the number of resulting kernels* for convolutions, thereby saving the total memory access latency for kernels.
4. Experimental results on ternary-weight VGG-16 demonstrate that our convolution optimization algorithm is very effective, reducing the total number of operations for all convolutions by 25.8-26.3%, thereby reducing the total number of execution cycles on hardware platform by 22.4% while using 2.7-3.8% fewer kernels over that of the convolution utilizing the common kernels extracted by the state-of-the-art algorithm.

In Chapter 4, this work is the first study to answer the following question: (**unfitted compression**) “*we attempted to aggressively compress the original DNN in the hope that the reduced parameters can be entirely contained in the on-chip memory under the budget constraint of accuracy loss, but failed in fitting them into the memory. Then, what should we do to mitigate the inherent off-chip memory access problem?*” As the complexity and size of DNNs grow rapidly in many applications, compressing DNNs is not being always successful in that the number of distinct weights by compression, together with indexes if hash function is not used, is not small enough to be contained in on-chip memory. In this state, the best solution is, in the inference stage, to use the number of off-chip memory accesses as minimal as possible. Fortunately, since

we have already known the distribution of shared (or clustered) weights produced by the compression as well as the entire index sequence to be accessed for processing inference computation on a given DNN architecture, we can make use of those information in two ways: (**memory layout**) arranging weights in off-chip memory so that the number of (block-based) off-chip memory accesses is minimized and (**block replacement**) requiring a mechanism to minimize the on-chip misses for the known index access sequence. Through experiments with the model of compressed AlexNet, it is shown that our solutions are able to reduce the total energy consumption of the off-chip memory accesses including the scanning overhead by 34.2% on average over the use of unoptimized memory layout and LRU replacement scheme.



## Chapter 2

# Bit-level Weight Pruning Techniques for High-Performance Neural Networks

## 2.1 Preliminary

### 2.1.1 Bit-level Weight Pruning in Binary Representation

An operation  $F$  that performs MAC on  $N$  pairs of weight  $W_i$  in  $B$ -bit fixed-point representation and input activation  $A_i$  of the same bits for  $0 \leq i \leq N - 1$  can be expressed as:

$$F = \sum_{i=0}^{N-1} (A_i \times W_i) = \sum_{b=0}^{B-1} \left( 2^b \times \sum_{i=0}^{N-1} (A_i \times w_i^b) \right) \quad (2.1)$$

where  $w_i^b$  represents the  $b^{\text{th}}$  bit value in  $W_i$ . ( $B$  is 8 or 16 in many DNN models.) Then,  $F$  can be evaluated in the following two steps:

- **Step 1.** *Bit-level multiplication:*

$$X^b = \sum_{i=0}^{N-1} (A_i \times w_i^b), \quad b = 0, \dots, B - 1 \quad (2.2)$$

- **Step 2.** *Shift-and-accumulation:*

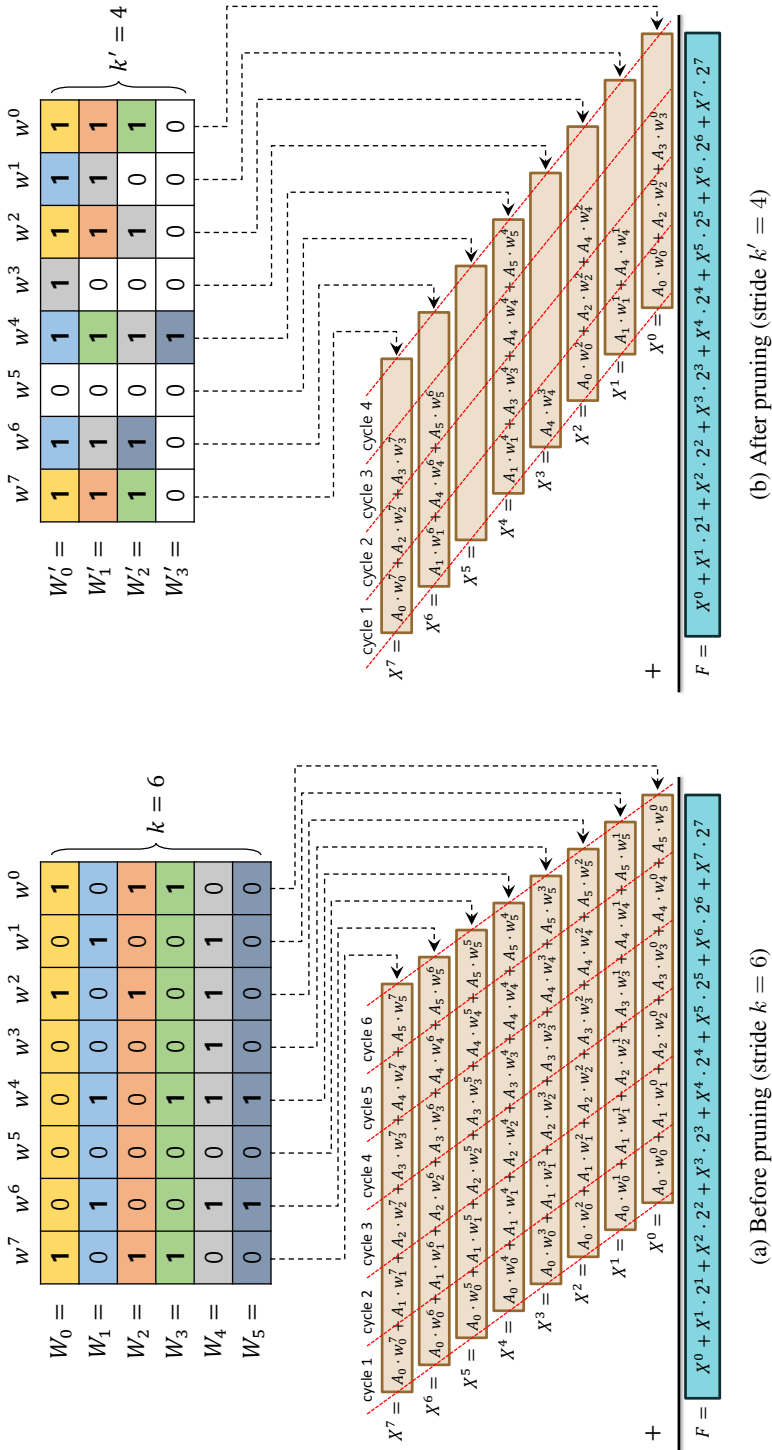


Figure 2.1: Conventional bit-level pruning [42] for weights in binary representation. (a) 6 weights of an 8-bit fixed-point representation before pruning. For performing Eq. 2.2 in parallel on each bit column  $b$ , 6 execution cycles are required. (b) 4 weights after the application of bit-level pruning. By reducing the number of weights from 6 to 4, 2 execution cycles are saved.



$$F = X^0 + X^1 \cdot 2^1 + X^2 \cdot 2^2 + \dots + X^{B-1} \cdot 2^{B-1} \quad (2.3)$$

The *bit-level weight pruning* is applied to the bit-level multiplication in Step 1 by pruning the computation of  $A_i \times w_i^b$  if  $w_i^b = 0$ . Since the weight values have already been known when performing inference, a bit column-wise condensed arrangement on the bit values in  $W_i, i = 0, \dots, N - 1$  is possible by pulling out the 0-bits from the arrangement to accelerate the computation of  $X^b, b = 0, \dots, B - 1$  in Eq.2.2.

A well-known conventional bit-level weight pruning called *weight kneading* [42] is illustrated in Fig. 2.1, in which the bit values of the six weights (i.e.,  $N = 6$ ), one row for each weight as shown in Fig. 2.1(a), are rearranged as shown in Fig. 2.1(b) by removing 0-bits and moving up 1-bits column-wise. Consequently, if a bit-level multiplication in Eq.2.2 for each row takes one clock cycle, the compressed arrangement of bit values enables to reduce the total number of execution cycles from 6 to 4.

We use notation  $k$  to refer the number of initial weights to be collectively considered for bit-level pruning, which we call *pruning stride* while we use notation  $k' (< k)$  to refer the number of weights compressed via bit-level pruning (e.g.,  $k = 6$  and  $k' = 4$  in Fig. 2.1).

In the weight kneading process, for each 1-bit, the information of the corresponding activation (i.e., which activation should be matched) should be retained. This information is the *activation selection index*, and  $\lceil \log_2 k \rceil$  bits are required for a set with  $k$  weights. In Fig. 2.1(b), 1-bits of different background colors indicate different activation selection indexes. A detailed description of the activation selection index is given in Sec. 2.5.2.

### 2.1.2 Bit-level Weight Pruning in Signed-digit Representation

The two-step MAC operation on the binary weight described in Sec. 2.1.1 can be extended to the ternary weight represented by signed-digits.  $w_i^b$  in both Eq.2.1 and Eq.2.2 has a value in  $\{0, 1\}$  for binary weights, but has a value in  $\{0, 1, -1\}$  for weights



in signed-digit representation. We want to cut the operations of  $A_i \times w_i^b$  for  $w_i^b = 0$  to process only the operations with  $w_i^b = 1$  or  $w_i^b = -1$  for calculating  $X^b$  in Eq.2.2.

Fig. 2.2(a) illustrates a bit-level pruning for weights in signed-digit representation comparable to that in Fig. 2.1(b), which shows bit-level pruning for weights in binary representation.

The MAC operation discussed in Sec. 2.1.1 can also be performed by the following two steps.

- **Step 1.** *Shift-and-add:*

$$Y_i = \sum_{b=0}^{B-1} \left( A_i \times w_i^b \times 2^b \right), \quad i = 0, \dots, N-1 \quad (2.4)$$

- **Step 2.** *Accumulation:*

$$F = Y_0 + Y_1 + \dots + Y_{N-1} \quad (2.5)$$

For weights in signed-digit representation, we apply bit-level pruning in a way to skip the calculation of  $A_i \times w_i^b \times 2^b$  if  $w_i^b = 0$  in Eq.2.4, which we call *bit-level intra-weight pruning*. Fig. 2.2(b) illustrates the vertical parallel computation using the bit-level intra-weight pruning in Eq.2.4 as opposed to that in Fig. 2.1(b) corresponding to the horizontal parallel computation using the bit pruning in Eq.2.2. We describe our proposed weight pruning technique for the horizontal parallel computation in Sec. 2.3, followed by our intra-weight pruning technique for the vertical parallel computation in Sec. 2.5. It should be noted that our exploitation of signed-digit representation is nothing else but to perform an effective bit-level weight pruning.

### 2.1.3 CSD Representation Conversion

A value in sign-magnitude or two's complement representation does not have a unique signed-digit representation. Canonical signed-digit (CSD) representation is a special form of signed-digit representation. First, let us consider an unsigned binary number

$X = (x_{n-1}, x_{n-2}, \dots, x_0)$  such that

$$X = \sum_{i=0}^{n-1} (x_i \times 2^i), \forall x_i \in \{0, 1\} \quad (2.6)$$

where  $x_{n-1} = 0$  to convert to  $Y = (y_{n-1}, y_{n-2}, \dots, y_0)$  in CSD representation such that  $y_i \in \{-1, 0, 1\}$ ,  $i = 0, \dots, n-1$ . Reitweisner [67] proposed a simple *right-to-left algorithm* to produce  $Y$ , as explained in the following.

It scans the bits in  $X$  from the rightmost bit toward the left and replaces 1-bit substrings delimited by 0-bits or rightmost boundary of size greater than one with substrings of one bit increased size such that their leftmost bits is 1, the rightmost bits are  $\bar{1}$ , and the rest are all 0-bits. For example, for  $X = 237 = (011101101)_2$ , substring 11 delimited by 0-bits is replaced with  $10\bar{1}$ , so that  $(011101101)_2$  becomes  $(011110\bar{1}01)$ . Then, the scanning continues and replaces maximal substring 1111 with  $1000\bar{1}$  to produce  $Y = (1000\bar{1}0\bar{1}01)$  in CSD representation.

For negative two's complement binary number, we transform it into sign-magnitude binary number  $X$  and apply the CSD conversion method to its positive binary number i.e.,  $-X$ . Once CSD representation  $Y$  for  $-X$  is obtained, we switch every 1-bits in  $Y$  to  $\bar{1}$ -bits and every  $\bar{1}$ -bits in  $Y$  to 1-bits to produce  $Y'$  in CSD representation for  $X$  in binary representation. For example, for  $X = -237 = (111101101)_2$  in sign-magnitude representation, we produce  $-X = 237 = (011101101)_2$ , which is then converted into  $Y = (1000\bar{1}0\bar{1}01)$  in CSD representation. Then, we obtain  $Y' = (\bar{1}0001010\bar{1})$  from  $Y$ .

The CSD representation can also be obtained directly without converting to sign-magnitude representation through the following process. First, for  $(100010011)_2$  in two's complement representation of  $X = -237$ , we flip each bit (e.g., 0-bits convert to 1-bits and 1-bits convert to 0-bits) and add 1 to obtain two's complement of  $X$  namely  $X' = (011101101)_2$ . Then, we apply the right-to-left algorithm to  $X'$  to produce  $-Z = (1000\bar{1}0\bar{1}01)$ . Finally, we switch every 1-bit in  $-Z$  to  $\bar{1}$ -bit and every  $\bar{1}$ -bit in  $-Z$  to 1-bit to produce  $Z = (\bar{1}0001010\bar{1})$ .

CSD representations produced by using the method has the following properties [67]:

1. There are no two or more consecutive non-zero bits in the CSD representations.
2. The converted CSD representation of a number is unique.
3. The converted CSD representation of a number has at most  $n/3 + 1/9 + O(2^{-n})$  number of non-zero bits for an  $n$ -bit number [68].

Note that property 3 provides the bound on the number of essential bits in CSD representations, which is the basis on building up DWP-intra, which will be described in Sec. 2.5.

## 2.2 Motivations

### 2.2.1 Inefficiency in Two's Complement Representation

Two's complement number system is commonly used to represent signed fixed-point values. One biggest advantage of two's complement over the other number systems is that the fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those of unsigned binary numbers. This property makes the system simple to implement. However, it is not true in the domain of bit-level weight pruning.

For example, for -13, its two's complement representation in 8 bits is  $(11110011)_2$  which includes 6 essential bits out of the 8 bits. On the other hand, if we use sign-magnitude representation, -13 is expressed as  $(10001101)_2$  which includes the leftmost sign-bit and 3 essential bits out of the remaining 7 bits. Fig. 2.3 shows the changes of the number of essential bits when converting to sign-magnitude representation for all negative numbers in 8 bits. For the case of numbers with small absolute value, the number of essential bits is smaller than that of the numbers with large absolute value. Thus, sign-magnitude representation is effective in reducing the number of essential

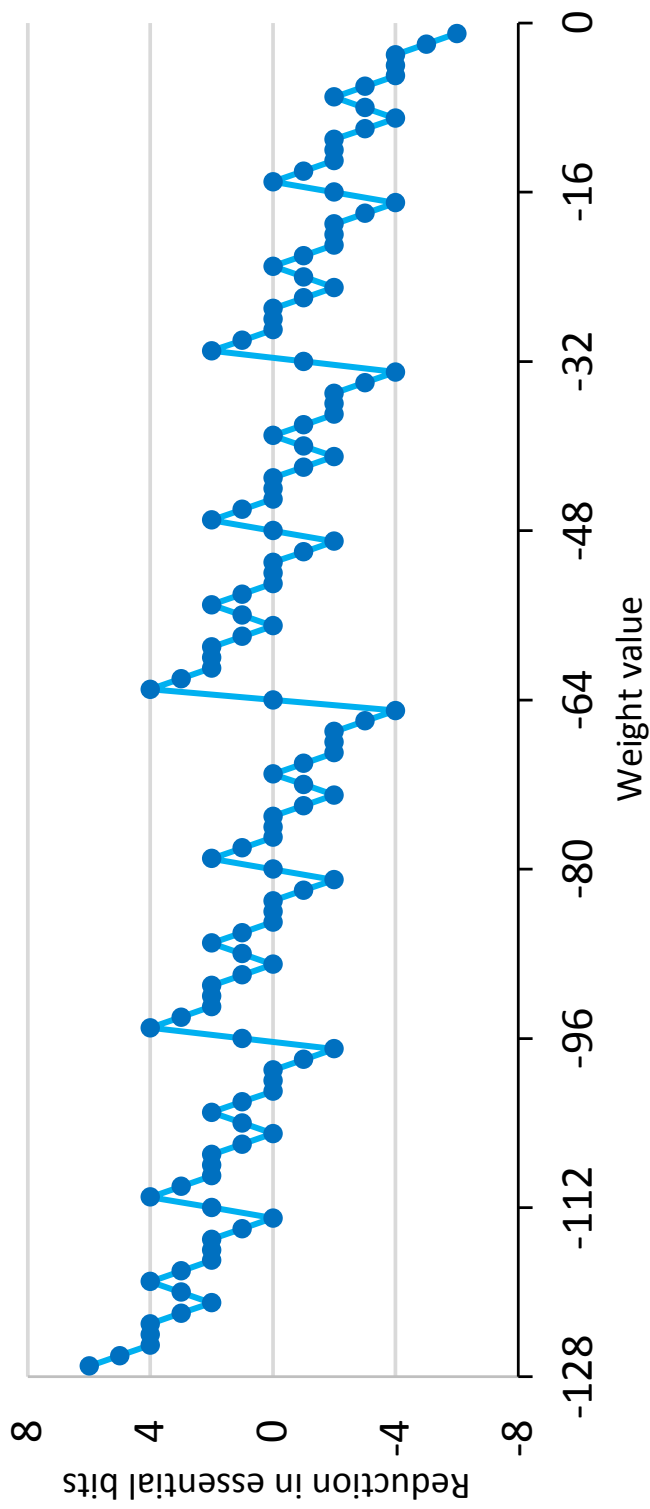


Figure 2.3: The reductions of the amount of essential bits for negative weights of 8 bits by converting them from two's complement representation to sign-magnitude representation.

bits when negative numbers with small absolute value occupy a majority. In addition, if we are able to handle the sign-bit efficiently<sup>1</sup> in sign-magnitude representation in performing the bit-level multiplication in *Eq.2.2*, it is desirable to apply bit-level pruning to weights in sign-magnitude representation rather than to weights in two’s complement representation.

Fig. 2.4 shows the comparison of the word/bit-level value distributions of the weights in AlexNet and VGG-16 trained with ImageNet dataset when expressing the weights in two’s complement and sign-magnitude representations. Fig. 2.4(a) shows the normalized weight value distributions, which form bell-shapes. Fig. 2.4(b) then shows essential bit (i.e., one-bit) distributions over the 16 bit-positions (LSB on the rightmost) when the weights are expressed in two’s complement and sign-magnitude representation. Since the number of essential bits in representing a near-zero negative weight in sign-magnitude representation is much smaller than that in two’s complement representation (the bars on bit-positions 11 to 14 in Fig. 2.4(b)), the sign-magnitude representation uses 31.3% fewer essential bits on AlexNet and 35.9% fewer essential bits on VGG-16 than the two’s complement representation.

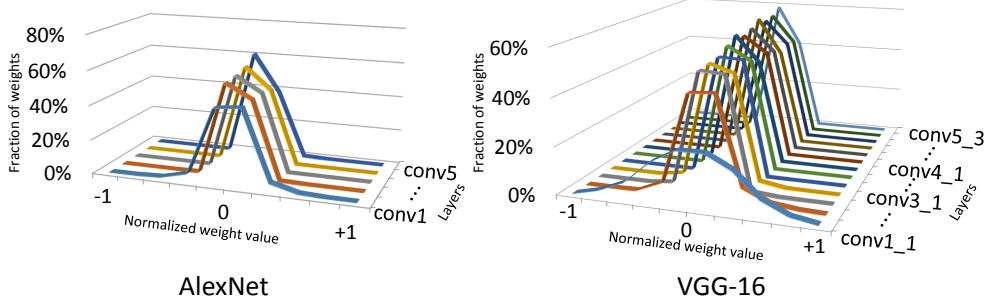
## 2.2.2 Inability to Exploit Signed-digit Representation

By converting the magnitude of the sign-magnitude representation or the two’s complement representation of every weight into the signed-digit representation, it allows to use the same or fewer number of essential bits. Considering a ternary numeral system in signed-digit representation using three digits of -1, 0, and 1, for example,  $(000011110)_2 (= 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 = 30)$  can be expressed as  $(0001000\bar{1}0)_2 (= 1 \cdot 2^5 + (-1) \cdot 2^1 = 30)$  in which  $\bar{1}$  denotes a digit value of -1, thereby saving two essential bits.

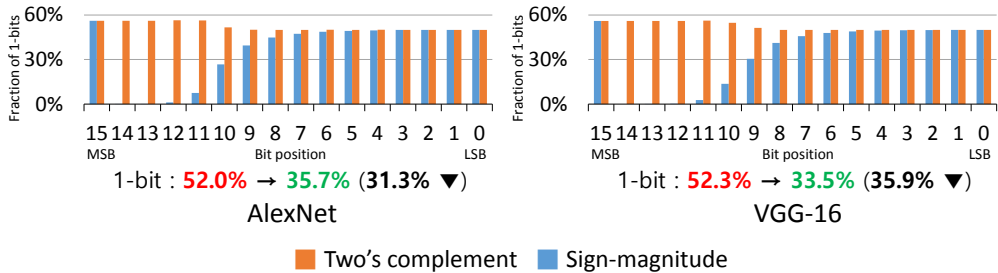
Utilizing the signed-digit representation for a deeper bit-level weight pruning requires to solve the following four problems:

---

<sup>1</sup>Our idea of efficiently handling the sign-bit will be described in Sec. 2.3.



(a) Normalized weight value distributions



(b) Essential bit (i.e., 1-bit) distributions and percentage of essential bits

Figure 2.4: Comparison of word/bit-level value distributions of the weights in AlexNet and VGG-16 trained with ImageNet dataset when using two's complement and sign-magnitude representations.



- P1. (**Multiple signed-digit representations**) Signed-digit representation for a number is not unique. For example, decimal number 103 ( $= (001100111)_2$ ) can be expressed as any of the following signed-digit representations of a fewer or equal number of essential bits:  $(00110100\bar{1})_2$ ,  $(010\bar{1}00111)_2$ , and  $(010\bar{1}0100\bar{1})_2$ . In Sec. 2.3, we *propose a method of selecting signed-digit representations that leads to maximal parallelism of bit-level multiplication in Eq.2.2.*
- P2. (**Memory space for representing ternary values**) Together with 0 and 1, signed-digit representation requires a memory space for encoding -1, which intuitively implies that for each digit, two bits are required, for example, 00, 01, and 10 respectively encode 0, 1, and  $\bar{1}$  in a signed-digit representation. Accordingly, the double memory space for storing weights in signed-digit representation would be required. In Sec. 2.4, we *devise a novel technique to use nearly the same amount of memory space as required for storing binary numbers of 0 and 1 though we use signed-digit representation.*
- P3. (**Sign-bit in sign-magnitude representation**) The sign-bit which is the leftmost bit in a sign-magnitude representation indicates whether the magnitude is positive ( $= 0$ ) or negative ( $= 1$ ). The sign-bit is one major obstacle to complicate the process of arithmetic operation, i.e., (1) depending on the sign-bit, the operation type of addition or subtraction is determined, and (2) by comparing the magnitudes, the sign-bit of resulting magnitude of the operation is set. In Sec. 2.3, we *propose a way of completely removing this burden by exploiting our extended notation of signed-digit representation.*
- P4. (**Supporting subtraction for  $\bar{1}$** ) Two's complement representation offers the most economical hardware for supporting subtraction as well as addition, but our signed-digit representation-based approach to minimizing the number of essential bits stems from sign-magnitude representation. In Sec. 2.4, we *propose an architectural technique which is able to seamlessly link the transformed*

*signed-digit representation to efficient hardware supporting two's complement operations* in performing the bit-level multiplication in Eq.2.2.

## 2.3 Signed-digit Representation-based Deeper Weight Pruning

Before generating signed-digit representations, Sec. 2.1.3 has introduced the CSD representation conversion method with our modification to fit into our pruning context. The first step in Sec. 2.3.1 is to generate all signed-digit representations with the minimal or near-minimal number of essential bits for every weight. The second step in Sec. 2.3.2 is then to select signed-digit representations among the ones obtained in the first step that leads to maximal parallelism on bit-level multiplication.

### 2.3.1 Generating Signed-digit Representations

Since all the weight values are known, generating possible signed-digit representations for weight values is processed *off-line*, and the generation is performed in two steps.

- **Step 1** (*Recursively applying the right-to-left algorithm*): For a weight value in sign-magnitude representation, we recursively apply right-to-left algorithm to the weight from the rightmost 1-bit string of size  $\geq 2$  to the left. Fig. 2.5(a) shows the *signed-digit enumeration tree* produced by recursively applying the conversion to  $(10110111)_2$  ( $= -55$ ) where 1 is sign-bit and the 0/1/ $\bar{1}$ -bit strings in red color are the signed-digit representations resulting from the 1-bit strings in the dotted circles of their parent nodes.

- **Step 2** (*Eliminating sign-bit for negative value*): Since signed-digit representation uses  $\bar{1}$  as well as 0 and 1, it is possible to convert the leftmost 1-bit for a negative value into 0-bit by converting 1-bit in every remaining bit to  $\bar{1}$  and  $\bar{1}$ -bit to 1. Fig. 2.5(b) shows the sign-bit (i.e., leftmost 1-bit) free signed-digit representations produced by



the application of such extended signed-digit conversion to every signed-digit representation obtained in Fig. 2.5(a).

Through Steps 1 and 2, we ensure that all possible signed-digit representations with a fewer or equal number of essential bits for every weight are available and there is no meaning on the leftmost sign-bit of every signed-digit representation. To boost up the opportunity of maximal parallelism of bit-level multiplication, rather than collecting the signed-digit representation of minimal essential bits only, we employ a user-defined *relaxing parameter*  $\gamma$  to additionally include the signed-digit representations of up to  $\gamma$  more essential bits over the minimal essential bits.<sup>2</sup>

### 2.3.2 Selecting Signed-digit Representations for Maximal Parallelism

We formulate the problem of selecting signed-digit representations that maximize bit-level parallelism into a variant problem of finding the shortest path in a graph. We describe our formulation using the example in Fig. 2.6, which illustrates a full flow of selecting signed-digit representations for maximal parallelism. The three columns at the left in Fig. 2.6 illustrate the signed-digit representation generation process performed in Sec. 2.3.1.

From the signed-digit representations, we construct a directed graph  $G(V, A, C)$  as shown in Fig. 2.6 where every signed-digit representation has a distinct node in  $V$  arranged horizontally in  $G$  for signed-digit representations for each weight (labeled as gray or blue), and  $V$  has two additional nodes (labeled as red): *src* having no entering arc, and *dest* having no leaving arc. There exists an arc between every pair of nodes  $v_{(\cdot)}^i$  and  $v_{(\cdot)}^{i+1}$  ( $v_{(\cdot)}^i \rightarrow v_{(\cdot)}^{i+1}$ ) where  $v_{(\cdot)}^i$  and  $v_{(\cdot)}^{i+1}$  represent nodes arranged in the  $i^{th}$  and  $(i + 1)^{th}$  horizontal lines in  $G$ , respectively. (In such convention,  $src = v_0^{-1}$  and  $dest = v_0^k$  where  $k$  is pruning stride.) We set a vector  $cost_{j_1, j_2}^{i+1} \in C$  for arc  $(v_{j_1}^i, v_{j_2}^{i+1}) \in A$  to a bit vector of the same bit-size of signed-digit representation such that for  $0 \leq b \leq B - 1$ ,  $c_{j_1, j_2}^{i+1}[b] = 1$ , if  $b^{th}$  bit value of  $j_2^{th}$  signed-digit representation for

---

<sup>2</sup>We set  $\gamma = 2$  for  $B = 8$  and  $\gamma = 4$  for  $B = 16$  in our experiments.

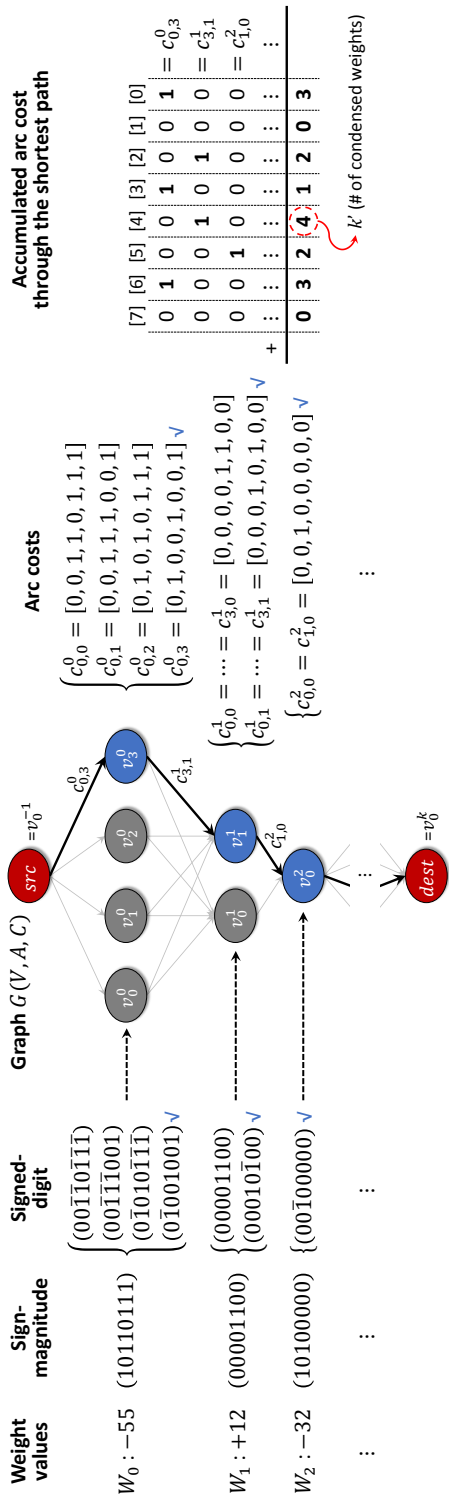


Figure 2.6: An illustration of full flow (from left to right) of selecting signed-digit representations for maximal parallelism of bit-level multiplication. The left of the graph illustrates the signed-digit representation generation process performed in Sec. 2.3.1. The graph in the middle shows our graph-based formulation of selecting signed-digit representations, and the arc cost setting is shown at the right of the graph. The alignment of essential bits for maximal parallelism taken from the selected signed-digit representations is placed at the right of the illustration.

$W_{i+1}$  is 1 or  $\bar{1}$ , and  $c_{j_1, j_2}^{i+1}[b] = 0$  otherwise, as illustrated in Fig. 2.6.<sup>3</sup>

Since we want to minimize the maximum among the numbers of essential bits in bit columns when the selected signed-digit representations are vertically stacked, as shown at the right in Fig. 2.6, we solve the signed-digit representation selection problem into a problem of finding a *multi-objective shortest path* (MOSP) from *src* to *dest* in  $G(V, A, C)$ . Since the decision version of MOSP problem is known to be NP-complete even when  $B = 2$  [69], we use Warburton’s polynomial-time  $\epsilon$ -approximation algorithm in [70], which guarantees the worst-case time and space bounds of  $O(rn^3(n/\epsilon)^{2r})$  and  $O(rn(n/\epsilon)^r)$  respectively, where  $n = |V|$  and  $r = B$ . The heavy line in  $G$  in Fig. 2.6 shows the MOSP solution and the two figures at the right show the corresponding arc costs and the arrangement of the selected signed-digit representations for parallel bit-level multiplication.

### 2.3.3 Extension to the Low-precision Weights

In the case of quantization using non-linear methods such as logarithm-based [35] and weighted-entropy-based [37], it is difficult to convert each weight into signed-digit representation. On the other hand, in the outlier quantization method [39], weights with small absolute values close to 0 are expressed in low-precision, and weights with large absolute values are expressed in full-precision. They have proved that 3.5% outlier weights (weights with large absolute values) and the remaining low-precision weights yield less than a 1% loss in top-5 accuracy in image classification. This outlier quantization method allows conversion to signed-digit representation because the weights have a partially linear spacing.

Since our bit-level weight pruning method can be applied to weights with various bit-size, we can propose a hybrid approach that performs bit-level weight pruning for both weights having small absolute values expressed in low-precision and outlier

---

<sup>3</sup>One exception is that our full hardware utilization to be explained later in Sec. 2.4.4 allows to set  $c_{j_1, j_2}^{i+1}[0] = c_{j_1, j_2}^{i+1}[B - 1] = 1/2$  if  $0^{th}$  bit value of  $j_2^{th}$  signed-digit representation for  $W_{i+1}$  is 1 or  $\bar{1}$ .

weights of large absolute values expressed in full-precision.

## 2.4 Supporting Hardware Architecture

The baseline of our signed-digit representation supporting architecture follows that in [42], but ours has a number of novel features that are unique to handling signed-digit representations.

### 2.4.1 Technique for Using a Single Bit to Encode Ternary Value

Storing a ternary value (0, 1, and  $\bar{1}$ ) in memory normally requires 2 bits. Thus, for  $k'$  ( $< k$ ) weights compressed from a set of initial weights of pruning stride  $k$ , a memory space of  $2k' \cdot B$  bits shall be needed. Thus, we propose a technique to reduce from the space requirement of  $2k' \cdot B$  bits to  $(k' + 1) \cdot B$  bits, which is also much lower than  $k \cdot B$  bits for storing initial  $k$  weights. (Note that in processing parallel bit-level multiplication on weights produced by bit-level pruning,  $k' \cdot B \cdot \log_2 k$  additional bits are commonly and essentially required to index the right ones among the  $k$  input activations for bit-level multiplication.) We describe our technique using the example in Fig. 2.7. An initial weight alignment in signed-digit representation produced by Sec. 2.3.2 with pruning stride  $k = 6$  is shown in Fig. 2.7(a), which is then column-wise condensed as shown in Fig. 2.7(b). To achieve the memory constraint of  $(k' + 1) \cdot B$  bits, we introduce  $B$  flags ( $f^0, f^1, \dots, f^{B-1}$ ), and define *ternary ordering rule*:

**Definition 1. (Ternary ordering rule):** Let  $\mathcal{L}^i$  be the list of all bit values on the  $i^{\text{th}}$  column in the condensed weight alignment of signed-digit representation. Thus,  $|\mathcal{L}^i| = k'$ . We then sort the elements in  $\mathcal{L}^i$  such that  $\bar{1}$  has the highest priority, and 1 has a higher priority over 0. We call such ordering of ternary values in  $\mathcal{L}^i$  *ternary ordering rule*. Fig. 2.7(c) shows the column-wise ordered arrangement from Fig. 2.7(b). We denote  $S_{\bar{1}}$ ,  $S_1$ , and  $S_0$  to the sublists of all  $\bar{1}$ , all 1, and all 0 values in a list  $\mathcal{L}^i$  satisfying





$\mathcal{L}^i = S_{\bar{1}} || S_1 || S_0$  where  $||$  means concatenation.

**Example 1.** In Fig. 2.7(c),  $\mathcal{L}^2 = [1, 1, 0, 0]$ , thus  $S_{\bar{1}} = []$ ,  $S_1 = [1, 1]$ , and  $S_0 = [0, 0]$  while  $\mathcal{L}^0 = [\bar{1}, \bar{1}, 1, 0]$ , thus  $S_{\bar{1}} = [\bar{1}, \bar{1}]$ ,  $S_1 = [1]$ , and  $S_0 = [0]$ .

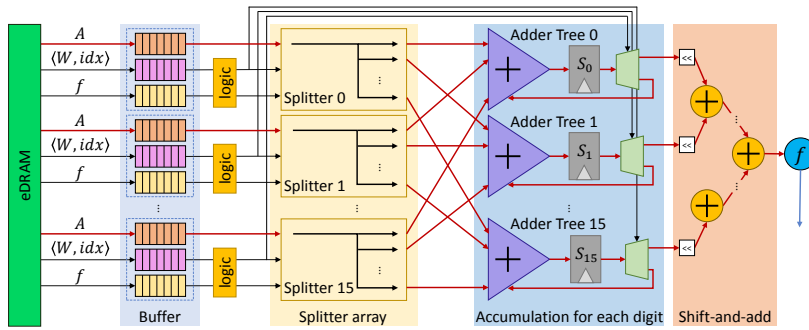
Then, the rules for setting flag  $f^i$  and  $k'$  memory bits in the  $i^{\text{th}}$  bit column are:

1. If  $S_1 = []$ , set  $f^i = 0$ , and all memory bits for  $S_{\bar{1}}$  to 1 and the rest to 0.
2. If  $S_1 \neq []$ , set  $f^i = 1$ , and all memory bits for  $S_1$  to 1 and the rest to 0.
3. When the ordered memory bits transit from 1 to 0, reset  $f^i$  to 0.

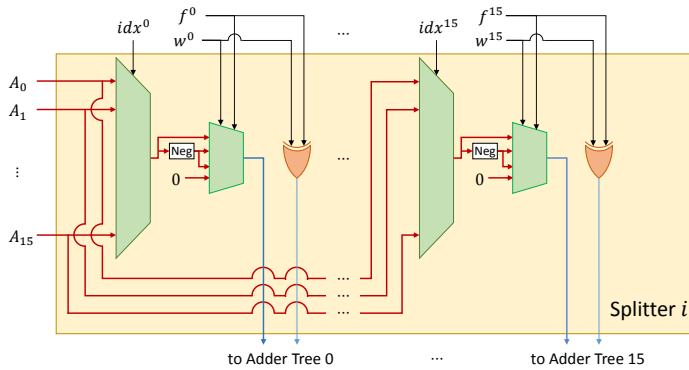
**Example 2.** Fig. 2.7(d) shows a set of cases in setting  $f^i$  and memory bits: For the  $0^{\text{th}}$  column, *Rule 2* is applied, which means memory bit 0 with flag 1 indicates the value is  $\bar{1}$ , thus subtraction (i.e.,  $A \times (-1) = -A$ ) will be performed. Then, during the subsequent accumulation process, when memory bit changes to 1, it means the value is 1, thus addition will be performed. Lastly, the flag is reset to 0 when memory bit changes to 0 according to *Rule 3*, as shown by the green dotted arrow in Fig. 2.7(d), which means no further operation will be performed. Likewise, *Rule 1*, *Rules 2 and 3*, and *Rule 1* are applied to the 1<sup>st</sup>, 2<sup>nd</sup>, and 3<sup>rd</sup> columns, respectively.

## 2.4.2 Structure of Supporting Architecture

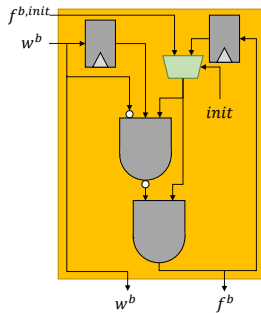
Fig. 2.8(a) shows the overall hardware structure. Initially, all values of flags  $f$ , activations  $A$ , pruned weights  $W$  aligned by ternary ordering rule, and indexes  $idx$  for activation selection are loaded from on-chip eDRAM to internal buffers, which are then transmitted to the corresponding splitters. For each splitter,  $\langle w^i, idx^i \rangle$  combinations ( $w^i$  and  $idx^i$  represent the  $i^{\text{th}}$  memory bit and activation selection index of  $W$ , as shown in Fig. 2.8(b)) are transmitted for every time step, and activations and flags are transmitted for every  $k'^{\text{th}}$  time step when corresponding initial set of weights are compressed into  $k'$  weights. It should be noted that before feeding to splitters, the circuit in the yellow boxes in Fig. 2.8(a) and Fig. 2.8(c) decodes the  $f_i$  value in conjunction with



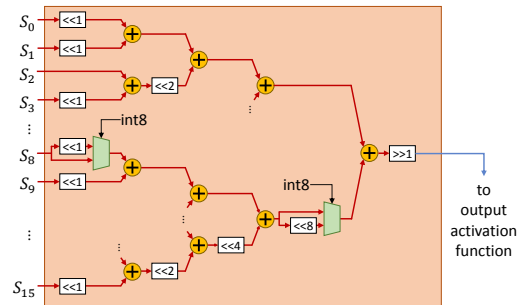
(a) Architecture processing 16 sets of condensed weights



(b) Distributing 16 activations to 16 adder trees



(c) Decoding logic



(d) Shift-and-accumulation for Eq.2.3

Figure 2.8: (a) Our proposed hardware architecture with 16 parallel lanes, each lane processing  $k'$  condensed weights, one weight at a time. 16 processing elements (PEs) are operating concurrently. (b) Microarchitecture of splitter for processing signed-digit representation. (c) Implementation for decoding a flag and memory bit. (d) Final adder tree for summing all bit-level partial-sums.

$i^{\text{th}}$  weight bit values to decipher if the encoding is addition ( $+A$ ), subtraction ( $-A$ ), or no ( $0 \cdot A$ ) accumulation. The decoded two signal values are then transmitted to the splitters to prepare one's complement representations  $\bar{A}$  for  $-A$  in the splitters (shown in Fig. 2.8(b)) and to complete the two's complement accumulation by adding extra 1 for  $-A$  (i.e.,  $-A = \bar{A} + 1$ ). The weights compressed through bit-level weight pruning as shown in Fig. 2.2(a) achieve performance improvement as it reduces the number of activations from the splitter array accumulated by the 16 adder trees. The final adder tree as shown in Fig. 2.8(d) shifts and sums all bit-level partial accumulations produced by the adder tree on each lane and delivers the result to the output activation function. Note that the architecture can speed up by  $2 \times$  if  $B = 8$ .

### 2.4.3 Memory Analysis

Like the prior architectures of DaDianNao [17] and Tetris [42], we have used eDRAM, which is more energy-efficient than off-chip DRAM and has higher storage density than SRAM. At 28nm technology node, 10MB SRAM occupies  $20.73\text{mm}^2$  [71] whereas eDRAM of the same size requires only  $7.27\text{mm}^2$  [72]. Also at the same technology node, while 256 bit read access to eDRAM consumes 0.0192nJ [73], read access to Micron DDR3 DRAM with the same bit wide requires 6.18nJ [74], which is 321 times more energy consumption.

For unpruned case, 512 bits must be loaded to the buffer every cycle in order to perform MAC operation for 16 input activations and 16 weights in 16-bit fixed-point representation. To perform the same amount of computation, Tetris [42] needs to load 16  $\langle W, idx \rangle$  combinations for every cycle and 256 input activations for every  $k'$  cycle. In other words, since  $idx$  has 4 bits when pruning stride  $k = 16$ ,  $1280 (= (1+4) \times 16 \times 16)$  bits for weight and  $\frac{4096}{k'}$  bits for input activation should be loaded for every cycle. In addition, our hardware has to load 256 bits of flags for every  $k'$  cycle. Therefore, Tetris [42] and our architecture must configure a wider internal bandwidth compared to the unpruned case, resulting in area overhead. We assume that  $k' = 4$

which is achievable minimum so that  $4.5\times$  wider internal bandwidth than the unpruned case is used.

#### 2.4.4 Full Utilization of Accumulation Adders

We have two observations regarding parallel bit-level multiplication:

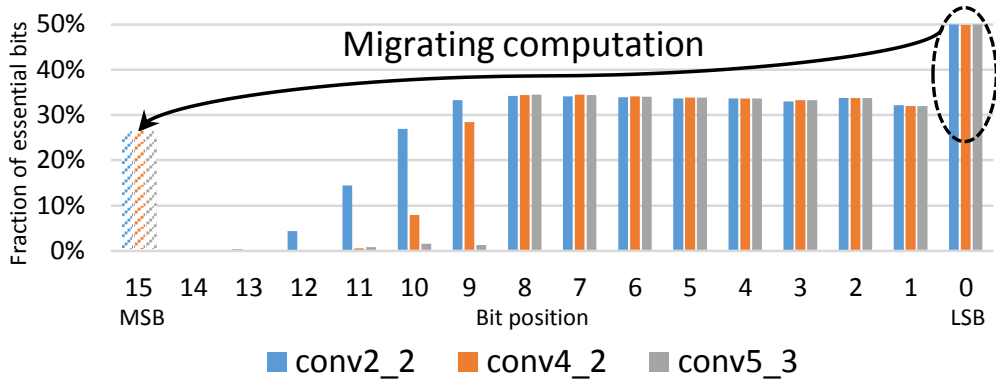
*Observation 1:* The sum of the sizes of  $S_{\bar{1}}$  and  $S_1$  in  $\mathcal{L}^0$  (i.e., the number of the ternary values of  $\bar{1}$  and 1 in the rightmost bit column in the signed-digit weight alignment) is the largest among all digits as shown in Fig. 2.9(a), which is a critical obstacle for boosting up maximal parallelism;

*Observation 2:* The elimination of the 1-bit in the leftmost bit in sign-magnitude weight representation (e.g., Fig. 2.5(b)) means to save one accumulation adder.

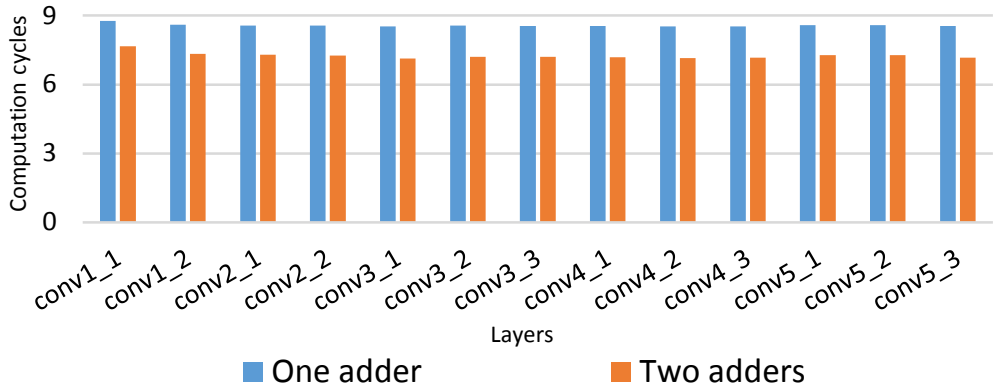
Thus, by equally partitioning the essential values in  $S_{\bar{1}}$  and  $S_1$  of  $\mathcal{L}^0$  into two parts and performing them concurrently by the two accumulation adder trees responsible for the  $0^{\text{th}}$  and  $(B-1)^{\text{th}}$  bit columns, the obstacle in *Observation 1* can be completely removed at no hardware cost as supported by *Observation 2*. As shown in Fig. 2.9(b), a considerable overall cycle reduction (16%) is possible. In the actual implementation as shown in Fig. 2.8(d), the  $i^{\text{th}}$  ( $i \geq 1$ ) bit column is processed in the  $(i+1)^{\text{th}}$  lane while the  $0^{\text{th}}$  bit column is processed in the  $0^{\text{th}}$  and  $1^{\text{st}}$  lanes.

#### 2.4.5 Modification for Hybrid Approach

In Sec. 2.3.3, a hybrid approach using low-precision weights has been described. It is possible to exploit this approach by slightly changing the final adder tree as shown in Fig. 2.10. Both 4-bit weights and 16-bit weights are supported by the modified final adder tree, and modes are identified through the control signal *int4*. Similar to what is mentioned in Sec. 2.4.2, this architecture can speed up by  $4\times$  when  $B = 4$ .



(a) Bit-wise comparison of the percentages of essential bits for a number of layers in VGG-16 with  $B = 16$  and  $k = 16$ .



(b) Reduction of total computation cycles when one more accumulation adder is allocated in (a) for processing all essential bits in the LSB position.

Figure 2.9: Effectiveness of using two accumulation adders for the essential bits in the least significant bit (LSB) position.

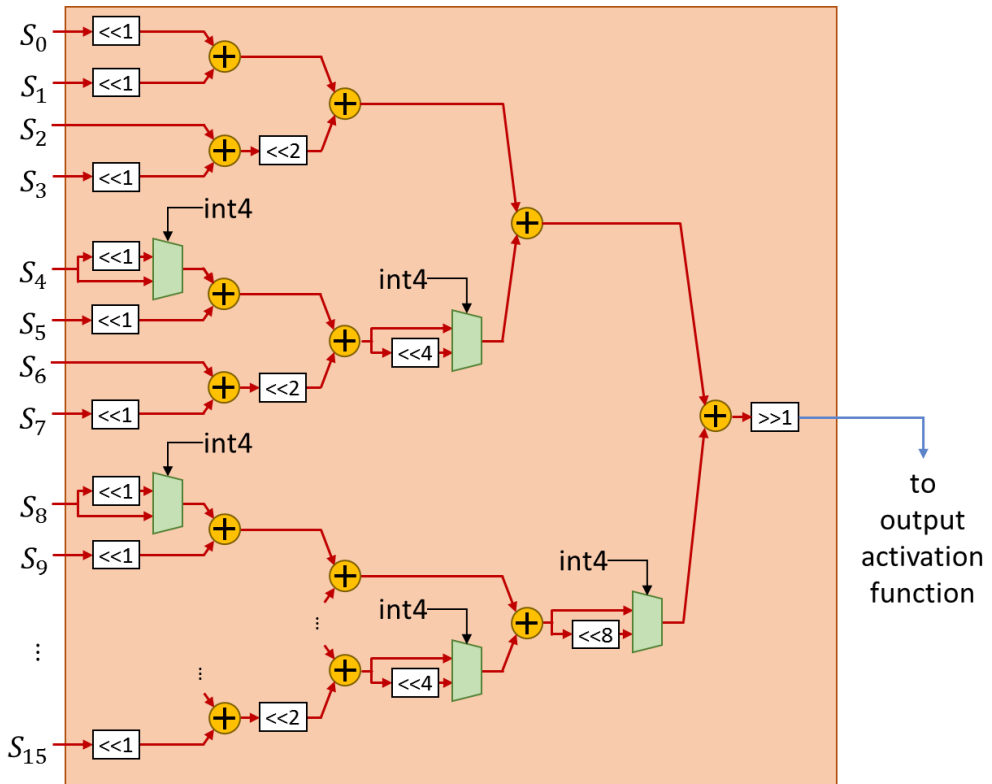


Figure 2.10: Modified final adder tree to support hybrid approach.

## 2.5 Bit-level Intra-weight Pruning

In Sec. 2.1.2 we have introduced DWP-intra, which is a variant of DWP for bit-level weight pruning. The usefulness of DWP-intra is the tight bound (i.e.,  $n/3 + 1/9 + O(2^{-n})$  in Sec. 2.1.3) on the worst latency of bit-level parallelism. This section is composed of converting weights into signed-digit representations, encoding logic, and supporting hardware architecture.

### 2.5.1 Signed-digit Representation Conversion

In order to get a small value of pruning stride  $k'$  for a set of weights by compressing them vertically, as illustrated in Fig. 2.2(a), DWP generates multiple signed-digit representation candidates for each weight. However, for DWP-intra, since compressing weights are performed horizontally, as shown in Fig. 2.2(b), the smallest value of the pruning stride  $k'$  is exactly the smallest number of essential bits among the (unique) CSD representations of the weights. As shown in Fig. 2.2(a) and Fig. 2.2(b), DWP-intra uses the transposed convention of DWP (e.g.,  $X^b$  and  $Y_b$ , and pruning stride  $k'$ ). We simply use Reitweiser's *right-to-left algorithm* to this end. Thus, the conversion ensures that the signed-digit representation of every weight has at most  $n/3 + 1/9 + O(2^{-n})$  number of essential bits according to the property 3 in Sec. 2.1.3.

### 2.5.2 Encoding Technique

We have described in Fig. 2.7 a technique of encoding ternary values for DWP. Since the only differences in processing DWP-intra with DWP are the compression direction and the step that performs shift operation, the encoding used in DWP needs to be slightly modified to fit into DWP-intra context as follows.

The activation selection index ( $idx$  in Fig. 2.8) that indicates the correct input activation corresponding to each essential bit on CSD representations in DWP will be replaced with the shifting size of the shift operation corresponding to that essential bit

for performing the shift-and-add operation in *Eq.2.4* in DWP-intra. (Bits of the same color in Fig. 2.2(a) have the same activation selection index, and bits of the same color in Fig. 2.2(b) have the same shifting size.) Since each essential bit has its own number of shifting sizes, the essential bits in the CSD representation of every weight will be rearranged according to a rule which is conceptually exactly the same as the ternary ordering rule applied in DWP employing a flag bit.

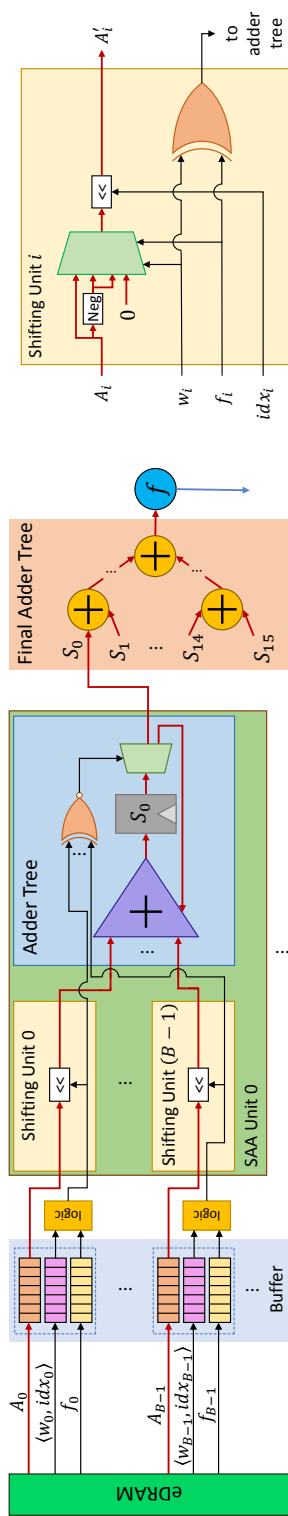
### 2.5.3 Supporting Hardware Architecture

The hardware components for DWP-intra and their logic are quite different from that for DWP in Sec. 2.4. The key differences are the following.

The shift-and-add (SAA) unit in Fig. 2.11(a) consists of  $B$  shifting units and one adder tree. Fig. 2.11(b) shows the details of the shifting unit that performs a shift operation of *Eq.2.6* suitable for signed-digit representation. For the  $i^{\text{th}}$  shifting unit, the input activation  $A_i$  shall be negated or zeroed according to  $w_i$  and a flag  $f_i$ , and is shifted to the left by  $idx_i$ , producing  $A'_i$ . The shifted input activations  $A'_i$  received from the  $B$  shifting units are summed up through the adder tree, and the outcome is transmitted to the final adder tree.

Once activations  $A$  and a set of encoded pruned weights  $W$  are loaded from an on-chip eDRAM into an internal buffer, they are transferred to the SAA units via decoding logics. Since DWP-intra requires the same amount of data as DWP for every cycle, it is configured with the same internal bandwidth as that of DWP. The implementation of splitter array and adder trees in Fig. 2.8(a) are replaced by 16 SAA units. The accumulations from the 16 SAA units are summed through the final adder tree and activation function and the output is stored in eDRAM again. The hardware architecture consists of 16 processing elements (PEs), and each PE is composed of 16 SAA units, a final adder tree, and an activation function.





(a) Architecture processing 16 sets of condensed weights including shift-and-add (SAA) units

(b) Shifting unit

Figure 2.1: (a) An architecture including conceptual structure of shift-and-add (SAA) units. Each SAA unit accepts  $B$  pairs of input activation and condensed weight and has  $B$  shifting units and an adder tree. (b) A structure of shifting unit. The decoded signals from the decoding logic which is similar to Fig. 2.7(c) determine whether the input activation is to be used as it is, or converted to a negative number, or left as 0. Then, the activation is shifted to the left by  $idx_i$ .

## 2.6 Experimental Results

For our experiment, we use several CNN models (AlexNet [1], VGG-16 [2], and ResNet-152 [4]) pre-trained with ImageNet dataset. For area and power evaluation, we compile our design using Synopsys Design Compiler with Nangate 45nm open cell library. The following two architectures are used as baselines. The first one is a state-of-the-art implementation Tetris [42] exploiting *weight kneading* which is a kind of column-wise bit-level weight pruning. The second one is PRA [47], which is originally designed for bit-level input activation pruning. In our experiment, we modify it to apply the proposed pruning technique in weights for a fair comparison.

Experiments are performed to assess (1) “how much our signed-digit representation is able to reduce the number of essential digits”, (2) “how much our accelerator is effective in performance and memory usage”, (3) “how much inference speed has been improved compared to the existing column-wise bit-level weight pruning technique”, and (4) “how much the area and power efficiency is improved compared to the existing neural network accelerator architectures”.

### 2.6.1 Essential Bits

Table 2.1 shows the comparison of the total number of essential bits used by AlexNet [1], VGG-16 [2], and ResNet-152 [4] when expressing every weight with two’s complement representation, sign-magnitude representation, and our signed-digit representations. We set the pruning stride  $k$  to 8 for a fair comparison. In short, our signed-digit representation conversion technique is able to represent the weights by using 52-74% fewer number of essential bits over that of the two’s complement representation. Since DWP-intra reduces the number of essential bits as many as possible via CSD representation, it uses the smaller number of essential bits than DWP, but the difference is negligible.

Table 2.1: The normalized number of essential bits for two’s complement, sign-magnitude, and our signed-digit representations.

Models	$B$	Methods		
		Two’s complement	Sign-magnitude	Our signed-digit for DWP (or DWP-intra)
AlexNet [1]	8	1.000	0.467	0.306
	16	1.000	0.685	0.477
VGG-16 [2]	8	1.000	0.433	0.260
	16	1.000	0.641	0.447
ResNet-152 [4]	8	1.000	0.480	0.315
	16	1.000	0.688	0.480

## 2.6.2 Memory Usage

Fig. 2.12 compares the memory sizes used by Tetris [42] and our DWP. Since DWP uses a much fewer number of essential bits, it directly affects the memory size for weights. Overall, our architecture uses up to 52% less memory space than that of the conventional architecture supporting parallel bit-level multiplication.

Fig. 2.13 compares the memory sizes occupied by the weights without compression and the weights encoded in our methods DWP and DWP-intra. As explained in Sec. 2.1.1, bit-level weight pruning accompanies the activation selection index (orange part in Fig. 2.13). Thus, both DWP and DWP-intra inevitably use more memory space than the case without pruning. Meanwhile, (i) in DWP, during the process of selecting a signed-digit representation combination for column-wise condensation described in Sec. 2.3.2, it could happen that a signed-digit representation of the minimum essential digit is not selected. For this reason, it uses a larger number of essential bits than DWP-intra, which always selects the signed-digit representation with the minimum essential bits. In addition, (ii) DWP often needs a large number of dummy 0-bits to make the vertical heights of all the bit columns of the compressed weights the same due to its uneven distribution of essential bits as illustrated in Fig. 2.9(a). By (i) and (ii), DWP requires up to 83% more memory space than DWP-intra.

## 2.6.3 Performance

Fig. 2.14 compares the performance of Tetris [42] and ours (DWP and DWP-intra) in terms of clock cycles required to perform a set of weights with bit-size of 8 and 16, and pruning stride of 8 and 16. The clock frequencies of the circuits in comparison are the same. Note that in experiment results,  $k$  and  $B$  used in DWP are  $B$  and  $k$  in DWP-intra, respectively.

We observe a considerable reduction of clock cycles, saving up to 64% cycles for DWP over the parallel bit-level multiplication without applying any compression technique, and 26-45% cycles over Tetris [42]. In addition, up to 61% fewer cycles

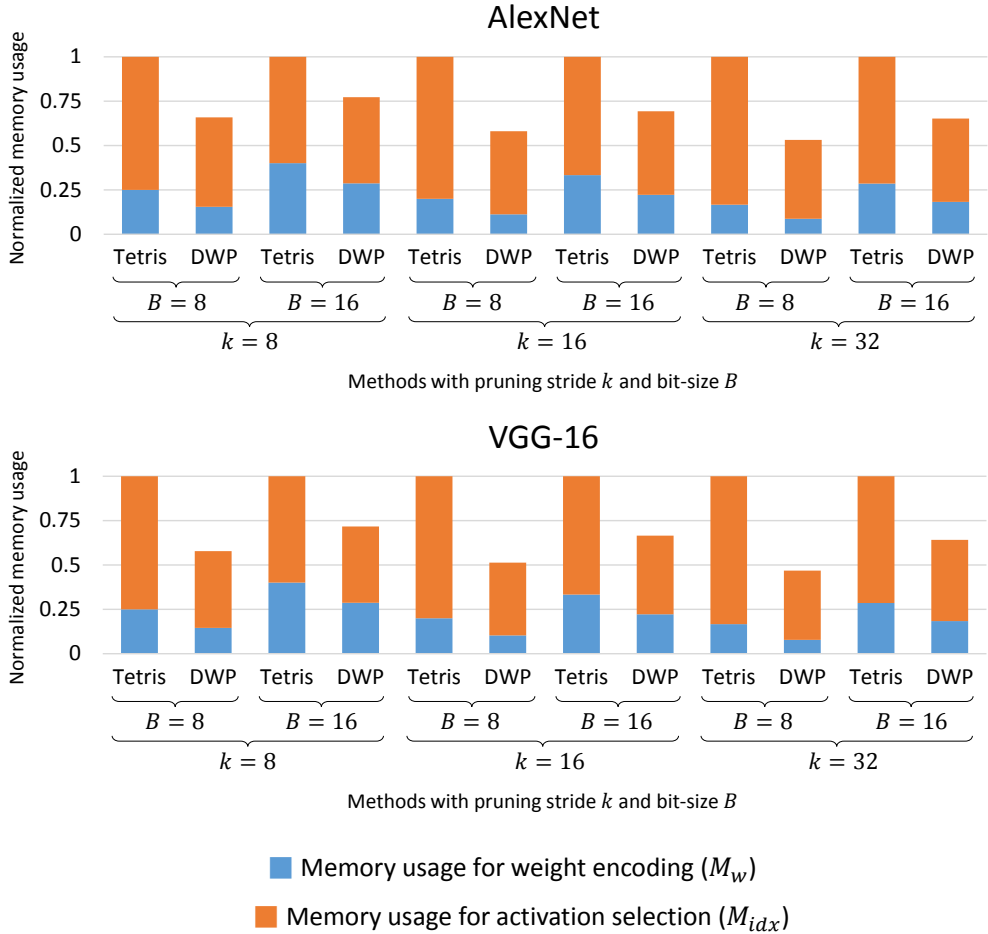


Figure 2.12: Comparison of memory usage by Tetris [42] and our DWP when bit-size  $B = 8$  and 16, and pruning stride  $k = 8, 16$  and 32.  $M_w$  includes the bits for weight encoding (Tetris and DWP) and flag (DWP) and  $M_{idx}$  includes the bits for activation selection in the splitters (Tetris and DWP).

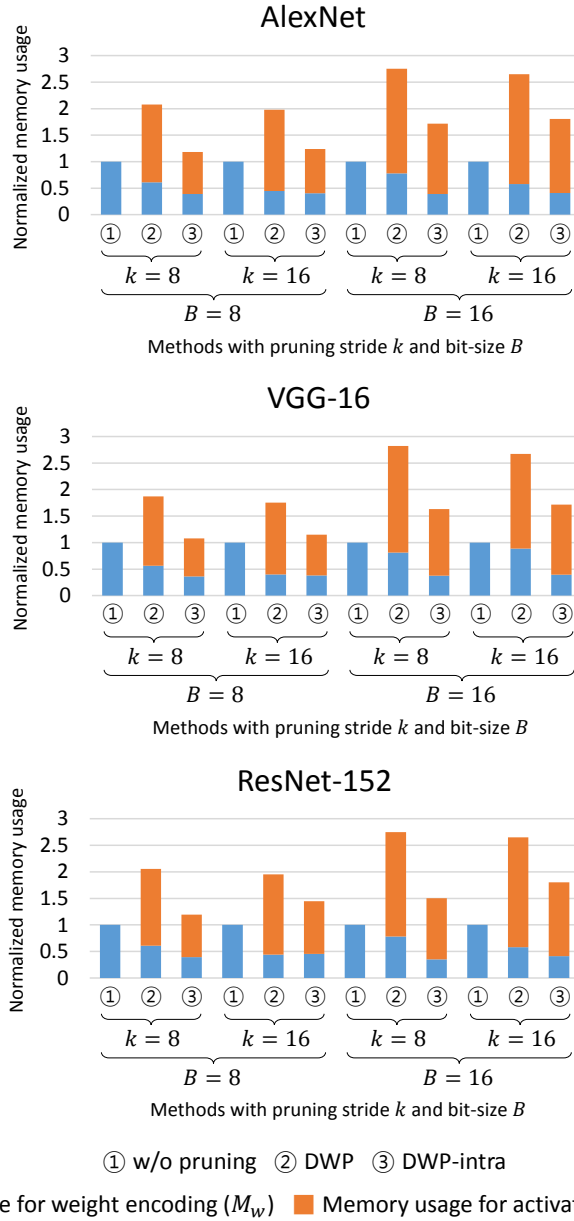


Figure 2.13: Comparison of memory usage by uncompressed case, DWP, and DWP-intra for several models when bit-size  $B = 8$  and 16, and pruning stride  $k = 8$  and 16. For a fair comparison, DWP with  $(B, k) = (16, 8)$  and  $(8, 16)$  has compared with DWP-intra with  $(B, k) = (8, 16)$  and  $(16, 8)$ , respectively.

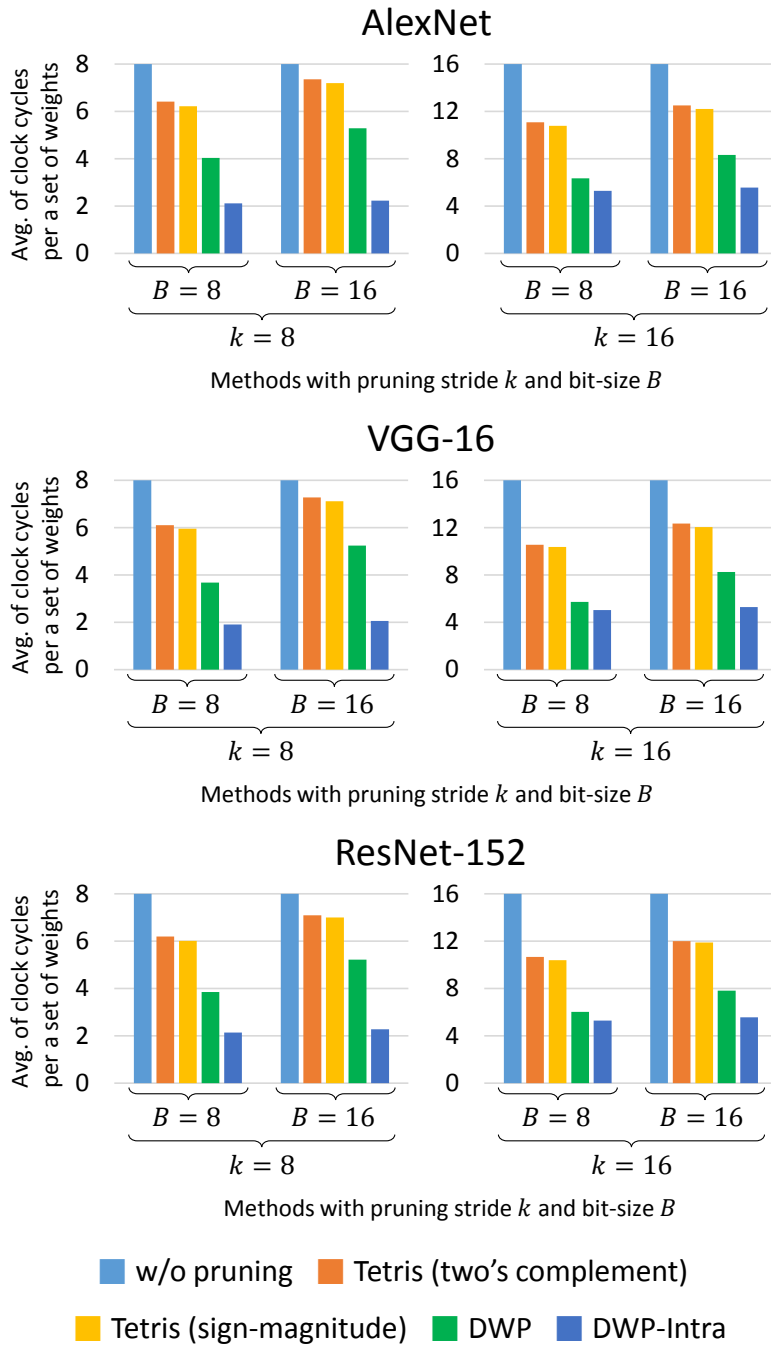


Figure 2.14: Comparison of the averaged number of clock cycles in performing a set of weights with  $B = 8$  and 16, and pruning stride  $k = 8$  and 16.

is used in DWP-intra compared with that in DWP. When DWP-intra is compared with the uncompressed case and Tetris [42], a reduction of 65-76% and 50-72% in the number of clock cycles are achieved, respectively.

Fig. 2.15 shows how much it affects the performance as the sparsity of the model (i.e., the ratio of zero weights) increases. For Tetris [42], the number of essential bits decreases as sparsity increases, and the average of clock cycles required to process a set of weight also decreases. However, for the case of ours using signed-digit representation, the performance improvement is less than that of the prior study. By increasing the sparsity of the model, the weights with the small absolute value are mostly eliminated. As shown in Fig. 2.9(a), since a part of the computation of the 0<sup>th</sup> bit column is processed after migration, the performance improvement is limited until the sparsity increases so that the weights of sufficiently large absolute value should be removed.

When applying the hybrid approach described in Sec. 2.3.3, Fig. 2.16 shows the result of comparing the average execution cycle after applying our proposed bit-level weight pruning for low precision weights. While the bit-size  $B$  decreases from 7 to 5, performance improvement is limited due to the reduced number of selectable signed-digit representation candidates. However, for  $B = 4$ , as the quantization interval increases, zero-valued weights after quantization increases significantly resulting in a significant performance improvement.

## 2.6.4 Area

Table 2.2 compares the areas of ours with other baselines, and Tables 2.3 and 2.4 show the area breakdown for DWP and DWP-intra. For fair comparison, both bit size  $B$  and pruning stride  $k$  are fixed to 16 and the size of I/O RAMs and buffers are set to be the same for all accelerator designs.

For the area breakdowns of both DWP and DWP-intra, I/O RAMs and buffers for storing activation and weight information occupy most of the area. Assessing only the parts that process MAC operations, in DWP the splitter array that distributes activa-



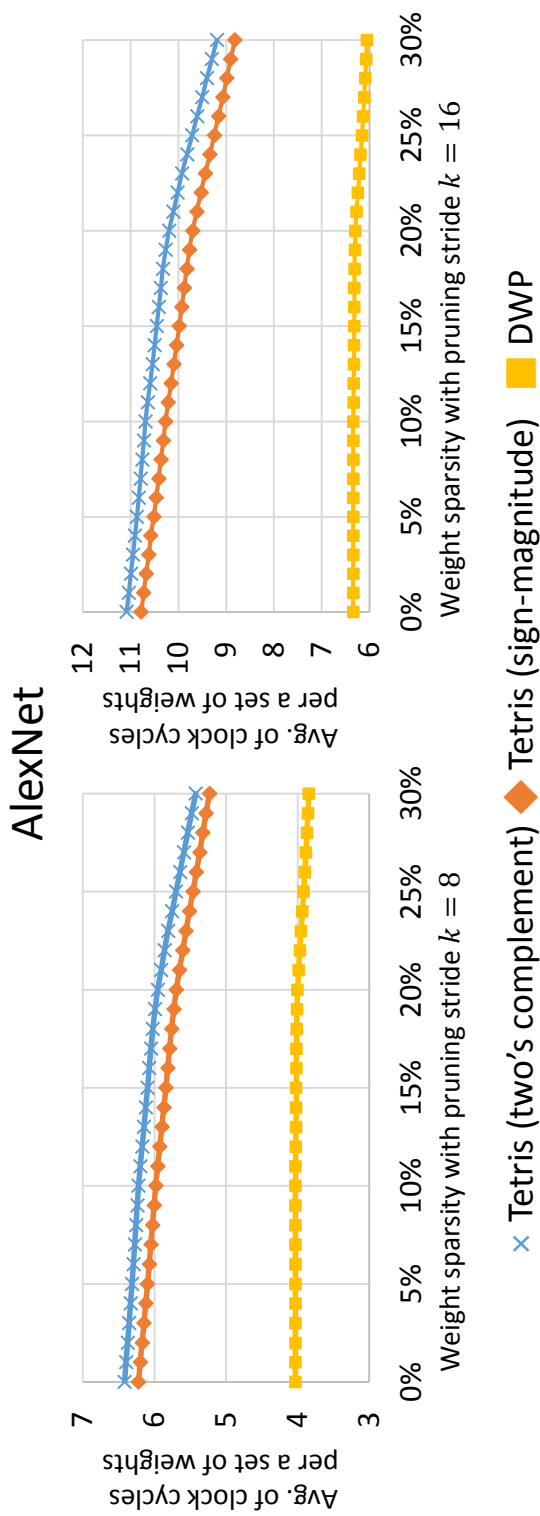


Figure 2.15: Comparison of the averaged number of clock cycles for various model sparsity in performing a set of weights with  $B = 8$  and pruning stride  $k = 8$  and  $16$ .

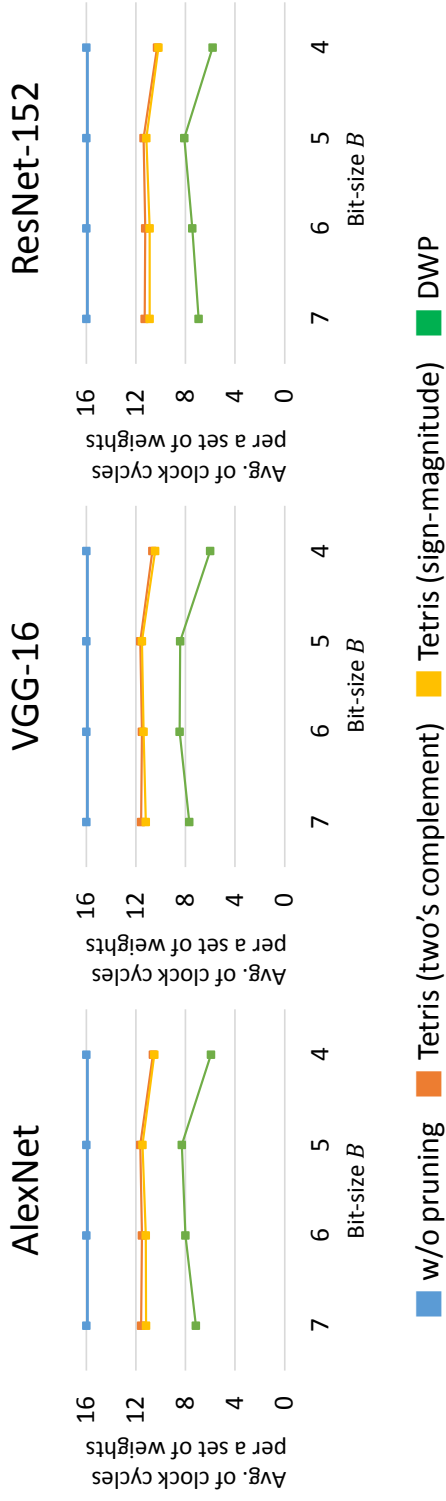


Figure 2.16: Comparison of the averaged number of clock cycles for uncompressed case, Tetris [42], and DWP in performing a set of weights with low precision and pruning stride  $k = 16$ .

Table 2.2: Area comparison for  $B = 16$  and  $k = 16$

Accelerators	Tetris [42]	PRA [47]	DWP	DWP-intra
Area ( $mm^2$ )	55.14	72.03	59.19	55.87

Table 2.3: Area breakdown of DWP for  $B = 16$  and  $k = 16$  ( $mm^2$ )

Item	I/O RAMs	Buffer	Decoding logic	Splitter array	Adder tree for each digit	Final adder tree	Activation function
Size	20KB/IPE	5KB	$16 \times 16$	$16 \times 16$	$16 \times 16$	$1 \times 16$	$1 \times 16$
Area	1.630	0.407	0.050	1.324	0.258	0.030	0.0005
Percentage	47.29%	11.82%	1.45%	38.41%	7.48%	0.88%	0.02%

Table 2.4: Area breakdown of DWP-intra for  $B = 16$  and  $k = 16$  ( $mm^2$ )

Item	I/O RAMs	Buffer	Decoding logic	Shift-and-add unit	Final adder tree	Activation function
Size	20KB/1PE	5KB	$16 \times 16$	$16 \times 16$	$1 \times 16$	$1 \times 16$
Area	1.630	0.407	0.050	1.368	0.037	0.0005
Percentage	46.67%	11.67%	1.44%	39.17%	1.05%	0.02%

tions to adder trees takes almost all areas, whereas in DWP-intra the area of the SAA unit that shifts and adds activations dominates other components.

As the complexity of the splitter increases and the decoding logic is added, DWP requires a slight more area ( $1.06\times$ ) than Tetris [42]. Since the process of extracting essential bits from the weights in two's complement representation in PRA [47] is performed by a hardware, area overhead is reached  $1.22\times$  over DWP. Although DWP-intra performs operations with wider bandwidth compared to DWP (e.g., DWP's final adder tree shown in Table 2.3 uses 16-bit input while DWP-intra shown in Table 2.4 uses wider 32-bit input), resulting in using excessive multiplexers in DWP, DWP-intra reduces the area by 5.6%. Compared to the state-of-the-art Tetris [42], DWP-intra has almost the same area.

### **2.6.5 Energy Efficiency**

Finally, Fig. 2.17 shows a comparison of EDP (energy-delay-product) for the computation by Tetris [42], PRA [47], and our DWP and DWP-intra. Although DWP has a complicated hardware structure over Tetris, a considerable inference speedup offsets the increase of energy consumption to a large extent, resulting in  $1.40\times$  EDP improvement on average. PRA has shown a higher EDP than Tetris due to its online weight encoding process in hardware architecture. DWP-intra has achieved almost same EDP as Tetris. As shown in Sec. 2.6.3, DWP-intra has a faster inference speed than DWP. However, unlike DWP, each activation in DWP-intra is shifted first and added later, resulting in an additional computational bandwidth and high power consumption. Thus, DWP-intra shows higher EDP numbers than DWP.

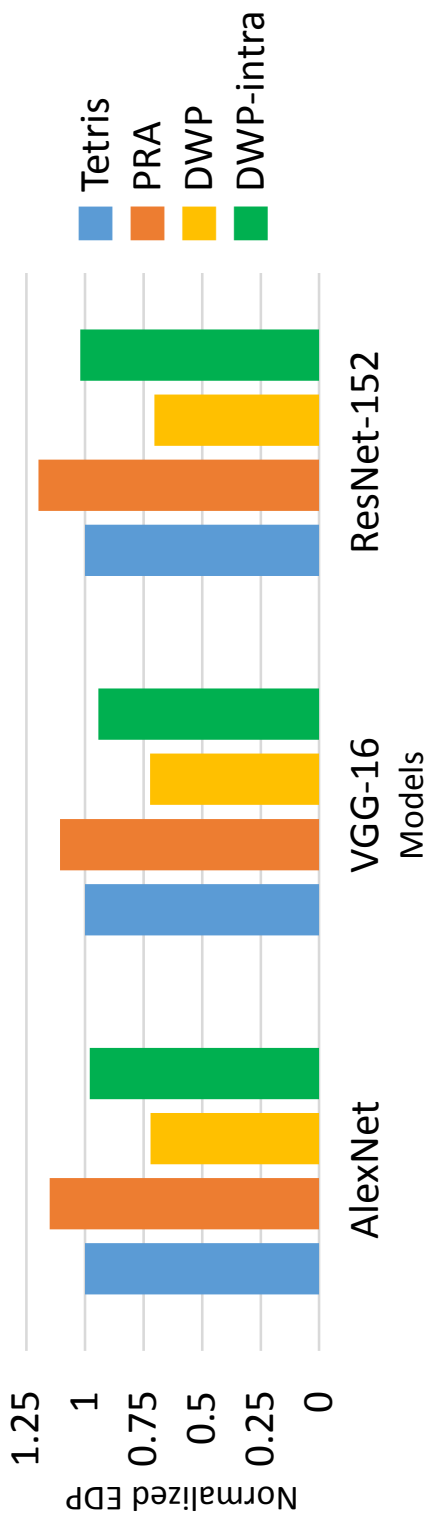


Figure 2.17: Comparison of the average of EDP with  $B = 16$  and  $k = 16$ . (The lower, the better.)





## Chapter 3

# Convolution Computation Techniques for High-Performance Neural Networks

### 3.1 Motivations

We start with a few definitions in the following:

**Definition 1.** (*partial kernel*): A kernel  $P$  is called a *partial kernel* of a kernel  $K$  if every non-zero value in  $P$  is embedded in the corresponding position in  $K$ . (For example, in Fig. 3.1(a), kernels  $K'_1$  and  $C_1$  are partial kernels of a kernel  $K_1$ .)

**Definition 2.** (*common kernel*): If a partial kernel is commonly included in all partial kernel sets of kernels, it is referred to as *common kernel*.

**Definition 3.** (*filtered kernel*): A kernel formed by filtering out the non-zero values of the partial kernel in the original kernel is called *filtered kernel*. Precisely, if  $P$  is a partial kernel of a kernel  $K$ , a kernel formed by  $(K - P)$  is classified as a filtered kernel.

#### 3.1.1 Limited Space Exploration for Common Kernels

Fig. 3.1(a) shows a step-by-step procedure of common kernel extraction performed by [64], in which from an original kernel set  $S = \{K_1, K_2, K_3\}$ , it selects, among all

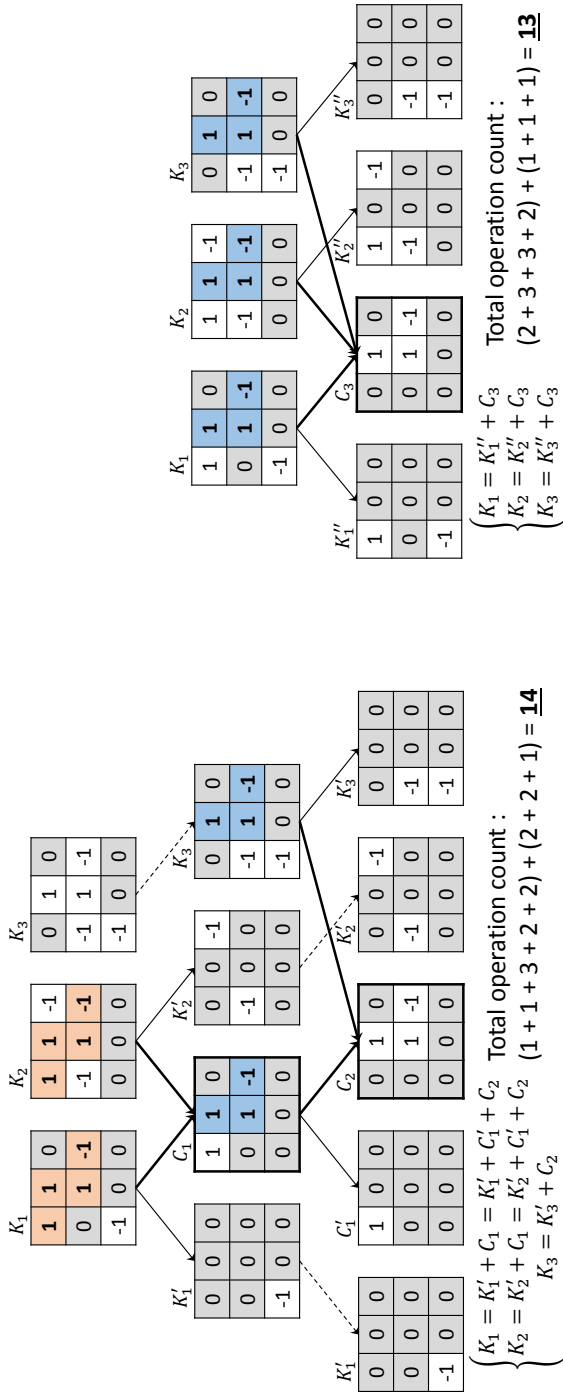


Figure 3.1: Examples showing the limitation of the common kernel extraction by [64]

common kernels of the pairs of kernels in  $\mathcal{S}$ , common kernel  $C_1$  of  $K_1$  and  $K_2$  since its use enables the maximal elimination of operation redundancy. As a result, the filtered kernels  $K'_1$  and  $K'_2$  respectively replace  $K_1$  and  $K_2$ , resulting in  $\mathcal{S} = \{K'_1, C_1, K'_2, K_3\}$  and the convolution evaluations with an input feature map  $A$  can be expressed as  $A \cdot K_1 = A \cdot K'_1 + A \cdot C_1$  and  $A \cdot K_2 = A \cdot K'_2 + A \cdot C_1$ . Then, at the next iteration, common kernel  $C_2$  is selected and filtered kernels  $C'_1$  and  $K'_3$  respectively replace  $C_1$  and  $K_3$ . Thus,  $\mathcal{S} = \{K'_1, C'_1, C_2, K'_2, K'_3\}$  and the convolution evaluations are expanded to  $A \cdot K_1 = A \cdot K'_1 + A \cdot C'_1 + A \cdot C_2$ ,  $A \cdot K_2 = A \cdot K'_2 + A \cdot C'_1 + A \cdot C_2$ , and  $A \cdot K_3 = A \cdot K'_3 + A \cdot C_2$ . The numbers of addition/subtraction operations for the convolutions on the kernels in  $\mathcal{S} = \{K'_1, C'_1, C_2, K'_2, K'_3\}$  are 1, 1, 3, 2, and 2, respectively, and once all convolutions on  $\mathcal{S}$  are evaluated,  $A \cdot K_1$ ,  $A \cdot K_2$ , and  $A \cdot K_3$  need 2, 2, and 1 more addition operations in the evaluation, respectively. Hence, the total number of operations is  $(1+1+3+2+2) + (2+2+1) = 14$ . On the other hand, Fig. 3.1(b) shows a common kernel extraction by exploring *all* kernels simultaneously. Consequently,  $C_3$  is identified as the common kernel that leads to the least count on the total number of operations. That is,  $A \cdot C_3$  takes 3 operations while  $A \cdot K_1 = A \cdot C_3 + A \cdot K''_1$ ,  $A \cdot K_2 = A \cdot C_3 + A \cdot K''_2$ , and  $A \cdot K_3 = A \cdot C_3 + A \cdot K''_3$  require  $(2+1) + (3+1) + (2+1) = 10$  additions/subtractions if the value of  $A \cdot C_3$  has been computed in advance. Thus, a total number of 13 operations are required for the convolutions, which is 7.1% fewer operations over that by [64].

### 3.1.2 Inability to Exploit Common Expressions of Convolution Values

The expanded expressions of kernel evaluations by using common kernels imply that an additional computation saving will be possible if there exists a common expression. For example,  $A \cdot K_1 = A \cdot K'_1 + A \cdot C'_1 + A \cdot C_2$  and  $A \cdot K_2 = A \cdot K'_2 + A \cdot C'_1 + A \cdot C_2$  in Fig. 3.2(a) contain  $A \cdot C'_1 + A \cdot C_2$  as a common expression. Consequently, by utilizing  $X = A \cdot C'_1 + A \cdot C_2$ , as shown in Fig. 3.2(b), 1 more operation can be saved.

$$A \cdot K_1 = A \cdot K'_1 + A \cdot C'_1 + A \cdot C_2$$

$$A \cdot K_2 = A \cdot K'_2 + A \cdot C'_1 + A \cdot C_2$$

$$A \cdot K_3 = A \cdot K'_3 + A \cdot C_2$$

Total operation count :

$$(1 + 1 + 3 + 2 + 2) + 2 + 2 + 1 = \underline{14}$$

(a) A total of 14 operations are required for convolution with input feature map  $A$  when the common and filters kernels in Fig. 3.1(a) are used.

$$X = A \cdot C'_1 + A \cdot C_2$$

$$A \cdot K_1 = A \cdot K'_1 + X$$

$$A \cdot K_2 = A \cdot K'_2 + X$$

$$A \cdot K_3 = A \cdot K'_3 + A \cdot C_2$$

Total operation count :

$$(1 + 1 + 3 + 2 + 2) + 1 + (1 + 1 + 1) = \underline{13}$$

(b) If common expression  $A \cdot C'_1 + A \cdot C_2$  is extracted and evaluated first, sharing the evaluated value results in 1 more operation saving.

Figure 3.2: Exploiting a common expression of convolution values

## 3.2 The Proposed Algorithm

Our algorithm ConvOpt, which minimizes the total number of operations required for the convolutions, performs two tasks: (Step 1) *extracting common kernels* in Sec. 3.2.1 and (Step 2) *extracting common convolutions* in Sec. 3.2.2. In addition, ConvOpt can tune in *minimizing the total number of resulting kernels* in Sec. 3.2.3, thereby reducing the total latency of memory accesses for kernels.

### 3.2.1 Common Kernel Extraction

Let  $\mathcal{S}_{init}$  be the original kernels to calculate their convolutions. Then, we set  $\mathcal{S} = \mathcal{S}_{init}$  and perform the following three steps:

**Step 1.1 (Generating all partial kernels):** We exhaustively collect all *partial kernels* with at least two non-zero weight values from all the kernels in  $\mathcal{S}$ .

*Time complexity:* For  $|\mathcal{S}|$  kernels of each kernel size  $l \times l$ , generating all partial kernels takes  $O(|\mathcal{S}| \cdot 2 \cdot 2^{l^2})$  time since the number of partial kernels in a kernel is bounded by  $O(2 \cdot 2^{l^2})$  with the consideration of opposite partial kernels and the time to insert a partial kernel to a hash table can be done in  $O(1)$ . We practically control run time by limiting the number non-zero values in the partial kernels (e.g.,  $\geq 3$ ).

For example, Fig. 3.3(a) shows the generation of all partial kernels of the kernels in  $\mathcal{S} = \{K_1, K_2, K_3\}$ . For example, for  $K_2$  with 6 non-zero values in Fig. 3.3(a), with the exception of the partial kernels with at most one non-zero value and the consideration of opposite kernels, there are  $2 \times (2^6 - 6 - 1) = 114$  partial kernels, which is much lower than the theoretical upper bound  $O(2 \cdot 2^{3^2})$ . Let  $H$  be the hash table that contains all distinct partial kernels in  $\mathcal{S}$ . Thus, theoretically  $|H| \leq \min\{|\mathcal{S}| \cdot 2 \cdot 2^{l^2}, 3^{l^2}\}$  and the actual values of  $|H|$  for CNN models are listed in Table 3.1.

Most of old CNN models like AlexNet [1] are composed of shallow layers with kernels of large size. However, CNN models of high performance such as VGGNet [2] and ResNet [4] are made up of deeper layers with kernels of small size, for which the

Table 3.1: Statistics of the number of partial kernels in ternary-weight CNN models.

CNN model	Top-5 error rate	Kernel size ( $l \times l$ )	#kernels (i.e., $ \mathcal{S} $ )	Theoretical bound of $ H $ ( $\min\{ \mathcal{S}  \cdot 2 \cdot 2^{l^2}, 3^{l^2}\}$ )	Actual value of $ H $
ALEXNET [1]	15.3%	$11 \times 11$ $5 \times 5$ $3 \times 3$	96 256 $\sim 384$	$5.1 \times 10^{38}$ $1.7 \times 10^{10}$ $2.0 \times 10^4$	$4.3 \times 10^{32}$ $2.1 \times 10^8$ $3.1 \times 10^3$
VGGNET [2]	7.3%	$3 \times 3$	$\sim 512$	$2.0 \times 10^4$	$2.1 \times 10^3$
RESNET [4]	3.6%	$7 \times 7$ $3 \times 3$	64 $\sim 512$	$7.2 \times 10^{16}$ $2.0 \times 10^4$	$2.5 \times 10^{13}$ $2.2 \times 10^3$

generation of all partial kernels can be done very quickly.

**Step 1.2 (Computing operation saving costs):** Let  $P_i$  be a partial kernel in  $H$ . For  $K_j$  in  $\mathcal{S}$ , we produce kernel  $K'_j$  that satisfies  $K_j = K'_j + P_i$  if  $P_i$  is a partial kernel of  $K_j$ , and  $K'_j = K_j$  otherwise. We call  $K'_j$  the *filtered kernel* of  $K_j$  with respect to  $P_i$ . Operation saving  $f(K_j, P_i)$ , which represents the number of addition and subtraction operations that can be saved for the convolution on  $K_j$  if  $P_i$  were utilized, can be expressed as:

$$f(K_j, P_i) = N_{op}(K_j) - N_{op}(K'_j) \quad (3.1)$$

where  $N_{op}(\cdot)$  is the number of non-zero weight values in the corresponding kernel. Then, we compute the total amount of operation saving  $\nabla Ops(P_i)$  for the convolutions on all kernels in  $\mathcal{S}$ :

$$\nabla Ops(P_i) = \sum_{j=1}^{|\mathcal{S}|} (f(K_j, P_i) - \alpha_j) - N_{op}(P_i), 1 \leq i \leq |H| \quad (3.2)$$

in which  $\alpha_j$  is 1 if  $f(K_j, P_i) \geq 1$  and 0, otherwise. For example, Fig. 3.3(b) shows the process of calculating the  $f(\cdot)$  and  $\nabla Ops(\cdot)$  values of some of partial kernels obtained in Fig. 3.3(a).

*Time complexity:* We construct a max-heap priority queue,  $Q_H$ , for the partial kernels in  $H$  according to the  $\nabla Ops(\cdot)$  values. Since computing  $\nabla Ops(\cdot)$  for each kernel in  $H$  can be done in  $O(|\mathcal{S}|)$  while  $|H| \leq |\mathcal{S}| \cdot 2 \cdot 2^{l^2}$ , and the construction of  $Q_H$  with  $|H|$  insertions can be done in  $O(|H|)$ , the total time is  $O(|\mathcal{S}|^2 \cdot 2 \cdot 2^{l^2})$ .

**Step 1.3 (Selecting common kernels):** We select the partial kernel in  $Q_H$  which has the largest value of  $\nabla Ops(\cdot)$ . If the  $\nabla Ops(\cdot)$  is a positive value, we remove its partial kernel from  $Q_H$  and update  $\mathcal{S}$  by replacing every kernel  $K_j$  in  $\mathcal{S}$  with its filtered kernel  $K'_j$  only if  $N_{op}(K'_j) \geq 1$ . Finally, we include the selected partial kernel, which we call now a *common kernel*, in  $\mathcal{S}$ . Note that in this updating process, we also maintain a linked-list set  $\mathcal{L}$  by including the information that the convolution output on  $K_j$  will be obtained later by using the convolution results on  $K'_j$  and  $P_i$ . Then, repeat Steps

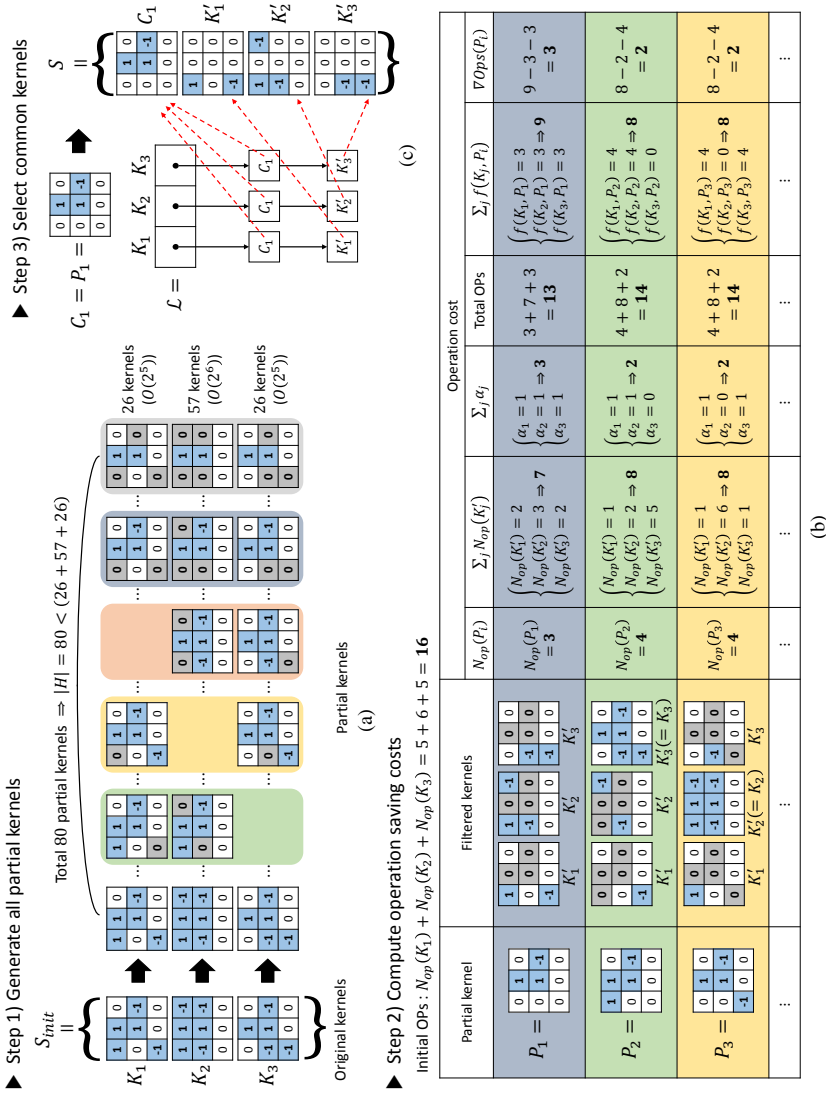


Figure 3.3: An example showing the common kernel extraction of ConvOpt. For simplicity, we do not consider opposite kernels in this example. (a) (Step 1.1) Generating all partial kernels. (b) (Step 1.2) Calculating the operation saving costs. (c) (Step 1.3) Iteratively selecting common kernels and update  $S$ .



1.1, 1.2, and 1.3 if the selected  $\nabla Ops(\cdot)$  is a positive value. Otherwise, stop and return  $\mathcal{L}$ .

*Time complexity:* The partial kernel selection and kernel replacement can be done in  $O(\log|Q_H| + |\mathcal{S}|) = O(|\mathcal{S}|)$ . Note that at each iteration, an incremental calculation of operation saving cost in Eq.3.1 and Eq.3.2, and incremental insertion to  $Q_H$  will be performed in practice.

For example, Fig. 3.3(c) shows the process of incrementally updating kernel set  $\mathcal{S}$  and its linked-list set  $\mathcal{L}$ , starting from the selected partial kernel  $P_1$  (common kernel  $C_1$ ) in Fig. 3.3(b).

### 3.2.2 Common Convolution Extraction

Once the linked-list set  $\mathcal{L}$  is obtained from Step 1, this step extracts common expressions from the addition and subtraction expressions of convolution results on common kernels for computing the convolutions on the original kernels. we employ a greedy method which iteratively performs the following two steps.

**Step 2.1 (Creating a counting graph):** We generate so-called a counting graph  $G(V, E, W)$  from  $\mathcal{L}$ , in which a node  $v_i \in V$  represents a distinct kernel in  $\mathcal{L}$ . There exists an edge  $(v_i, v_j) \in E$  if at least two kernels in  $\mathcal{S}_{init}$  contain the two kernels corresponding to  $v_i$  and  $v_j$  in their convolution evaluation expressions and  $w(v_i, v_j) \in W$  indicates the number of such original kernels. For example, Fig. 3.4(a) shows an example of generating graph  $G$  from a linked-list set  $\mathcal{L}$ .

*Time complexity:* This step can be done in  $O(|\mathcal{S}_{init}|)$  since the length of linked-list for each kernel in  $\mathcal{S}_{init}$  is limited to a small number ( $\leq 4$ ) for CNN models with small-size kernels.

**Step 2.2 (Selecting common expressions of convolution evaluations):** From the counting graph  $G(V, E, W)$ , we select an edge with the largest  $w(\cdot)$  value. Then, we select the two kernels on the edge as a common expression of convolution evaluations.

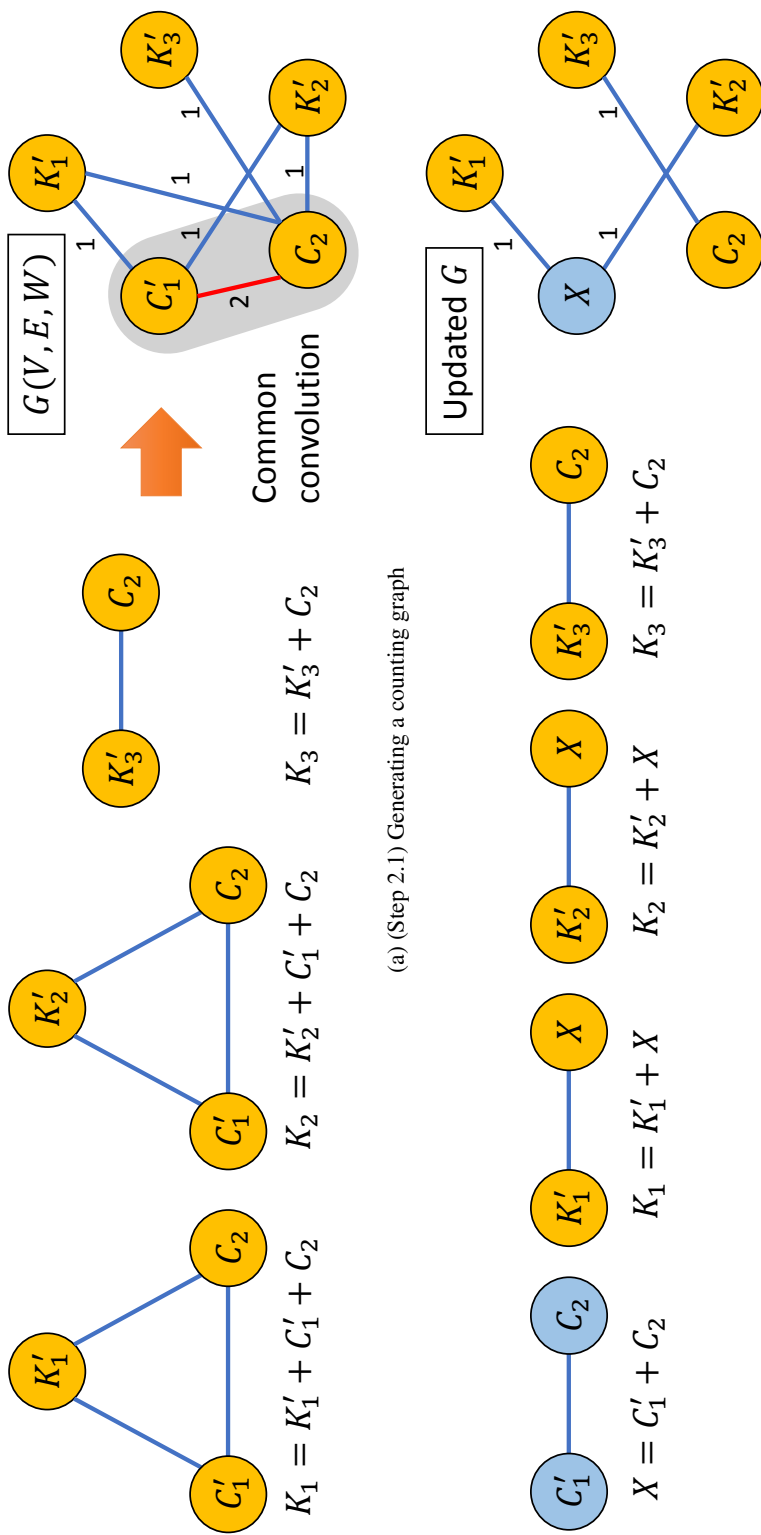


Figure 3.4: An example showing the common convolution extraction of ConvOpt.

We then update  $G$  by creating a new node which represents the two corresponding nodes and updating the weights of adjacent edges, and repeat this step until there is no edge in  $G$  with an edge weight greater than 2 or the space for storing evaluation results of all common expressions exceeds a memory limit. For example, Fig. 3.4(b) shows the process of selecting a common expression from the counting graph produced by Step 2.1.

*Time Complexity* : The iterative process of this step can be done in  $O(|E| \cdot \log |E|) = O(|\mathcal{S}|^2 \cdot \log |\mathcal{S}|)$ .

### 3.2.3 Memory Access Minimization

The objective of ConvOpt described in Sec. 3.2.1 (for common kernel extraction) and Sec. 3.2.2 (for common convolution extraction) is to minimize the number of addition and subtraction operations required for the convolutions in binary- or ternary-weight CNNs. Together with this, since the memory access would also contribute a significant delay if it were not performed in parallel with convolution computation, we need to analyze the impact of ConvOpt on the number of external memory accesses for kernels and find a way to take into account minimizing the total latency caused by the memory accesses.

1. The common convolution extraction in Step 2 of ConvOpt influences no or very little on the external memory accesses since the convolution evaluation will be done internally in the computing processors.
2. The common kernel extraction in Step 1 of ConvOpt certainly affects the memory accesses for kernels since using the common kernels extracted implies they should be stored in external memory. However, fortunately, as our experimental data shown in Sec. 3.4, the total number of kernels to be stored after the application of ConvOpt is usually lower than the number of original kernels. Considering the impact of memory accesses on performance, we want to diver-

sify the applicability of ConvOpt:

- **ConvOpt-op**: we place the primary importance on minimizing the number of operations, i.e., exactly performing Steps 1 and 2 in Sec. 3.2.1 and Sec. 3.2.2.
- **ConvOpt-mem**: we place the primary importance on minimizing the number of kernels, thereby indirectly minimizing external memory accesses for kernels. Thus, unlike ConvOpt-op, ConvOpt-mem uses the *kernel saving* (or *memory access saving*)  $\nabla N_{ker}(P_i, \mathcal{S})$  in Eq.3.3 as the primary cost in selecting a partial kernel in  $H$  in Step 1.3 while using  $\nabla Ops(\cdot)$  in Eq.3.2 as the secondary cost to break ties if exist.

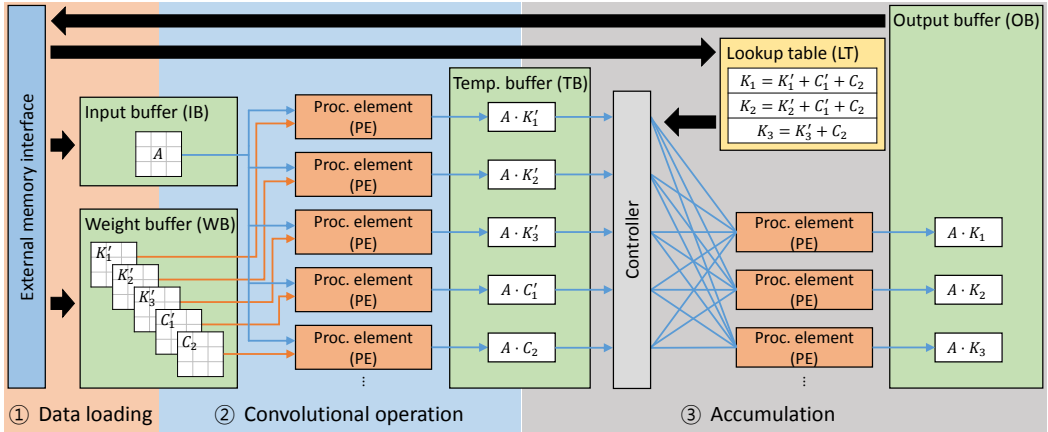
$$\nabla N_{ker}(P_i, \mathcal{S}) = |\mathcal{S}| - N_{ker}(P_i, \mathcal{S}), 1 \leq i \leq |H| \quad (3.3)$$

where  $N_{ker}(P_i, \mathcal{S})$  represents the number of distinct kernels in the kernel set updated from  $\mathcal{S}$  in the current iteration when  $P_i$  is selected in the current iteration as a common kernel.

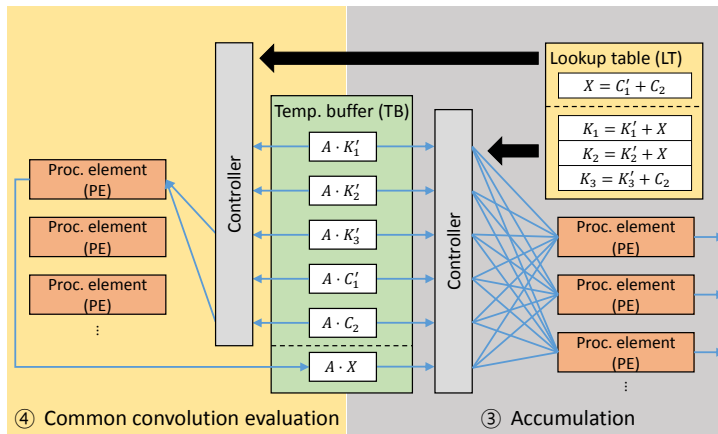
### 3.3 Hardware Implementation

The architecture utilizing common kernels obtained from Step 1 of ConvOpt is shown in Fig. 3.5(a) [64], and the architecture supporting common convolutions obtained from Step 2 of ConvOpt as well as common kernels is shown in Fig. 3.5(b).

For a convolutional layer, the architecture in Fig. 3.5(a) performs the sequence of three subtasks (i.e., ①  $\rightarrow$  ②  $\rightarrow$  ③): ① loading the input feature map, the kernels in the linked-list set  $\mathcal{L}$ , and the linked-list information in  $\mathcal{L}$  to produce the convolution results on the original kernels into input buffer block (IB), weight buffer block (WB), and lookup table (LT) addressed by the indices of the original kernels, respectively; ② broadcasting the contents in IB to processing elements (PEs), (In our experiments, the number of PEs for parallel processing is set to 16 or 32.) performing convolution on every kernel in WB with the data in PEs, and storing the convolution results to



(a) An architecture supporting Step 1 of ConvOpt [64], performing ① (data loading) → ② (convolution on common kernels) → ③ (accumulation of convolution values for original kernels).



(b) An architecture fully supporting ConvOpt, performing ① → ② → ④ (evaluation of common expressions of convolutions) → ③.

Figure 3.5: A hardware architecture supporting ConvOpt

temporary buffer block (TB); ③ bringing the intermediate convolution results in TB to PEs, accumulating the convolution values for each original kernel according to LT, and storing the final result to the output buffer block (OB).

To exploit the common convolution values obtained in Step 2 of ConvOpt, the supporting architecture needs to be updated, as shown in Fig. 3.5(b), which includes one more subtask in between ② and ③ (i.e., ① → ② → ④ → ③): ④ accumulating convolution values for *each common convolution* in LT, and storing the result to TB. In addition, to maximize the parallelism on PEs, we employed an off-line scheduler for kernel convolutions in ② and common convolution evaluations in ④ to balance loads on PEs.

## 3.4 Experimental Results

### 3.4.1 Experimental Setup

The CNN model tested in our experiment is the ternary-weight VGG-16 [2] trained with ImageNet dataset, on which the state-of-the-art work of common kernel extraction in [64] was tested as well. Since the binary-weight CNN is a constrained version of the ternary-weight one, we consider in the experiment the ternary-weight model only. Our ConvOpt algorithm written in Python-3 code runs on a server equipped with an Intel i7-8700k CPU running at 4.70GHz and a 32GB DDR4 RAM. We use our custom cycle-accurate simulator for performance evaluation on the hardware architecture.

### 3.4.2 Assessing Effectiveness of ConvOpt-op and ConvOpt-mem

Table 3.2 shows the total number of addition and subtraction operations produced by No-sharing (the convolution without exploitation of common kernels), Local-sharing (the existing work in [64] of common kernel sharing), and our ConvOpt-op and ConvOpt-mem, which not only extract common kernels globally but also exploit common expressions of convolutions, performing in ② to ④ in Fig. 3.5 for the

convolutional layers in VGG-16. In summary, **ConvOpt** uses 63% and 25-26% fewer operations over the convolution without using common kernels and the state-of-the-art work in [64] that exploits common kernels, respectively.

On the other hand, Table 3.3 summarizes the total number of kernels (which affects the data loading (i.e., memory access) time performed in ① in Fig. 3.5) used by **No-sharing**, **Local-sharing**, and our **ConvOpt-op** and **ConvOpt-mem** for performing convolution operations on VGG-16. Clearly, **Local-sharing**, **ConvOpt-op**, and **ConvOpt-mem** which take advantage of common kernel sharing use 86% fewer kernels. In addition, by exploiting global search space, both **ConvOpt** methods use 2.7-3.8% fewer kernels than that of **Local-sharing**. In comparison with **ConvOpt-op**, **ConvOpt-mem** which places a more emphasis (i.e., *Eq.3.3*) on reducing the number of kernels over that of **ConvOpt-op** reduces the number of kernels by 2.51K (1.1%) further.

Fig. 3.6 shows the breakdown of the operation counts in Table 3.2 for the convolutions of four layers in VGG-16. Since **No-sharing** does not extract common kernels, it produces the convolution results by performing ② only (blue bars in Fig. 3.6). **Local-sharing** performs ② for the convolution on the common kernels, which is much fewer than that of the original kernels, and produces the final convolution results using the intermediate ones. Thus, the operation count on ② is greatly reduced, but ③ (orange bars in Fig. 3.6) requires more operations. On the other side, **ConvOpt-op** and **ConvOpt-mem** use fewer operations over that of **Local-sharing** by fully utilizing the convolution results for the common kernels (in ②) and common convolutions (in ④, green bars in Fig. 3.6) on performing ③. In comparison with **ConvOpt-mem**, **ConvOpt-op** uses more kernels, consequently requires more operations in ② (blue bars) and ④ (green bars), but tends to require fewer operations in ③.

Table 3.2: Comparison of the *total number of operations* produced by No-sharing (the convolution without exploitation of common kernels), Local-sharing (the existing work in [64] of common kernel sharing), and ConvOpt-op & ConvOpt-mem performing in ② to ④ in Fig. 3.5 for the convolutional layers in VGG-16.

Layer	No-sharing	Local-sharing [64]	ConvOpt-op	ConvOpt-mem
CONV1_1	33.9M	20.8M	16.8M	16.8M
CONV1_2	656.7M	393.8M	315.3M	315.3M
CONV2_1	371.1M	197.0M	150.2M	150.2M
CONV2_2	729.0M	388.3M	304.9M	304.9M
CONV3_1	326.0M	157.4M	116.7M	117.2M
CONV3_2	675.4M	323.6M	233.8M	235.0M
CONV3_3	691.3M	328.1M	242.1M	243.5M
CONV4_1	282.6M	124.2M	85.8M	87.3M
CONV4_2	553.5M	243.6M	165.4M	169.2M
CONV4_3	572.8M	251.0M	171.6M	175.0M
CONV5_1	139.5M	62.0M	41.8M	42.6M
CONV5_2	105.6M	49.6M	32.2M	32.5M
CONV5_3	73.8M	36.6M	22.9M	22.9M
Total #operations	5.21G	2.58G	1.90G	1.91G
Reduction over No-sharing	-	50.6%	<b>63.5%</b>	<b>63.3%</b>
Reduction over Local-sharing	N/A	-	<b>26.3%</b>	<b>25.8%</b>



Table 3.3: Comparison of the *total number of kernels* used by No-sharing (the convolution without exploitation of common kernels), Local-sharing (the existing work in [64] of common kernel sharing), and ConvOpt-op & ConvOpt-mem performing convolutions in ② in Fig. 3.5 for the convolutional layers in VGG-16.

Layer	No-sharing	Local-sharing [64]	ConvOpt-op	ConvOpt-mem
CONV1_1	0.19K	0.09K	0.09K	0.09K
CONV1_2	4.10K	1.72K	1.70K	1.70K
CONV2_1	8.19K	2.77K	2.66K	2.66K
CONV2_2	16.38K	5.69K	5.46K	5.46K
CONV3_1	32.77K	7.38K	7.14K	7.08K
CONV3_2	65.54K	14.39K	14.07K	13.86K
CONV3_3	65.54K	14.73K	14.29K	14.13K
CONV4_1	131.07K	17.63K	16.79K	16.66K
CONV4_2	262.14K	33.96K	33.14K	32.26K
CONV4_3	262.14K	34.49K	33.35K	32.94K
CONV5_1	262.14K	34.57K	33.24K	32.99K
CONV5_2	262.14K	29.22K	29.07K	28.66K
CONV5_3	262.14K	23.54K	23.23K	23.23K
Total #kernels	1,634.50K	220.15K	214.23K	211.72K
Reduction over No-sharing	-	86.5%	<b>86.9%</b>	<b>87.0%</b>
Reduction over Local-sharing	N/A	-	<b>2.7%</b>	<b>3.8%</b>

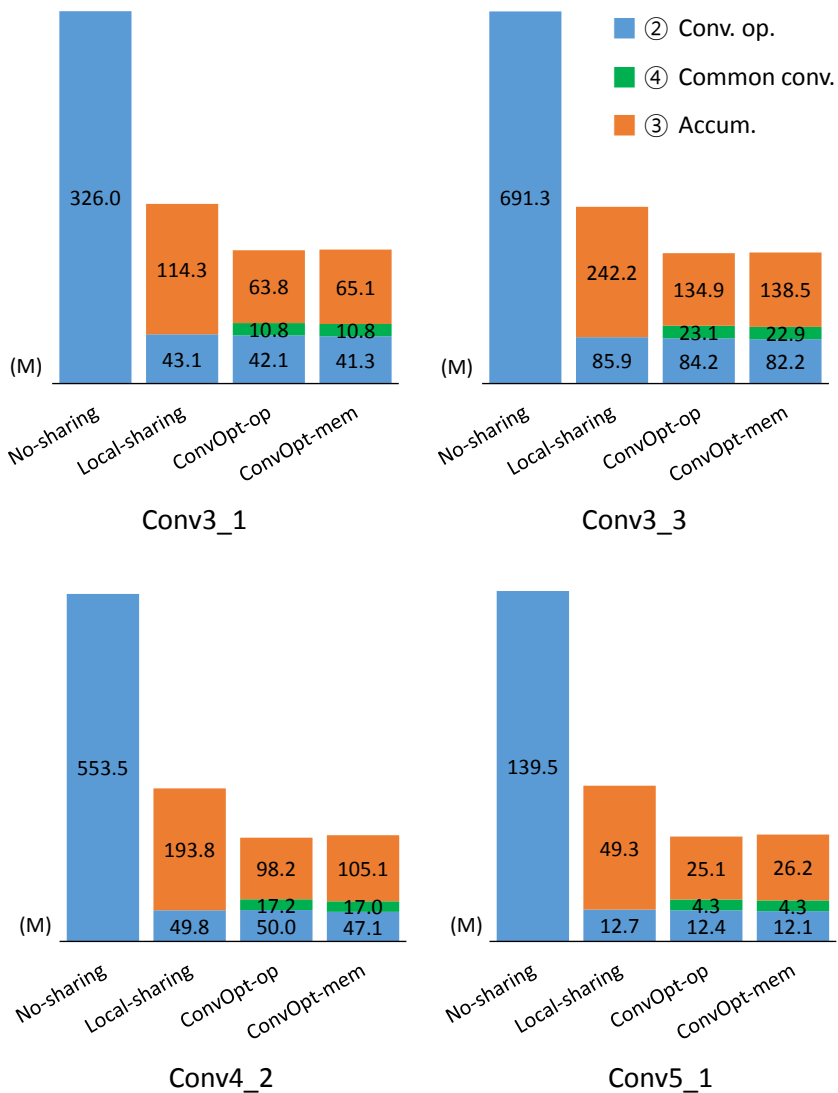


Figure 3.6: A breakdown of operation counts used by No-sharing, Local-sharing [64], ConvOpt-op, and ConvOpt-mem for the convolutional layers in VGG-16. Blue, green, and orange bars stand for the operation counts in convolution on common or original kernels (②), calculating common expressions of intermediate convolution results (④), and accumulation to produce final convolution outputs (③), respectively.

Table 3.4: Comparison of the *total number of execution cycles* on the hardware platform in Fig. 3.5 when using the kernels produced by Local-sharing [64], ConvOpt-op, and ConvOpt-mem for image inferencing with VGG-16.

#PEs	Method	Local-sharing [64]	ConvOpt-op	ConvOpt-mem
16 PEs	#Cycles ② + ③ + ④	368.2M	285.8M	286.4M
	Reduction over Local-sharing	-	<b>22.4%</b>	<b>22.2%</b>
32 PEs	#Cycles ② + ③ + ④	212.5M	183.3M	182.0M
	Reduction over Local-sharing	-	<b>13.7%</b>	<b>14.3%</b>

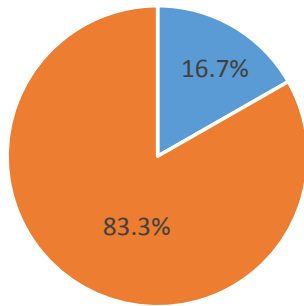
### 3.4.3 Measuring Performance through Hardware Implementation

Table 3.4 summarizes the total number of execution cycles on the hardware architecture in Fig. 3.5 with 16 and 32 processing elements when using the kernels produced by Local-sharing [64], ConvOpt-op, and ConvOpt-mem for image inferencing with VGG-16. We assume that retrieving the kernels from the external memory and loading them to the internal memory blocks IB, WB, and LT (i.e., ①) in Fig. 3.5 will be processed in parallel with the execution of ②, ③, and ④. Overall, ConvOpt-op uses up to 22.4% smaller number of clock cycles over Local-sharing [64].

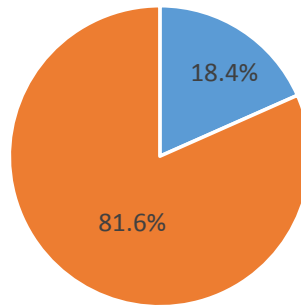
The breakdown of the execution cycles on the hardware platform in Fig. 3.5 is shown in Fig. 3.7. Due to the utilization of intermediate results produced by common convolution evaluation in ④ (green pies), the cycle portion taken by the accumulation in ③ (orange pies) is shrunk. Note that the operation count saving by ConvOpt over Local-sharing shown in Table 3.4 unbends as the number of PEs used increases from 16 to 32, i.e., 22%  $\rightarrow$  13-14%. This is because more parallelism implies shortening the total execution cycles, thus relatively lower percentage on the cycle count reduction. For example, convolution on 68 common kernels can be completed in five iterations on 16 PEs. (There are 28 to 68 common kernels on average for each input channel in VGG-16 layers.) However, on 32 PEs three iterations are enough where in the last iteration 28 among 32 PEs are in idle state.

### 3.4.4 Running Time of ConvOpt

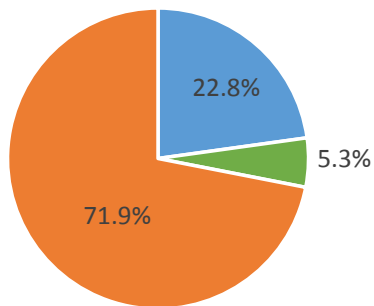
Table 3.5 shows the comparison of the running time (hour:min:sec) spent by Local-sharing [64] and ConvOpt-op for all convolutional layers in VGG-16. (We found no non-trivial difference in running time between ConvOpt-op and ConvOpt-mem.) Except for the layer of CONV1\_1, which size is very small, despite our ConvOpt-op method requires performing two steps of common kernel extraction and common convolution extraction, it uses 64.5-96.5% less running time than Local-sharing [64]. In short, ConvOpt-op achieves an overall speedup of 18.4 $\times$ .



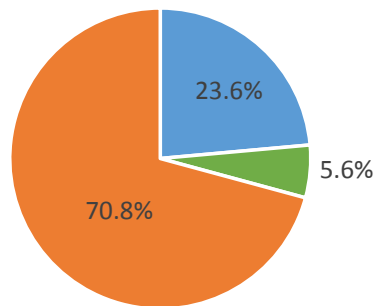
Local-sharing using 16 PEs



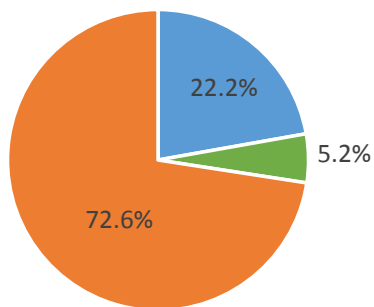
Local-sharing using 32 PEs



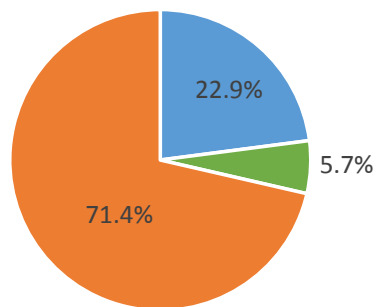
ConvOpt-op using 16 PEs



ConvOpt-op using 32 PEs



ConvOpt-mem using 16 PEs



ConvOpt-mem using 32 PEs

■ ② Conv. op. ■ ④ Common conv. ■ ③ Accum.

Figure 3.7: A breakdown of the execution cycles on the hardware platform in Fig. 3.5. Blue, green, and orange pies stand for the execution cycles in convolution on common kernels (②), calculating common expressions of intermediate convolution results (④), and accumulation to produce final convolution outputs (③), respectively.

Table 3.5: Comparison of *running time* (hour:min:sec) spent by Local-sharing [64] and ConvOpt-op for VGG-16.

Layer	CONV1_1	CONV1_2	CONV2_1	CONV2_2	CONV3_1	CONV3_2	CONV3_3
Local-sharing [64]	0:00:01	0:00:31	0:04:42	0:12:52	1:15:09	2:50:56	3:04:10
ConvOpt-op	0:00:02	0:00:11	0:01:03	0:02:00	0:06:11	0:10:55	0:11:56
Reduction	-100.0%	64.5%	77.7%	84.5%	91.8%	93.6%	93.5%
Speedup	0.5×	2.8×	4.5×	6.4×	12.2×	15.7×	15.4×
Layer	CONV4_1	CONV4_2	CONV4_3	CONV5_1	CONV5_2	CONV5_3	Total
Local-sharing [64]	13:00:26	24:43:47	14:24:45	12:58:16	5:35:36	1:55:25	80:06:36
ConvOpt-op	0:27:27	0:56:27	0:52:56	0:53:11	0:24:39	0:13:52	4:20:50
Reduction	96.5%	96.2%	93.9%	93.2%	92.7%	88.0%	94.6%
Speedup	28.4×	26.3×	16.3×	14.6×	13.6×	8.3×	<b>18.4×</b>

## Chapter 4

# Memory Layout and Block Replacement Techniques for High-Performance Neural Networks

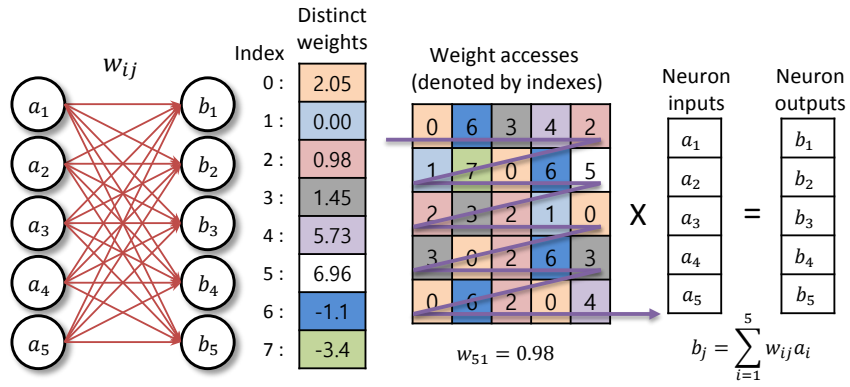
### 4.1 Motivation

Fig. 4.1 illustrates how a careful arrangement (i.e., memory layout) of weights in off-chip memory and a block replacement policy exploiting the complete index sequence influence the resulting total number of off-chip memory accesses. We assume that DNN compression with weight sharing produces a total number of 8 distinct weights, thus requiring off-chip memory of size 8. Further, we assume the size of on-chip memory is 4 and the block size for off-chip memory access is 2. Fig. 4.1(a) shows the values of 8 weights and the access sequence of the weights by index for performing MAC (multiply-and-accumulate) operations<sup>1</sup>, and Fig. 4.1(b) depicts a memory configuration for storing/accessing the weights produced by the unfitted DNN compression.

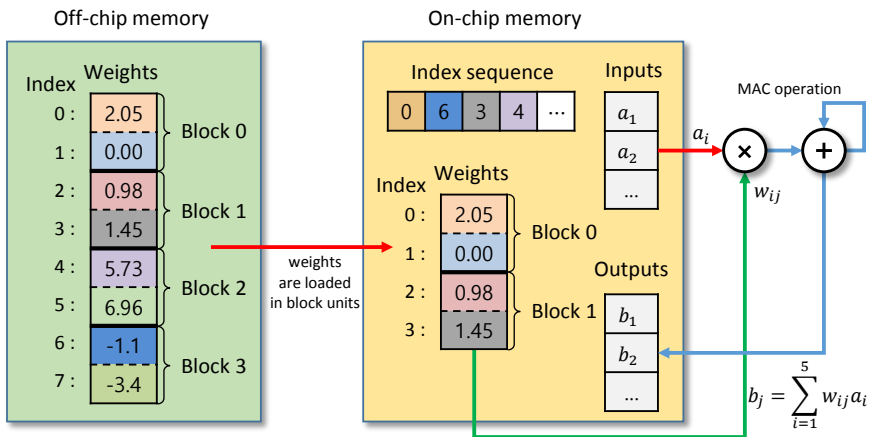
Fig. 4.2 describes, for the index sequence, shared weights, and memory configuration in Fig. 4.1, how frequently the off-chip memory accesses occur according to unoptimized memory layout and LRU (least recently used) block replacement (Fig. 4.2(a)),

---

<sup>1</sup>Note that the index sequence corresponds to the order of performing the MAC operations in DNN and highly depends on the underlying DNN computing architecture. In this work, the index sequence is given.



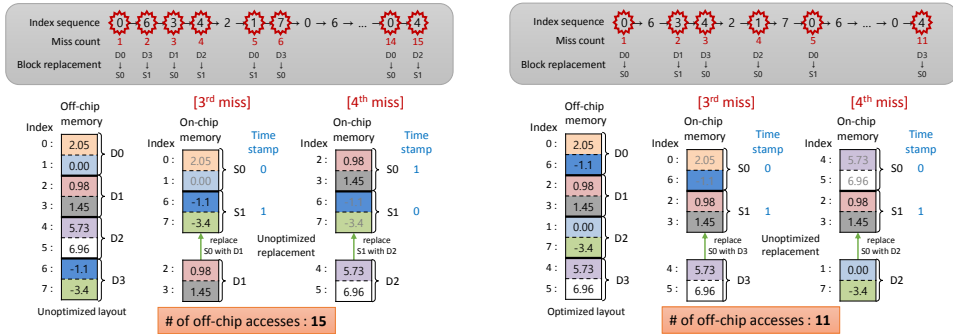
(a) Index sequence by weight sharing



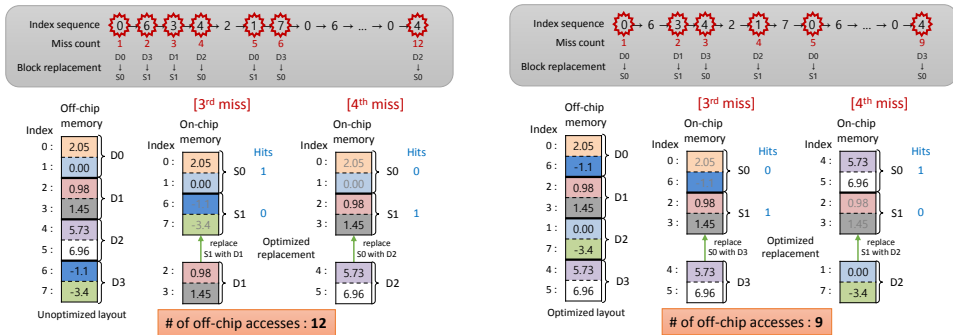
(b) Memory configuration for (a)

Figure 4.1: An example of DNN with unfitted compression to illustrate our motivation of reducing the number of off-chip memory accesses in Fig. 4.2. (a) MAC (multiply-and-accumulate) operations in a layer of DNN and index sequence for accessing the weights, assuming DNN compression with weight sharing results in a total number of 8 weights. (b) The configuration of off-chip (containing 8 weights) and on-chip memories (containing up to 4 weights). The block size for access is 2.





(a) Unoptimized layout + Unoptimized replacement (b) Optimized layout + Unoptimized replacement



(c) Unoptimized layout + Optimized replacement (d) Optimized layout + Optimized replacement

Figure 4.2: Four cases of handling memory layout and block replacement for the DNN in Fig. 4.1. (a) Off-chip memory accesses resulting from the use of unoptimized memory layout and LRU replacement scheme. (b) Off-chip memory accesses resulting from the use of our optimized memory layout and LRU replacement scheme. (c) Off-chip memory accesses resulting from the use of unoptimized memory layout and our optimized replacement scheme. (d) Off-chip memory accesses resulting from the use of our optimized memory layout and our optimized replacement scheme.

our optimized layout and LRU replacement (Fig. 4.2(b)), unoptimized layout and our optimized replacement (Fig. 4.2(c)), and our optimized layout and our optimized replacement (Fig. 4.2(d)). The comparison of the four cases clearly reveals that it is essential to devise techniques for memory layout and replacement scheme that are able to minimize the number of off-chip memory accesses for DNNs with unfitted compression.

## 4.2 Algorithms for Off-chip Memory Access Optimization for DNNs with Unfitted Compression

The inputs to our algorithms are:

- $\mathcal{S} = (i_1, i_2, \dots)$ : a complete index sequence.
- $\mathcal{W} = \{w_1, w_2, \dots\}$ : a set of distinct weights obtained from the weight sharing in DNN compression.
- $ws(i_k)$ : a function that maps index  $i_k \in \mathcal{S}$  to a weight in  $\mathcal{W}$ .  $ws(\cdot)$  is also given by the result of DNN compression.
- $|M_{off}|$ : size of off-chip memory in terms of weight count and  $|M_{off}| \geq |\mathcal{W}|$ .
- $|M_{on}|$ : size of on-chip memory in terms of weight count and  $|M_{on}| \leq |M_{off}|$ .
- $|B|$ : block size for transferring weights in single access in terms of weight count and  $|B| \leq |M_{on}|$ .

The proposed algorithms for solving the memory layout and block replacement problems are described in the following two subsections.

### 4.2.1 Algorithm for Off-chip Memory Layout

Our strategy to tackle the off-chip memory layout problem is that our algorithm allows being broadly applicable to *every* kind of block replacement with an equal impact. As

a result, we assume  $|M_{on}| = |B|$  to minimally link the selection of block replacement scheme to our memory layout solution. The memory layout problem for DNNs with unfitted compression then can be described as:

**Memory layout problem for DNNs with unfitted compression:** For  $\mathcal{S}$ ,  $\mathcal{W}$ ,  $M_{on}$  with  $|M_{on}| = |B|$ , and  $M_{off}$  partitioned by  $|B|$ , find a function  $\phi(w_j)$  that maps weight  $w_j \in \mathcal{W}$  to  $\{0, 1, \dots, |M_{off}| - 1\}$  (i.e., addresses) of  $M_{off}$  that satisfies (1)  $\phi(w_{j_1}) \neq \phi(w_{j_2})$  if  $w_{j_1} \neq w_{j_2}$  for  $w_{j_1}, w_{j_2} \in \mathcal{W}$  and (2) the number of two consecutive accesses  $i_j$  and  $i_{j+1}$  in  $\mathcal{S}$  such that  $ws(i_j)$  and  $ws(i_{j+1})$  are in the same block is maximized.

The memory layout problem is translated into finding an arrangement of the  $|\mathcal{W}|$  weights to  $M_{off}$  so that the number of consecutive accesses of which the corresponding weights are in the same block of  $M_{off}$  is maximized. We propose a greedy algorithm called Mem-layout which iteratively performs the following three steps:

1. **Generating an access graph  $G$ :** From  $\mathcal{S}$ ,  $\mathcal{W}$ , and  $ws(\cdot)$ , we create a graph, called *access graph*  $G(N, E, nSize(\cdot), eSize(\cdot))$  where each node in  $N$  indicates a distinct weight in  $\mathcal{W}$  and there is an edge  $(n_i, n_j) \in E$  if there exists a pair of consecutive index accesses in  $\mathcal{S}$  of which their accessed weights of different values are exactly those of  $n_i$  and  $n_j$ . Node size  $nSize(n_i)$  for every  $n_i \in N$  is set to 1, and edge size  $eSize(n_i, n_j)$  is set to the number of occurrences of consecutive index accesses that form edge  $(n_i, n_j)$ .
2. **Merging a pair of nodes in  $G$ :** We select the edge,  $(n_i, n_j)$ , that has the largest value of  $eSize(\cdot)$  and  $nSize(n_i) + nSize(n_j) \leq |B|$ . If exists, we merge the two terminal nodes of the edge into one in  $G$ . Otherwise, we stop.
3. **Updating  $G$ :** The value of  $nSize(\cdot)$  of the merged node in Step 2 is set to the value of  $nSize(\cdot) + nSize(\cdot)$  for the nodes before merging. Similarly, the values of  $eSize(\cdot)$  for the edges connecting the merged node are updated, by summing the  $eSize(\cdot)$  values of the constituent edges before merging. Finally,  $N$  and  $E$

are updated accordingly. Then, repeat step 2.

Fig. 4.3 shows an example of the stepwise procedure of the application of Mem-layout for the weights and index sequence in Fig. 4.1 with  $|B| = 4$ . The initial access graph constructed by step 1 of Mem-layout is shown in Fig. 4.3(a), from which the two nodes (shown in red color) with the largest  $eSize(\cdot)$  value ( $= 3$ ) are chosen by step 2 of Mem-layout to be merged as indicated in Fig. 4.3(b). Then, step 3 of Mem-layout produces the updated access graph, as shown in Fig. 4.3(c). By repeating this process through Fig. 4.3(c) and Fig. 4.3(d), we generate the final graph shown in Fig. 4.3(e), resulting in a memory layout of two blocks, one containing  $w_1, w_3, w_4$ , and  $w_7$ , the other containing the rest.

*Time complexity:* The total number of iterations of Mem-layout is  $(|B| - 1) \cdot \frac{|\mathcal{W}|}{|B|}$ , which is nearly  $|\mathcal{W}|$ . Step 1 requires  $O(|\mathcal{S}|)$  time in total, step 2 takes  $O(\log|\mathcal{W}|)$  time at each iteration by using maximum heap implementation, and step 3 needs constant time in each iteration. Thus the total time of Mem-layout is bounded by  $O(|\mathcal{S}| + |\mathcal{W}| \cdot \log|\mathcal{W}|)$ .

## 4.2.2 Algorithm for On-chip Memory Block Replacement

If the future access sequence is unknown, as the ordinary program execution, the conventional strategies are to make use of the history of the prior access sequence. The most well-known schemes are LRU, which evicts the block in on-chip memory that is the least recently accessed by exploiting the temporal access locality, and MRU, which evicts the block that is most recently used, considering the uniform distribution of accesses. The other schemes are FIFO/LIFO, which discard the first/last accessed block, and RR, which evicts an arbitrary block.

On the other hand, when the future access sequence is completely known, it is theoretically feasible to devise an optimal replacement scheme. The MIN algorithm in [75, 76, 77] shows that it can guarantee a minimum number of block replacement. However, it incurs overhead for scanning the index sequence whenever a block miss occurs. The potential performance degradation caused by the scanning process can be

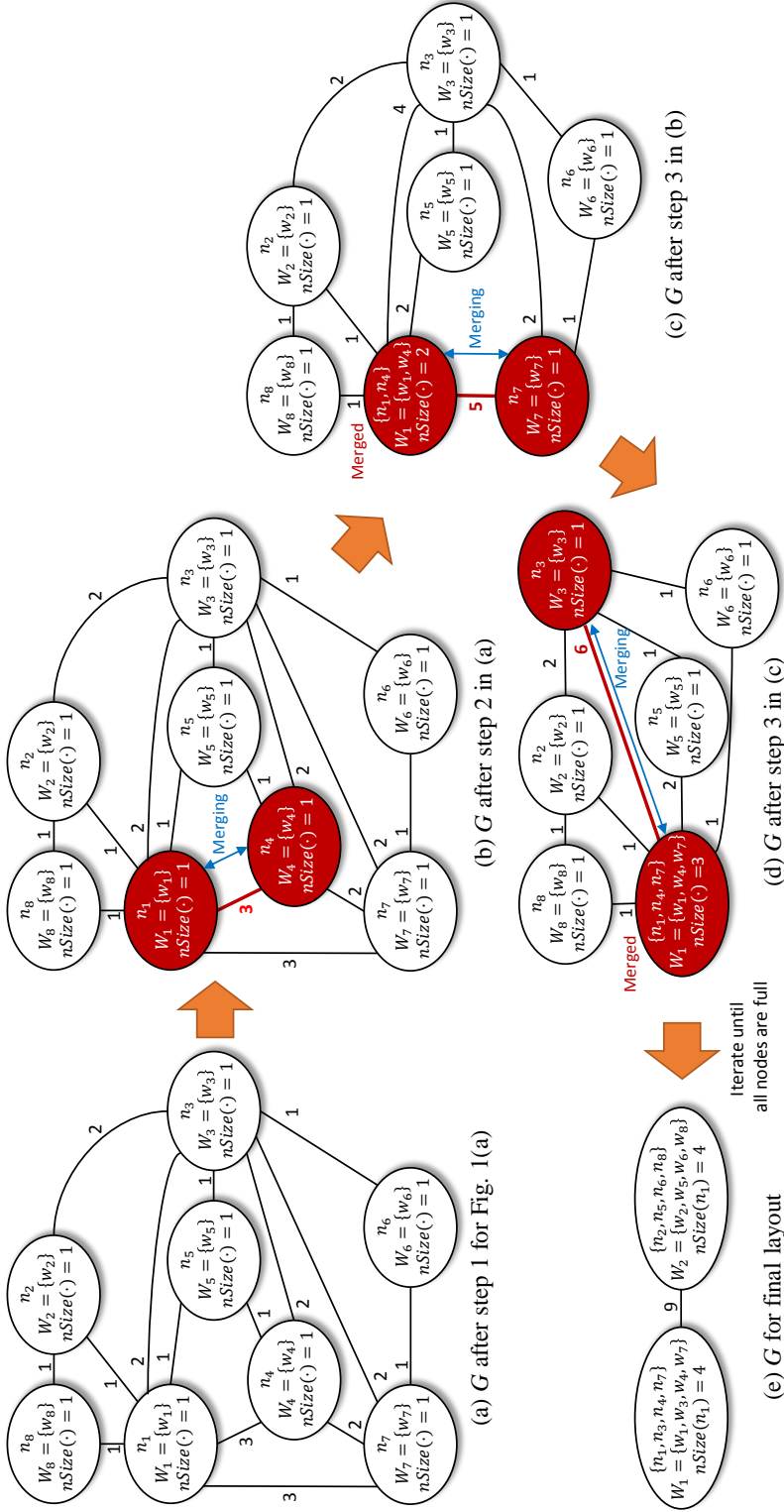


Figure 4.3: An example of Mem-layout. (a) An access graph  $G$  for the index sequence in Fig. 4.1. (b) Select nodes  $n_1$  and  $n_4$  whose edge has the largest  $eSize(\cdot)$  value. (c) Update  $G$  by merging  $n_1$  and  $n_4$ , then repeat this process by selecting  $\{n_1, n_4\}$  and  $n_7$ . (d) Merge  $\{n_1, n_4, n_7\}$  and  $n_3$ . (e) The iteration stops since there is no pair of nodes to merge that meets  $nSize(\cdot) + nSize(\cdot) \leq |B| = 4$ .

overcome by performing the current off-chip memory access and the access sequence scanning for the next replacement block concurrently, but the increase of energy is unavoidable. Here, our objective focuses on minimizing energy consumption.

The block replacement problem for DNNs with unfitted compression can be described as:

**Block replacement problem for DNNs with unfitted compression:** For  $\mathcal{S}$ ,  $\mathcal{W}$ ,  $M_{on}$  with  $|M_{on}| \geq |B|$ ,  $M_{off}$  partitioned by  $|B|$ , and a memory layout result (i.e., by  $\phi(\cdot)$ ), select a block for replacement when a block miss occurs, with the objective of minimizing the quantity of  $E_{tot}$ :

$$E_{tot} = E_{off} + E_{scan} \quad (4.1)$$

where  $E_{off}$  and  $E_{scan}$  represent the amounts of energy consumed by the off-chip memory accesses and the process of scanning the index sequence for selecting blocks for replacement, respectively.

To minimize the  $E_{tot}$  cost in Eq.4.1, we propose a modified version of MIN, called MIN- $k$  that scans the future index sequence up to  $k$  (*scanning distance*) when a block miss happens. Furthermore, to reduce the redundant scanning of indexes, we employ a bookkeeping mechanism using an array of size  $k$ . The value of scanning distance  $k$  is chosen experimentally, which is normally set to a number in  $8 \sim 20$  with nearly no miss-rate increase. Specifically, MIN- $k$  employs three hardware components for the process of selecting a block for eviction:

- $r_p$ : a register that locates the position in the index sequence where the current block miss occurs.
- $r_q$ : a register that locates the position in the index sequence, from which forward or backward scanning will be performed to find a block to evict. Whenever  $r_p > r_q$ , we reset  $r_q$  to  $r_p$  to constrain  $r_q \geq r_p$ . In addition, we constrain the scanning length by:

$$r_q - r_p \leq k \quad (4.2)$$

- $hitCount[\cdot]$ : an array in which each of its element records the number of accesses to a particular block for the interval  $[r_p, r_q]$  in the index sequence. As the values in  $r_p$  and  $r_q$  dynamically change, the value of each element in  $hitCount[\cdot]$  will be updated accordingly. Since  $r_q - r_p \leq k$ , the array size of  $hitCount[\cdot]$  is  $k$ .

We explain how MIN-k selects a block for replacement when block miss occurs by checking/performing the following three actions:

- *Checking 1*: Is there one and only one element in  $hitCount[\cdot]$  of value 0? If the answer is “yes”, the block corresponding to the element will be the one for replacement.

- *Checking 2*: If the answer to *Checking 1* is “no”, we consider two mutually exclusive cases:

A. ( $hitCount[\cdot]$  has more than one element of value 0): perform a *recursive forward scanning* by incrementing  $r_q$  until (A.1) the answer to *Checking 1* is “yes”, (A.2) Eq.4.2 is not met or there is no more index to scan. For A.1, we perform the action in *Checking 1* while for A.2, we randomly select a block whose corresponding element in  $hitCount[\cdot]$  has value 0.

B. ( $hitCount[\cdot]$  has no element of value 0): perform a *recursive backward scanning* by decrementing  $r_q$  until (B.1) the answer to *Checking 1* is “yes”. Then, we perform the action in *Checking 1*. (Note that since backward scanning always decreases the count value in  $hitCount[\cdot]$ , it eventually returns “yes” for *Checking 1*.)

A block diagram corresponding to the procedure of MIN-k that selects a block in on-chip memory for replacement when a block miss occurs is shown in Fig. 4.4.

*Time complexity*: Since both forward and backward scanning for every replacement is bounded by  $O(k)$ , the total run time is  $O(k \cdot N)$  where  $N$  is the number of block misses. We apply MIN-k by varying the value of  $k$  and can find the value of  $k$  that leads to the smallest amount of total energy consumed by the off-chip memory accesses ( $\sim N$ ) and

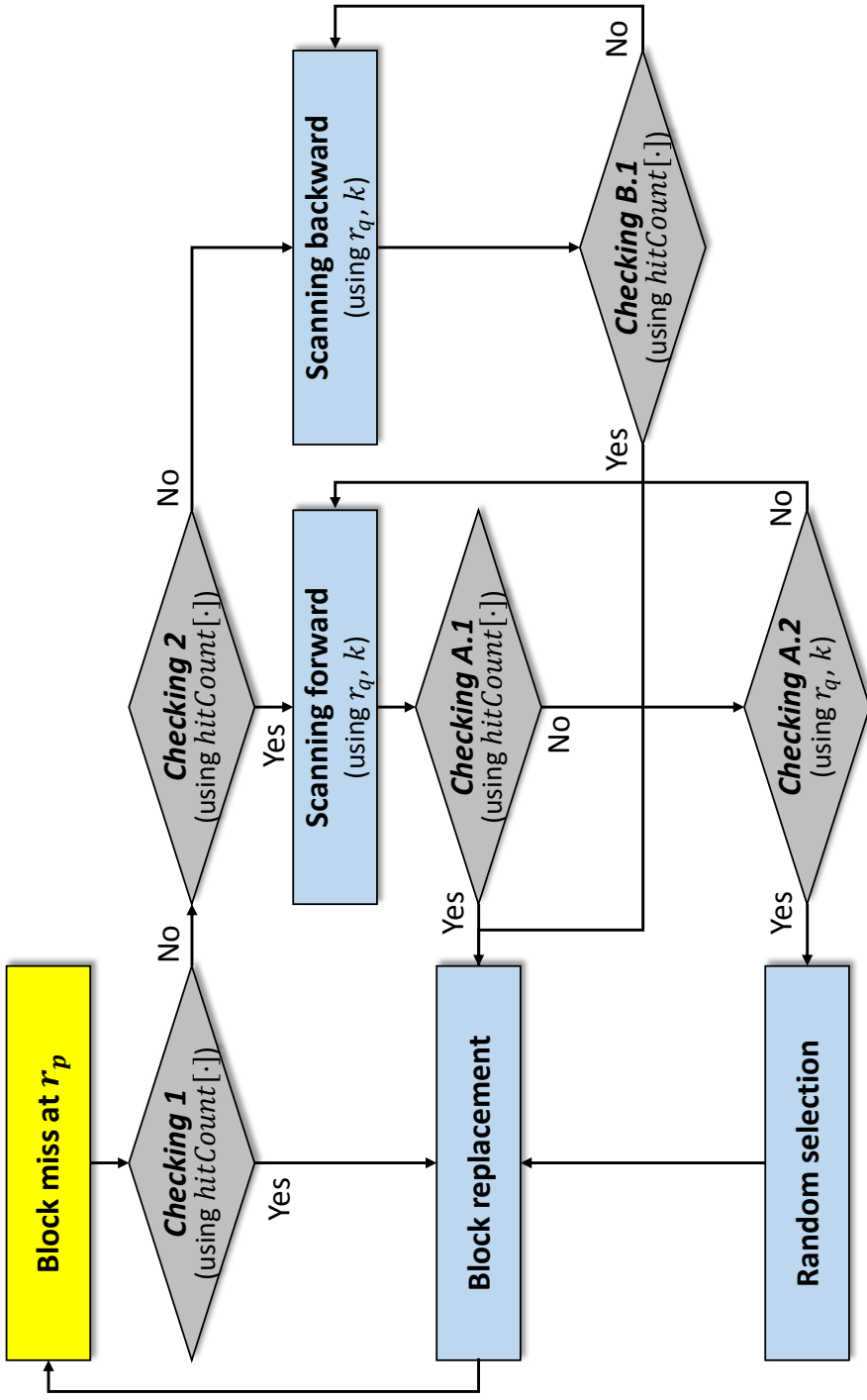


Figure 4.4: Flow of selecting a block for replacement by MIN-k.



on-chip scanning overhead ( $\sim k$ ). (Our experiments show that the scanning overhead in terms of the energy consumption is considerably less than the energy-saving by MIN-k, which means  $N$  is a dominant factor that should be minimized.)

### 4.2.3 Exploitation of Parallel Computing

As shown in Fig. 4.1(a), in a layer with  $m$  input nodes and  $n$  output nodes, up to  $m \times n$  multiplications, and the length of the index sequence is also needed by that amount. Because it takes a long time to process one multiplication operation at a time, parallelism can be applied to improve throughput by processing multiple multiplication operations at the same time.

As shown in Fig. 4.5, each processing element placed in parallel performs independent MAC operation, and each PE calculates by dividing the amount of whole computation required to process a single layer. Each PE has its own on-chip memory which stores input activations and weight information needed for each calculation from common off-chip memory. A unique weight sequence is determined according to the order in which each PE calculates, and there are weight index sequences corresponding to the number of PEs. Since all PEs load common weight blocks from off-chip memory, an optimized memory layout for all index sequences is required. By creating a single access graph from given index sequences and repeating the merge process using the proposed Mem-layout, a single optimized layout for index sequences can be obtained.

Also, since each PE has a different index sequence, it is possible to perform optimal block replacement dynamically by scanning each index sequence when MIN-k is applied independently to each PE to minimize the off-chip memory accesses.

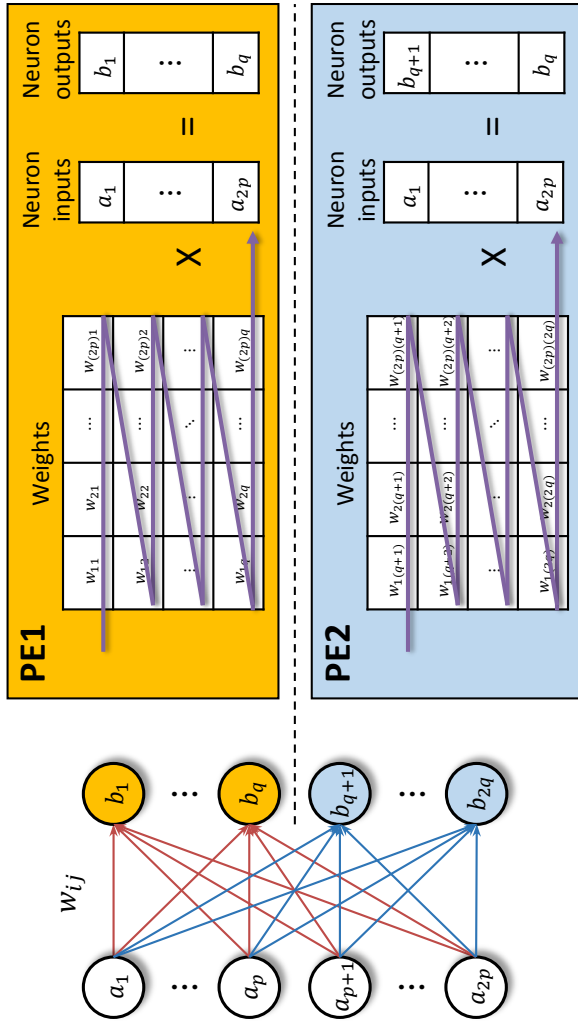


Figure 4.5: An example of calculating one layer with two PEs. Each PE includes on-chip memory and MAC operation unit as shown in Fig. 4.1 (b). The red line represents the calculation performed at PE1 and the blue line represents the calculation performed at PE2. Each PE has their own weight index sequence.

Table 4.1: Statistics of the number of weights before and after the compression. Note: column *before* represents the total number of weight accesses while column *after* represents the number of distinct weights.

AlexNet by [41]	before	after	top 4 / top 8 / top 16 (# accesses)
FC6	2,332,737	252	21.34% / 31.45% / 47.21%
FC7	979,690	173	14.32% / 24.23% / 40.97%
FC8	530,823	254	8.06% / 15.38% / 28.15%

## 4.3 Experimental Results

### 4.3.1 Experimental Setup

The architecture our work targets is the one containing small on-chip memory, which is suitable for embedded applications requiring limited (low cost) on-chip memory resources. An example is the XA6SLX4 model of the Xilinx Spartan 6 FPGA with 12 BRAMs, each of which is capable of storing up to 2.25KB of data, thus, a total of 27KB. For such architecture, careful management of off-chip memory layout and data access is essential because not only the values of weight parameters in DNN model but also the input and output activations and intermediate results should be stored to and accessed from the off-chip memory.

We tested our proposed algorithms **Min-layout** and **MIN-k** on the compressed DNN of AlexNet by **DeepCompression** in [41] from which we used the three fully connected layers FC6, FC7, and FC8. (We obtained the compressed DNN from the Caffe model uploaded on the authors' github.) The statistics of the number of weights before and after the compression are summarized in Table 4.1. The last column of the table indicates that the most frequently accessed 16 weights of the compressed AlexNet take a portion of 28-47% of the total number of weight accesses. The experiments are performed in two-fold: (1) to assess the effectiveness of **Mem-layout** for reducing the number of off-chip memory accesses and (2) to assess the effectiveness of **MIN-k** combined with **Mem-layout** for reducing the energy consumption i.e.,  $E_{tot}$  in Eq.4.1.

### 4.3.2 Assessing the Effectiveness of Mem-layout

Table 4.2 lists the numbers of off-chip weight accesses using the memory layouts before and after the application of **Mem-layout** for two cases: the number of blocks (i.e.,  $|M_{on}|/|B|$ ) in on-chip memory is 4 or 8, assuming the size of a block (i.e.,  $|B|$ ) is 4. The reduction numbers in the table show that **Mem-layout** uses consistently less or

Table 4.2: The numbers of off-chip weight accesses before and after applying our Mem-layout under various conventional schemes of block replacement, assuming  $|B| = 4$ .

AlexNet FC layer	Eviction scheme	# blocks in $M_{on} = 4$		# blocks in $M_{on} = 8$	
		Unopt. / Mem-layout	Red.	Unopt. / Mem-layout	Red.
FC6	FIFO	1,750K / 1,520K	13.1%	1,400K / 1,127K	19.5%
	LIFO	1,794K / 1,434K	20.1%	1,297K / 849K	34.6%
	LRU	1,732K / 1,484K	14.3%	1,349K / 1,045K	22.5%
	MRU	2,009K / 1,894K	5.7%	1,870K / 1,772K	5.3%
	RR	1,762K / 1,534K	13.0%	1,406K / 1,132K	19.5%
FC7	FIFO	762K / 733K	3.8%	573K / 532k	7.09%
	LIFO	701K / 635K	9.3%	460K / 381K	17.2%
	LRU	757K / 724K	4.3%	553K / 504K	8.9%
	MRU	876K / 877K	-0.1%	810K / 819K	-1.2%
	RR	762K / 733K	3.8%	574K / 532K	7.2%
FC8	FIFO	446K / 440K	1.5%	371K / 360k	2.9%
	LIFO	420K / 412K	1.7%	324K / 351K	-8.4%
	LRU	445K / 438K	1.5%	364K / 353K	3.2%
	MRU	484K / 482K	0.3%	454K / 452K	0.4%
	RR	447K / 440K	1.5%	371K / 360K	2.9%

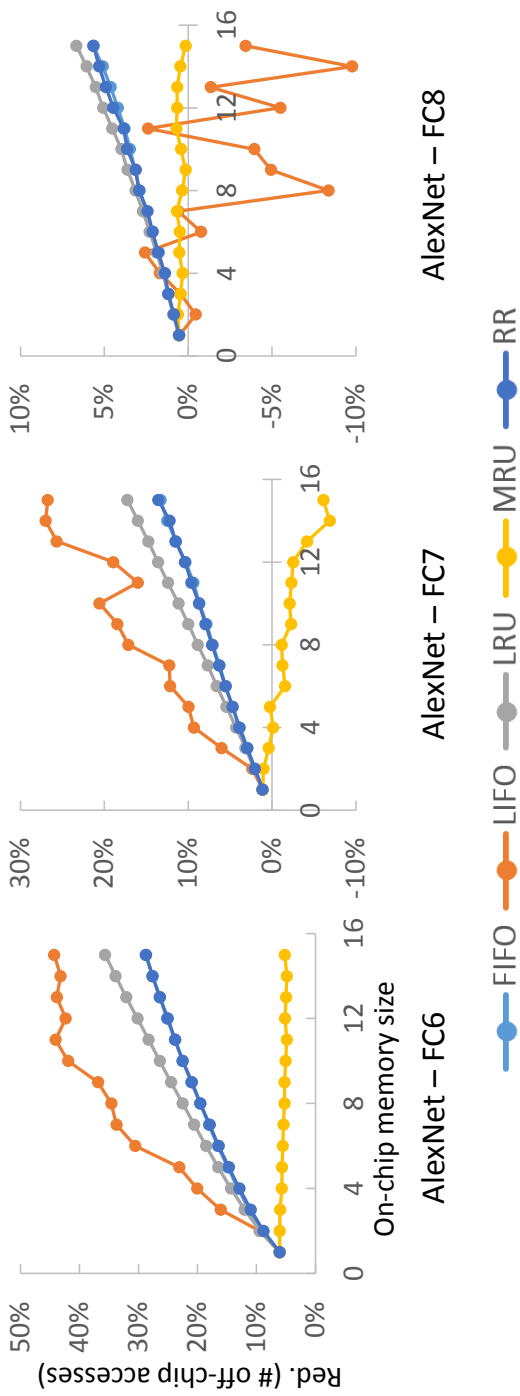


Figure 4.6: Changes in the number of off-chip memory accesses by Mem-layout for the conventional replacement schemes under various sizes, in terms of the number of blocks, of on-chip memory (The size of a unit block is 4 in terms of the number of weights.)

an equal number of off-chip accesses for every combination of replacement schemes and network layers (i.e., access sequences) except that of LIFO on FC8 layer with  $|M_{on}|/|B| = 8$ , which comes from the discrepancy between the use of  $|M_{on}|/|B| = 1$  in Mem-layout to unlink replacement scheme and the use of  $|M_{on}|/|B| = 4$  in experiments. The trend of off-chip memory access saving as well as the discrepancy for the LIFO on FC8 is further validated by the curves shown in Fig. 4.6. Nevertheless, Mem-layout reduces the number of off-chip accesses on FC6 and FC7 with LIFO scheme by up to 45% and 28%, respectively.

### 4.3.3 Assessing the Effectiveness of MIN-k Combined with Mem-layout

Table 4.3 shows the amount of saving ( $\nabla E_{off}$  in Eq.4.1) of the energy consumption for the off-chip memory accesses and the energy overhead ( $E_{scan}$  in Eq.4.1) by scanning index sequence for block replacement by our MIN-k combined with Mem-layout over that by LRU replacement scheme with initial off-chip memory layout. The energy numbers per on-chip and off-chip memories are taken from the energy datasheet at 45nm technology in [15]. The comparison of the values of  $\nabla E_{off}$  and  $E_{scan}$  shows that the energy penalty  $E_{scan}$  is two or three orders of magnitude less than the energy saving  $\nabla E_{off}$ . Overall, the total energy-saving ( $\nabla E_{tot}$  in Eq.4.1) by MIN-k and Mem-layout is 34.2% on average.

Finally, Fig. 4.7 shows the changes of the values of  $E_{off}$  (blue curves) and  $E_{scan}$  (red curves) for various off-chip memory sizes ( $= |M_{on}|/|B|$ ) as the value of scanning distance parameter  $k$  changes. The sharp increase of the value of  $E_{scan}$  implies that using a long scanning distance rapidly increases the amount of energy consumed by the index scanning process. On the other hand, the slow decrease of the value of  $E_{off}$  indicates that using a scanning distance longer than 20 does not much help save the energy consumed by the off-chip memory accesses.

Table 4.3: Energy reduction by our MIN-k and Mem-layout over the conventional approach.

AlexNet	Parameter	# off-chip accesses		Energy consumption (mJ)	
FC layer	$( M ^*, k)$	Conv. <sup>†</sup>	Ours <sup>‡</sup>	$\nabla E_{off} / E_{scan}$	$\nabla E_{tot}$
FC6	(4, 8)	1,732K	1,273K	587.27 / 2.93	584.33
	(4, 16)	1,732K	1,079K	835.91 / 3.44	832.47
	(8, 16)	1,349K	751K	765.27 / 2.91	762.36
	(8, 32)	1,349K	622K	929.80 / 3.43	926.37
FC7	(4, 8)	757K	614K	182.70 / 1.25	181.46
	(4, 16)	757K	514K	310.81 / 1.60	309.21
	(8, 16)	553K	332K	283.13 / 1.23	281.90
	(8, 32)	553K	286K	341.78 / 1.66	340.12
FC8	(4, 8)	445K	394K	65.01 / 0.68	64.34
	(4, 16)	445K	326K	151.94 / 0.83	151.10
	(8, 16)	364K	252K	144.29 / 0.67	143.62
	(8, 32)	364K	213K	193.85 / 0.96	192.88
Average					<b>34.2%</b>

\* $|M| = |M_{on}|/|B|$ , <sup>†</sup>Unoptimized layout + LRU,

<sup>‡</sup>Optimized layout from Mem-layout + MIN-k



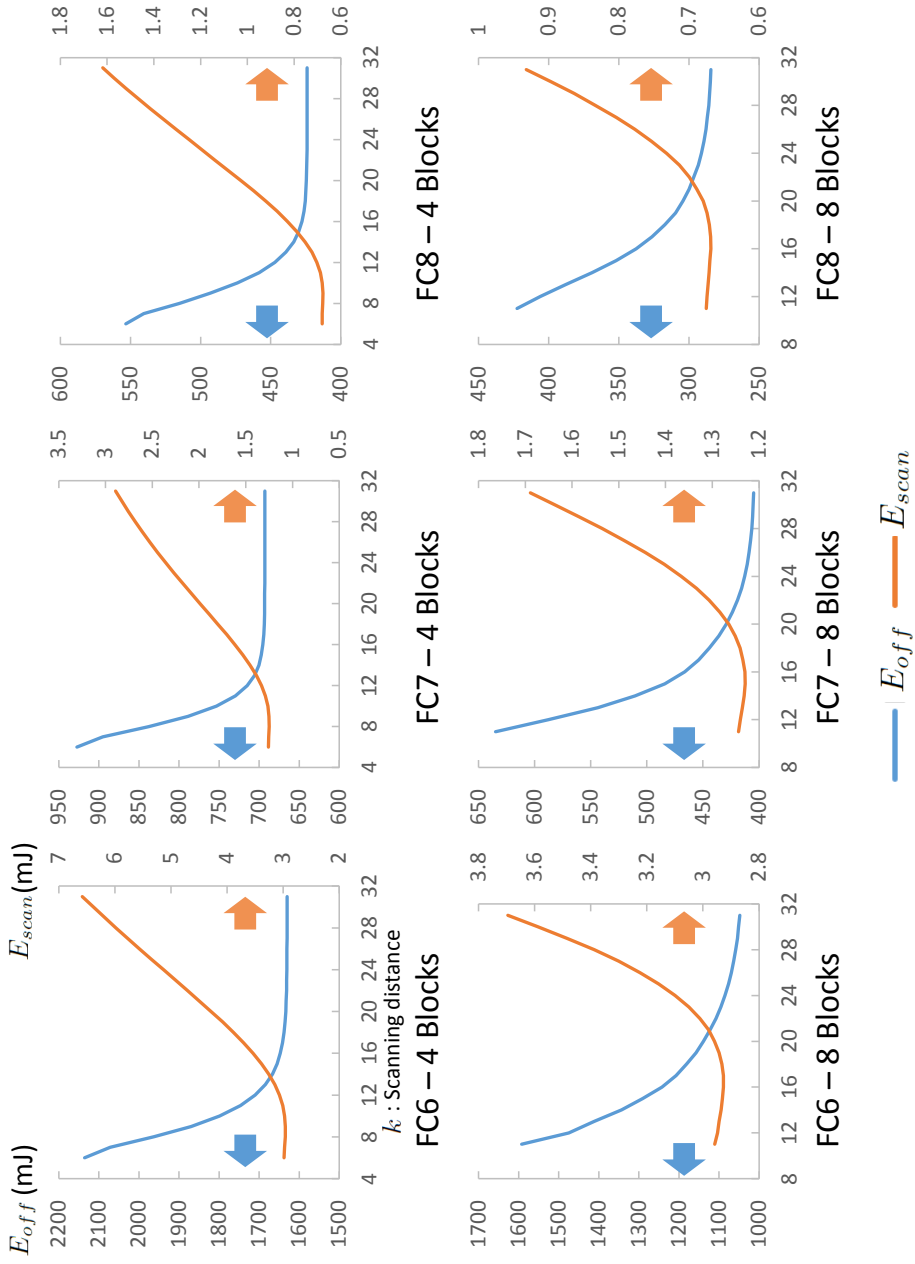


Figure 4.7: Curves showing the trade-off between  $E_{off}$  and  $E_{scan}$  as the value of scanning distance parameter  $k$  changes.



## Chapter 5

### Conclusions

#### 5.1 Bit-level Weight Pruning Techniques for High-Performance Neural Networks

The section proposed an effective technique to resolve the inherent limitation of the bit-level weight pruning: the maximal computation speedup was bounded by the total number of non-zero bits of the weights, but the bound had consistently been considered as ‘unoptimizable’. Specifically, based on the notion of canonical signed digit (CSD) encoding, we (1) proposed a transformation technique which converted the two’s complement representation of every weight into a set of signed-digit representations of the minimal or near-minimal number of essential bits, (2) formulated the problem of selecting signed-digit representations of weights that maximized the parallelism of bit-level multiplication into a multi-objective shortest path problem and solved it efficiently, and (3) proposed a supporting novel acceleration architecture at no additional non-trivial hardware cost. In addition, we (4) proposed a variant to support bit-level parallel multiplication with the capability of predicting a tight worst-case latency of the parallel processing. Through experiments, it was shown that our proposed approach reduced the number of essential bits by 69% on AlexNet, 74% on VGG-16, and 68% on ResNet-152, by which our accelerators of DWP and DWP-intra sped up the infer-

ence computation time up to  $2.22\times$  and  $3.57\times$  over the conventional bit-level weight pruning, respectively. Furthermore, ours improved EDP by up to  $1.42\times$  with a slight area overhead than that of the existing state-of-the-art bit-level pruning architecture.

## **5.2 Convolution Computation Techniques for High-Performance Neural Networks**

The section presented a new algorithm ConvOpt for extracting common kernels and convolutions to maximally eliminate the redundant operations among the convolutions in binary- and ternary-weight convolutional neural networks. Precisely, ConvOpt employed two engines, (1) performing *a new algorithm for common kernel extraction* to overcome the limited and local view of the conventional method and (2) applying *a new concept called common convolution extraction* to maximally eliminate the redundancy in the convolution operations. Besides, ConvOpt was able to (3) tune in *minimizing the number of resulting kernels* for convolutions, thereby saving the total memory access latency for kernels. Experimental results on ternary-weight VGG-16 showed that our convolution optimization ConvOpt was able to reduce the total number of operations by 25.8-26.3%, thereby reducing the total number of execution cycles on hardware platform by 22.4% while using 2.7-3.8% fewer kernels over that of the convolutions utilizing the common kernels extracted by the state-of-the-art algorithm in [64].

## **5.3 Memory Layout and Block Replacement Techniques for High-Performance Neural Networks**

The section introduced a couple of new problems, called off-chip memory layout problem and on-chip block replacement problem, that could arise when DNN compression led to an unsuccessful weight compression in that off-chip memory was necessarily

needed to store all the weights, and proposed effective solutions called Mem-layout and MIN-k for minimizing the total energy consumption by the off-chip memory accesses and the scanning overhead of selecting blocks for replacement. Experiments with the model of AlexNet compression showed that our algorithms were able to save the energy consumption by 32.4% on average over the conventional approach.



# Bibliography

- [1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Advances in Neural Information Processing Systems (NIPS)*, pp. 1097–1105, 2012.
- [2] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *International Conference on Learning Representations (ICLR)*, 2015.
- [3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going Deeper with Convolutions,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.
- [4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.
- [5] F. Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1251–1258, 2017.
- [6] J. Hu, L. Shen, and G. Sun, “Squeeze-and-Excitation Networks,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7132–7141, 2018.

- [7] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, “GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism,” *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 103–112, 2019.
- [8] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Computation*, MIT Press, vol. 9, no. 8, pp. 1735–1780, 1997.
- [9] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling,” *Advances in Neural Information Processing Systems (NIPS) Deep Learning and Representation Learning Workshop*, 2014.
- [10] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft COCO: Common Objects in Context,” *European Conference on Computer Vision (ECCV)*, pp. 740–755, 2014.
- [11] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *Advances in Neural Information Processing Systems (NIPS)*, pp. 91–99, 2015.
- [12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 779–788, 2016.
- [13] J. Long, E. Shelhamer, and T. Darrell, “Fully Convolutional Networks for Semantic Segmentation,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3431–3440, 2015.
- [14] H. Noh, S. Hong, and B. Han, “Learning Deconvolution Network for Semantic Segmentation,” *IEEE International Conference on Computer Vision (ICCV)*, pp. 1520–1528, 2015.



- [15] M. Horowitz, “Energy Table for 45nm Process, Stanford VLSI Wiki,” [Online]. Available: <https://sites.google.com/site/seecproject>. [Accessed: 17-Oct-2020].
- [16] Berkeley AI Research, “Caffe Model Zoo,” [Online]. Available: [https://caffe.berkeleyvision.org/model\\_zoo](https://caffe.berkeleyvision.org/model_zoo). [Accessed: 17-Oct-2020].
- [17] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, “DaDianNao: A Machine-Learning Supercomputer,” *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 609–622, 2014.
- [18] 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,” *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 269–284, 2014.
- [19] M. N. Bojnordi and E. Ipek, “Memristive Boltzmann Machine: A Hardware Accelerator for Combinatorial Optimization and Deep Learning,” *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–13, 2016.
- [20] H. Pourmeidani, S. Sheikhfaal, R. Zand, and R. F. DeMara, “Probabilistic Interpolation Recoder for Energy-Error-Product Efficient DBNs with p-bit Devices,” *IEEE Transactions on Emerging Topics in Computing (TETC)*, 2020.
- [21] M. Jaderberg, A. Vedaldi, and A. Zisserman, “Speeding up Convolutional Neural Networks with Low Rank Expansions,” *British Machine Vision Conference (BMVC)*, 2014.
- [22] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, “Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation,” *Advances in Neural Information Processing Systems (NIPS)*, pp. 1269–1277, 2014.

- [23] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, “Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition,” *International Conference on Learning Representations (ICLR)*, 2015.
- [24] C. Tai, T. Xiao, Y. Zhang, X. Wang, and E. Weinan, “Convolutional Neural Networks with Low-rank Regularization,” *International Conference on Learning Representations (ICLR)*, 2016.
- [25] P. Wang and J. Cheng, “Accelerating Convolutional Neural Networks for Mobile Applications,” *ACM International Conference on Multimedia (ACM-MM)*, pp. 541–545, 2016.
- [26] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, “Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications,” *International Conference on Learning Representations (ICLR)*, 2016.
- [27] H. Zhou, J. M. Alvarez, and F. Porikli, “Less is More: Towards Compact CNNs,” *European Conference on Computer Vision (ECCV)*, pp. 662–677, 2016.
- [28] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning Structured Sparsity in Deep Neural Networks,” *Advances in Neural Information Processing Systems (NIPS)*, pp. 2082–2090, 2016.
- [29] S. Anwar, K. Hwang, and W. Sung, “Structured Pruning of Deep Convolutional Neural Networks,” *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–18, 2017.
- [30] Y. He, X. Zhang, and J. Sun, “Channel Pruning for Accelerating Very Deep Neural Networks,” *IEEE International Conference on Computer Vision (ICCV)*, pp. 1389–1397, 2017.

- [31] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, “Learning Efficient Convolutional Networks through Network Slimming,” *IEEE International Conference on Computer Vision (ICCV)*, pp. 2736–2744, 2017.
- [32] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, “Pruning Filters for Efficient ConvNets,” *International Conference on Learning Representations (ICLR)*, 2017.
- [33] H. Wang, Q. Zhang, Y. Wang, and H. Hu, “Structured Probabilistic Pruning for Convolutional Neural Network Acceleration,” *British Machine Vision Conference (BMVC)*, 2018.
- [34] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, “Quantized Convolutional Neural Networks for Mobile Devices,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4820–4828, 2016.
- [35] D. Miyashita, E. H. Lee, and B. Murmann, “Convolutional Neural Networks using Logarithmic Data Representation,” *arXiv preprint arXiv:1603.01025*, 2016.
- [36] C. Zhu, S. Han, H. Mao, and W. J. Dally, “Trained Ternary Quantization,” *International Conference on Learning Representations (ICLR)*, 2017.
- [37] E. Park, J. Ahn, and S. Yoo, “Weighted-Entropy-Based Quantization for Deep Neural Networks,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7197–7205, 2017.
- [38] S.-C. Zhou, Y.-Z. Wang, H. Wen, Q.-Y. He, and Y.-H. Zou, “Balanced Quantization: An Effective and Efficient Approach to Quantized Neural Networks,” *Journal of Computer Science and Technology (JCST)*, vol. 32, no. 4, pp. 667–682, 2017.

- [39] E. Park, D. Kim, and S. Yoo, “Energy-Efficient Neural Network Accelerator Based on Outlier-Aware Low-Precision Computation,” *International Symposium on Computer Architecture (ISCA)*, pp. 688–698, 2018.
- [40] W. Chen, J. Wilson, S. Tyree, K. Weinberger, and Y. Chen, “Compressing Neural Networks with the Hashing Trick,” *International Conference on Machine Learning (ICML)*, pp. 2285–2294, 2015.
- [41] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *International Conference on Learning Representations (ICLR)*, 2016.
- [42] H. Lu, X. Wei, N. Lin, G. Yan, and X. Li, “Tetris: Re-architecting Convolutional Neural Network Computation for Machine Learning Accelerators,” *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, 2018.
- [43] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing,” *International Symposium on Computer Architecture (ISCA)*, pp. 1–13, 2016.
- [44] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks,” *International Symposium on Computer Architecture (ISCA)*, pp. 27–40, 2017.
- [45] D. Kim, J. Ahn, and S. Yoo, “ZeNA: Zero-Aware Neural Network Accelerator,” *IEEE Design & Test (D&T)*, vol. 35, no. 1, pp. 39–46, 2017.
- [46] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, “Stripes: Bit-Serial Deep Neural Network Computing,” *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2016.

- [47] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O’Leary, R. Genov, and A. Moshovos, “Bit-Pragmatic Deep Neural Network Computing,” *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 382–394, 2017.
- [48] S. Sharify, A. D. Lascorz, K. Siu, P. Judd, and A. Moshovos, “Loom: Exploiting Weight and Activation Precisions to Accelerate Convolutional Neural Networks,” *Design Automation Conference (DAC)*, pp. 1–6, 2018.
- [49] Z. Chen, G. J. Blair, H. T. Blair, and J. Cong, “BLINK: Bit-Sparse LSTM Inference Kernel Enabling Efficient Calcium Trace Extraction for Neurofeedback Devices,” *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 217–222, 2020.
- [50] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, “SqueezeNet: AlexNet-level Accuracy with 50x Fewer Parameters and <0.5MB Model Size,” *arXiv preprint arXiv:1602.07360*, 2016.
- [51] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4510–4520, 2018.
- [52] X. Zhang, X. Zhou, M. Lin, and J. Sun, “ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices,” *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6848–6856, 2018.
- [53] D. Hammerstrom, “A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning,” *International Joint Conference on Neural Networks (IJCNN)*, pp. 537–544, 1990.
- [54] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep Learning with Limited Numerical Precision,” *International Conference on Machine Learning (ICML)*, pp. 1737–1746, 2015.

- [55] M. Courbariaux, Y. Bengio, and J.-P. David, “Training Deep Neural Networks with Low Precision Multiplications,” *International Conference on Learning Representations (ICLR)*, 2015.
- [56] M. Courbariaux, Y. Bengio, and J.-P. David, “BinaryConnect: Training Deep Neural Networks with Binary Weights during Propagations,” *Advances in Neural Information Processing Systems (NIPS)*, pp. 3123–3131, 2015.
- [57] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks,” *European Conference on Computer Vision (ECCV)*, pp. 525–542, 2016.
- [58] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients,” *arXiv preprint arXiv:1606.06160*, 2016.
- [59] M. Kim and P. Smaragdis, “Bitwise Neural Networks,” *International Conference on Machine Learning (ICML) Workshop on Resource-Efficient Machine Learning*, 2015.
- [60] K. Hwang and W. Sung, “Fixed-Point Feedforward Deep Neural Network Design Using Weights +1, 0, and -1,” *IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 1–6, 2014.
- [61] F. Li, B. Zhang, and B. Liu, “Ternary Weight Networks,” *arXiv preprint arXiv:1605.04711*, 2016.
- [62] Z. Lin, M. Courbariaux, R. Memisevic, and Y. Bengio, “Neural Networks with Few Multiplications,” *International Conference on Learning Representations (ICLR)*, 2016.

- [63] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, “Ternary Neural Networks for Resource-Efficient AI Applications,” *IEEE International Joint Conference on Neural Networks (IJCNN)*, pp. 2547–2554, 2017.
- [64] S. Zheng, Y. Liu, S. Yin, L. Liu, and S. Wei, “An Efficient Kernel Transformation Architecture for Binary- and Ternary-Weight Neural Network Inference,” *Design Automation Conference (DAC)*, 2018.
- [65] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, “YodaNN: An Architecture for Ultra-Low Power Binary-Weight CNN Acceleration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 48–60, 2018.
- [66] H. Kim, J. Sim, Y. Choi, and L.-S. Kim, “A Kernel Decomposition Architecture for Binary-weight Convolutional Neural Networks,” *Design Automation Conference (DAC)*, 2017.
- [67] G. W. Reitwiesner, “Binary Arithmetic,” *Advances in Computers*, Elsevier, vol. 1, pp. 261–265, 1960.
- [68] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*, John Wiley & Sons, 2007.
- [69] M. Ehrgott, *Multicriteria Optimization*, Springer Science & Business Media, vol. 491, 2005.
- [70] A. Warburton, “Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems,” *Operations Research*, INFORMS, vol. 35, no. 1, pp. 70–79, 1987.
- [71] N. Maeda, S. Komatsu, M. Morimoto, K. Tanaka, Y. Tsukamoto, K. Nii, and Y. Shimazaki, “A 0.41  $\mu$ A Standby Leakage 32 kb Embedded SRAM with Low-Voltage Resume-Standby Utilizing All Digital Current Comparator in 28

- nm HKMG CMOS,” *IEEE Journal of Solid-State Circuits*, IEEE, vol. 48, no. 4, pp. 917–923, 2013.
- [72] G. Wang, D. Anand, N. Butt, A. Cestero, M. Chudzik, J. Ervin, S. Fang, G. Freeman, H. Ho, B. Khan, B. Kim, W. Kong, R. Krishnan, S. Krishnan, O. Kwon, J. Liu, K. McStay, E. Nelson, K. Nummy, P. Parries, J. Sim, R. Takalkar, A. Tessier, R. M. Todi, R. Malik, S. Stiffler, and S. S. Iyer, “Scaling Deep Trench Based eDRAM on SOI to 32nm and Beyond,” *IEEE International Electron Devices Meeting (IEDM)*, pp. 1–4, 2009.
- [73] K. C. Huang, Y. W. Ting, C. Y. Chang, K. C. Tu, K. C. Tzeng, H. C. Chu, C. Y. Pai, A. Katoch, W. H. Kuo, K. W. Chen, T. H. Hsieh, C. Y. Tsai, W. C. Chiang, H. F. Lee, A. Achyuthan, C. Y. Chen, H. W. Chin, M. J. Wang, C. J. Wang, C. S. Tsai, C. M. Oconnell, S. Natarajan, S. G. Wu, I. F. Wang, H. Y. Hwang, and L. C. Tran, “A High-Performance, High-Density 28nm eDRAM Technology with High-K/Metal-Gate,” *IEEE International Electron Devices Meeting (IEDM)*, pp. 1–4, 2011.
- [74] Micron, “DDR3 SDRAM RDIMM,” [Online]. Available: [https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/parity\\_rdim/jsf18c1gx72pdz.pdf](https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/parity_rdim/jsf18c1gx72pdz.pdf). [Accessed: 19-Jan-2021].
- [75] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation Techniques for Storage Hierarchies,” *IBM Systems Journal*, IBM, vol. 9, no. 2, pp. 78–117, 1970.
- [76] B. Van Roy, “A Short Proof of Optimality for The MIN Cache Replacement Algorithm,” *Information Processing Letters*, Elsevier, vol. 102, no. 2-3, pp. 72–73, 2007.



- [77] W. Vogler, “Another Short Proof of Optimality for The MIN Cache Replacement Algorithm,” *Information Processing Letters*, Elsevier, vol. 106, no. 5, pp. 219–220, 2008.



# 초 록

인공 신경망 연산을 수행하고자 하는 수요가 꾸준히 증가하고 있지만, 깊은 인공 신경망에는 과도한 메모리와 계산 비용이 수반되기 때문에 많은 설계 문제가 있다. 본 논문에서는 인공 신경망 추론 연산을 효과적으로 처리하기 위한 여러 가지 새로운 기술을 연구한다.

첫 번째로, 최대 계산 속도 향상이 가중치의 0 아닌 비트의 총 수에 의해 제한되는 한계의 극복을 시도한다. 구체적으로, 부호있는 숫자 인코딩에 기반한 본 연구에서, (1) 모든 가중치의 2의 보수 표현을 필수 비트를 최소로 하는 부호있는 숫자 표현의 집합으로 변환하는 변환 기법을 제안하며, (2) 가중치의 비트 단위 곱셈의 병렬성을 최대화하는 가중치의 부호있는 숫자 표현을 선택하는 문제를 숫자 인덱스 (열 단위) 압축 최대화를 달성하도록 다목적 최단 경로 문제로 공식화하여 근사 알고리즘을 사용하여 효율적으로 해결하며, (3) 주요 하드웨어를 추가로 포함하지 않고 앞서 제안한 기법을 지원하는 새로운 가속기 아키텍처(DWP)를 제안한다. 또한, 우리는 (4) 병렬 처리에서 최악의 지연 시간을 엄격하게 예측할 수 있는 기능이 포함된 비트 단위 병렬 곱셈을 지원하도록 다른 형태의 DWP를 제안한다. 실험을 통해 본 연구에서 제안하는 접근 방법은 필수 비트 수를 AlexNet에서 69%, VGG-16에서 74%, ResNet-152에서 68%까지 줄일 수 있음을 보여주었다. 또한 이를 지원하는 가속기는 추론 연산 시간을 기존의 비트 단위 가중치 가지치기 방법에 비해 최대 3.57배까지 감소시켰다.

두 번째로, 이진 및 삼진 가중치의 컨볼루션 인공 신경망에서 컨볼루션 간의 중복 연산을 최대한 제거하기 위하여 공통 커널 및 컨볼루션을 추출하는 새로운 알

고리즘을 제시한다. 구체적으로, (1) 기존 방법에서 공통 커널 후보의 국부적이고 제한적인 탐색을 극복하기 위한 새로운 공통 커널 추출 알고리즘을 제안하고, 이후에 (2) 컨볼루션 연산에서의 중복성을 최대한으로 제거하기 위한 새로운 개념의 공통 컨볼루션 추출을 적용한다. 또한, 우리의 알고리즘은 (3) 컨볼루션에 대해 최종적으로 도출된 커널 수를 최소화하여 커널에 대한 총 메모리 접근 지연 시간을 절약할 수 있다. 삼진 가중치의 VGG-16에 대한 실험 결과로 모든 컨볼루션에 대한 총 연산 수를 25.8-26.3% 감소시켜, 최신 알고리즘으로 추출한 공통 커널을 사용하는 컨볼루션에 비해 2.7-3.8% 더 적은 커널을 사용하는 동안 하드웨어 플랫폼에서의 총 수행 사이클을 22.4% 감소시켰으므로 우리가 제안한 컨볼루션 최적화 알고리즘이 매우 효과적임을 보였다.

마지막으로, 우리는 압축된 DNN의 모든 고유 가중치들을 온-칩 메모리에 완전히 포함할 수 없는 경우 정확도 유지를 위해 “부적합 압축”을 사용하는 DNN 솔루션을 제안한다. 구체적으로, 가중치의 접근 시퀀스가 주어지면, (1) 첫 번째 문제는 오프-칩 메모리의 메모리 접근 수(접근에 의해 소비되는 에너지)를 최소화하도록 오프-칩 메모리에 가중치를 배열하는 것이고, (2) 두 번째 문제는 블록 교체를 위한 인덱스 탐색에 소비되는 오버헤드와 오프-칩 메모리 접근에 소모되는 총 에너지의 최소화를 목적으로 하여 블록 미스 발생 시 온-칩 메모리에서 교체될 가중치 블록을 선택하는 전략을 고안하는 것이다. 압축된 AlexNet 모델을 사용한 실험을 통해 우리의 솔루션은 최적화되지 않은 메모리 레이아웃 및 LRU 교체 방법을 사용하는 경우에 비해 탐색 오버헤드를 포함하여 오프-칩 메모리 접근에 필요한 총 에너지 소비를 평균 34.2%까지 줄일 수 있음을 보였다.

**주요어:** 깊은 인공 신경망, 비트 단위 가중치 가지치기, 부호 있는 숫자 표현, 공통 커널 추출, 공통 컨볼루션 추출, 부적합 압축

**학번:** 2015-20943