Efficient Deep Neural Network Computation on Processors

by

Jiecao Yu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2019

Doctoral Committee:

Professor Scott A. Mahlke, Chair
Assistant Professor Reetuparna Das
Assistant Professor Hun-Seok Kim
Assistant Professor Andrew Lukefahr, Indiana University
Professor Trevor N. Mudge

Jiecao Yu
jiecaoyu@umich.edu
ORCID iD: 0000-0003-2085-0312

© Jiecao Yu 2019

ACKNOWLEDGMENTS

This is an exciting and challenging journey.

I would like to express my sincere gratitude to my advisor, Prof. Scott Mahlke. His insights and advice helped shape every part of this dissertation. He gave me much freedom to try the research directions I am interested in. With his guidance, I learned a lot about how to find and solve critical problems.

I want to thank Prof. Andrew Lukefahr. Our collaboration starts the very first day I came to Michigan. He is not only a great mentor but also a great friend. I am grateful to Prof. Reetuparna Das. Her guidance is essential for finishing this work. Also, I want to thank Prof. Trevor Mudge and Prof. Hun-Seok Kim. As the committee members, they have provided valuable insights for improving my work.

I am proud of being a member of the Compilers Creating Custom Processors (CCCP) Research Group. Shruti Padmanabha, Ankit Sethia, Mehrzad Samadi, Daya S Khudia, Janghaeng Lee, Gaurav Chadha, Jason Park, Anoushe Jamshidi, John Kloosterman, Babak Zamirai, Jonathan Bailey, Sunghyun Park, Salar Latifi, Ze Zhang, Armand Behroozi, Pedram Zamirai and Brandon Nguyen have been great colleagues who I can share insights and discuss problems with. Also, I thank my friends across the whole world for sharing my happiness and sadness.

Above all, I would like to thank my parents, Guojun Yu and Lijun Hu. It would not have been possible for me to complete this work without their support.

TABLE OF CONTENTS

Acknowledgments . . . . **ii**

List of Figures . . . . **vi**

List of Tables . . . . **ix**

Abstract . . . . **x**

Chapter

1 Introduction **1**

1.1 DNN Computation Reduction 2
1.1.1 Network Compression 2
1.1.2 Domain Transformation 3
1.2 Contributions 4
1.2.1 Hardware Parallelism Aware Pruning 4
1.2.2 Sub-byte DNN on IoT Microcontrollers 5
1.2.3 Winograd-Domain Pruning 5

2 Background and Related Work **7**

2.1 Deep Neural Networks 7
2.2 Network Compression 8
2.2.1 Pruning 8
2.2.2 Low-Precision DNNs 9
2.3 Hardware-Based DNN Acceleration 9
2.4 Bit-Level Computation Optimization 10
2.5 Winograd Convolution 10

3 Hardware Parallelism Aware Pruning **12**

3.1 Introduction 12
3.2 Background and Motivation 15
3.2.1 DNN Weight Pruning 15
3.2.2 Challenges 16
3.3 Scalpel 20
3.3.1 Overview 20
3.3.2 Hardware with Different Parallelism 21
3.3.3 SIMD-Aware Weight Pruning 22

3.3.4 Node Pruning . . . 25
3.3.5 Combined Pruning . . . 29
3.4 Experiment Methodology . . . 32
3.5 Evaluation Results . . . 34
3.5.1 Microcontroller - Low Parallelism . . . 34
3.5.2 CPU - Moderate Parallelism . . . 37
3.5.3 GPU - High Parallelism . . . 38
3.6 Summary . . . 40

4 Sub-Byte DNN on IoT Microcontrollers . . . 41

4.1 Introduction . . . 41
4.2 Background and Motivation . . . 44
4.2.1 Sub-Byte DNNs . . . 44
4.2.2 Challenges . . . 46
4.3 Sub-byte DNN Deploying Pipeline . . . 48
4.3.1 Uniform LQ-Net . . . 49
4.3.2 Bitset Self-Loop Multiplication . . . 52
4.4 GoLo Extensions . . . 59
4.4.1 Simple Arithmetic Instructions . . . 59
4.4.2 DMA-Based Bitset Multiply . . . 61
4.4.3 Double-Width DMA-Based Bitset Multiply . . . 64
4.5 Evaluation Methodology . . . 65
4.5.1 Hardware and DNN Benchmarks . . . 65
4.5.2 Performance, Area, Energy Measurement . . . 66
4.6 Evaluation Results . . . 67
4.6.1 Sub-Byte DNN Accuracy . . . 67
4.6.2 Computation Performance . . . 68
4.6.3 Cost Utilization and Energy Efficiency . . . 68
4.6.4 Storage Requirement . . . 70
4.7 Summary . . . 71

5 Pruning for Winograd-Domain Sparsity . . . 72

5.1 Introduction . . . 72
5.2 Background and Motivation . . . 74
5.3 Spatial-Winograd Pruning . . . 76
5.3.1 Spatial Structured Pruning . . . 77
5.3.2 Winograd Direct Pruning . . . 79
5.4 Experiments . . . 83
5.4.1 CIFAR-10 and CIFAR-100 . . . 84
5.4.2 ImageNet . . . 84
5.4.3 Sparsity Distribution . . . 87
5.4.4 ResNet-18 Pruning with Varied Sparsities across Layers . . . 89
5.4.5 Computation Performance . . . 90
5.4.6 Effectiveness of Importance Factor Matrix . . . 92
5.5 Summary . . . 94

6 Conclusion and Future Directions **95**

6.1 Conclusion 95
6.2 Future Directions 96

Bibliography **99**

LIST OF FIGURES

Figures

2.1 Deep Neural Networks (DNNs) structure. DNNs integrate convolutional (CONV) layers and fully-connected (FC) layers into an end-to-end structure. 8

3.1 (A) Dense weight matrix; (B) Sparse weight matrix; (C) Compressed Sparse Rows (CSR) format for sparse matrices. 16

3.2 Relative execution time (Time Microcontroller/ CPU/ GPU), model sizes, and MAC operations of the networks pruned with the traditional pruning technique in Deep Compression. NIN and AlexNet are not tested on the microcontroller due to its limited storage size. 17

3.3 Execution time breakdown of the original networks (Dense) and the pruned networks (Sparse). The expected execution time of pruned networks (Expected) shows the relative MAC operations remaining. The second CONV layer (*conv2*) in AlexNet is tested. 18

3.4 Numbers of cache access and cache miss in the computation of the second CONV layer (*conv2*) in AlexNet on Intel Core i7-6700 CPU. The original dense layer (Dense), the sparse layer with 80% (Sparse-0.80) and 60% (Sparse-0.60) of weights removed are tested. The matrices are randomly generated. 19

3.5 Overview of Scalpel. 20

3.6 Main steps of SIMD-aware weight pruning. 23

3.7 (A) Weights grouping; (B) Sparse weight matrix after pruning weight groups; (C) Modified CSR format for SIMD-aware weight pruning. 23

3.8 Relative execution time of sparse matrix multiplication on ARM Cortex-M4 with respect to the original dense matrix-vector or matrix-matrix multiplication. MV/MM-Sparse show the results for sparse Matrix-Vector (MV) and Matrix-Matrix (MM) multiplication, respectively. MV/MM-SIMD Sparse shows the corresponding performance with nonzero elements grouped and aligned as SIMD-aware weight pruning does. All matrices have the size of 100 x 100 and are randomly generated. 25

3.9 Relative execution time for sparse matrix-matrix multiplication (Sparse) on NVIDIA GTX Titan X with respect to original execution time (Dense). The matrices have the sizes of 4096x4096 and 4096x50. 26

3.10 Main steps of node pruning. 27

3.11 Mask layers. Node *A-3* with $\alpha_3 = 0$ can be removed. The whole mask layer A' will be removed after pruning all redundant nodes. 27

3.12 Relative execution time for sparse matrix-vector multiplication (FC layers) on Intel Core i7-6700. The matrix size is 4096 x 4096 and the vector size is 4096. MKL-Dense/Sparse show the results of dense and sparse weight matrix with the Intel MKL library. 30
3.13 Relative execution time for sparse matrix-matrix multiplication (CONV layers) on Intel Core i7-6700. The weight matrix and input matrix have the size of 128 x 1200 and 1200 x 729, respectively. 31
3.14 Relative performance speedups of the original models (original), traditional pruning, optimized pruning and Scalpel on ARM Cortex-M4 microcontroller. NIN and AlexNet are not tested due to the limited storage size of the microcontroller. 35
3.15 Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for ARM Cortex-M4 microcontroller. 35
3.16 Relative accuracy against pruning rate with different metrics for importance measurement: maximum absolute value (MAX), mean absolute value (MEAN) and root-mean-square (RMS). 36
3.17 Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on Intel Core i7-6700 CPU. 37
3.18 Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for Intel Core i7-6700 CPU. 37
3.19 Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on NVIDIA GTX Titan X GPU. 38
3.20 Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for NVIDIA GTX Titan X GPU. 39

4.1 Conventional end-to-end visual IoT system. 42
4.2 Conventional computation algorithm of convolutional layers on IoT microcontrollers. . . 44
4.3 Prediction accuracy of NIN with different input/weight precisions. 46
4.4 Examples of unpacking sub-byte inputs/weights. 46
4.5 Relative execution time breakdown of NIN with different input/weight precisions. Unpacking inputs/weights from the sub-byte formats will lead to an execution time increase. 48
4.6 Storage requirements breakdown of NIN with different input/weight precisions. Input matrix (Input Mat) and input feature maps (Input FMs) have to be stored in limited SRAM storage, e.g., 128KB or 256KB. 49
4.7 Overview of the sub-byte DNN deploying pipeline. 49
4.8 Quantization in gated recurrent units (GRUs). 50
4.9 Overview of the bitset self-loop convolution. 52
4.10 SRAM storage requirement of NIN. The conventional sub-byte convolution algorithm is annotated as *Unpacking (Unpack.)*. 54
4.11 Relative computation performance of NIN with different convolution algorithms. The baseline uses 16-bit integers for both inputs and weights. 56
4.12 Bitset multiplication. $\&$, $\overline{\mathcal{B}}$, $\oplus$, $\overline{\mathcal{B} \oplus \mathcal{H}}$ are bitwise AND, NOT, XOR, XNOR operations, respectively. 57
4.13 Relative computation performance of NIN with the GoLo extensions. 61
4.14 Architecture support for the microprogramming instructions. 62

4.15 Accuracy of sub-byte VGG-8 and ResNet-20. The result for NIN is shown in Figure 4.3. 67
4.16 Relative performance of networks with 4-bit inputs/ternary weights(4/2) and 2-bit inputs/binary weights(2/1). 68
4.17 Energy efficiency of the tested networks. 69
4.18 SRAM requirements of networks with different input precisions. 70

5.1 Conventional Winograd convolution and sparse Winograd convolution ($m = 4$, $n = 3$). 74
5.2 Overview of the spatial-Winograd pruning. 76
5.3 Pruning of (a) VGG-nagadomi on CIFAR-10 (b) ConvPool-CNN-C on CIFAR-100 with uniform sparsity across the pruned layers. 85
5.4 Pruning of ResNet-18 on ImageNet with uniform sparsity across the pruned layers. . . 86
5.5 Number of data values that need to be accessed, including weights, inputs and outputs for ResNet-18 computation. 87
5.6 Distribution of 2D filters with different numbers of weights pruned. 88
5.7 Sparsity of different locations of Winograd-domain weights for: (a) filters with 20 weights pruned; (b) filters with at least one weight remaining. Darker locations have higher sparsities. 88
5.8 Accuracy loss of ResNet-18 when incurring 60% Winograd-domain sparsity into different layers. Spatial structured pruning is applied with no retraining. 89
5.9 Relative arithmetic operations (OPs) required for different models. 91
5.10 Relative execution time for different models. 91
5.11 Effectiveness of employing importance factor matrix $\boldsymbol{F}$ in (a) Winograd pruning and (b) Winograd retraining. 93

LIST OF TABLES

Tables

3.1 Hardware platforms with different parallelism. 21
3.2 DNN benchmarks. 33
3.3 Results overview. 34
3.4 Percentage of nodes removed by node pruning in each layer. Output layers are not included. 40

4.1 Synthesis results of integrated circuit modules. 60
4.2 DNN benchmarks. 66
4.3 Microcontroller (Micro.) die areas, including CPU, flash and SRAM, and normalized manufacturing (Manu.) costs for different SRAM sizes under 90nm. 71

5.1 The sparsity for the pruned convolutional layers when pruning the second convolutional layer in each residual block of ResNet-18. 90

ABSTRACT

Deep neural networks (DNNs) have become a fundamental component of various applications. They are trained with a large amount of data to make accurate predictions. However, conventional DNN models have high computation and storage cost. These costs make it challenging to deploy DNN-based algorithms on existing processors.

There are various algorithms proposed to reduce DNN computation. These algorithms can be divided into two main categories: network compression and domain transformation. Network compression removes the redundancy in DNN models by either pruning unimportant parameters or lowering the parameter precisions. It helps reduce both the required computation and storage space. For domain transformation, convolution operations are converted into different domains and replaced with less computation. Nevertheless, these algorithms are designed without considering the characteristics of underlying processors, which may lead to a degradation in computation performance and an increase in model size.

This thesis solves this challenge by customizing the computation reduction algorithms for the processor architecture, and augmenting the hardware to provide better support for DNN computation. The first part of this thesis proposes to customize DNN pruning techniques for the underlying processors by matching the pruned network structure to the parallel hardware organization. Two techniques are introduced: SIMD-aware weight pruning and node pruning. SIMD-aware weight pruning maintains weights in aligned fixed-size groups to fully utilize the SIMD support. Node pruning removes redundant nodes instead of individual weights to reduce computation without sacrificing the dense matrix format. These two techniques are combined based on the hardware

parallelism and layer types.

Besides pruning, I investigate deploying sub-byte DNN models on microcontrollers. Due to the incompatibility between the sub-byte data formats and the byte-addressable memory hierarchy, using sub-byte weights and inputs will cause significant performance degradation. To address this issue, a new convolution algorithm is proposed to perform multiply-accumulate computations with bitwise logic operations. Extra instruction set architecture (ISA) extensions are introduced to accelerate the computation further.

The last part focuses on accelerating DNN computation by combining pruning techniques and Winograd convolution. These two techniques cannot be directly combined because Winograd transformation fills in the sparsity resulting from spatial-domain pruning. To achieve a higher Winograd-domain sparsity, I propose a new pruning method, spatial-Winograd pruning. As the first step, spatial-domain weights are pruned in a structured way, which efficiently transfers the spatial-domain sparsity into the Winograd domain. For the next step, pruning and retraining are performed directly in the Winograd domain to increase the sparsity further.

With these proposed techniques, this thesis solves the conflicts between the existing processor architectures and the computation reduction algorithms, enabling efficient DNN computation.

CHAPTER 1

Introduction

Deep neural networks (DNNs), especially convolutional neural networks (CNNs), have become ubiquitous in various application domains including computer vision [1], natural language processing [2], and speech recognition [3]. By extracting high-level features learned from an immense amount of data, DNNs achieve and even exceed human-level accuracy on various artificial intelligence (AI) tasks [4, 5].

However, the superior accuracy of DNN models comes at the cost of a significant amount of parameters and computation. As an example, the ResNet-152 model [6], which achieves 94.29% top-5 accuracy on ImageNet challenge [7], has 60M parameters and requires 11.3×10^9 multiply-accumulate (MAC) operations. This high storage and computation cost limits the deployment of larger and deeper DNN models.

Meanwhile, although various FPGA/ASIC accelerators have been proposed [8, 9, 10], most DNN workloads are still running on the existing processors: microcontrollers, CPUs and Graphics Processing Units (GPUs). Therefore, it is of great importance to perform DNN computation efficiently on those processors.

Numerous techniques were introduced to reduce the parameters or the required computation of DNN models, trying to improve the DNN computation efficiency on the processors. They can be separated into two main categories: network compression which removes the internal redundancy of DNN models, and domain transformation which converts the DNN computation into different domains.

These conventional techniques have a common problem which is the neglect of the characteristics of the underlying processors: single instruction multiple data (SIMD) support, branch prediction, memory hierarchy, etc. Due to this issue, counter-intuitively, many existing approaches cannot improve or even hurt the computation efficiency. Besides, techniques from different categories may not be directly compatible with each other, which limits their adoption.

The focus of this thesis is to provide a hardware-software co-design solution to enable efficient DNN computation on the processors. On the one hand, the conventional computation reduction approaches are customized or combined based on the characteristics of the underlying hardware to fully utilize the reduction in parameters or computation. On the other hand, the processor architectures need to be augmented to provide better support for the customized DNN computation.

1.1 DNN Computation Reduction

There are two main categories of techniques for reducing DNN computation: network compression and domain transformation. Along with the high computation cost, DNN models have a large amount of internal redundancy. This redundancy can be removed by network compression, eliminating unnecessary parameters, or lowering the precision of model parameters. It can help reduce both the model parameters and the required computation. For domain transformation, the convolution operations in the DNN computation are transformed from the spatial domain into different domains, and get replaced with fewer calculations.

1.1.1 Network Compression

One typical method for removing DNN redundancy is pruning. It deletes unimportant parameters from DNN models. In a conventional DNN model, a large portion of the parameters, e.g., those close to zero, have negligible effects on the outputs. Pruning can remove these parameters, and the pruned network is retrained to regain the original accuracy. However, removing individual parameters incurs sparsity into the network, which causes irregularity in the control flow and the memory

access pattern. On the existing processors, the incurred irregularity requires extra instructions in the computations. It also leads to inefficient utilization of the architecture support, e.g., SIMD units, designed for accelerating regular computation patterns. In this case, conventional pruning techniques may even degrade the computation performance. Besides, encoding the sparse structure, including the position of the non-zero weights, requires extra parameters. It costs much more storage space for the pruned DNN models.

Besides pruning, reducing the precision of the model parameters and intermediate results helps reduce the DNN redundancy. There are many research attempts to use sub-byte (precision$\leq$8-bit) values, e.g., binary (1-bit) values, to represent weights or intermediate results [11, 12, 13]. Although the number of required calculations will not decrease, the bit width of each calculation and, therefore, the effective computation is reduced. Using sub-byte values can also help reduce the memory footprint of DNN computation. However, the memory hierarchy of existing processors is commonly byte-addressable. In this case, the parameters and intermediate results with precisions lower than 8 bits (one byte) cannot be directly fetched from the memory and fed into the execution units. Instead, they need to be unpacked from the sub-byte data formats, and the related computation can then be performed. This unpacking process requires extra computation and will lead to an increase in the computation latency.

1.1.2 Domain Transformation

For the inference computation of DNN models, most of the execution time is spent on convolutional layers. Convolutional layers perform the convolution operations between the weight parameters and the input data to generate the corresponding outputs.

For domain transformation techniques, using Winograd convolution [14] as an example, the weights and the inputs of the convolution operations are first transformed into the Winograd domain. Then the element-wise multiplications between the Winograd-domain weights and inputs will be performed to generate the Winograd-domain outputs. As the last step, the Winograd-domain outputs are transformed back into the original spatial domain as the final outputs. In this

case, the convolution operations in the original spatial domain are replaced with the element-wise multiplications in the Winograd domain, which helps reduce the required computation.

However, transforming weights into the target domain, e.g., the Winograd domain, dramatically increases the number of weight parameters and therefore the memory footprint of the computation. Considering the cache hierarchy in the processors benefits much for the computation with a lower memory footprint, the memory footprint increase caused by domain transformation may eliminate the benefit of the computation reduction.

An intuitive solution for the memory footprint increase is to apply the network compression techniques, but network compression and domain transformation are not directly compatible with each other. For instance, the pruned network loses its sparsity through the domain transformation, and the required precisions may become higher for the domain transformed computation than for conventional sub-byte networks.

1.2 Contributions

This thesis addresses the limitations of the conventional DNN computation reduction techniques by customizing or combining these algorithms and, meanwhile, augmenting the target processors. Chapter 3 describes a new pruning algorithm which customizes pruning techniques based on the characteristics of the underlying hardware parallelism. In Chapter 4, I propose a sub-byte convolution algorithm with a series of instruction set architecture (ISA) extensions to accelerate DNN computation on microcontrollers. As the last part, in Chapter 5, I introduce a new pruning method combining the pruning techniques with the domain transformation algorithms to speed up the DNN computation further.

1.2.1 Hardware Parallelism Aware Pruning

Different processors provide support for different levels of hardware parallelism, e.g., data-level parallelism. If the sparse structure of the pruned DNN models cannot utilize the underlying hard-

ware support efficiently, pruning may cause performance degradation compared with the original dense structure. To overcome this problem, hardware parallelism aware pruning customizes the pruning granularity based on the characteristics of target processors. Instead of removing individual weights, this pruning method deletes weights by groups. The group structure is designed to keep a fine-grain regularity in the pruned DNN models so that the underlying hardware support can be fully utilized. Also, keeping a fine-grain regularity can help reduce the overhead of encoding the irregular sparse structure. Across the microcontroller, CPU, and GPU, mean speedups of 3.54x, 2.61x, and 1.25x are achieved while the model sizes are reduced by 88%, 82%, and 53%.

1.2.2 Sub-byte DNN on IoT Microcontrollers

Microcontrollers used in internet-of-things (IoT) nodes usually have extremely limited computation and storage resource. Therefore, it is difficult to run large DNN models on microcontrollers due to their high computation and storage cost. Using sub-byte values for the parameters and intermediate results can help reduce the memory footprint, but introduces unacceptable computation overhead for unpacking the sub-byte formats. This thesis proposes a new convolution algorithm to replace the MAC operations with bitwise logic operations, which helps avoid the weight unpacking and dramatically reduces the computation overhead of sub-byte DNNs. Based on this convolution algorithm, a series of extensions to the existing Arm v7-M ISA is introduced, which further accelerates the computation and, at the same time, keep the generality of microcontrollers. Compared to the 16-bit baseline, using 4-bit inputs and ternary weights, the computation performance can be improved by 3.37x, and the energy efficiency is increased by 3.42x.

1.2.3 Winograd-Domain Pruning

Winograd convolution reduces the required computation of DNN models but also dramatically increases memory access. Pruning is a potential solution to reduce the memory footprint, but commonly applied in the original spatial domain. Performing Winograd convolution requires transforming model parameters into the Winograd domain, and this transformation will full fill

the sparsity incurred by pruning. Therefore, this thesis proposes a two-step pruning technique, spatial-Winograd pruning, to achieve a higher Winograd-domain sparsity. In the first step, the spatial-domain weights are pruned in a structured way, transferring more spatial-domain sparsity into the Winograd domain. As the second step, the Winograd-domain weights are directly pruned to increase the sparsity further. This technique can help combine the performance benefit from both pruning and Winograd convolution. Compared with the conventional spatial convolution, spatial-Winograd pruning reduces the required computation by $8.61\times$ and achieves a $2.08\times$ speedup on average across the tested models.

CHAPTER 2

Background and Related Work

2.1 Deep Neural Networks

The fundamental building block of all DNNs is the neuron. A neuron combines its weighted inputs and the bias value to determine the output activation via the activation function. DNNs integrate convolutional (CONV) layers and fully-connected (FC) layers into an end-to-end multi-layer network [6]. Figure 2.1 shows a typical DNN used for image classification. It consists of two CONV layers followed by two FC layers. In FC layers, all input values are connected to every neuron. For CONV layers, as shown in Figure 2.1, they consist of a stack of 2D matrices named feature maps. The convolution operations are performed between the input feature maps and the weights to generate the outputs.

FC layers perform matrix-vector multiplication, and CONV layers perform matrix-matrix multiplication. Weights of each layer can be grouped into a weight matrix. For FC layers, the input values are stored in a 1D vector (input vector). Then, the weight matrix can be multiplied with the input vector to generate the output which is also a 1D vector.

For CONV layers, the input feature maps will be rearranged into the input matrix $\boldsymbol{I}$ through the image-to-column (im2col) function. Then the weight matrix $\boldsymbol{W}$ is multiplied with the input matrix as

$$\boldsymbol{O} = f(\boldsymbol{W} \cdot \boldsymbol{I}) \tag{2.1}$$

to generate the output matrix $\boldsymbol{O}$. $f()$ is the element-wise activation function. The output matrix

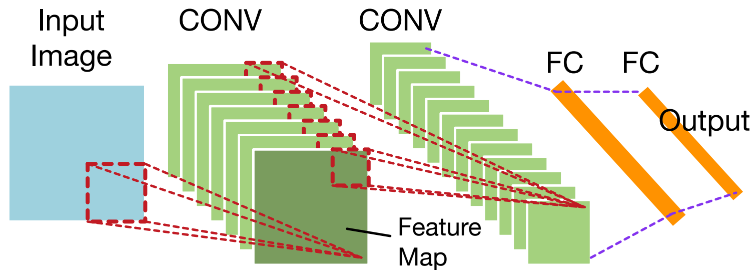


Figure 2.1: Deep Neural Networks (DNNs) structure. DNNs integrate convolutional (CONV) layers and fully-connected (FC) layers into an end-to-end structure.

then becomes the input feature maps to the subsequent layer.

2.2 Network Compression

DNN models have significant redundancy [15] which leads to redundant computation and storage. Various techniques have been proposed to remove this redundancy.

2.2.1 Pruning

Weight pruning has been used to remove redundant parameters and connections inside a network. LeCun et al. [16] calculate the saliencies of the weights based on the Hessian matrix of the loss function and remove low-saliency weights. Hassibi et al. [17] employ Lagrange Multiplier to find the weight with low saliencies. However, the calculation of the Hessian matrix of the loss function needs unacceptable computation. Han et al. [18] directly remove low-value weights and retrain the network to keep the original network performance. Guo et al. [19] use mask matrices to incorporate connection splicing into the entire pruning process, which can help avoid incorrect pruning.

For node pruning, He et al. [20] introduce a DNN reshape method which measures the importance of each node with a certain importance function. It works for fully-connected networks without convolutional layers. Miconi et al. [21] propose a method to make network structure differentiable. It imposes a penalty on the outgoing weights from each neuron to determine the importance of each neuron. However, it only works for simple recurrent networks.

2.2.2 Low-Precision DNNs

Another typical method for network compression is to reduce the numerical precision of the parameters and the intermediate results. Gupta et al. [22] demonstrate that DNN can be trained with the 16-bit fixed-point representation using stochastic rounding. Vanhoucke et al. [23] use 8-bit quantization for activations and weights to accelerate computation.

For compressing weight values, Courbariaux et al. [24] and Li et al. [25] propose to use binary and ternary weights for DNN models, respectively. Courbariaux et al. [26] and Rastegari et al. [11] proposed to use binary inputs with binary weights to further compress the input values and simplify the computation. Also, more works [27, 28, 13] were proposed to improve the accuracy of low-precision DNNs.

2.3 Hardware-Based DNN Acceleration

For DNN acceleration, various ASIC accelerators [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 9, 39], in-memory processing techniques [40, 41], FPGA solutions[42, 43, 44] and system designs[45, 46, 47] are proposed.

As examples, DianNao [48] introduces a customized machine learning architecture accelerating DNN/CNN computation by tiling both input and output to improve data reuse opportunities. Minerva [39] reduces the overall power consumption by aggressive data type optimization, selective operation pruning, and SRAM voltage reduction. Han et al. [9] propose EIE, a dedicated hardware accelerator which utilizes the sparsity in compressed DNNs. Shafiee et al. [37] use the memristor crossbar to perform effective MAC operations. In-situ AI [49] provides support for both inference and diagnose tasks with IoT nodes built on mobile GPU and FPGA to deploy DNNs in IoT system.

In comparison, this thesis focuses on DNN computation on existing processors, and the proposed augments still maintain the hardware generality.

2.4 Bit-Level Computation Optimization

Bit-level optimization for DNN computation is widely adopted in ASIC and FPGA accelerator designs. Stripes [50] proposes to use bit-serial compute units, which enables using different numerical precisions across different layers. Based on this, Pragmatic [30] avoids processing zero bits to further improve the energy efficiency. Also, multiple other ASIC [32, 36] and FPGA [43, 44] designs are proposed to provide low-precision support.

However, on the existing processors, performing bit-level computation becomes difficult since the memory hierarchy is not bit-addressable, and decoding low-precision formats incurs extra computation overhead. Bitslice Vectors [51] proposes to use the software bitslice format and bitwise instructions to build arithmetic operators. This thesis adopts the same insight but provides a different computation algorithm which is optimized for low-precision DNNs on the low-power IoT microcontrollers.

2.5 Winograd Convolution

As typical domain transformation algorithms, Winograd convolution [14] and FFT convolution [52] transform the computation into the Winograd and frequency domain, respectively. The convolution operations can then be replaced by element-wise multiplications. For the typical convolution kernel size of 3×3, Winograd convolution can achieve more than twofold speedup over highly optimized spatial convolution algorithms, and typically requires fewer flops than FFT-based approaches [53]. Therefore, this thesis focuses on Winograd convolution.

To combine pruning with Winograd convolution, Liu et al. [54] propose to directly mask out Winograd-domain weights and use backpropagation to train the spatial-domain weights. However, directly setting Winograd-domain weights to zero will cause an inconsistency between the spatial domain and the Winograd domain. To address the inconsistency between the spatial and Winograd domain, Li et al. [53] propose the sparse Winograd convolution. Weight values are stored in the Winograd domain instead of the spatial domain. Both pruning and retraining are applied directly

to Winograd-domain weights. Based on sparse Winograd convolution, Liu et al. [55] introduce the Winograd-ReLU pruning. It moves the ReLU function from the spatial domain into the Winograd domain, which incurs sparsity into the Winograd-domain inputs and helps further reduce the required computation.

CHAPTER 3

Hardware Parallelism Aware Pruning

3.1 Introduction

Weight pruning compresses DNN models by removing their internal redundancy. One such method, Deep Compression [56, 18], reduces the number of weights in AlexNet and VGG-16 by 9x and 13x, respectively. The compressed networks achieve 3-4x layerwise speedup on both CPUs and GPUs.

To investigate weight pruning more deeply, we reimplemented Deep Compression and measured the performance of five popular networks across three hardware platforms (ARM Cortex-M4 Microcontroller, Intel Core i7-6700 CPU, NVIDIA GTX Titan X GPU). We discovered surprising results. For 8/15 configurations, the performance of the networks after weight pruning was actually *worse* than before pruning. Pruning hurts performance despite removing an average of 80% of the weights. As an example, the execution time of AlexNet on the CPU increases by 25% even though 89% of its weights are removed. For the remaining configurations, a performance gain was observed with weight pruning, but that performance speedup lagged far behind the actual reduction in multiply-accumulate (MAC) operations. For instance, weight pruning can remove 76% of the MAC operations in LeNet-5 [57], but the execution time on the microcontroller is only reduced by 16%. For the tested hardware, a performance loss was consistently observed on the GPU while modest performance gains were observed on the microcontroller. The CPU yielded mixed results with some networks achieving a gain and others a loss.

To understand these counter-intuitive results, we need to examine weight pruning in more depth as well as the structure of the networks and the interplay of both with the underlying hardware. DNNs consists of two types of layers: fully-connected layers and convolutional layers. They perform matrix-vector and matrix-matrix multiplication, respectively. Weight pruning techniques [56, 16, 17] measure the importance of each weight and remove those deemed unimportant, resulting in both memory storage and computation reductions. After weight pruning, redundant weights and related MAC operations are removed. The matrix computation in the pruned networks becomes sparse, thus the remaining weights are stored in a sparse matrix format.

The sparsity of pruned networks often leads to a performance decrease in DNN computation. Sparse weight matrices lose the regular structure of dense matrices, and sparse matrix multiplication needs extra computation to decode the sparse format. The performance impact of weight pruning is heavily intertwined with the underlying hardware. On microcontrollers, weight pruning consistently improves DNN performance. The simple architecture of microcontrollers cannot hide the memory access latency. The reduction in model size can, therefore, make up for the computation inefficiencies of sparse matrix multiplication. However, the reduction in execution time is still much lower than the computation reduction. For GPUs, weight pruning consistently loses performance. The sparse matrix computation cannot make optimal usage of the supported hardware, e.g. memory coalescing. Also, dense matrix optimizations, like matrix tiling, are less effective.

For CPUs, the effect of weight pruning varies for different networks and depends on the computation breakdown between fully-connected and convolutional layers. For fully-connected layers, weight pruning can improve performance because the total memory footprint is critical to matrix-vector multiplication. But for convolutional layers that perform matrix-matrix multiplication, the matrix data will be reused multiple times, and there is limited benefit from the memory footprint reduction. Therefore, the inefficiencies inherent in the sparse matrix format will hurt the performance of convolutional layers on CPUs. In addition to the performance decrease, another challenge for weight pruning is that a significant amount of data is necessary to record the sparse matrix structure. Each nonzero weight needs one extra column index to record its position. This

extra overhead reduces the impact of weight pruning across all hardware platforms.

To address these challenges, we propose *Scalpel* to customize DNN pruning to the underlying data-parallel hardware structure. Scalpel consists of two methods: *SIMD-aware weight pruning* and *node pruning*. It creates a pruned network that can be efficiently executed on the target hardware platform. For hardware with low parallelism like microcontrollers, SIMD-aware weight pruning removes redundant weights but forces the remaining weights to be in groups. All groups are sized to the SIMD width, thereby, improving performance by ensuring the SIMD units are fully utilized. It also decreases the remaining model size since weights in the same group can share the same column index. For hardware with high parallelism like GPUs, node pruning removes redundant nodes in DNNs by using mask layers to dynamically find out and remove unimportant nodes. Removing nodes maintains the dense format of weight matrices, so the computation will not suffer from the sparsity caused by traditional weight pruning methods. For hardware platforms with moderate parallelism like desktop CPUs, SIMD-aware weight pruning and node pruning can be synergistically combined and applied together to reduce both execution time and model sizes. The pruned DNN models generated by Scalpel do not suffer a loss in prediction accuracy compared with the original models.

This pruning technique makes the following contributions:

- We demonstrate that DNN weight pruning is not a panacea, but rather its impact is closely coupled to both the structure of the network (fully connected vs. convolutional layers) as well as the data-parallel structure of the target hardware. The blind application of traditional weight pruning often results in performance loss, hence a closer examination of this topic is warranted.

- We propose Scalpel that creates a pruned network that is customized to the hardware platform that it will execute. Scalpel provides a method to improve the computation speed and reduce the model sizes of DNNs across processors ranging from microcontrollers to GPUs with no accuracy loss.

- SIMD-aware weight pruning is introduced as a refinement to traditional weight pruning. It puts contiguous weights into groups of size equal to the SIMD width. Extra data for recording the sparse matrix format is reduced along with providing higher utilization of SIMD units.

- A new method, node pruning, is proposed to compress the DNN model by removing redundant nodes in each layer. It does not break the regular structure of DNNs, thus avoiding the overheads of sparsity caused by existing pruning techniques.

- We compare the performance of Scalpel to prior DNN pruning techniques across three hardware platforms: microcontroller, CPU and GPU. Across these hardware platforms, Scalpel achieves geometric mean performance speedups of 3.54x, 2.61x, and 1.25x while reducing the model sizes by 88%, 82%, and 53%. In comparison, traditional weight pruning achieves mean speedups of 1.90x, 1.06x, 0.41x across the three platforms.

3.2 Background and Motivation

Deep neural networks (DNNs) often include a significant amount of redundant weights. Weight pruning with retraining can safely remove those redundant weights with no reduction in the prediction accuracy.

3.2.1 DNN Weight Pruning

In DNN models, fully-connected(FC) layers perform matrix-vector multiplication, and convolutional(CONV) layers perform matrix-matrix multiplication.

As an example, FC layers perform the computation

$$\boldsymbol{y} = f(\boldsymbol{W} \cdot \boldsymbol{x}) \tag{3.1}$$

where $\boldsymbol{y}$ is the output activation vector, $\boldsymbol{W}$ is the weight matrix and $\boldsymbol{x}$ is the input vector. f is the

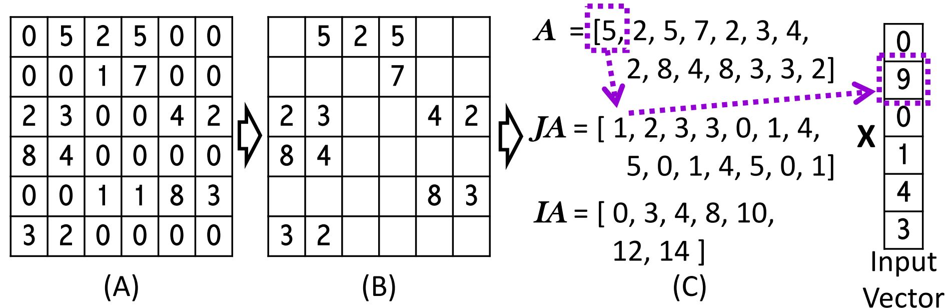


Figure 3.1: (A) Dense weight matrix; (B) Sparse weight matrix; (C) Compressed Sparse Rows (CSR) format for sparse matrices.

element-wise non-linear activation function. Bias values can be appended to weights matrix with corresponding input values equal to 1 and, therefore, are neglected.

As shown in Figure 3.1, weight pruning removes redundant weights and the dense weight matrix $\boldsymbol{W}$ (Figure 3.1(A)) is converted into a sparse matrix $\boldsymbol{W}^{sparse}$ (Figure 3.1(B)). Every output element $y_i \in \boldsymbol{y}$ should be calculated as

$$y_i = f(\sum_{W_{i,j}^{sparse} \neq 0, j \in [0, n-1]} W_{i,j}^{sparse} x_j) \tag{3.2}$$

where multiple-accumulate operations with zero weight have been removed. n is the number of columns in $\boldsymbol{W}$.

After weight pruning, the input vector or input matrix is still dense for each layer. The FC and CONV layers need to perform sparse matrix-vector and sparse matrix-matrix multiplication, respectively.

Deep Compression [56, 18] provides a typical weight pruning technique. Weights with absolute values lower than the thresholds are removed, and the remaining network is retrained. The steps of removing weights and retraining are iteratively applied to generate the pruned DNN model.

3.2.2 Challenges

Weight pruning can dramatically decrease DNN model sizes and the number of MAC operations. However, several challenges exist for traditional DNN pruning techniques.

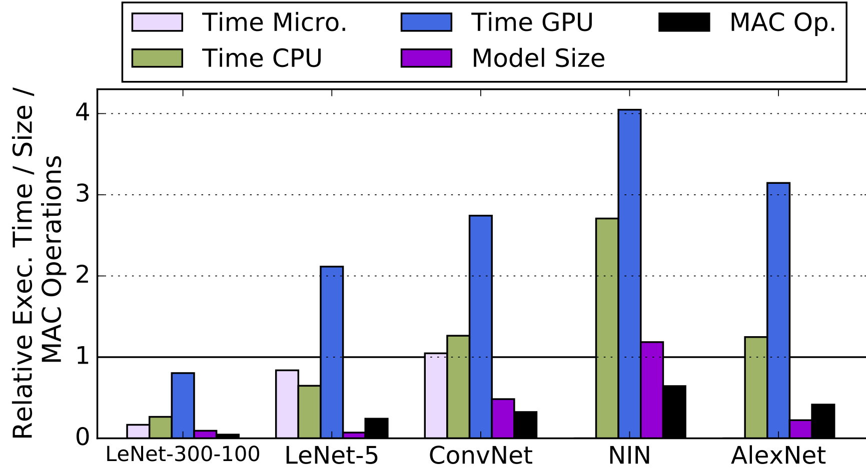


Figure 3.2: Relative execution time (Time Microcontroller/ CPU/ GPU), model sizes, and MAC operations of the networks pruned with the traditional pruning technique in Deep Compression. NIN and AlexNet are not tested on the microcontroller due to its limited storage size.

The first challenge is that the sparse weight matrix spends too much extra data to record the sparse matrix format. For example, as shown in Figure 3.1 (C), the compressed sparse rows (CSR) format use three 1-D arrays (**A**, **IA**, **JA**) to store a $m \times n$ matrix $\boldsymbol{W}$. **A** holds all the nonzero values. **IA** records the index into **A** of the first nonzero element in each row of $\boldsymbol{W}$. **JA** stores the column indexes of the nonzero elements. Since the index array **JA** has the same size with the data array **A**, more than half of the data in the CSR format is used to store the sparse matrix format.

The second challenge is that weight pruning can hurt the DNN computation performance. Figure 3.2 shows the relative execution time, model sizes and MAC operations of network models pruned by the weight pruning method in Deep Compression with respect to the original DNN models. The first three bars show the relative execution time on the microcontroller, CPU and GPU. As shown in the figure, the relative execution time is much higher than the relative model sizes and MAC operations. Weight pruning hurts the performance of LeNet-5 (on GPU), ConvNet, NIN and AlexNet, which causes an execution time increase for these networks.

We generate a breakdown of the execution time for the dense and the sparse DNN models on the CPU, as shown in Figure 3.3. The expected execution time of the sparse network (Expected) shows the relative MAC operations remaining, which ignores the overheads of the sparse storage

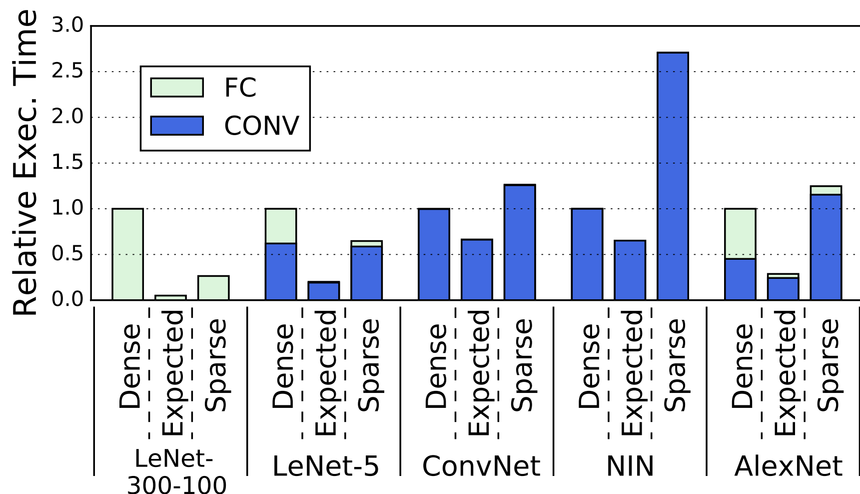


Figure 3.3: Execution time breakdown of the original networks (Dense) and the pruned networks (Sparse). The expected execution time of pruned networks (Expected) shows the relative MAC operations remaining. The second CONV layer (*conv2*) in AlexNet is tested.

implementation. Execution times are presented relative to the unpruned baseline (Dense). The real execution time of the FC layers is significantly reduced by traditional weight pruning, which is similar to the results shown in the previous work [56]. For both FC and CONV layers, there is a large gap between the real execution time of the pruned networks and the expected values. The performance gains are lagging significantly behind the reduction in MAC operations. Lastly, weight pruning is ineffective for CONV layers, leading to substantial increases in execution time for ConvNet, NIN and AlextNet.

The gap between real and expected execution times occurs because the sparse weight matrices lose the regular structure of their dense counterparts. For example, all rows have the same size in dense matrices, which does not hold for sparse matrices. Assume the sparse weight matrix is stored in CSR format. To perform sparse matrix-vector multiplication, as the dashed lines in Figure 3.1 show, for each nonzero weight stored in array **A**, we first need to load the corresponding column index from **JA** and use that to load the input value from the input vector. This indexing of input values requires additional computation and memory accesses. Worse yet, since sparse matrices lose the regular structure, many optimizations applicable to dense matrices, e.g. matrix tiling, cannot be applied. For CPU computation, FC layers do not suffer as much from these reasons

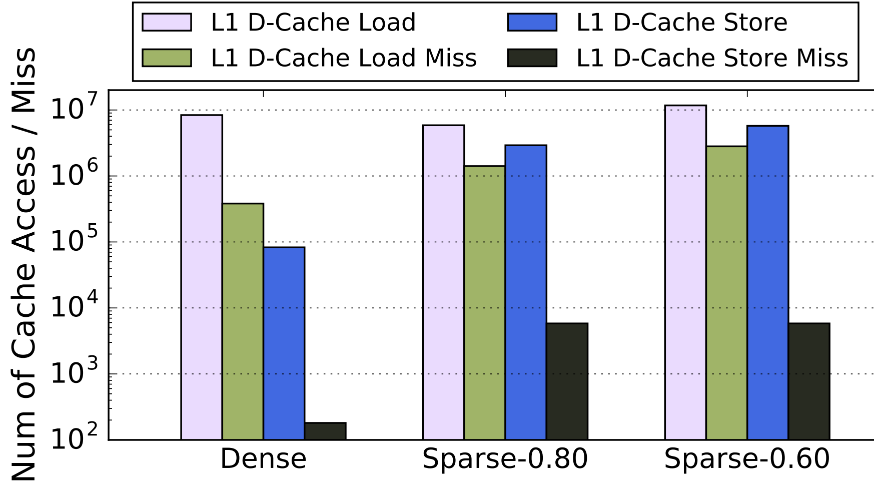


Figure 3.4: Numbers of cache access and cache miss in the computation of the second CONV layer (*conv2*) in AlexNet on Intel Core i7-6700 CPU. The original dense layer (Dense), the sparse layer with 80% (Sparse-0.80) and 60% (Sparse-0.60) of weights removed are tested. The matrices are randomly generated.

because they have a much lower computation/memory access ratio compared with CONV layers.

To understand the impact of pruning on memory access behavior in more detail, we consider a representative layer from AlexNet, the second layer (*conv2*). This layer performs a matrix-matrix multiplication because it is a CONV layer. Figure 3.4 shows the numbers of cache accesses and cache misses for this layer broken down by loads and stores. We measure three cases: original dense layer (Dense), sparse layer with 80% (Sparse-0.80) and 60% (Sparse-0.60) of the weights removed. We profile the computation with Callgrind. Sparse-0.60 represents the actual pruning rate for this layer, while Sparse-0.80 represents the pruning rate necessary to reach the break-even performance point. As shown in the figure, weight pruning leads to large increases in L1 D-Cache load misses, stores, and store misses due to the sparse representation. With 60% pruning, these overheads are not counter-balanced by the computation reduction, hence a net performance loss is observed. At 80% pruning, the combination of modestly lower memory access overheads and higher reductions in computations results in break-even performance. Pruning rates of more than 80% are necessary to overcome the memory access overhead for this layer and achieve a net performance gain.

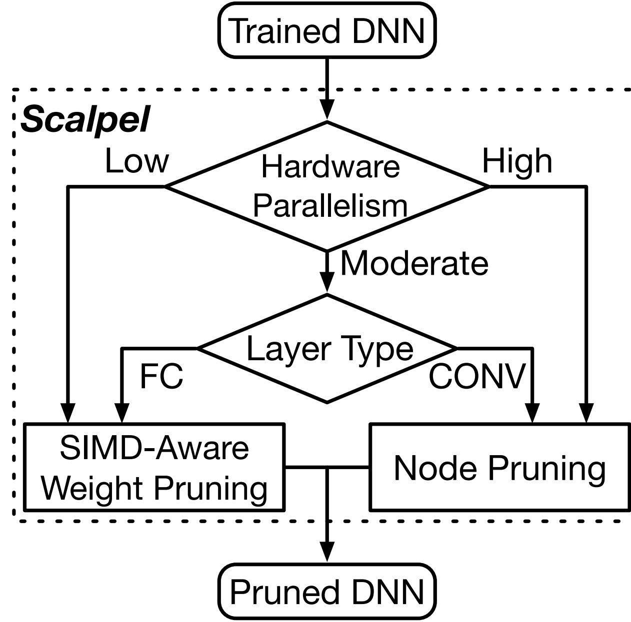


Figure 3.5: Overview of Scalpel.

3.3 Scalpel

To address the challenges from traditional pruning techniques, we propose Scalpel in this chapter. It consists of two methods: SIMD-aware weight pruning and node pruning. Scalpel customizes the DNN pruning for different hardware platforms based on their parallelism.

3.3.1 Overview

Figure 3.5 shows the overview of Scalpel. The first step of Scalpel is profiling and determining the parallelism level of the hardware platform. All general-purpose hardware platforms are divided into three categories based on their internal parallelism: low parallelism, moderate parallelism, and high parallelism.

For low-parallelism hardware, SIMD-aware weight pruning is applied. It prunes weights in groups and forces the remaining weights to be in aligned groups. All groups have the same size as the SIMD width and the weights in the same group share the same column index, reducing the overhead of the sparse format.

For high-parallelism hardware, node pruning is applied. It removes the DNN redundancy by

Table 3.1: Hardware platforms with different parallelism.

	Parallelism		
	Low	Moderate	High
Example	Micro-controller	CPU	GPU
Memory Hierarchy	No cache	Deep cache hierarchy	High bandwidth / long latency
Memory Size	∼100KB SRAM	∼8MB SRAM	2-12GB DRAM

removing redundant nodes instead of redundant weights. Removing nodes does not break the regular structure of dense weight matrices.

For hardware with moderate parallelism, we use a combination of SIMD-aware weight pruning and node pruning. SIMD-aware weight pruning is employed for fully-connected layers and node pruning is applied to convolutional layers.

By customizing pruning technique for different hardware platforms, Scalpel can reduce both the DNN model size and execution time across all the tested processors without accuracy loss for DNNs.

3.3.2 Hardware with Different Parallelism

We divide the existing processors into three main categories based on their parallelism level. Table 3.1 demonstrates the basic characteristics of these three categories.

Low Parallelism: Low-power processors like microcontrollers usually have a low parallelism. These processors contain in-order cores with a ”shallow” pipeline and have no cache. They also have very limited storage. SIMD units are employed to accelerate the computation, but their width is still limited. As an example, ARM Cortex-M4 has a 3-stage in-order pipeline and a 2-way SIMD unit. The test board we use only has 128KB SRAM and 512KB flash.

Moderate Parallelism: Out-of-order processors, for example Intel Core i7-6700 CPU, can be classified as moderate-parallelism hardware. In addition to SIMD units, the instruction-level parallelism (ILP) and the memory-level parallelism (MLP) are utilized to accelerate the computation. To

fully utilize ILP and MLP, moderate-parallelism processors require a deep cache hierarchy. They are usually connected to a large off-chip DRAM and, therefore, the storage size is sufficiently large to be considered unlimited.

High Parallelism: High-parallelism hardware like GPUs exploits thread-level parallelism (TLP) to further improve the parallelism to accelerate the computation. It focuses on high computation throughput and, therefore, DNN model sizes are not critical. Applications for high-parallelism hardware tend to be bandwidth sensitive, which requires a memory hierarchy with high bandwidth. Like moderate-parallelism hardware, the storage size can be considered unlimited.

Multiple inputs for the same DNN can be processed in a batch to reduce the memory access. The weight matrices can be loaded once and shared for the computation of multiple different inputs. Batch processing will increase the computation throughput but also the latency. On low-parallelism and moderate-parallelism hardware, we set the batch size to 1 because real-time applications need the DNN computation to be completed within a short latency. However, for high-parallelism hardware, the computation is throughput-driven, and DNN computation with a large batch size can still meet the latency requirement. In this case, we set the batch size to 50 for DNN computation on high-parallelism hardware. Han et al. [56] set batch size to 1 for GPU testing, which is actually unpractical.

3.3.3 SIMD-Aware Weight Pruning

For low-parallelism hardware, we apply SIMD-aware weight pruning to DNNs. The main steps of SIMD-aware weight pruning are shown in Figure 3.6. We use ARM Cortex-M4 microcontroller as an example of low-parallelism hardware. It has a 2-way SIMD unit for 16-bit fixed-point numbers.

The first step is weights grouping. All the weights are divided into aligned groups with the same size equal to the supported SIMD width. Figure 3.7 (A) shows a simple example of weights grouping. All groups have a size of 2 which is the SIMD width of Cortex-M4.

The second step is pruning weight groups. We calculate the Root-Mean-Square (RMS) of each group and use it to measure the importance of weight groups. Groups with RMS value below a

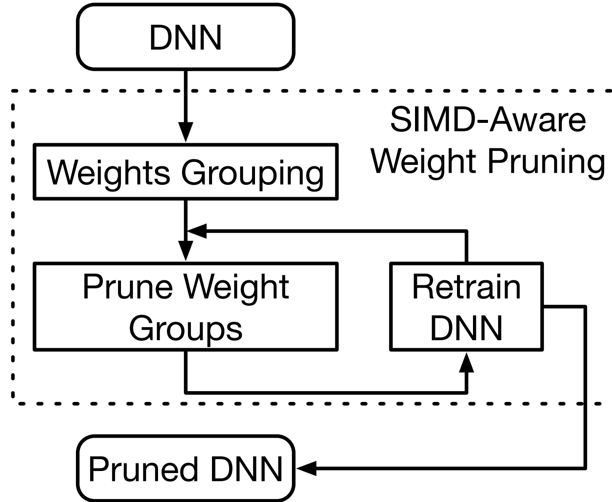


Figure 3.6: Main steps of SIMD-aware weight pruning.

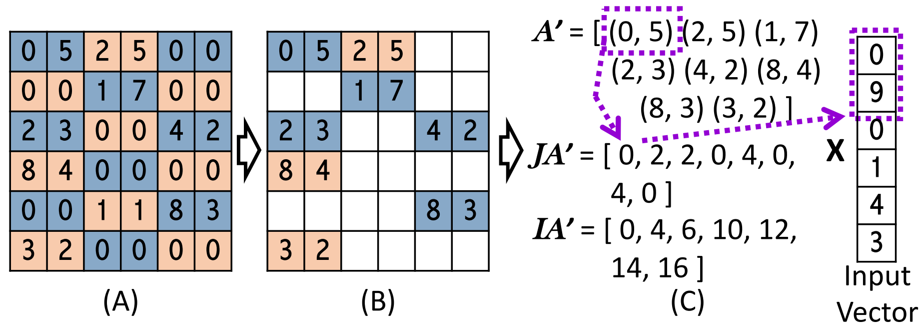


Figure 3.7: (A) Weights grouping; (B) Sparse weight matrix after pruning weight groups; (C) Modified CSR format for SIMD-aware weight pruning.

threshold are removed. Figure 3.7 (B) shows an example of the weight matrix after pruning weight groups. Then the pruned weight matrix will be retrained. The steps of pruning weight groups and retraining DNN are iteratively applied until the retrained DNN cannot keep the original accuracy.

SIMD-aware weight pruning works layer by layer. The execution time for each layer will be generated at the beginning, and the pruning process starts with the layer of the highest execution time. Every time after retraining the pruned DNN, the execution time of each layer will be updated. The new slowest layer will be pruned in the next iteration if the retrained DNN does not lose the original accuracy. We will not prune the layers which have low redundancy and cannot get a performance improvement through the SIMD-aware weight pruning.

During SIMD-aware weight pruning, we need to adjust the dropout ratio. Dropout is a widely

used technique for preventing overfitting [58]. During network training, dropout is implemented by keeping a neuron active with some probability $(1-d)$, or setting it to zero otherwise. This procedure can be regarded as sampling the neural network, and only the sampled part of the network needs to be updated through this iteration of training. For next iteration, the network should be re-sampled. SIMD-aware weight pruning will remove connections and reduce the DNN model capacity. Here we use the same technique with Han et al. [18] to adjust the dropout ratio. Assuming for the layer i, c_i is the number of the connections where c_i^o is for the original network and c_i^r is for the remaining network. We can adjust the dropout ratio as

$$d^r = d^o \sqrt{\frac{c_i^r}{c_i^o}} \tag{3.3}$$

where d^o is the original dropout ratio and d^r is the adjusted dropout ratio for retraining the remaining network after pruning weight groups.

After SIMD-aware weight pruning, we use a modified CSR format to record the sparse weight matrices. The modified CSR format, shown in Figure 3.7(C), includes three 1-D arrays: $\mathbf{A}'$, $\mathbf{IA}'$ and $\mathbf{JA}'$. $\mathbf{A}'$ stores all the nonzero weight groups with the original order. $\mathbf{IA}'$ records the index into $\mathbf{A}'$ of the first nonzero element in each row of $\boldsymbol{W}$. $\mathbf{JA}'$ stores the column index of each group. Only the column index of the first element in each group is recorded. In real computation, as the dashed arrows in Figure 3.7 show, we can load the nonzero weights in the array $\mathbf{A}'$ in groups. Then only one index from array $\mathbf{JA}'$ is used to load the corresponding input values. Since the input values are now also in contiguous addresses, they can be loaded with a single SIMD instruction. With the input values and weights loaded, the SIMD unit then performs the computation.

SIMD-aware weight pruning can reduce both model sizes and execution time of DNNs on low-parallelism hardware. Using one column index for each weight group can dramatically reduce the storage size of the indexes array $\mathbf{JA}'$ and the entire model size. For DNN computation, loading multiple contiguous input values with one SIMD instruction can reduce the computation instructions. The reduction in model size can also reduce the memory footprint. Therefore, the DNN

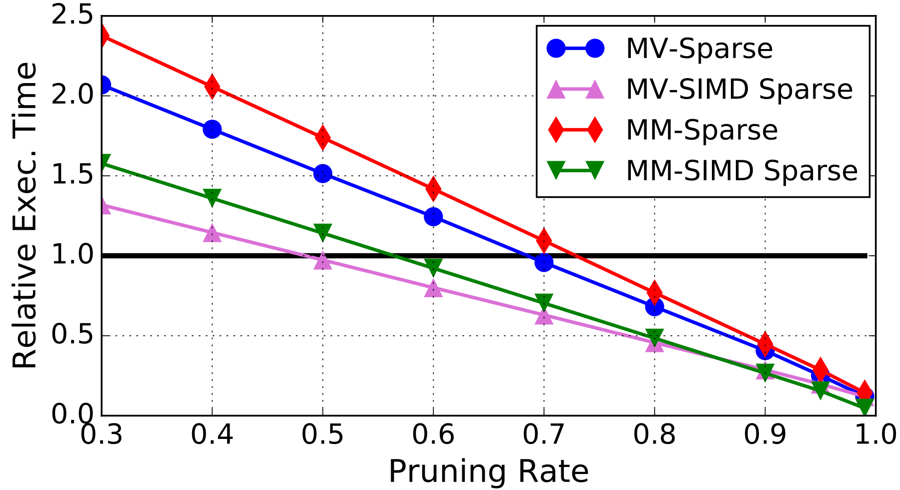


Figure 3.8: Relative execution time of sparse matrix multiplication on ARM Cortex-M4 with respect to the original dense matrix-vector or matrix-matrix multiplication. MV/MM-Sparse show the results for sparse Matrix-Vector (MV) and Matrix-Matrix (MM) multiplication, respectively. MV/MM-SIMD Sparse shows the corresponding performance with nonzero elements grouped and aligned as SIMD-aware weight pruning does. All matrices have the size of 100 x 100 and are randomly generated.

computation performance can be improved with SIMD-aware weight pruning.

Figure 3.8 shows the peak performance benefit from SIMD-aware weight pruning. The x-axis is the pruning rate which means how much weights we can remove from the weight matrix. For sparse matrix-vector (MV-Sparse) and matrix-matrix (MM-Sparse) multiplication, we need to remove more than 68% and 73% of the weight matrix to decrease the execution time, respectively. However, with SIMD-aware weight pruning (MV-SIMD Sparse / MM-SIMD Sparse), we only need to remove 48% and 56% of the weights.

3.3.4 Node Pruning

For hardware with high parallelism, node pruning is employed to remove the redundancy in DNNs. We use NVIDIA GTX Titan X GPU as an example of high-parallelism hardware.

Traditional weight pruning techniques will decrease the performance of all DNN layers on high-parallelism hardware. Figure 3.9 shows the relative execution time of sparse matrix-matrix multiplication on GPU against the pruning rate. The two matrices have the sizes of 4096x4096

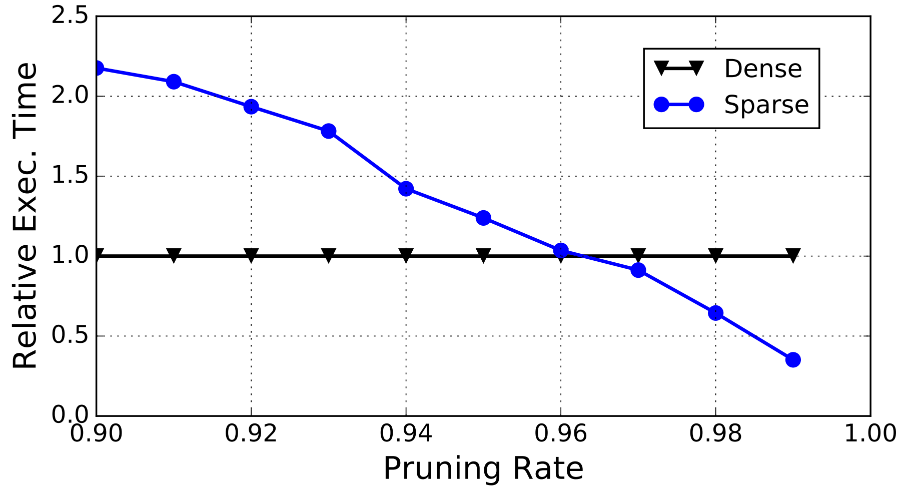


Figure 3.9: Relative execution time for sparse matrix-matrix multiplication (Sparse) on NVIDIA GTX Titan X with respect to original execution time (Dense). The matrices have the sizes of 4096x4096 and 4096x50.

and 4096x50. It estimates the performance of a fully-connected layer with 4096 inputs and 4096 outputs. The computation batch size is set to 50. As shown in the figure, more than 96% of the weights need to be removed to achieve a performance speedup. However, without a loss of accuracy, it is unpractical to remove that much weights from DNN layers. The matrix sparsity caused by weight pruning will hurt the computation performance of all layers.

To avoid this performance decrease, node pruning removes DNN redundancy by removing entire nodes instead of weights. It uses mask layers to dynamically find out unimportant nodes and block their outputs. The blocked nodes are removed after the training of mask layers. After removing all redundant nodes, mask layers are removed, and the network is retrained to get the pruned DNN model.

One neuron in the fully-connected layers or one feature map in the convolutional layers is considered as one node. Removing nodes in DNNs only shrinks the size of each layer but will not incur sparsity into the network. The remaining DNN model after node pruning keeps the regular dense DNN structure and will not suffer from the overheads of network sparsity.

Figure 3.10 shows the main steps of node pruning. First, we will add a mask layer for each DNN layer except the input and output layers. Figure 3.11 gives an example of a mask layer for

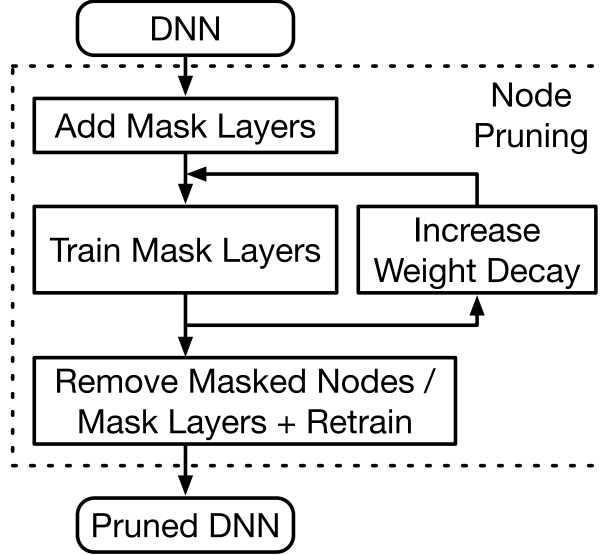


Figure 3.10: Main steps of node pruning.

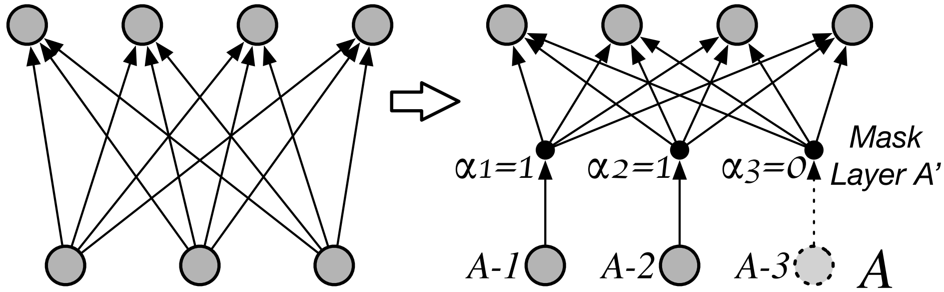


Figure 3.11: Mask layers. Node *A-3* with $\alpha_3 = 0$ can be removed. The whole mask layer A' will be removed after pruning all redundant nodes.

fully-connected layers. The output values of layer *A* need to go through the mask layer A' before propagated to the next layer. Each node in the mask layer holds two parameters α and β. α is a boolean variable ($\alpha \in \{0, 1\}$) and β is a floating-point number between 0 and 1 ($0 \leqslant \beta \leqslant 1$). Let array $\boldsymbol{y}$ and $\boldsymbol{y}'$ to be the output activation array of the original layer *A* and the mask layer A'. For $y'_i \in \boldsymbol{y}'$ and $y_i \in \boldsymbol{y}$, we have

$$y'_i = \alpha_i \cdot y_i \tag{3.4}$$

With α_i set to 0, the corresponding node can be considered as removed because the output y'_i is fixed to 0.

For convolutional layers, the activations in the feature map i will go through the same mask

node i and produce a masked feature map keeping the same size. Therefore, convolutional layers are pruned at a granularity of feature maps, and we are considering each feature map as one node.

The next step is to train the mask layers. For a single node i in the mask layer, α_i and β_i are both initialized to 1:

$$\alpha_i|_0 = 1, \quad \beta_i|_0 = 1.0 \tag{3.5}$$

In training iteration $k \geqslant 1$, α_i is calculated as

$$\alpha_i|_k = \begin{cases} 1, & t + \epsilon \leqslant \beta_i|_k \\ \alpha_i|_{k-1}, & t \leqslant \beta_i|_k < t + \epsilon \\ 0, & \beta_i|_k < t \end{cases} \tag{3.6}$$

where t $(0 < t < 1)$ is a threshold shared by all the mask layers. ϵ is a small value to make training more stable that α_i will not ”ping pong” between 0 and 1. β_i is updated through back-propagation and truncated to $[0, 1]$.

We use the L1 regularization to control the number of nodes got removed. Regularization is used to penalize the magnitude of parameters. For the mask layer, the penalty of each parameter β_i can be calculated as

$$R_i^{L1} = \lambda|\beta_i| = \lambda\beta_i \tag{3.7}$$

where λ is the weight decay (regularization strength). It forces β_i to be close to zero. If the corresponding node is not important, β_i will be decreased to be lower than threshold t and the node is temporarily removed. In case some removed nodes are found important, they will be retained through the DNN training. Since the threshold t is fixed in node pruning, increasing the weight decay λ will increase the penalty for β_i and decrease more parameters β to be lower than t. More nodes will, therefore, be removed.

Node pruning also needs to adjust the dropout ratio. Different from SIMD-aware weight pruning, the dropout ratio is dynamic updated during the step of training mask layers. In iteration k,

the dropout ratio is calculated as

$$d|_k = d|_0 \times \frac{n|_k}{n|_0} \tag{3.8}$$

where $d|_0$ is the initial dropout ratio, $n|_0$ is the initial number of nodes and $n|_k$ is the number of remaining nodes in iteration k.

In the step of training mask layers, the parameters of other layers are not fixed. Weights and biases in other layers are trained to fit the new DNN architecture with nodes removed.

After training mask layers, the weight decay of L1 regularization on the mask layers is increased. Then more nodes will be removed in the next iteration of training mask layers. The two steps of training mask layers and increasing weight decay will be iteratively applied until retraining cannot retain the DNN accuracy.

The last step of node pruning is removing masked nodes, removing mask layers, and retraining the network. All the nodes and feature maps with corresponding α value equal to zero are removed. For example, in Figure 3.11, the node *A-3* with $\alpha_3 = 0$ will be removed. The mask layers are then removed, and output activations of remaining nodes can be directly propagated to the next layer. The remaining network is retrained to get the final pruned DNN.

3.3.5 Combined Pruning

For hardware with moderate parallelism, for example Intel Core i7-6700 CPU, the SIMD-aware weight pruning and node pruning can be combined and applied to the same DNN. To limit the computation latency for real-time application, DNN computation usually has a small batch size on moderate-parallelism hardware. Here the batch size is fixed to 1, and a small batch size will give similar results.

DNNs can be split into two parts: fully-connected layers and convolutional layers. With a batch size of 1, fully-connected layers perform matrix-vector multiplication, and convolutional layers perform matrix-matrix multiplication. In this case, Scalpel applies SIMD-aware weight pruning to fully-connected layers and node pruning to convolutional layers.

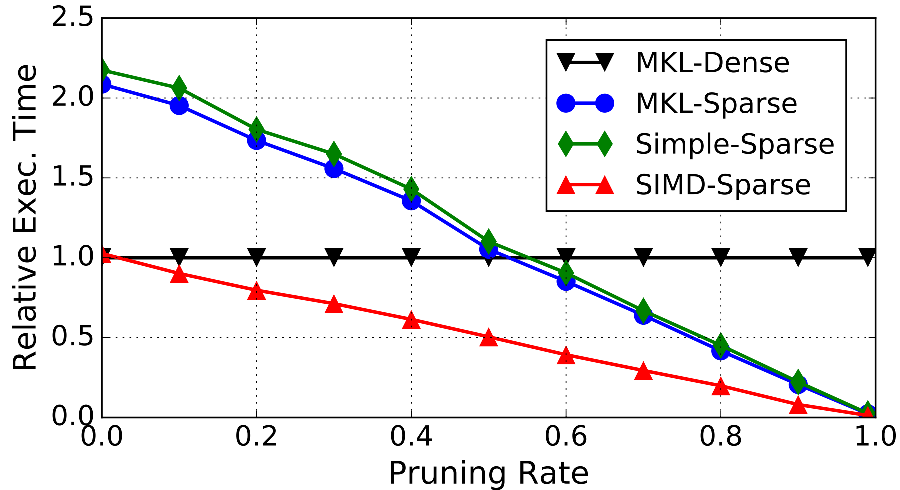


Figure 3.12: Relative execution time for sparse matrix-vector multiplication (FC layers) on Intel Core i7-6700. The matrix size is 4096 x 4096 and the vector size is 4096. MKL-Dense/Sparse show the results of dense and sparse weight matrix with the Intel MKL library.

Figure 3.12 and Figure 3.13 profiles the relative execution time for sparse matrix-vector and matrix-matrix multiplication on Intel Core i7-6700 CPU, respectively. MKL-Dense and MKL-Sparse show the results of dense and sparse weight matrix with the Intel MKL library. Simple-Sparse is our implementation of the sparse library. SIMD-Sparse shows the results with nonzero elements grouped and aligned as in SIMD-aware weight pruning.

Figure 3.12 shows the relative execution time reduction of matrix-vector multiplication with SIMD-aware weight pruning. Intel i7-6700 has an 8-way SIMD unit for 32-bit floating-point numbers. Therefore, SIMD-aware weight pruning removes weights in groups of 8. Comparing to the sparse matrix-vector multiplication with the Intel MKL library (MKL-Sparse), the SIMD-aware weight pruning (SIMD-Sparse) can dramatically improve the computation performance. The percentage of weights need to be removed for performance speedup decreases from 52% to 3%. Therefore, for fully-connected layers which perform matrix-vector multiplication, the SIMD-aware weight pruning can be applied to improve the performance and reduce model sizes.

SIMD-aware weight pruning can dramatically decrease the execution time on moderate-parallelism hardware for three reasons. First, by reducing the number of indexes, the memory footprint for the computation decreases. Second, weights are grouped and aligned with SIMD-aware weight prun-

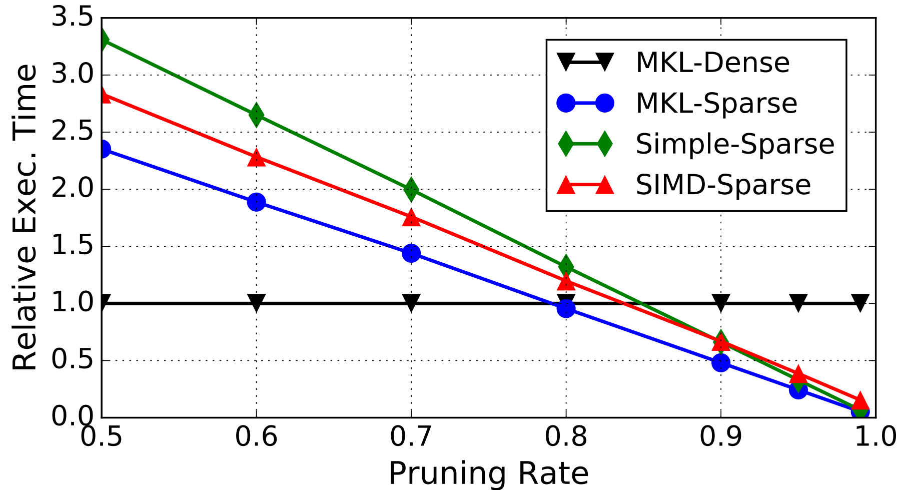


Figure 3.13: Relative execution time for sparse matrix-matrix multiplication (CONV layers) on Intel Core i7-6700. The weight matrix and input matrix have the size of 128 x 1200 and 1200 x 729, respectively.

ing, which increases the spatial locality of reading weights and corresponding inputs. Third, the number of computation instructions decreases since we only need to load one index for each group and the corresponding input values can be loaded with SIMD instructions.

Figure 3.13 is the corresponding relative execution time reduction for matrix-matrix multiplication. Compared to the sparse matrix-matrix multiplication with the Intel MKL library (MKL-Sparse), SIMD-aware weight pruning (SIMD-Sparse) cannot improve the execution performance. To achieve a performance speedup with traditional pruning technique or SIMD-aware weight pruning, at least 79% of the weights need to be removed. However, without accuracy loss, it is difficult to remove that much weights from convolutional layers for DNNs running on moderate-parallelism hardware. Convolutional layers have much less redundancy compared with full-connected layers since all weights need to be reused in the convolution operations. Therefore, weight pruning will hurt the computation performance and cannot be applied to convolutional layers. In this case, Scalpel applies node pruning to convolutional layers since it keeps the regular structure of weight matrices to avoid the performance decrease caused by weight pruning.

To apply both SIMD-aware weight pruning and node pruning to the same network, we will first use node pruning to remove redundant nodes in the convolutional layers. Then the convolutional

layers are fixed, and the SIMD-aware weight pruning is employed to prune redundant weights from the fully-connected layers.

3.4 Experiment Methodology

Hardware platforms We test Scalpel on three different processors: microcontrollers, CPU and GPU. They are representatives of hardware with low, moderate and high parallelism, respectively.

1) Microcontroller - low parallelism. We use ARM Cortex-M4 microcontroller which has a 3-stage in-order pipeline and 2-way SIMD units for 16-bit fixed-point numbers. The test board has 128KB SRAM and 512KB flash storage. To run the benchmarks, we use libraries directly from ARM for the dense matrix-vector/matrix multiplication. Libraries for sparse matrix-vector/matrix multiplication are written in-house.

2) CPU - moderate parallelism. We use Intel Core i7-6700 CPU, which is a Skylake class core. It supports 8-way SIMD instructions for 32-bit floating-point numbers. To run the benchmarks, we use MKL BLAS GEMV/ GEMM to implement the original dense model and the convolutional layers with node pruning. MKL Sparse BLAS CSRMV/ CSRMM is used for the sparse models generated by existing weight pruning techniques. Libraries of sparse matrix multiplication for SIMD-aware weight pruning are written in-house.

3) GPU - high parallelism. We use NVIDIA GTX Titan X which is a state-of-the-art GPU for deep learning and included in the NVIDIA Digits Deep Learning DevBox machine [59]. We use cuDNN to implement the original dense model and the convolutional layers with node pruning. cuBLAS GEMV/ GEMM is used for profiling the performance of the dense matrix-vector/matrix multiplication. cuSPARSE CSRMV/ CSRMM is used for the sparse model generated by existing weight pruning techniques.

Benchmarks. We compare the performance and the model size of five DNNs: LeNet-300-100 [57], LeNet-5 [57], ConvNet [60], Network-in-Network (NIN) [61] and AlexNet [62].

Table 3.2 shows the DNN benchmarks and their structures. NIN consists of 9 convolutional

Table 3.2: DNN benchmarks.

Networks	Num of Layers		Test Dataset	Error Rate
	CONV	FC		
LeNet-300-100	0	3	MNIST	1.50%
LeNet-5	2	2		0.68%
ConvNet	3	1	CIFAR-10	18.14%
NIN	9	0		10.43%
AlexNet	5	3	ImageNet	19.73% (top-5)

layers (CONV) and no fully-connected layers (FC). Recent DNN designs [6, 61, 63] contain no or only small-size fully-connected layers to reduce the model size. NIN is chosen as an example to test the performance of Scalpel on those networks. For NIN and AlexNet, we get the original models from Caffe Model Zoo [64]. Also, due to their large sizes, NIN and AlexNet are not tested on ARM Cortex-M4 microcontroller.

All pruned DNN models generated with Scalpel have *no accuracy loss* compared with the original DNNs. Caffe [64] is used for DNN training and pruning.

Experiment baselines. We use the original dense DNN models as the baseline. Performance speedups and model sizes shown in section 3.5 are relative values with respect to the corresponding original DNN models.

We compare Scalpel to two weight pruning techniques:

1) Traditional pruning. The pruning technique proposed by Han et al. [18] is implemented as the traditional pruning.

2) Optimized pruning. After traditional pruning, for layers with more than 50% of the weights remaining, we revert them back to the original dense format. We keep the nonzero weights and add zeros to the sparse weight matrix to convert it back into a dense matrix. This technique combined with traditional pruning is considered as the optimized pruning. Recording the layers with more 50% weights remaining into a sparse format will increase the model size. Also, not enough weights are removed to achieve a performance improvement for these layers. Therefore, optimized pruning can reduce both the model size and the execution time compared with the traditional pruning.

Table 3.3: Results overview.

Hardware	DNNs	Speedup	Relative Size
Micor-controller	LeNet-300-100	9.17x	6.93%
	LeNet-5	3.51x	6.72%
	ConvNet	1.38x	40.95%
CPU	LeNet-300-100	6.86x	7.08%
	LeNet-5	4.15x	5.20%
	ConvNet	1.58x	44.28%
	NIN	1.22x	81.16%
	AlexNet	2.20x	13.06%
GPU	LeNet-300-100	1.08x	66.83%
	LeNet-5	1.59x	11.67%
	ConvNet	1.14x	45.40%
	NIN	1.17x	81.16%
	AlexNet	1.35x	76.52%

3.5 Evaluation Results

Table 3.3 is the overview of the results. Scalpel achieves mean speedups of 3.54x, 2.61x, and 1.25x on the microcontroller, CPU and GPU, respectively. It also reduces the model sizes by 88%, 82%, and 53% on average.

3.5.1 Microcontroller - Low Parallelism

Scalpel is first tested on ARM Cortex-M4 microcontroller which is considered as the low-parallelism hardware. SIMD-aware weight pruning is applied to LeNet-300-100, LeNet-5, and ConvNet. Since Cortex-M4 has a 2-way SIMD unit, the size of weight groups is set to 2.

Figure 3.14 and Figure 3.15 shows the relative performance speedups and relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel.

For all tested networks, Scalpel achieves better performance and lower model sizes than traditional pruning and optimized pruning. The performance speedup can be up to 9.17x, and the model size is reduced by up to 93.28%.

Traditional pruning and optimized pruning have the same performance speedups and model sizes on LeNet-300-100. This is because LeNet-300-100 is a fully-connected network and all

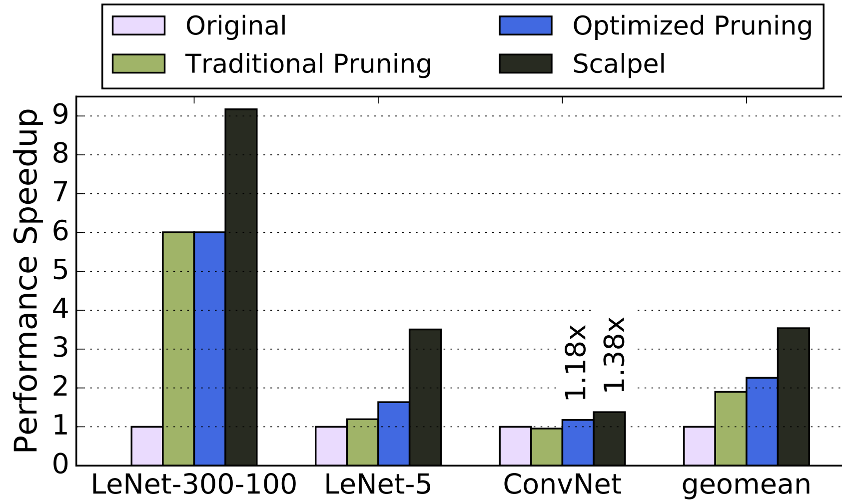


Figure 3.14: Relative performance speedups of the original models (original), traditional pruning, optimized pruning and Scalpel on ARM Cortex-M4 microcontroller. NIN and AlexNet are not tested due to the limited storage size of the microcontroller.

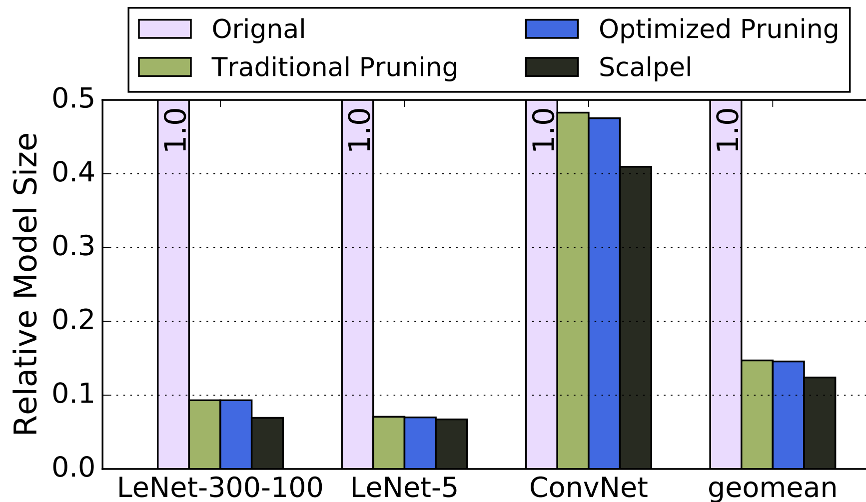


Figure 3.15: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for ARM Cortex-M4 microcontroller.

layers can have more than 50% of weights removed. But for ConvNet, the first convolutional layer has a relatively small size of 2400 weights. There is little redundancy inside this layer, and few weights can be removed. Therefore, it is not pruned in the optimized pruning, which helps improve the performance and reduce the model size.

How to measure the importance of each group is important for SIMD-aware weight pruning since weight groups with low importance will be removed in the step of pruning weight groups.

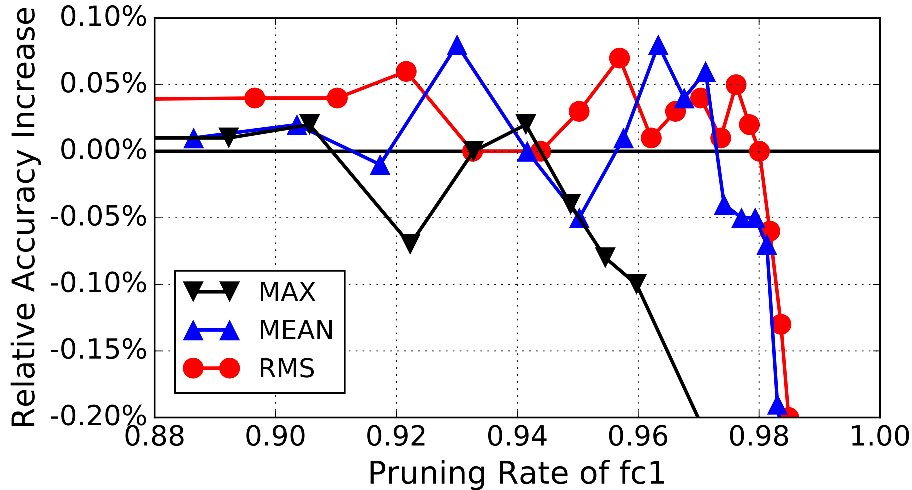


Figure 3.16: Relative accuracy against pruning rate with different metrics for importance measurement: maximum absolute value (MAX), mean absolute value (MEAN) and root-mean-square (RMS).

We test three different types of metrics for importance measurement: maximum absolute value (MAX), mean absolute value (MEAN) and root-mean-square (RMS).

In these metrics, MAX only takes the weight with highest absolute value into consideration, but all weights in the group should be considered to determine the group importance. For MEAN, large weight values and small weight values have the same impact on the group importance. However, in real DNN, larger weight tend to be more important and should have a larger impact. RMS considers all the weight values, and larger weight value will have a large impact on the group importance. Therefore, compared to the other two metrics, RMS is expected to help SIMD-aware weight pruning remove more redundant weights.

We apply SIMD-aware weight pruning on the first layer of LeNet-300-100 (*fc1*) with these three different metrics to test the real effect. The group size is chosen to be 2. Figure 3.16 shows the curves of relative accuracy against pruning rate for the three metrics. For each line, every dot is the result of one iteration for the SIMD-aware weight pruning since we are applying the two steps of pruning weight group and increasing weight decay iteratively. As expected, without accuracy loss, using RMS for the SIMD-aware weight pruning has the highest pruning rate of 98.0%.

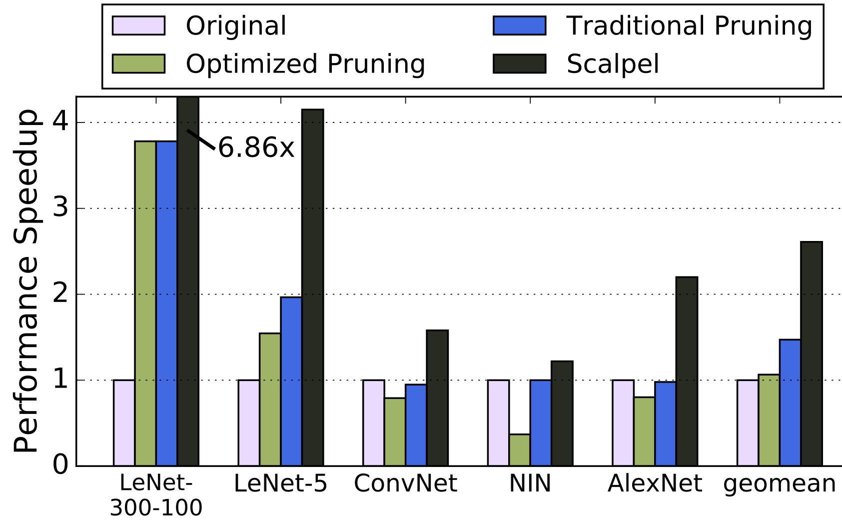


Figure 3.17: Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on Intel Core i7-6700 CPU.

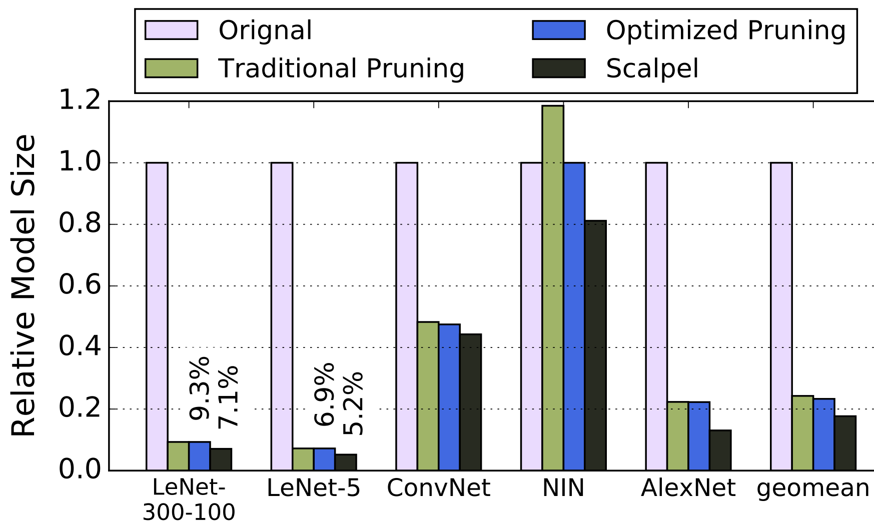


Figure 3.18: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for Intel Core i7-6700 CPU.

3.5.2 CPU - Moderate Parallelism

For Intel Core i7-6700 CPU, SIMD-aware weight pruning and node pruning are combined and applied to LeNet-300-100, LeNet-5, ConvNet, NIN and AlexNet. The processor has an 8-way SIMD unit and, therefore, the size of weight groups in SIMD-aware weight pruning is set to 8.

Figure 3.17 and Figure 3.18 are the relative performance speedups and relative model sizes

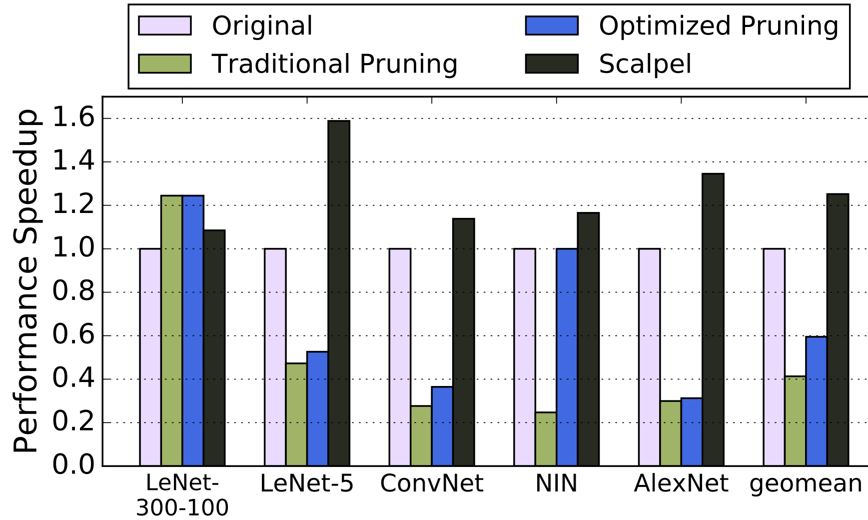


Figure 3.19: Relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel on NVIDIA GTX Titan X GPU.

of the original models, traditional pruning and Scalpel. For all tested networks, Scalpel achieves better performance and lower model size than traditional pruning and optimized pruning. The performance speedup can be up to 6.86x, and the relative size is reduced by up to 94.8%.

Notice that NIN pruned with optimized pruning holds the same performance and the same model size with the original model. This is because none of the layers in NIN can have more than 50% of weights removed through traditional pruning. Converting to sparse is expected to yield performance loss. Therefore, all the layers are not pruned for NIN with the optimized pruning. For the same reason, traditional pruning hurts the computation performance and increases the model size of NIN.

3.5.3 GPU - High Parallelism

For NVIDIA GTX Titan X GPU which is considered as high-parallelism hardware, node pruning is applied to keep the regular structure of DNNs. Figure 3.19 shows the relative performance speedups of the original models, traditional pruning, optimized pruning and Scalpel. Batch size is set to 50. For LeNet-5, ConvNet, NIN and AlexNet, Scalpel has much higher speedups compared to traditional pruning and optimized pruning. However, for LeNet-300-100, the speedup from

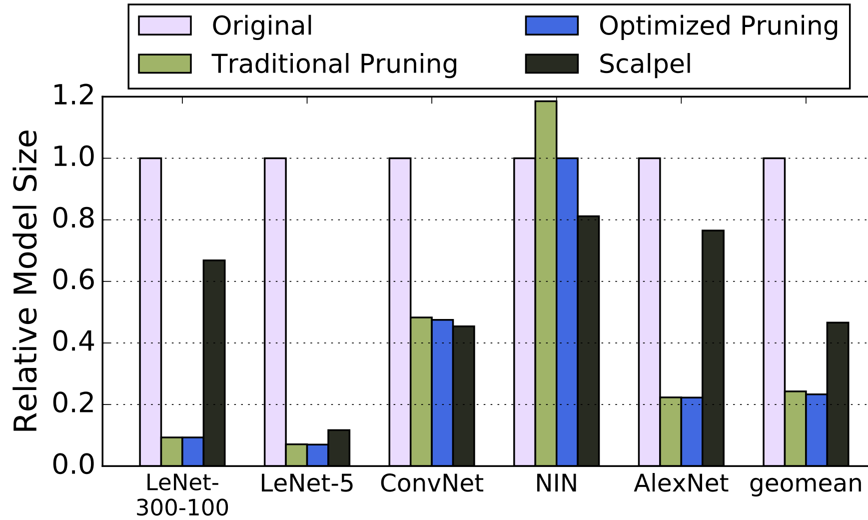


Figure 3.20: Relative model sizes of the original models, traditional pruning, optimized pruning and Scalpel for NVIDIA GTX Titan X GPU.

Scalpel is lower. The reason is that LeNet-300-100 has very tiny size and it is difficult to map the computation efficiently onto the GTX Titan X GPU. Another possible reason is the matrix tiling strategy in cuBLAS. A small change in the matrix size may lead to a large difference in the detailed matrix tiling strategy. As a result, although we can remove 31% and 32% of the nodes from the first and the second layers in LeNet-300-100, respectively, the computation performance is improved by only 8.50%.

The corresponding relative model sizes are shown in Figure 3.20. Models generated by Scalpel, except ConvNet and NIN, have higher model sizes than those generated by traditional and optimized pruning. It is because node pruning is applied now and we need to remove DNN redundancy at a granularity of nodes to keep the regular structure. But traditional and optimized pruning can prune DNN redundancy at a finer granularity of weights. Therefore, node pruning cannot reduce the model size as much as traditional pruning and optimized pruning do. However, the high-parallelism hardware is designed for high-throughput, and the DNN model size is not critical. Scalpel is more beneficial than traditional and optimized pruning on high-parallelism hardware since it can dramatically increase the computation throughput.

Table 3.4 gives the percentage of nodes we can remove from each layer in DNNs. As an

Table 3.4: Percentage of nodes removed by node pruning in each layer. Output layers are not included.

DNNs	Percentage of Nodes Removed in Each Layer
LeNet-300-100	31% (fc1)- 32% (fc2)
LeNet-5	50% (conv1)- 68% (conv2)- 65% (fc3)
ConvNet	28% (conv1)- 25% (conv2)- 49% (conv3)
NIN	28% (conv1)- 20% (cccp1)- 5% (cccp2)- 2% (conv2)- 14% (cccp3)- 8% (cccp4)- 22% (conv3)- 48% (cccp5)
AlexNet	3% (conv1)- 20% (conv2)- 24% (conv3)- 18%(conv4)-0%(conv5)-17%(fc6)-23%(fc7)

example, NIN consists of 9 convolutional layers, and each feature map is considered as one node in the convolutional layers. Layers close to the input or output have more redundancy and tend to be less important than the layers in the middle of NIN. 28% and 48% of the nodes can be removed from the first layer (conv1) and the eighth layer (cccp5), respectively.

3.6 Summary

In this chapter, we propose *Scalpel* to customize DNN pruning for different hardware platforms based on their parallelism. It includes two techniques: *SIMD-aware weight pruning* and *node pruning*. For low-parallelism hardware, SIMD-aware weight pruning is applied to keep remaining weights in aligned groups to fully utilize the SIMD units. All groups have the same size equal to the SIMD width. For high-parallelism hardware, node pruning removes redundant nodes. It avoids the sparsity in weights matrices caused by traditional pruning techniques. SIMD-aware weight pruning and node pruning can be combined and applied to DNN models for moderate-parallelism hardware. On the microcontroller, CPU and GPU, Scalpel achieves mean speedups of 3.54x, 2.61x, and 1.25x while reducing the model sizes by 88%, 82%, and 53%.

CHAPTER 4

Sub-Byte DNN on IoT Microcontrollers

4.1 Introduction

In conventional DNN-based IoT systems, the computation of the DNN models is not performed locally, but remotely on the cloud. For example, Figure 4.1 shows a typical visual IoT system. Ultra-low-power cameras on the sensor nodes collect the visual data, but then, the data is wirelessly transmitted to the cloud for further processing. DNN inferences on the cloud will perform the actual analysis. However, this cloud-based approach fails for many reasons: limited or no network connectivity, data privacy and security concerns, and finally wireless transmission energy. Therefore, the goal of this work is to enable local DNN computation on resource-impoverished microcontrollers.

The computation resource on low-power microcontrollers is extremely limited. A typical method to reduce DNN computation is using sub-byte weights and input activations. Here "sub-byte" refers to precision$\leq$8-bit. DNN models generally contain lots of redundant parameters and computation which are not necessary for maintaining high prediction accuracy. Using sub-byte weights and inputs can eliminate unnecessary numeric precision, which simplifies the computation operations and reduces memory access.

Past works have found that using 8-bit integers for both weights and inputs can maintain the original accuracy [23]. Compressing the weights further, Courbariaux et al. [24] and Li et al. [25] propose using binary (-1/+1) and ternary (-1/0/+1) weights, respectively, at the cost of small levels

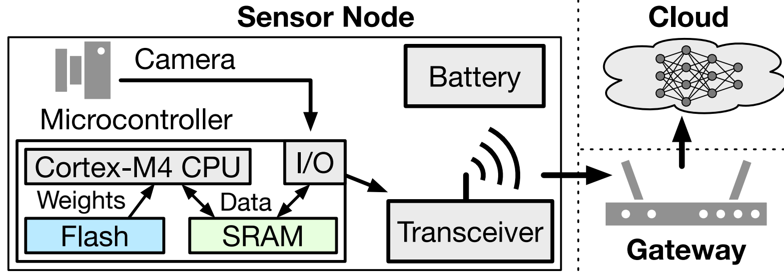


Figure 4.1: Conventional end-to-end visual IoT system.

of accuracy loss. For further compression on inputs, binarized neural networks [26] and XNOR-Net [11] use binary values for both weights and inputs. This further reduces the storage cost, but the binarization of input values will significantly hurt the prediction accuracy. Here using binary/ternary weights and 1- to 8-bit (1∼8-bit) inputs provides a feasible trade-off space between memory requirements and DNN accuracy.

However, counter-intuitively, *directly using sub-byte weights and inputs hurts DNN computation performance*. This is because of the incompatibility between the sub-byte data format and the byte-addressable memory hierarchy. To save memory space, sub-byte values are stored in bit-streams (sub-byte data format). As neither SRAM nor flash is bit-addressable, we need to first unpack the sub-byte inputs and weights from the bitstreams into byte-addressable numbers, e.g., 8-bit integers, before performing the corresponding DNN computation. This unpacking process consumes extra computation, yielding increases in the execution time. As an example, for the Network-in-Network (NIN) [61] model on CIFAR-10 dataset [65], using ternary weights and 4-bit inputs leads to a 3.8× increase in the execution time vs. the 16-bit baseline.

Besides, although using sub-byte inputs and weights can reduce the memory cost, existing SRAM space may still not be enough. A typical microcontroller offers only 9KB to 256KB SRAM and 64KB to 1MB flash [66, 67, 68, 69, 70]. For DNN computation, constant values, e.g., weights, are stored in the non-volatile flash. The intermediate values, including the input feature maps and input matrix (generated by unrolling the input feature maps), are generated at runtime and must reside in on-chip SRAM. For NIN, using ternary weights reduces the weight values to 264KB, within the available flash. But for input values, using even binary (1-bit) values requires 174KB of

SRAM. Thus using sub-byte inputs may still exceed the available SRAM, e.g., 128KB on our test board [66]. Worse, a higher input precision, e.g., 4-bit, is critical for maintaining a high accuracy but costs even more SRAM space. Larger SRAM sizes for IoT nodes, e.g., 256KB, are possible but drive up unit prices and energy consumption.

To address these challenges, we propose a software-hardware co-design pipeline to deploy sub-byte DNN models on microcontrollers efficiently. The proposed pipeline includes three main stages: uniform LQ-Net, bitset self-loop convolution and GoLo extensions. As the first stage, we provide a sub-byte DNN training framework, uniform LQ-Net, to generate sub-byte models which can be effectively accelerated in the following stages. In the next stage, we introduce a new convolution algorithm, bitset self-loop convolution. It overwrites input feature maps with the output pixels, reducing the SRAM requirement. Also, we reconstruct how we load sub-byte values to mitigate the unpacking overhead. Rather than extracting all bits of a value into a single register, the same binary digits from multiple input values (called a bitset) is loaded and processed. The multiply-accumulate (MAC) operations then proceed bit-serially using bitwise operations.

To further accelerate the bitset self-loop convolution, we propose a series of instruction extensions named GoLo to the Arm v7-M ISA [71]. It contains two categories of instructions: simple arithmetic instructions and more complex microprogramming instructions. With simple arithmetic instructions, we provide hardware support for two repeating operations: POPCOUNT and XNOR_AND. The second category, microprogramming instructions, includes the (double-width) Direct Memory Access (DMA)-based bitset multiply instructions. Repeating operations are fused into a DMA-based computation pipeline, further reducing execution latency and energy consumption.

This chapter makes the following contributions:

- We demonstrate that a naive implementation of sub-byte DNN models hurts the computation performance on microcontrollers and may still cost more SRAM space than available.
- A software-hardware co-design pipeline is provided to deploy sub-byte networks on microcontrollers efficiently. Both CNNs and recurrent neural networks (RNNs) are supported.

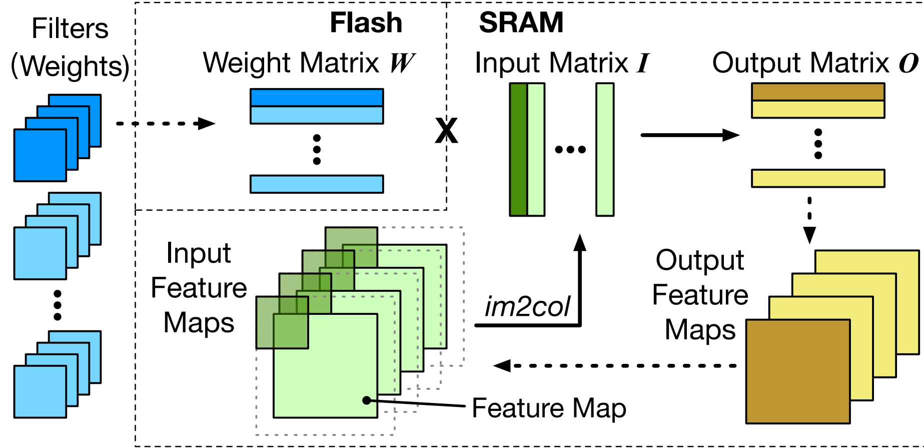


Figure 4.2: Conventional computation algorithm of convolutional layers on IoT microcontrollers.

- We show that, for low-power low-cost microcontrollers, extending existing ISA can help effectively accelerate the computation while minimizing the hardware overhead.
- The proposed pipeline is tested on the Arm Cortex-M4 microcontroller. Using ternary weights and 4-bit inputs, it can improve the computation performance and energy efficiency by 3.37x and 3.42x on average, respectively.

4.2 Background and Motivation

Deep Neural Networks (DNNs) often include a significant amount of internal redundancy. Using sub-byte weights and inputs is a typical method to remove this redundancy.

4.2.1 Sub-Byte DNNs

Deep convolutional neural networks (CNNs) integrate convolutional (CONV) layers and fully-connected (FC) layers into an end-to-end multi-layer network [72, 73]. In FC layers, all input values are connected to every neuron. Computationally, the input values are stored in a $N \times 1$ matrix $\boldsymbol{I}$ (1D vector). The weight matrix is multiplied with the input vector to generate the output vector. CONV layers consist of a stack of 2D matrices called feature maps. Figure 4.2 shows the detailed computation in CONV layers. The input feature maps are rearranged into the input matrix

$\boldsymbol{I}$ through the image-to-column (im2col) function. Then the weight matrix $\boldsymbol{W}$ is multiplied with the input matrix as $\boldsymbol{O} = f(\boldsymbol{W} \cdot \boldsymbol{I})$ to generate the output matrix $\boldsymbol{O}$. $f()$ is the element-wise activation function. The output matrix then becomes the input feature maps to the subsequent layer. In both FC and CONV layers, the computation is mainly consumed by the matrix multiplication $\boldsymbol{W} \cdot \boldsymbol{I}$.

Modern microcontrollers contain two types of memory: SRAM and flash. Flash is generally considered read-only during the computation, e.g., DNN inferences, due to the high latency and energy cost of writes. As shown in Figure 4.2, the weight values, which will not be modified during inferences, can be stored on the flash memory. However, the input values, including the input feature maps and the input matrix, are generated at runtime and must be kept in the SRAM. Once the input feature maps have been unrolled into the input matrix, the output feature maps can overwrite the SRAM space occupied by the input feature maps.

Using sub-byte weights and inputs is a typical method to remove the internal redundancy of DNN models and reduce the computation requirement. As an example, XNOR-Net [11] proposes using binary values for both weights and input activations. The i-th row, $\boldsymbol{W}_{i,:}$, of the weight matrix, corresponding to the i-th filter, can be binarized as

$$\boldsymbol{W}'_{i,:} = \alpha_i \cdot \mathrm{sign}(\boldsymbol{W}_{i,:}) \quad \alpha_i = \mathrm{mean}(|\boldsymbol{W}_{i,:}|) \tag{4.1}$$

$\mathrm{sign}()$ is the sign function and $\mathrm{sign}(\boldsymbol{W})$ is a binary matrix. Similarly, the j-th column in the input matrix $\boldsymbol{I}_{:,j}$, corresponding to the j-th convolution window, is binarized as

$$\boldsymbol{I}'_{:,j} = \beta_j \cdot \mathrm{sign}(\boldsymbol{I}_{:,j}) \quad \beta_j = \mathrm{mean}(|\boldsymbol{I}_{:,j}|) \tag{4.2}$$

The corresponding output pixel $\boldsymbol{O}_{i,j}$ can be generated by

$$\boldsymbol{O}_{i,j} = f(\boldsymbol{W}'_{i,:} \cdot \boldsymbol{I}'_{:,j}) = f[\alpha_i \beta_j (\mathrm{sign}(\boldsymbol{W}_{i,:}) \cdot \mathrm{sign}(\boldsymbol{I}_{:,j}))] \tag{4.3}$$

where both $\mathrm{sign}(\boldsymbol{W})$ and $\mathrm{sign}(\boldsymbol{I})$ are binary matrices. The multiplication between binary values can

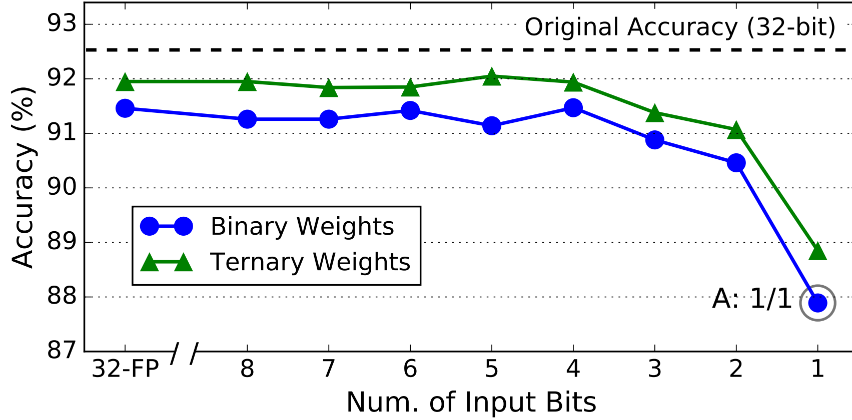


Figure 4.3: Prediction accuracy of NIN with different input/weight precisions.

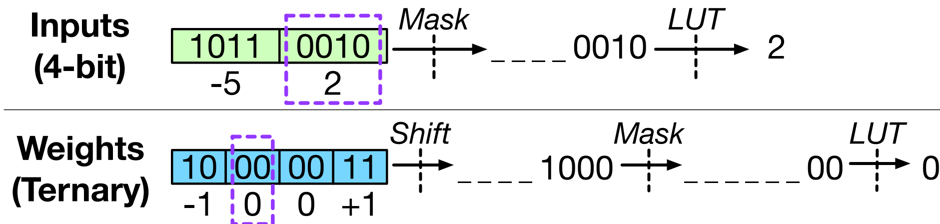


Figure 4.4: Examples of unpacking sub-byte inputs/weights.

be replaced by XNOR operations. As an example, using the bit 0/1 to represent the value -1/+1, $-1 \times +1 = -1$ can be converted to $0 \text{ XNOR } 1 = 0$. Then the popcount operation, which counts the number of bits "1", is used to perform the accumulation.

Based on LQ-Net [74], we build a framework to train networks with binary/ternary weights and 1$\sim$8-bit inputs. The details are discussed in Section 4.3.1. Figure 4.3 shows the accuracy of NIN with varied input precisions. Weight binarization reduces the accuracy by 1.1%, but further binarizing the inputs leads to a large accuracy decrease of 4.6%. A high input precision, e.g., 4-bit, is necessary to achieve high accuracy.

4.2.2 Challenges

Although using sub-byte weights and inputs helps reduce the computation requirement, it is challenging to deploy sub-byte networks on microcontrollers efficiently.

The main challenge is that weights and inputs stored in sub-byte data formats require extra

unpacking operations, which hurts computation performance. Since the memory hierarchy of microcontrollers is not bit-addressable, sub-byte values need to be stored as bitstreams, and cannot be directly accessed. Instead, these values need to be unpacked before being used for computation. Here “unpacking” means extracting sub-byte values from the bitstream into a format, e.g., 8-bit/16-bit signed integers, suitable for general-purpose instructions. Figure 4.4 gives two unpacking examples. For 4-bit inputs, two values are stored in one byte. Unpacking the second input ($2 = (0010)_2$) requires masking out the lower four bits and using a lookup table (LUT) to determine the corresponding 8-bit signed value. The ternary weights can be unpacked through a similar 2-bit process. Here we map 2-bit values 10/00/11 to ternary weights -1/0/+1. For binary weights, 0/1 map to -1/+1.

These unpacking operations lead to a significant increase in execution time. Figure 4.5 shows the execution time breakdown for NIN on the test board with different weight/input precisions. The baseline uses 16-bit signed integers for both weights and inputs, which fully utilizes the SIMD support and achieves the best performance [75]. Except for inputs and weights both being binary, sub-byte configurations lead to a significant execution time increase up to 4.3x. Input unpacking overheads are higher than weight unpacking as unpacked weights are reused for multiple inputs. Using binary weights and inputs improves performance vs. the baseline by replacing MAC with XNOR and popcount operations. This removes the need for weight unpacking, and input unpacking is only necessary for the im2col function. Yet, this configuration significantly impacts the network accuracy. For NIN, the prediction accuracy loss is 4.6%, i.e., point A in Figure 4.3.

Besides the performance degradation, another challenge is that using sub-byte input values may still not meet the SRAM limitation. Figure 4.6 shows the memory requirements of NIN with different input and weight precisions. Our test board includes an Arm Cortex-M4 CPU with 128KB SRAM and 512KB flash [66]. Both ternary and binary weights require < 512KB storage and can fit onto the flash memory. However, for input values including the input feature maps and the input matrix, using 4-bit values will only reduce the corresponding storage cost to 396KB. This storage cost still exceeds the SRAM size (128KB) on the test board. Using a larger SRAM may help

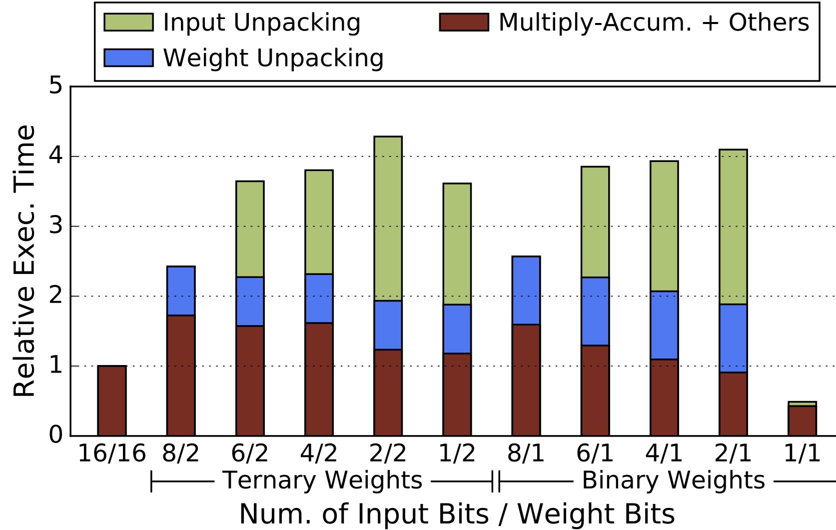


Figure 4.5: Relative execution time breakdown of NIN with different input/weight precisions. Unpacking inputs/weights from the sub-byte formats will lead to an execution time increase.

solve this issue but will dramatically increase the manufacturing cost. Note here we assume no overheads for other data, i.e., the runtime stack and global variables, which also requires a portion of the SRAM space.

4.3 Sub-byte DNN Deploying Pipeline

To address those challenges, we propose a new pipeline to deploy sub-byte networks on microcontrollers efficiently. It provides support for sub-byte networks with binary/ternary weights and 1- to 8-bit activations. Figure 4.7 shows an overview of the proposed pipeline which consists of three main parts. We first provide a training framework, uniform LQ-Net, to prepare high-accuracy sub-byte CNN and RNN models for the following steps. A new computation algorithm, bitset self-loop convolution, is then introduced to accelerate the sub-byte computation. As the third part of the pipeline, we propose new extensions to the Arm v7-M ISA to further improve the computation performance with minimum hardware overhead.

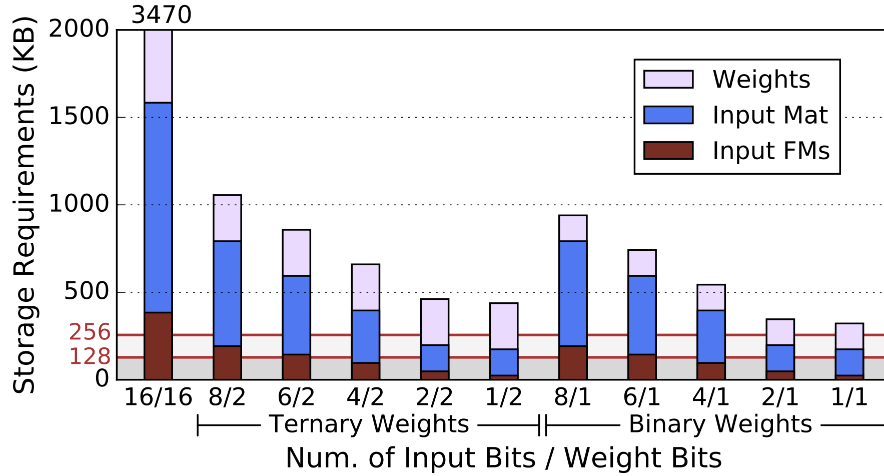


Figure 4.6: Storage requirements breakdown of NIN with different input/weight precisions. Input matrix (Input Mat) and input feature maps (Input FMs) have to be stored in limited SRAM storage, e.g., 128KB or 256KB.

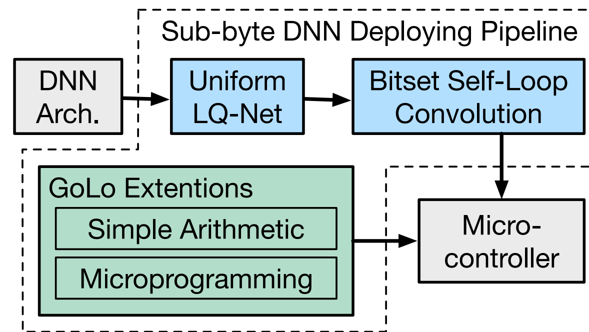


Figure 4.7: Overview of the sub-byte DNN deploying pipeline.

4.3.1 Uniform LQ-Net

The first step of the proposed pipeline is to train sub-byte DNN models based on the network architectures provided by users. We develop a new training framework based on LQ-Net [74]. LQ-Net is one of the state-of-the-art frameworks for CNN quantization using non-uniform units. We modify it to use uniform quantization instead for efficient computation and add the support for RNNs.

Weight Quantization: Each row, $\boldsymbol{W}_{i,:}$, of the weight matrix contains the weights for one filter in

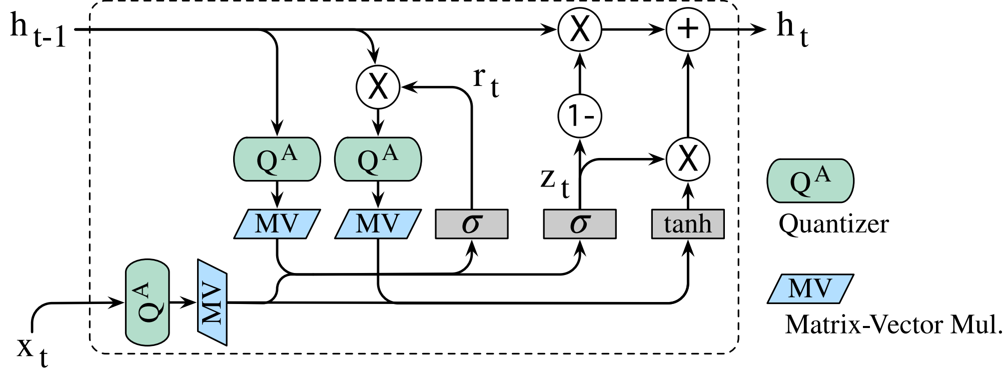


Figure 4.8: Quantization in gated recurrent units (GRUs).

CONV layers or one neuron in FC layers. It can be quantized into binary or ternary values as

$$\boldsymbol{W}'_{i,:} = \mathrm{Q}^W(\boldsymbol{W}_{i,:}) = \begin{cases} v_i^w \cdot \mathrm{sign}(\boldsymbol{W}_{i,:}), & \text{binary} \\ v_i^w \cdot \mathrm{clamp}(\mathrm{round}(\frac{\boldsymbol{W}_{i,:}}{v_i^w})), & \text{ternary} \end{cases} \tag{4.4}$$

The $\mathrm{clamp}()$ function limits the rounded values to [-1, 1]. v^w is a trainable parameter.

Activation Quantization: In conventional CNN models, the input activations for each CONV layer is generated by the ReLU function of the previous layer. Therefore, the input activations will be non-negative ($\boldsymbol{I} \geq 0$).

But for RNN models, the activations before quantization include negative values. Figure 4.8 shows how we quantize the computation in gated recurrent units (GRUs). Other RNN units, e.g., LSTM [76], can be quantized similarly. In the time step t, the detailed computation with quantization is

$$\boldsymbol{z}_t = \mathrm{sigmoid}\big(\boldsymbol{W}'_{z,x}\mathrm{Q}_x^A(\boldsymbol{x}_t) + \boldsymbol{W}'_{z,h}\mathrm{Q}_h^A(\boldsymbol{h}_{t-1}) + \boldsymbol{s}_z\big) \tag{4.5}$$

$$\boldsymbol{r}_t = \mathrm{sigmoid}\big(\boldsymbol{W}'_{r,x}\mathrm{Q}_x^A(\boldsymbol{x}_t) + \boldsymbol{W}'_{r,h}\mathrm{Q}_h^A(\boldsymbol{h}_{t-1}) + \boldsymbol{s}_r\big) \tag{4.6}$$

$$\begin{aligned} \boldsymbol{h}_t =& (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \\ & \boldsymbol{z}_t \odot \tanh\big(\boldsymbol{W}'_{h,x}\mathrm{Q}_x^A(\boldsymbol{x}_t) + \boldsymbol{W}'_{h,h}\mathrm{Q}_{r,h}^A(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{s}_h\big) \end{aligned} \tag{4.7}$$

$\odot$ is the element-wise multiplication. $\boldsymbol{x}_t$ and $\boldsymbol{h}_t$ are the input vector and hidden state vector. $\boldsymbol{z}_t$ and $\boldsymbol{r}_t$ are the update and reset gate vector. Each $\boldsymbol{W}'$ is a quantized weight matrix and $\boldsymbol{s}$ is the bias

vector. $\boldsymbol{x}_t$, $\boldsymbol{h}_{t-1}$ and the intermediate vector $(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1})$ are considered as the activation values $\boldsymbol{I}$ and are quantized right before the corresponding matrix-vector multiplication. These vectors contain negative values, and $\boldsymbol{Q}_x^A$, $\boldsymbol{Q}_h^A$, $\boldsymbol{Q}_{r,h}^A$ are the quantizers for each activation vector, respectively.

In this case, the activations $\boldsymbol{I}$ in CNN and RNN models need to be quantized using similar but different methods as

$$\begin{gathered}\boldsymbol{I}' = \mathrm{Q}^A(\boldsymbol{I}) = v^a \cdot \Big(\mathrm{clamp}(\mathrm{round}(\frac{\boldsymbol{I}}{v^a} - \gamma)) + \gamma\Big) \\ \mathrm{clamp}()\ \mathrm{range},\ \gamma = \begin{cases} [-2^{k-1}, 2^{k-1}-1],\ 2^{k-1} & \mathrm{CNN} \\ [-2^{k-1}, 2^{k-1}-1],\ 0.5 & \mathrm{RNN} \end{cases}\end{gathered} \tag{4.8}$$

where v^a is also a trainable parameter. We choose $\gamma = 2^{k-1}$ for CNN and then the clamp() results can always be represented by k-bit signed integers, which benefits the future computation acceleration.

Quantizer Training: The weight and activation quantization have a similar format as

$$\boldsymbol{W}'_{i,:} = v_i^w \cdot \boldsymbol{B}_{i,:} \quad \boldsymbol{I}' = v^a \cdot (\boldsymbol{H} + \gamma) \tag{4.9}$$

where $\boldsymbol{B}$ and $\boldsymbol{H}$ are both integer tensors. v_i^w and v^a are trainable parameters which can be trained with the same algorithm. Using v^a as an example, in the s-th training iteration, we perform the following two steps:

$$① \ \boldsymbol{I}' = \mathrm{Q}^A(\boldsymbol{I}) = v_{s-1}^a \cdot (\boldsymbol{H} + \gamma) \tag{4.10}$$

$$② \ v_s^a = (1-m)v_{s-1}^a + m \cdot \frac{\sum[(\boldsymbol{H}+\gamma) \odot \boldsymbol{I}]}{\sum[(\boldsymbol{H}+\gamma) \odot (\boldsymbol{H}+\gamma)]} \tag{4.11}$$

m is the moving average factor fixed to 0.9. The training of v^w can be considered as a similar case with $\gamma = 0$.

For a variable x, the gradient of function $\mathrm{clamp}(\mathrm{round}(x))$ is 0 at almost everywhere, which makes the backpropagation problematic [74, 26]. To address this issue, we follow the same strategy in LQ-Net. For activation quantization, the gradient is set to 1 for x in the clamp() range and 0

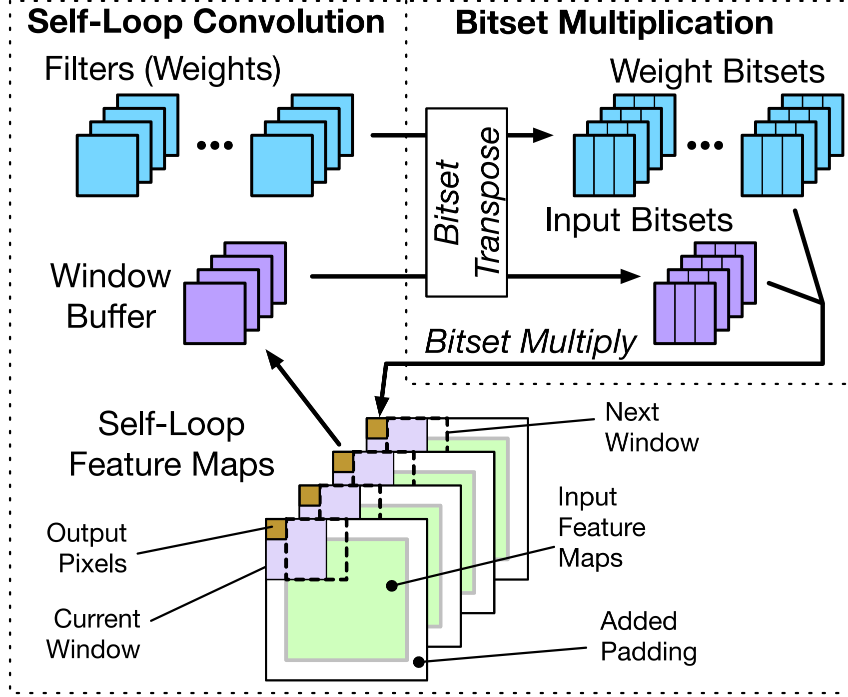


Figure 4.9: Overview of the bitset self-loop convolution.

elsewhere. For weight quantization, the gradient is set to 1 everywhere.

After quantization, the output $\boldsymbol{O}_{i,j}$ is calculated as

$$\boldsymbol{O}_{i,j} = f(\boldsymbol{W}'_{i,:} \cdot \boldsymbol{I}'_{:,j}) = f(v_i^w v^a (\boldsymbol{B}_{i,:} \cdot \boldsymbol{H}_{:,j}) + v_i^w v^a \gamma \sum \boldsymbol{B}_{i,:}) \tag{4.12}$$

The second item $v_i^w v^a \gamma \sum \boldsymbol{B}_{i,:}$ is a fixed value and can be computed offline. Therefore, in DNN inferences, the main computation we need to perform is $\boldsymbol{B}_{i,:} \cdot \boldsymbol{H}_{:,j}$.

4.3.2 Bitset Self-Loop Multiplication

As the second part, we introduce a new convolution algorithm, bitset self-loop convolution. It consists of two parts: self-loop convolution and bitset multiplication. For self-loop convolution, padded input feature maps are directly overwritten by output feature maps. For bitset multiplication, MAC operations between sub-byte numbers are replaced with bitwise logic operations to mitigate the unpacking overhead.

Self-Loop Convolution For self-loop convolution, we first implement an optimized implicit generalized matrix-matrix multiplication (implicit-GEMM) algorithm. Different from Figure 4.2, we do not generate the input matrix at once. Instead, we generate and process each column one after another. Here each column corresponds to a convolution window. The input values for a single column are unpacked and stored as 16-bit integers in a buffer space. The unpacked input column can then be shared across all the weight filters and, therefore, the input unpacking overhead is amortized. Also, there is no need to store the entire input matrix in SRAM, which helps dramatically reduce the SRAM requirement.

However, with the optimized implicit-GEMM, we still need to allocate separate SRAM spaces for input and output feature maps. To further reduce the SRAM requirement, we propose the self-loop convolution in which input and output feature maps share the same SRAM space. Figure 4.9 shows the detailed steps. For the input feature maps, we first remap the input feature maps to add paddings as configured in the DNN architecture. The padded input feature maps are named as the self-loop feature maps. Next, as the same with the optimized implicit-GEMM, the input values in the current sliding window are unpacked into 16-bit signed integers and kept in the window buffer. The window buffer will be multiplied with all the weight filters to generate the output pixels. As the last step, the output pixels can be written back to the self-loop feature maps at the top-left corner of the current window. The remapping in the first step ensures that the top-left corner of all sliding windows physically exist in SRAM and, therefore, can be overwritten by the output pixels. As shown in Figure 4.9, the next window will not cover the top-left corner of the current window. This means the input pixels overwritten by the outputs will not be reused in the future computation.

By overwriting the padded input feature maps with the output pixels, the self-loop convolution further reduces the SRAM requirement. We only need to allocate the SRAM space for the self-loop feature maps and the window buffer. Figure 4.10 shows the SRAM requirements of NIN with different convolution algorithms. For our test board with 128KB SRAM, the optimized implicit-GEMM can help enable using 2-bit inputs. With the self-loop convolution, we now can fit 4-bit inputs into the SRAM.

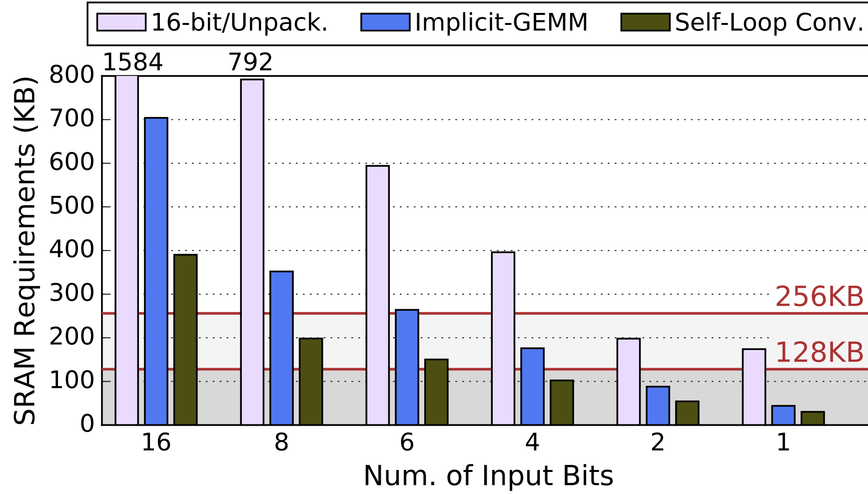


Figure 4.10: SRAM storage requirement of NIN. The conventional sub-byte convolution algorithm is annotated as *Unpacking (Unpack.)*.

In both the optimized implicit-GEMM and self-loop convolution, the unpacked inputs will be reused by all the weight filters. Each input pixel only needs to be unpacked once in the computation. The input unpacking overhead is dramatically reduced, which helps accelerate the computation. We measure the performance of NIN on the test board with self-loop convolution, and the results are included in Figure 4.11. Except for the configuration of both inputs and weights being binary, the self-loop convolution can achieve a significant performance improvement. The implicit-GEMM delivers a similar performance since the overhead of remapping is negligible.

Bitset Multiplication With self-loop convolution, we still need to unpack the sub-byte weights and perform the computation using 16-bit or 32-bit integers. In this case, much computation and energy are wasted on unnecessary bits. To fully utilize the reduction in input/weight precision, we propose bitset multiplication. It separates the computation for each binary digit of input values and, therefore, lower input precisions require less computation.

Suppose we need to multiply a 4-bit signed integer, $\boldsymbol{h} = (h_3h_2h_1h_0)_2$, with a ternary weight, $\boldsymbol{b} = (b_1b_0)_2$. For the ternary weight $\boldsymbol{b}$, $(11)_2$, $(00)_2$, $(10)_2$ represent +1, 0, -1, respectively. The

multiplication $\boldsymbol{b} \cdot \boldsymbol{h}$ can be written as

$$\boldsymbol{b} \cdot \boldsymbol{h} = \left\{ \begin{array}{rr} \boldsymbol{h},\ \boldsymbol{b} = & 1 \\ 0,\ \boldsymbol{b} = & 0 \\ -\boldsymbol{h},\ \boldsymbol{b} = & -1 \end{array} \right\} = \begin{cases} \boldsymbol{h} + 0,\ \boldsymbol{b} = \quad 1 \\ 0 + 0,\ \boldsymbol{b} = \quad 0 \\ \overline{\boldsymbol{h}} + 1,\ \boldsymbol{b} = -1 \end{cases} \tag{4.13}$$

Recall, with $\boldsymbol{h}$ in the two's complement notation [77], we have $-\boldsymbol{h} = \overline{\boldsymbol{h}} + 1$ where $\overline{\boldsymbol{h}}$ is the bitwise NOT operation. As $\boldsymbol{h}$ is signed, we can expand Equation 4.13 as

$$\boldsymbol{b} \cdot \boldsymbol{h} = \begin{cases} (-8 \cdot h_3 + 4 \cdot h_2 + 2 \cdot h_1 + h_0)\ + 0,\ \boldsymbol{b} = \quad 1 \\ (-8 \cdot\ 0 + 4 \cdot\ 0 + 2 \cdot\ 0 +\ 0)\ + 0,\ \boldsymbol{b} = \quad 0 \\ (-8 \cdot \overline{h_3} + 4 \cdot \overline{h_2} + 2 \cdot \overline{h_1} + \overline{h_0})\ + 1,\ \boldsymbol{b} = -1 \end{cases} \tag{4.14}$$

It can then be converted to

$$\begin{aligned} \boldsymbol{b} \cdot \boldsymbol{h} &= \textstyle\sum_{i=0}^{3} \left[(-1)^{(i==3)} \cdot 2^i \cdot g(\boldsymbol{b}, h_i)\right] + l(\boldsymbol{b}) \\ g(\boldsymbol{b}, h_i) &= \left\{ \begin{array}{l} h_i,\ \boldsymbol{b} = \ 1 \ = (11)_2 \\ 0\,,\ \boldsymbol{b} = \ 0 \ = (00)_2 \\ \overline{h_i},\ \boldsymbol{b} = -1 = (10)_2 \end{array} \right\} \\ &= b_1 \,\&\, b_0 \,\&\, h_i + b_1 \,\&\, \overline{b_0} \,\&\, \overline{h_i} \\ &= b_1 \,\&\, \overline{(b_0 \oplus h_i)} \\ l(\boldsymbol{b}) &= \left\{ \begin{array}{l} 0,\ \boldsymbol{b} = \ 1 \ = (11)_2 \\ 0,\ \boldsymbol{b} = \ 0 \ = (00)_2 \\ 1,\ \boldsymbol{b} = -1 = (10)_2 \end{array} \right\} = b_1 \,\&\, \overline{b_0} \end{aligned} \tag{4.15}$$

where $(-1)^{(i==3)}$ equals to -1 when $i = 3$, and $+1$ otherwise. $\&$, $\overline{(b \oplus h)}$ are bitwise AND, XNOR operations. Based on Equation 4.15, for N inputs $\boldsymbol{h}^0, \ldots, \boldsymbol{h}^{N-1}$ and weights $\boldsymbol{b}^0, \ldots, \boldsymbol{b}^{N-1}$,

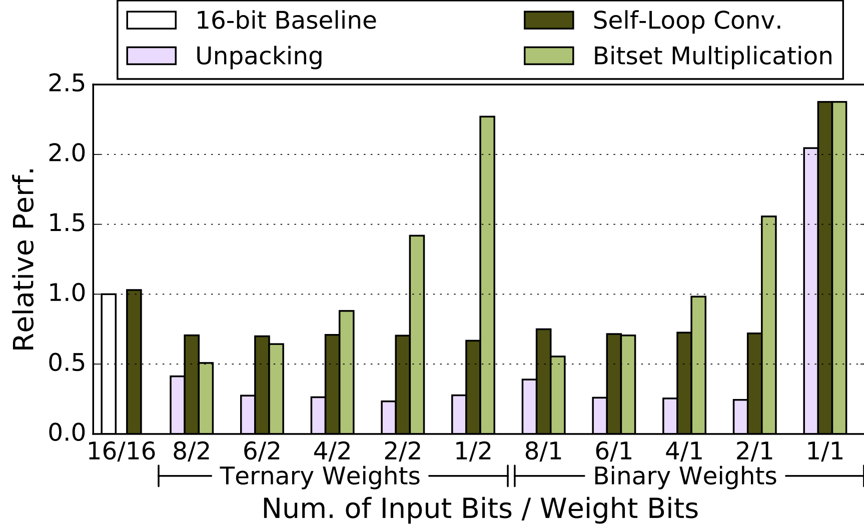


Figure 4.11: Relative computation performance of NIN with different convolution algorithms. The baseline uses 16-bit integers for both inputs and weights.

we can calculate their MAC result as

$$\sum_{k=0}^{N-1} \boldsymbol{b}^k \cdot \boldsymbol{h}^k = \sum_{i=0}^{3} \left[(-1)^{(i==3)} \cdot 2^i \cdot \sum_{k=0}^{N-1} g(\boldsymbol{b}^k, h_i^k) \right] + \sum_{k=0}^{N-1} l(\boldsymbol{b}^k) \tag{4.16}$$

Here we first perform the computation, $\sum_{k=0}^{N-1} g(\boldsymbol{b}^k, h_i^k)$, related to the i-th digit of the inputs and then calculate their weighted sum as the final output. With this strategy, we can separate the computation related to each input digit, and now using a lower input precision requires less computation. Here we use signed integers for $\boldsymbol{h}$ and, therefore, input values need to be quantized to signed integers in Equation 4.8.

We now combine the bits on the same digit of multiple sub-byte values into the same group called a bitset:

$$\mathcal{B}_t = (b_t^0 \dots b_t^{N-1})_2 \qquad \mathcal{H}_i = (h_i^0 \dots h_i^{N-1})_2 \tag{4.17}$$

where $t \in [0, 1]$ and $i \in [0, 3]$. We can then parallelize the computation and accumulation of $g()$

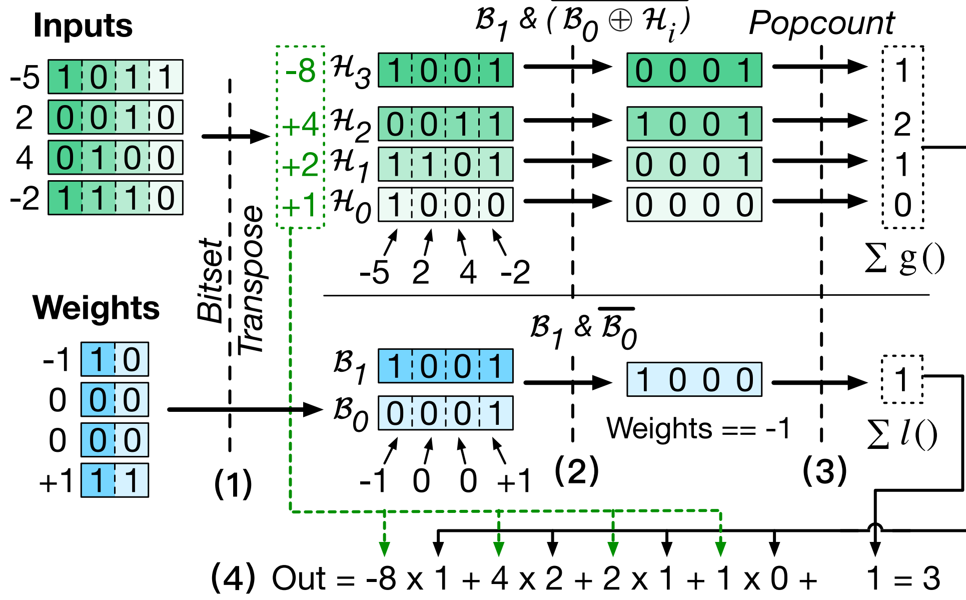


Figure 4.12: Bitset multiplication. &, $\overline{\mathcal{B}}$, $\oplus$, $\overline{\mathcal{B} \oplus \mathcal{H}}$ are bitwise AND, NOT, XOR, XNOR operations, respectively.

and $l()$ using bitwise operations between bitsets, and popcount operations:

$$\begin{aligned} \sum_{k=0}^{N-1} g(\boldsymbol{b}^k, h_i^k) &= \text{popcount}(\mathcal{B}_1 \,\&\, \overline{(\mathcal{B}_0 \oplus \mathcal{H}_i)}) \\ \sum_{k=0}^{N-1} l(\boldsymbol{b}^k) &= \text{popcount}(\mathcal{B}_1 \,\&\, \overline{\mathcal{B}_0}) \end{aligned} \tag{4.18}$$

Based on this analysis, Figure 4.12 shows the detailed steps of bitset multiplication. 4-bit inputs are used as an example here, and the same strategy supports various input precisions. In the first step (bitset transpose), inputs and weights in the original sub-byte formats are transposed to generate the bitsets ($\mathcal{B}_t$ and $\mathcal{H}_i$ in Equation 4.17). An efficient bitset transpose algorithm is implemented based on the Parabix Transform transpose [78]. The steps (2) and (3) then perform the functions $g()$ and $l()$, and their accumulation as shown in Equation 4.18. In the final step (4), the weighted sum of the accumulation results is calculated as in Equation 4.16 to get the final output.

The first three steps for the fixed weight values can be performed offline, and the value of $\sum_{k=0}^{N-1} l(\boldsymbol{b}^k)$ can be statically saved. For binary weights, the weight bitsets will be stored as $\mathcal{B}_0$, and $\mathcal{B}_1$ can be removed from the computation. The remaining computation steps will stay the

same.

With the bitset multiplication, the weight unpacking overhead can be avoided, and lower input precisions require less computation. We test the computation performance of the bitset multiplication on NIN, and the results are shown in Figure 4.11. For 8-bit and 6-bit input values, the bitset multiplication will slow down the computation because the popcount operation is not included in Arm v7-M ISA [71]. A software implementation of popcount takes at least 11 instructions, incurring much overhead. However, for input precisions$<$6-bit, the reduction in the required computation can help achieve better performance. Therefore, for the bitset self-loop convolution, the bitset multiplication is adopted for input precision$<$6-bit. With input precisions$\geq$6-bit, we use the self-loop convolution without the bitset multiplication.

Using the bitset multiplication will not affect the SRAM cost. As shown in Figure 4.9, besides the self-loop feature maps, the window buffer is still required. Since SIMD support is not needed for the Parabix Transform transpose, we use 8-bit integers instead of 16-bit for the window buffer, which halves the SRAM cost. However, an extra SRAM space is required for storing the generated bitsets. In this case, the total SRAM cost keeps the same.

For fully-connected (FC) layers, the bitset multiplication is used for all input precisions. Since FC layers only have one input vector, the weight unpacking overhead cannot be amortized by reusing one unpacked weight for multiple input values. Therefore, the relative overhead of weight unpacking becomes higher. The bitset multiplication can avoid the overhead of weight unpacking and achieve better performance.

For high-end processors, without a bit-addressable memory hierarchy, the computation for sub-byte DNN models still suffers from the weight/input unpacking overhead. In this case, the bitset multiplication can also be utilized to accelerate the computation.

4.4 GoLo Extensions

To further accelerate bitset self-loop convolution with minimal area and energy overhead, we propose a series of instruction extensions, GoLo, to the existing Arm v7-M ISA. The GoLo extensions can be divided into two categories: POPCOUNT and XNOR_AND as simple arithmetic instructions, and the (double-width) DMA-based bitset multiply instructions as microprogramming instructions. The microprogramming instructions require the simple arithmetic instructions to be supported. More extensions can help achieve better performance and energy efficiency, but with a higher manufacturing cost. Chip designers can choose which categories to implement.

Recently, various ultra-low-power accelerators are proposed to handle reduced precision computation, e.g., Lattice [79] and UNPU [80]. However, this chapter focuses on extending a (re)programmable microcontroller for two reasons. First, IoT systems largely already include a microcontroller for support tasks, i.e., power management or radio processing, thus extending it minimizes changes to existing designs. In contrast, accelerators will likely require higher die area overheads, increasing manufacturing costs. Second, the rapid algorithmic evolution of DNN computation is likely to quickly render accelerator designs obsolete. Meanwhile, microcontrollers need to support various applications other than DNN inferences, but highly-optimized accelerators lose too much programming flexibility and cannot be utilized for other applications. Thus, accelerators are likely to increase development and unit costs, *and* have a higher likelihood of obsolescence than ISA extensions to existing microcontroller designs.

4.4.1 Simple Arithmetic Instructions

The first category of the GoLo extensions contains two simple arithmetic instructions: POPCOUNT and XNOR_AND. As shown in Figure 4.12, the operations applied in the second and third steps are the same for all input bitsets. In the second step, we need to perform XNOR between $\mathcal{B}_0$ and $\mathcal{H}_i$ (i $\in \{0,1,2,3\}$), and then AND with $\mathcal{B}_1$. In the third step, we use popcount to perform the accumulation. Since these two steps are applied for all input bitsets, fusing them into

Table 4.1: Synthesis results of integrated circuit modules.

Inst.	Modules	E/Op(pJ)	Delay(ns)	Area Over.(μm^2)
Simple Arith.	POPCOUNT	0.45	0.84	924.4 (0.78%*)
	XNOR_AND	0.13	0.31	545.2 (0.46%)
Micro-prog.	Latch	1.59	0.22	776.3 (0.65%)
	Controller	1.62	1.81	1525.7 (1.28%)

*Percentages are relative to the Arm Cortex-M4 CPU area ($0.119\mathrm{mm}^2$) excluding FPU, debug modules and SRAM.

two instructions can help improve the computation performance of the bitset multiplication. The proposed two instructions have the following formats:

- **POPCOUNT** ***Rd*****,** ***Rn***: compute the number of ”1” bits (Hamming weight) of *Rn* and store the result into *Rd*. *Rd* and *Rn* specify the destination and operand register, respectively.
- **XNOR_AND** ***Rd*****,** ***Rn*****,** ***Rm*****,** ***Ra***: compute the XNOR of *Rn* and *Rm*, perform the AND operation between the XNOR result and *Ra*, and store the result into *Rd*. *Rd* is the destination register. *Rn*, *Rm* and *Ra* specify the operand registers.

The execution units required for these two instructions can be integrated into the existing Arithmetic Logic Unit (ALU). The POPCOUNT unit can be implemented as an adder tree. We synthesized both execution units and Table 4.1 shows the results. The instruction decoder also needs to be extended, but we assume negligible area overhead. The basic Arm Cortex-M4 CPU, not including SRAM, has an area cost of $0.119\mathrm{mm}^2$ [81]. In this case, the total area overhead of the first two instruction extensions is only 1.23%.

Although included in the x86 ISA [82], POPCOUNT is not included in the Arm v7-M ISA due to limited applications, e.g., cryptography, which are not the target of low-power Arm microcontrollers. With the traditional MAC algorithm, POPCOUNT only benefits networks with both weights and inputs being binary. However, with the bitset multiplication, we demonstrate that POPCOUNT can benefit general computation between sub-byte numbers.

We test the computation performance with the proposed instructions for NIN. The results are shown in Figure 4.13. For all input precisions, the computation performance can be dramatically

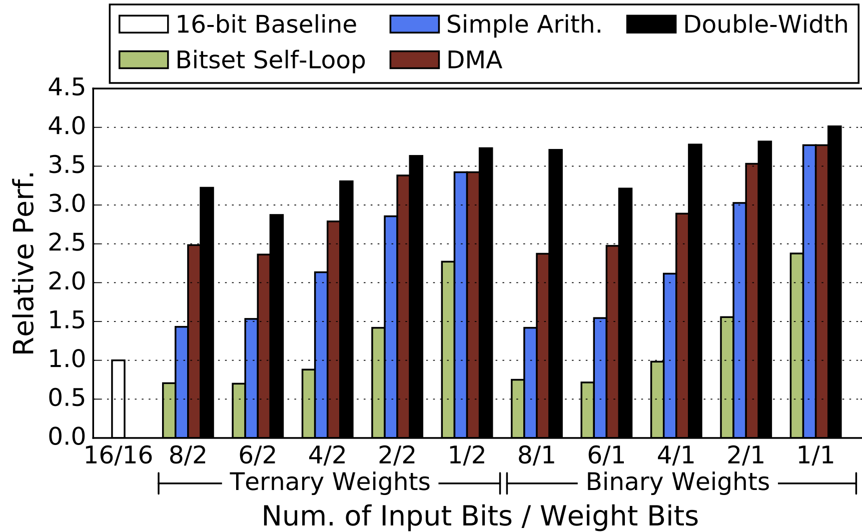


Figure 4.13: Relative computation performance of NIN with the GoLo extensions.

improved. With 4-bit inputs and ternary weights, adding the simple arithmetic instructions improves the relative performance from 0.88x to 2.13x. Now bitset multiplication can also improve the computation performance for input precisions≥6-bit, and will be adopted for all input precisions.

4.4.2 DMA-Based Bitset Multiply

As discussed above, in the second and third steps of bitset multiplication, we need to perform the XNOR_AND and popcount operations for all input bitsets. In the fourth step, the computation for each bitset is also the same. The popcount results will be multiplied by the corresponding magnitude and get accumulated to the temporary outputs. The only difference is the magnitude value. In this case, an effective solution to further accelerate the computation is to fuse the second, third, and fourth step into a hardware pipeline.

Figure 4.14 shows the proposed hardware pipeline integrated into the microcontroller architecture. Here we use the Arm Cortex-M4 microcontroller as an example. The same extensions can also be applied to other microcontrollers. The processor pipeline includes three stages: fetch, decode and execute. The light shadowed blocks are extra modules we need. The microprogramming

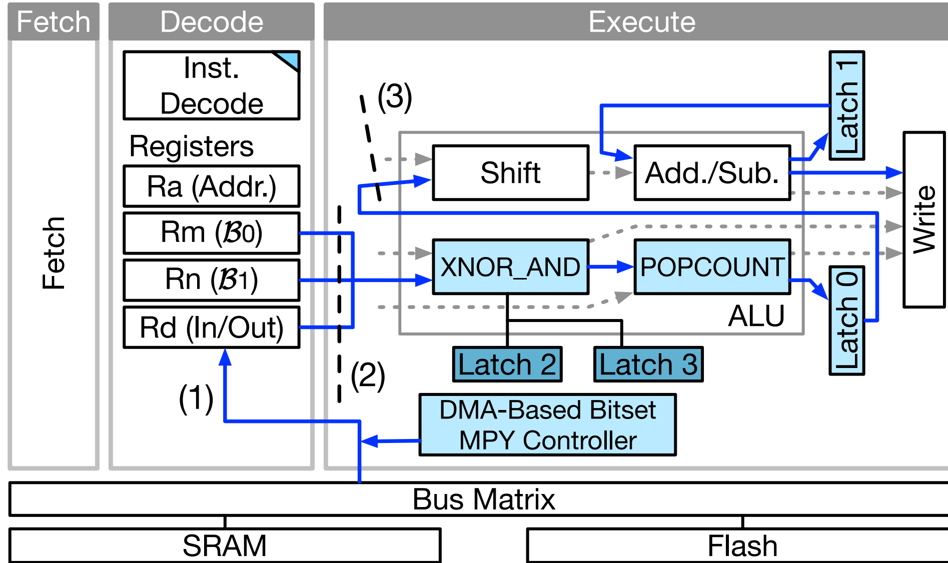


Figure 4.14: Architecture support for the microprogramming instructions.

instructions require the simple arithmetic instructions to be supported since the POPCOUNT and XNOR_AND units are reused here.

Before activating the proposed computation pipeline, we first need to load the weight bitsets $\mathcal{B}_1$ and $\mathcal{B}_0$ into the registers *Rn* and *Rm*, respectively. With k-bit inputs and ternary weights, each group of 32 input values will be transposed into k input bitsets. Each bitset has a size of 32 which equals to the register length. Similarly, each group of 32 weight values will be transposed into 2 weight bitsets.

The proposed computation pipeline shown in Figure 4.14 includes three stages. For the first stage, the temporary output stored in the register *Rd* is loaded into *Latch 1*. The input bitset address in *Ra* is stored into the controller and get incremented by one in each cycle to fetch the input bitsets continuously into *Rd*. In the second stage, the input bitset in *Rd* and the weight bitsets in *Rn* and *Rm* are used to perform the XNOR_AND and popcount operations. The temporary result is stored in *Latch 0*. In the last stage, the temporary result in *Latch 0* is multiplied with the corresponding magnitude and get accumulated onto the temporary output in *Latch 1*. Since the magnitudes are all power of 2, the multiplication can be replaced by shift operations. After processing all input bitsets, the temporary output will be written back to *Rd*. For binary weights, a block of 32 weight

values will be stored in only one bitset $\mathcal{B}_0$. *Rn* will store a constant number of 0xFFFF_FFFF, and other steps are the same with ternary weights. With this pipeline, we overlap the fetching of the next input bitset with the XNOR_AND/popcount operations of the current input bitset, accelerating the computation.

During the computation, the input bitset address stored in the controller is incremented each cycle and sent to SRAM to load the next input bitset. It can be considered as setting up a DMA access from SRAM to registers. Therefore, the multiplication process with this fused computation pipeline is named as the DMA-based bitset multiply. The incremented address will be kept in the controller and will not be written back to *Ra* until all required input bitsets are fetched into *Rd*.

We introduce two instructions to program the pipeline: DMA-Based Bitset Multiply (DBM) and a configure instruction (DBM_CONF). They have the following formats:

- **DBM *Rd*, *Rn*, *Rm*, [*Ra*]**: activative the computation pipeline for bitset multiplication. *Rd* specifies the register which stores the loaded input bitsets and the temporary output value. *Rn* and *Rm* specify the operand registers storing the weight bitsets $\mathcal{B}_1$ and $\mathcal{B}_0$. *Ra* specifies the operand register which stores the starting address of the input bitsets.

- **DBM_CONF *Rn***: configure the number of input bits. *Rn* specifies the operand register.

For each layer, the DBM_CONF instruction is first called to configure the input precision which is recorded in the controller. After bitset transposing finishes (step (1) in Figure 4.12), the DBM instruction will be used for the bitset multiplication.

In the ALU block shown in Figure 4.14, the dashed arrows show the dataflow paths after we add the first two simple arithmetic instructions, and the solid arrows are the dataflow paths added for the DMA-based bitset multiply instructions.

The general-purpose registers of Arm Cortex-M4 microcontrollers have 3 read ports and 1 write port [75]. As shown in Figure 4.14, after all three computation stages are full filled, we need to write data into register *Rd*, and read from registers *Rd*, *Rn*, *Rm* in each cycle. In this case, the proposed architecture does not require more read or write ports for the registers.

The proposed three-stage computation pipeline can be tightly integrated into the existing Arm Cortex-M4 pipeline. We synthesize the integrated modules, and the results are listed in Table 4.1. Including the execution units of POPCOUNT and XNOR_AND, the total area overhead is only 3.8%.

We test the computation performance improvement from the DMA-based bitset multiply instructions on NIN. The results, annotated as *DMA*, are shown in Figure 4.13. The performance across most input precisions can be further improved. Higher improvement is observed for higher input precisions. For example, compared to the performance with the POPCOUNT and XNOR_AND instructions, the computation performance for 4-bit inputs and ternary weights is improved by 30%. For 8-bit inputs, the relative improvement increases to 73%. This is because, for inputs with higher precisions, the computation needs to process more input bitsets and, therefore, benefits more from the DMA-based bitset multiply instructions.

4.4.3 Double-Width DMA-Based Bitset Multiply

In the proposed computation pipeline, each input bitset is only used by one weight filter after being fetched. In this case, we introduce the double-width DMA-based bitset multiply to reuse the fetched input bitsets, which helps reduce memory access and improve computation performance. It keeps the same DMA-computation pipeline but processes one input bitset with the weight bitsets from *two* different weight filters in parallel, saturating available memory bandwidth.

The weight bitsets for the first output channel are loaded with the original load instruction in Arm v7-M ISA. To avoid adding extra read ports for the registers, the weight bitsets for the second output channel need to be loaded into the latches. The added latches (*Latch 2 and 3*) are shown as the dark shadowed blocks in Figure 4.14. Additionally, we extend the XNOR_AND and POPCOUNT units to support two parallel 32-bit computations. The POPCOUNT results are stored as two separate 16-bit values concatenated into a single 32-bit result in *Latch 0*. For the Shift and Add./Sub. units, we break the carry logic between the higher and lower 16 bits, but maintain the original 32-bit computation width. *Latch 1* and register *Rd* also remain 32-bit, storing

two concatenated 16-bit temporary output values. The hardware design for the DMA-based bitset multiply remains otherwise unchanged. In this case, supporting the double-width DMA-based bitset multiply instructions results in an area overhead of 6.4%.

For double-width DMA-based bitset multiply, two instructions are needed: Double-Width DMA-Based Bitset Multiply (DWDBM) and a setup load instruction (DWDBM_LOAD), with the following formats:

- **DWDBM** ***Rd*****,** ***Rn*****,** ***Rm*****, [*****Ra*****]**: identical to DBM except for the high-order 16 bits and low-order 16 bits of *Rd* storing the temporary outputs for the first and the second output channels, respectively. Weight bitsets for the second output channel are read from the latches.
- **DWDBM_LOAD** ***Ld*****,** ***Rn*****, [*****Ra*****]**: loads the weight bitset into the latch *Ld*. *Rn* and *Ra* record the input precision and the address of the weight bitsets for the second weight filter, respectively.

This extension yields improved computation performance across all input precisions on NIN, illustrated as *Double-Width* in Figure 4.13. With the bitset multiplication further accelerated, using 6-bit inputs has lower performance than 8-bit inputs due to the unpacking overhead in the bitset transpose operation.

4.5 Evaluation Methodology

4.5.1 Hardware and DNN Benchmarks

Arm Cortex-M series processors are typical embedded processors used in IoT sensor nodes. Therefore, we use Arm Cortex-M4 microcontroller as the test platform. The test board, STM32 Nucleo-F411RE [66], includes 128KB SRAM and 512KB flash memory. The clock frequency is 100MHz.

For DNN benchmarks, we include three CNN models (NIN [61], VGG-8 [24] and ResNet-20 [72]) on CIFAR-10 dataset [65] and one RNN model (CRNN [83]) on Google speech commands dataset [84]. Details are listed in Table 4.2. The networks chosen as benchmarks cover various

Table 4.2: DNN benchmarks.

Networks	Num of FMs/Neurons in Each Layer			32-bit FP Accuracy
	CONV	GRU	FC	
NIN	192-160-96-(192)x5-10	- -	- -	91.32%
VGG-8	(128)x2-(256)x2-(512)x2	- -	-1024x2-10	93.78%
ResNet-20	16-(16)x6-(32)x6-(64)x6	- -	-10	91.25%
CRNN	100	-136-136	-188-12	95.94%

layer types. NIN contains CONV layers with large (5×5) and small (1×1) filters. VGG-8 includes large FC layers, and ResNet-20 includes skip connections. CRNN contains two GRU layers.

4.5.2 Performance, Area, Energy Measurement

For performance measurement, we run all computation directly on the Arm Cortex-M4 microcontroller and profile the execution cycles. For configurations which require additional SRAM space, we run part of the computation to estimate the total execution time. To estimate the performance with the new instructions proposed in GoLo, we use existing instructions to hold the cycles for the proposed instructions and also perform the necessary memory access.

We synthesize the extra circuit modules using Synopsys Design Compiler (DC) under the IBM 45nm SC12 Standard Cell library to measure their area and energy costs. All the results are scaled up to 90nm which is the feature size of the Cortex-M4 CPU. The area cost of the 128KB SRAM is 1.58mm^2 which is estimated by CACTI [85].

For energy measurement, we use the STM32 Power Shield board [86] to measure the energy cost of the entire microcontroller including the SRAM and flash memory. To estimate the energy cost for the computation with the GoLo extensions, as described above, existing instructions are used to hold the computation cycles for the new instructions and also perform the necessary memory access. Also, the energy costs of extra circuit modules, which is generated by the synthesis, are used to adjust the energy numbers to get a better estimation.

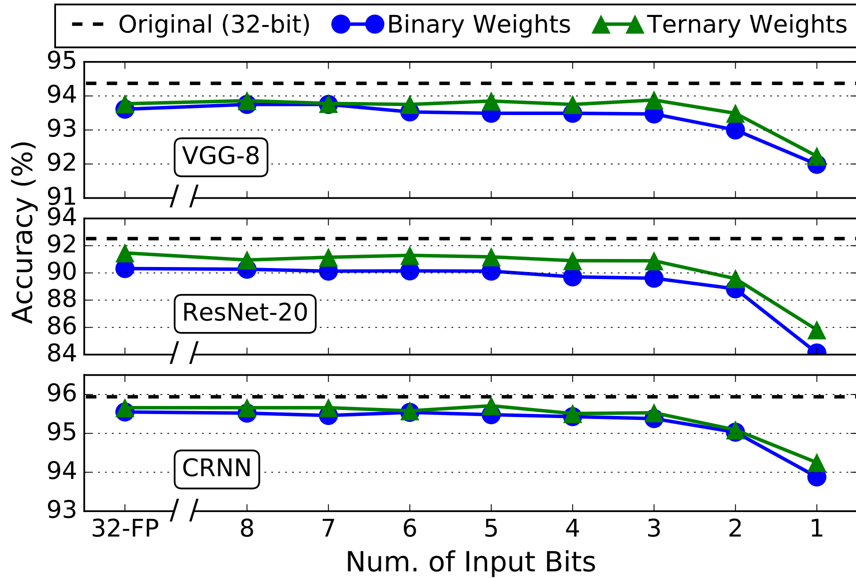


Figure 4.15: Accuracy of sub-byte VGG-8 and ResNet-20. The result for NIN is shown in Figure 4.3.

4.6 Evaluation Results

4.6.1 Sub-Byte DNN Accuracy

We first test the training framework on the DNN benchmarks. Figure 4.15 shows how the accuracies of VGG-8, ResNet-20 and CRNN change for different input precisions. The result of NIN is shown previously in Figure 4.3.

For all four networks, with ternary/binary weights, using input precisions≥4-bit will not cause a large accuracy reduction. However, when the input precision becomes lower than 4-bit, there is a sharp accuracy drop. 4-bit input precision can be considered as a balancing point which helps save much computation and storage and, at the same time, does not dramatically hurt network accuracy. In the following experiments, we focus on two configurations: (1) 4-bit inputs with ternary weights and (2) 2-bit inputs with binary weights.

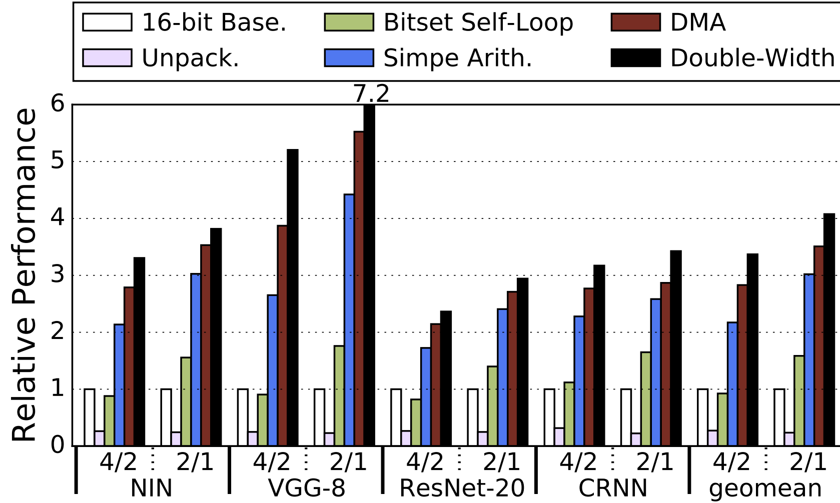


Figure 4.16: Relative performance of networks with 4-bit inputs/ternary weights(4/2) and 2-bit inputs/binary weights(2/1).

4.6.2 Computation Performance

Figure 4.16 shows the relative computation performance with 4-bit inputs/ternary weights and 2-bit inputs/binary weights. The baseline here is the performance with 16-bit inputs/weights. On average, bitset self-loop multiplication can improve the relative performance from 0.27x and 0.24x to 0.92x and 1.58x, respectively. With the GoLo extensions, the relative performance is further improved to 3.37x and 4.08x.

The GoLo extensions help achieve a much higher relative performance on VGG-8 compared to ResNet-20. This is because the layer sizes in ResNet-20 are much smaller. Each unpacked convolution window will not be used for many filters. The overhead of bitset transposing cannot be amortized much. Therefore, ResNet-20 suffers more from bitset transposing and has a lower relative performance than VGG-8.

4.6.3 Cost Utilization and Energy Efficiency

For microcontrollers, an important design metric is to provide higher performance while maintaining an ultra low cost. Since the manufacturing cost is tightly correlated to the chip area, we define the cost utilization as throughput per area, i.e., Giga operations per second (GOPS)/mm^2.

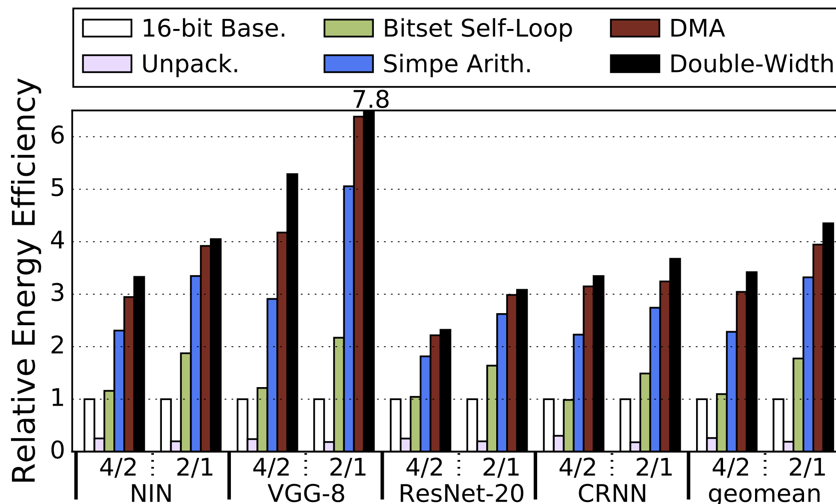


Figure 4.17: Energy efficiency of the tested networks.

Using 4-bit inputs and ternary weights, our pipeline achieves a cost utilization of 3.83 GOPS/mm^2. As a comparison, typical ASIC designs, UNPU [80] and Eyeriss [87], have the cost utilization of 25.2 GOPS/mm^2 and 1.61 GOPS/mm^2, respectively. These results are calculated under the 90nm process and 100MHz frequency. The SRAM area is excluded. UNPU uses 4-bit inputs/weights, and Eyeriss uses 16-bit. Compared to the ASIC designs, our proposed pipeline can help achieve a comparable cost utilization with negligible design efforts, although the performance is still much lower.

Figure 4.17 shows the relative energy efficiency for the tested networks. The energy efficiency is defined to be the number of images that can be processed per energy unit (frames/J). Using the conventional convolution algorithm for sub-byte inputs and weights will hurt the energy efficiency a lot. With the bitset self-loop convolution algorithm, for 4-bit inputs/ternary weights and 2-bit inputs/binary weights, we can achieve an energy efficiency higher than the 16-bit baseline. The GoLo extensions provide a significant energy efficiency improvement up to 7.8x. On average, the energy efficiency can be improved by 3.42x and 4.30x for 4-bit inputs/ternary weights and 2-bit inputs/binary weights, respectively.

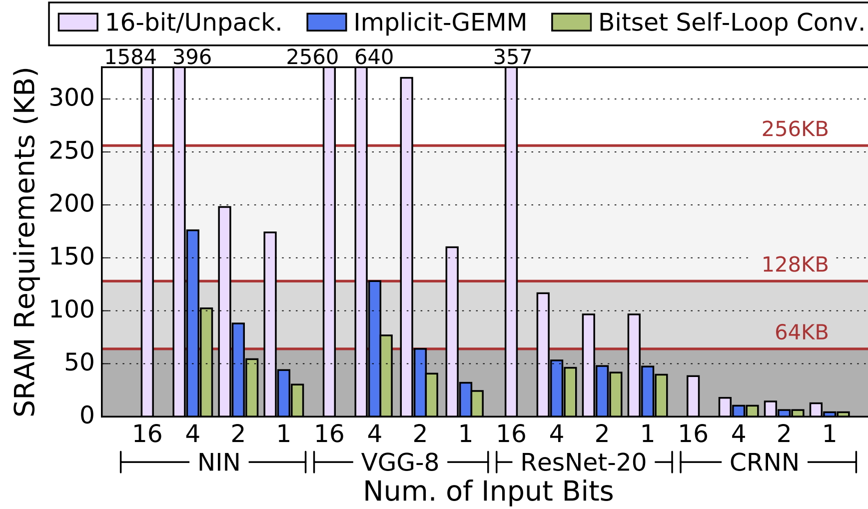


Figure 4.18: SRAM requirements of networks with different input precisions.

4.6.4 Storage Requirement

When running DNN models on microcontrollers, weights are stored in the flash memory as read-only data. The input values of each layer are generated through computation and, therefore, stored in SRAM. Flash memory has a much larger size than SRAM and can be expanded at a lower manufacturing cost. As such, we focus our discussion on the SRAM requirements, and the weight value storage in flash is not discussed.

Figure 4.18 shows the SRAM requirements for networks with different input precisions. With the conventional convolution algorithm, even using 1-bit inputs cannot reduce the SRAM requirements of NIN and VGG-8 lower than 128KB (the SRAM size of the test board). The optimized implicit-GEMM algorithm reduces the SRAM requirements but still requires more than 128KB when using 4-bit inputs. The bitset self-loop convolution cuts down the SRAM cost further, allowing NIN and VGG-8 to fit into the SRAM. For ResNet-20, we use 16-bit integers to store the feature maps for the skip connections to maintain high accuracy. Therefore, compared to NIN and VGG-8, the SRAM requirement reduction for ResNet-20 is smaller. CRNN requires much less SRAM than the CNN models, and even the 16-bit baseline can fit in the SRAM.

Another potential solution for the SRAM requirements is simply increasing the SRAM size in microcontrollers. However, it is likely to cause a notable increase in the manufacturing cost.

Table 4.3: Microcontroller (Micro.) die areas, including CPU, flash and SRAM, and normalized manufacturing (Manu.) costs for different SRAM sizes under 90nm.

SRAM Size (KB)	Micro. Area (mm^2)	Norm. Manu. Cost	% Area for SRAM	NIN/VGG-8/ResNet-20 4-bit Inputs Fit SRAM: Y(es)/N(o)		
				Unpack.	Implicit-GEMM	Bitset Self-loop
64	1.00	0.57×	81.9%	N/N/N	N/N/Y	N/N/Y
128	1.76	1.00×	89.7%	N/N/Y	N/N/Y	Y/Y/Y
256	3.28	1.86×	94.5%	N/N/Y	Y/Y/Y	Y/Y/Y
512	6.26	3.56×	97.1%	Y/N/Y	Y/Y/Y	Y/Y/Y
1024	12.57	7.14×	98.6%	Y/Y/Y	Y/Y/Y	Y/Y/Y

Table 4.3 shows the area and manufacturing costs for different SRAM sizes with a 90nm process. SRAM areas are estimated using CACTI [85]. A 2MB flash is also modeled, at 0.062mm^2 area [88]. With the conventional unpacking algorithm, 4-bit inputs require increasing the SRAM size to 1024KB, leading to a 7.14x increase in the manufacturing cost. In comparison, with the bitset self-loop convolution, all CNN models fit within the current 128KB SRAM size.

4.7 Summary

In this chapter, we propose a new pipeline to enable efficient sub-byte DNN inference on IoT microcontrollers. In bitset self-loop convolution, padded input feature maps are directly overwritten by output feature maps, and the multiplications between sub-byte values are replaced by bitwise logic operations between bitsets. For further acceleration, we propose a series of ISA extensions, including simple arithmetic instructions and microprogramming instructions. Compared to the 16-bit baseline, using 4-bit inputs and ternary weights, the computation performance can be improved by 3.37x, and the energy efficiency is increased by 3.42x.

CHAPTER 5

Pruning for Winograd-Domain Sparsity

5.1 Introduction

Pruning techniques and Winograd/FFT convolution are two typical methods to reduce the required computation of CNN models. Pruning removes redundant weight parameters, inducing sparsity into the network. On the other hand, Winograd convolution [14] and FFT convolution [52] transform the computation into different domains. The convolution operations can then be replaced by element-wise multiplications. For the typical convolution kernel size of 3×3, Winograd convolution can achieve more than twofold speedup over highly optimized spatial convolution algorithms, and typically requires less computation than FFT-based approaches [53]. Therefore, in this chapter, we focus on Winograd convolution.

For Winograd convolution, replacing convolution operations with element-wise multiplications can help reduce the required computation but need to perform more memory access. Since memory access is expensive on conventional processors, the increase of memory access will limit the performance benefit we can get from the computation reduction.

Pruning is a potential solution to remove unnecessary memory access. However, the pruning techniques and Winograd convolution are not directly compatible with each other. Sparse weight matrices, which are generated by pruning, lose most of the sparsity after the Winograd transformation from the spatial (original) domain to the Winograd domain. The remaining sparsity is much lower than what we need for improving the computation performance of DNN inferences.

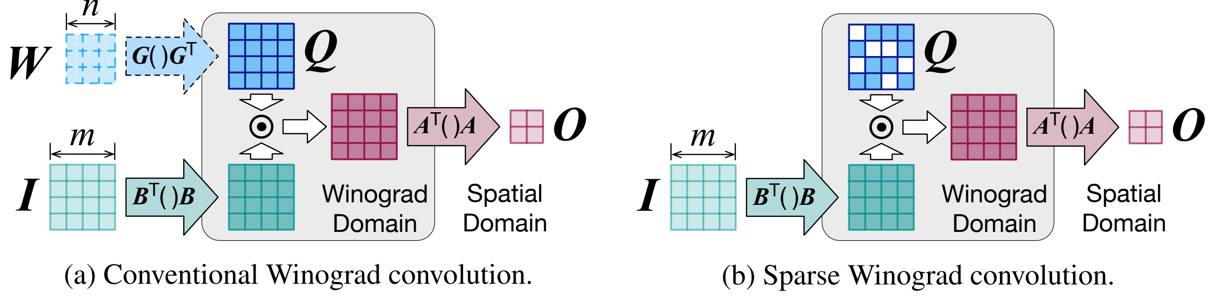
To increase the Winograd-domain sparsity, Li et al. [53] propose to perform pruning and retraining directly on Winograd-domain weights. However, it requires using an extremely small learning rate, e.g., 200x smaller for AlexNet, in retraining and is difficult to apply to deep networks. Besides, Winograd-ReLU pruning [55] moves ReLU function into the Winograd domain, which helps increase Winograd-domain sparsity but requires changes in the network structure.

In this chapter, to further improve the sparsity of Winograd-domain weights without changing the network structure, we propose a new pruning method, *spatial-Winograd pruning*. It includes two parts: *spatial structured pruning and Winograd direct pruning*.

In spatial structured pruning, we introduce a structured pruning method to transfer the spatial-domain sparsity into the Winograd domain efficiently. It can help avoid Winograd-domain retraining in this part and accelerate the pruning process. After spatial structured pruning, weights of the pruned layers will be converted to and kept in the Winograd domain.

As the second part, for Winograd direct pruning, we perform pruning and retraining entirely in the Winograd domain to improve the sparsity further. We present a new approach to measuring the importance of each Winograd-domain weight based on its impact on output activations. Also, we propose to use an importance factor matrix to adjust the gradients of Winograd-domain weights, which makes it much faster to retrain deep networks directly in the Winograd domain without changing the network structure.

Across three tested DNN models, spatial-Winograd pruning achieves the Winograd-domain sparsities of 73%, 69% and 74%, respectively, with $< 0.1\%$ accuracy loss. Compared with the baseline implementation using the conventional spatial convolution, the required computation can be reduced by $8.61\times$ on average. We also implement sparse Winograd convolution on an quad-core Arm Cortex-A53 processor. With spatial-Winograd pruning, the inference computation performance of the tested models is improved by $2.08\times$ on average.



(a) Conventional Winograd convolution. (b) Sparse Winograd convolution.

Figure 5.1: Conventional Winograd convolution and sparse Winograd convolution ($m = 4$, $n = 3$).

5.2 Background and Motivation

Winograd convolution [14] is a typical algorithm to reduce the arithmetic complexity of CNNs. It transforms the computation into the Winograd domain, and the convolution operations can then be replaced by element-wise multiplications. We call the domain, in which the conventional convolution operation is executed, to be the spatial domain.

The basic block of Winograd convolution works on a 2D input tile, $\boldsymbol{I}$, with a size of $m \times m$ and a 2D weight filter, $\boldsymbol{W}$, with a size of $n \times n$. In this case, the 2D output tile generated, $\boldsymbol{O}$, will have a size of $(m-n+1) \times (m-n+1)$. For a typical convolutional layer, the input feature maps are first disassembled into input tiles and, after the Winograd convolution, the output tiles will be reassembled into the output feature maps.

Figure 5.1a shows how conventional Winograd convolution works. As the first step, the weight filter $\boldsymbol{W}$ and the input tile $\boldsymbol{I}$ are converted into the Winograd domain using the predefined matrices $\boldsymbol{G}$ and $\boldsymbol{B}$. Element-wise multiplication is then applied to the Winograd-domain weight filter, $\boldsymbol{G}\boldsymbol{W}\boldsymbol{G}^\top$, and input tile, $\boldsymbol{B}^\top\boldsymbol{I}\ \boldsymbol{B}$, to generate the Winograd-domain output tile with a size of $m \times m$. In the last step, the output tile is converted back into the spatial domain with another predefined matrix $\boldsymbol{A}$. With $\odot$ as the Hadamard (element-wise) product, the entire process can be written as

$$\boldsymbol{O} = \boldsymbol{A}^\top[(\boldsymbol{G}\boldsymbol{W}\boldsymbol{G}^\top) \odot (\boldsymbol{B}^\top\boldsymbol{I}\ \boldsymbol{B})]\boldsymbol{A} \tag{5.1}$$

The transform/inverse-transform matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{G}$ are only determined by m and n. These matrices contain many repeating elements and applying them requires only few multiplications.

In this case, considering only the element-wise multiplication between $\boldsymbol{GWG}^\top$ and $\boldsymbol{B}^\top\boldsymbol{I}\ \boldsymbol{B}$, the Winograd convolution can reduce the number of multiplications from $(m-n+1)^2n^2$ to m^2. However, the number of data values need to be fetched, including inputs, outputs and weights, increases from $m^2+(m-n+1)^2+n^2$ to $3m^2$.

FFT-based convolution is another typical domain transformation technique. However, the FFT-based method requires using complex numbers. In comparison, Winograd convolution works with real numbers and requires much fewer operations, especially for small kernels (e.g., 3×3). Since more networks are using small kernels, this chapter focuses on Winograd convolution.

In addition to Winograd convolution, pruning is also a well-explored method to reduce CNN computation. Han et al. [56] propose to perform pruning and retraining iteratively, which can help reduce the computation by up to $5\times$. To fully utilize the sparsity incurred by pruning to accelerate CNN computation, Wen et al. [89] and Yu et al. [73] prune networks in a structured way: weights are clustered into groups with hardware-friendly structures and then get pruned in groups.

However, Winograd convolution is not directly compatible with conventional pruning algorithms. The transformation $\boldsymbol{GWG}^\top$ fills in the zeros in the sparse weight filters generated by pruning. There have been several research attempts to solve this problem.

Liu and Turakhia [54] propose to directly mask out Winograd-domain weights and use backpropagation to train the spatial-domain weights. However, the linear transformation mapping spatial-domain weights to Winograd-domain weights is non-invertible as there are more Winograd-domain weights than spatial-domain. Directly setting Winograd-domain weights to zero will cause an inconsistency between the spatial domain and the Winograd domain. This inconsistency will lead to a significant accuracy loss or a low sparsity on networks, e.g., AlexNet, for large datasets [53].

To address the inconsistency between the spatial and Winograd domains, Li et al. [53] propose the sparse Winograd convolution. Figure. 5.1b shows how it works. Weight values are stored in the Winograd domain instead of the spatial domain. Both pruning and retraining are applied directly to Winograd-domain weights. This native pruning algorithm achieves $>90\%$ sparsity on

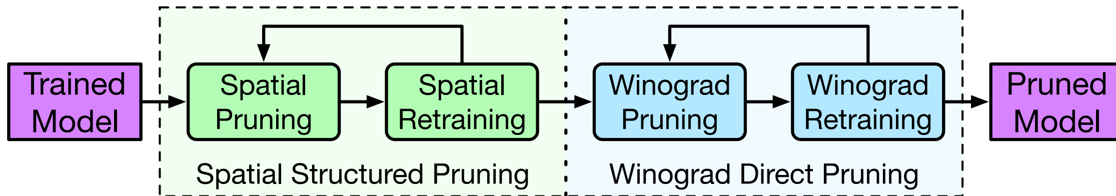


Figure 5.2: Overview of the spatial-Winograd pruning.

AlexNet [62] but cannot provide a high sparsity for deep networks [55]. Also, direct retraining in the Winograd domain requires an extremely small learning rate, e.g., 200x smaller for AlexNet, which makes the retraining much slower.

Based on the sparse Winograd convolution, Liu et al. [55] introduce the Winograd-ReLU pruning. It moves the ReLU function from the spatial domain into the Winograd domain. In this case, the computation of Winograd convolution becomes

$$\boldsymbol{O} = \boldsymbol{A}^{\top}[(\boldsymbol{G}\boldsymbol{W}\boldsymbol{G}^{\top}) \odot \mathrm{ReLU}(\boldsymbol{B}^{\top}\boldsymbol{I}\,\boldsymbol{B})]\boldsymbol{A} \tag{5.2}$$

The Winograd-domain inputs also become sparse, which helps further reduce the required computation. Besides, a higher sparsity in Winograd-domain weights can be achieved. However, with weight filters being sparse, it is challenging to utilize both the weight and input sparsities for CNN acceleration on general-purpose processors due to more irregularity in the access pattern and control flow. Also, it is not easy to directly apply Winograd-ReLU pruning to conventional CNN models since the new computation in Equation 5.2 does not correspond to the original convolution operation. It requires changing the network structure and retraining the network from scratch.

5.3 Spatial-Winograd Pruning

In this chapter, to achieve a high Winograd-domain weight sparsity on deep CNN models without changing network structures, we propose the *spatial-Winograd pruning*. As shown in Figure 5.2, it consists of two parts: spatial structured pruning and Winograd direct pruning.

In spatial structured pruning, spatial-domain weights are pruned in a structured way and then

retrained to regain the original accuracy. After spatial structured pruning, the weights of the pruned model will be transferred into and kept in the Winograd domain. Winograd direct pruning then performs pruning and retraining directly onto the weights in the Winograd domain. The pruning and retraining steps in both spatial structured pruning and Winograd direct pruning will be iteratively executed until we achieve the desired sparsity or the produced model loses much accuracy.

5.3.1 Spatial Structured Pruning

The first part of the spatial-Winograd pruning is spatial structured pruning. Spatial-domain weights which affect the same Winograd-domain weight are clustered into the same group. Less important weight groups are removed, and the pruned network will be retrained to regain accuracy. This structured pruning method helps transfer more spatial-domain sparsity into the Winograd domain.

Spatial Pruning For each spatial-domain filter $\boldsymbol{W}$, we need to generate a mask $\boldsymbol{M}^{spatial}$ to indicate the redundant weights. Assuming $\boldsymbol{W}$ has a size of $n \times n$ and the Winograd-domain filter $\boldsymbol{Q}$ is $m \times m$, we have $\boldsymbol{Q} = \boldsymbol{G}\boldsymbol{W}\boldsymbol{G}^{\top}$. Each element of the Winograd-domain filter, $\boldsymbol{Q}_{i,j}$, is the weighted sum of the spatial-domain weights:

$$\boldsymbol{Q}_{i,j} = \sum_{0 \leqslant u,v \leqslant n-1} (\boldsymbol{S}_{i,j,u,v} \cdot \boldsymbol{W}_{u,v}) \qquad 0 \leqslant i,j \leqslant m-1 \tag{5.3}$$

where $\boldsymbol{S}$ is a 4D tensor containing the weight coefficients of the spatial-domain weights and is only determined by m and n.

To calculate the tensor $\boldsymbol{S}$, we first introduce an equivalent transformation: for two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ with a size of m, and a matrix $\boldsymbol{C}$ with a size of $m \times m$, we have

$$\boldsymbol{a}^{\top}\boldsymbol{C}\boldsymbol{b} = \vec{\mathbf{1}}^{\top}[(\boldsymbol{a}\boldsymbol{b}^{\top}) \odot \boldsymbol{C}]\,\vec{\mathbf{1}} \tag{5.4}$$

where $\vec{\mathbf{1}}$ is a vector of size m and all its entries are 1.

For the spatial-domain weight filter $\boldsymbol{W}$, with the weight transform matrix $\boldsymbol{G}$, we have $\boldsymbol{Q} =$

$\boldsymbol{G}\boldsymbol{W}\boldsymbol{G}^\top$. Each element in the Winograd-domain weight filter $\boldsymbol{Q}$ is calculated as

$$\begin{aligned}\boldsymbol{Q}_{i,j} &= \boldsymbol{G}_{i,:} \cdot \boldsymbol{W} \cdot (\boldsymbol{G}_{j,:})^\top \\ &= \vec{\mathbf{1}}^{\top}\big[\big((\boldsymbol{G}_{i,:})^\top \boldsymbol{G}_{j,:}\big) \odot \boldsymbol{W}\big]\ \vec{\mathbf{1}} \\ &= \sum_{0\leqslant u,v\leqslant n-1} (\boldsymbol{G}_{i,u} \cdot \boldsymbol{G}_{j,v} \cdot \boldsymbol{W}_{u,v})\end{aligned} \tag{5.5}$$

where $0 \leqslant i, j \leqslant m-1$. In this case, compared with Equation 5.3, each element in the coefficient tensor $\boldsymbol{S}$ can be calculated as

$$\boldsymbol{S}_{i,j,u,v} = \boldsymbol{G}_{i,u} \cdot \boldsymbol{G}_{j,v} \tag{5.6}$$

where $0 \leqslant i, j \leqslant m-1, 0 \leqslant u, v \leqslant n-1$.

For each Winograd-domain weight $\boldsymbol{Q}_{i,j}$, we can create a set $\mathbb{D}_{i,j}$ containing the spatial-domain weights which affect the value of $\boldsymbol{Q}_{i,j}$. $\mathbb{D}_{i,j}$ is defined as

$$\mathbb{D}_{i,j} = \{\boldsymbol{W}_{u,v} \mid \boldsymbol{S}_{i,j,u,v} \neq 0,\ 0 \leqslant u, v \leqslant n-1\} \tag{5.7}$$

In this case, for each weight group $\mathbb{D}_{i,j}$, we use a function $h(\mathbb{D}_{i,j})$ to measure its importance. In this chapter, we use the maximum norm function as h

$$h(\mathbb{D}_{i,j}) = \max(\{\ |\boldsymbol{W}_{u,v}| \mid \boldsymbol{W}_{u,v} \in \mathbb{D}_{i,j}\}) \tag{5.8}$$

With a specific threshold $t^{spatial}$, if $h(\mathbb{D}_{i,j}) < t^{spatial}$, then $\mathbb{D}_{i,j}$ is considered as redundant and all weights included need to be removed. In this case, the corresponding $\boldsymbol{Q}_{i,j}$ will be fixed to 0 and also removed. The set of redundant weights for entire $\boldsymbol{W}$ is the union of all redundant $\mathbb{D}_{i,j}$ and can be calculated as

$$\mathbb{D} = \bigcup_{0\leqslant i,j\leqslant m-1, h(\mathbb{D}_{i,j})<t^{spatial}} \mathbb{D}_{i,j} \tag{5.9}$$

Here we define $\mathbb{D}_{i,j}$ in a structured way based on the relation between spatial-domain weights and Winograd-domain weights. It helps transfer as much spatial-domain sparsity into Winograd-

domain sparsity as possible. The mask matrix $\boldsymbol{M}^{spatial}$ can be generated by

$$\boldsymbol{M}_{u,v}^{spatial} = \begin{cases} 0 & \boldsymbol{W}_{u,v} \in \mathbb{D} \\ 1 & \boldsymbol{W}_{u,v} \notin \mathbb{D} \end{cases} \qquad 0 \leqslant u, v \leqslant n-1 \tag{5.10}$$

Spatial Retraining After spatial pruning, we can perform the spatial retraining with conventional training algorithms, e.g., stochastic gradient descent (SGD). The removed weights are fixed to 0 by applying $\boldsymbol{W} = \boldsymbol{W} \odot \boldsymbol{M}^{spatial}$ after each training iteration. $\odot$ is element-wise multiplication. The steps of spatial pruning and spatial retraining will be iteratively performed until the retrained model loses much accuracy. The threshold $t^{spatial}$ is gradually increased to incur more sparsity into the network.

In spatial structured pruning, both pruning and retraining steps are performed in the spatial domain. It helps avoid the Winograd-domain retraining to accelerate the pruning process but, at the same time, incurs high Winograd-domain sparsity.

5.3.2 Winograd Direct Pruning

After spatial structured pruning, as in sparse Winograd convolution, weights of the pruned model will be transferred into and kept in the Winograd domain. In Winograd direct pruning, we measure the importance of each weight based on its impact on output activations, and unimportant weights are removed. The pruned network is then retrained in the Winograd domain, for which an importance factor matrix is deployed to adjust the weight gradients.

Winograd Pruning Similar to spatial pruning, in Winograd pruning, we need to generate a mask matrix $\boldsymbol{M}^{Winograd}$ for each Winograd-domain filter $\boldsymbol{Q}$ to indicate the redundant weights. With the weight filter $\boldsymbol{Q}$ in the Winograd domain, the output tile $\boldsymbol{O}$ is calculated as

$$\boldsymbol{O} = \boldsymbol{A}^{\top}[\boldsymbol{Q} \odot (\boldsymbol{B}^{\top}\boldsymbol{I}\ \boldsymbol{B})]\boldsymbol{A} \tag{5.11}$$

Each output element can be considered as the weighted sum of the products of weights and inputs

$$\boldsymbol{O}_{x,y} = \sum_{0 \leqslant i,j,s,t \leqslant m-1} (\boldsymbol{H}_{x,y,i,j,s,t} \cdot \boldsymbol{Q}_{i,j} \cdot \boldsymbol{I}_{s,t}) \qquad 0 \leqslant x, y \leqslant m-n \tag{5.12}$$

where $\boldsymbol{H}$ is a 6D tensor containing the weight coefficients of different products ($\boldsymbol{Q}_{i,j} \cdot \boldsymbol{I}_{s,t}$) and is only determined by m and n.

Tensor $\boldsymbol{H}$ can be calculated using a similar strategy as calculating tensor $\boldsymbol{S}$. With the Winograd-domain weight filter $\boldsymbol{Q}$, the output tile $\boldsymbol{O}$ is calculated as

$$\boldsymbol{O} = \boldsymbol{A}^\top [\boldsymbol{Q} \odot (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})] \boldsymbol{A} \tag{5.13}$$

Each element $\boldsymbol{O}_{x,y}$ is calculated as

$$\boldsymbol{O}_{x,y} = (\boldsymbol{A}_{:,x})^\top [\boldsymbol{Q} \odot (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})] \boldsymbol{A}_{:,y} \tag{5.14}$$

where $0 \leqslant x, y \leqslant m-n$. Based on Equation 5.4, we have

$$\boldsymbol{O}_{x,y} = \vec{\boldsymbol{1}}^{\ \top} [(\boldsymbol{A}_{:,x} (\boldsymbol{A}_{:,y})^\top) \odot \boldsymbol{Q} \odot (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})]\ \vec{\boldsymbol{1}} \tag{5.15}$$

Let $\boldsymbol{V} = (\boldsymbol{A}_{:,x} (\boldsymbol{A}_{:,y})^\top) \odot \boldsymbol{Q} \odot (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})$, then

$$\boldsymbol{V}_{i,j} = (\boldsymbol{A}_{i,x} \boldsymbol{A}_{j,y}) \cdot \boldsymbol{Q}_{i,j} \cdot (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})_{i,j} \tag{5.16}$$

where $0 \leqslant i, j \leqslant m-1$. The element $(\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})_{i,j}$ can be calculated as

$$\begin{aligned} (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})_{i,j} &= (\boldsymbol{B}_{:,i})^\top \boldsymbol{I}\ \boldsymbol{B}_{:,j} \\ &= \vec{\boldsymbol{1}}^{\ \top} [(\boldsymbol{B}_{:,i} (\boldsymbol{B}_{:,j})^\top) \odot \boldsymbol{I}]\ \vec{\boldsymbol{1}} \\ &= \sum_{0 \leqslant s,t \leqslant m-1} [(\boldsymbol{B}_{s,i} \boldsymbol{B}_{t,j}) \cdot \boldsymbol{I}_{s,t}] \end{aligned} \tag{5.17}$$

Based on Equation 5.15, 5.16 and 5.17, we have

$$\begin{aligned}
\boldsymbol{O}_{x,y} &= \sum_{0\leqslant i,j\leqslant m-1} \boldsymbol{V}_{i,j} \\
&= \sum_{0\leqslant i,j\leqslant m-1} [(\boldsymbol{A}_{i,x}\boldsymbol{A}_{j,y}) \cdot \boldsymbol{Q}_{i,j} \cdot (\boldsymbol{B}^\top \boldsymbol{I}\ \boldsymbol{B})_{i,j}] \\
&= \sum_{0\leqslant i,j\leqslant m-1} \Big[(\boldsymbol{A}_{i,x}\boldsymbol{A}_{j,y}) \cdot \boldsymbol{Q}_{i,j} \cdot \sum_{0\leqslant s,t\leqslant m-1} [(\boldsymbol{B}_{s,i}\boldsymbol{B}_{t,j}) \cdot \boldsymbol{I}_{s,t}]\Big] \\
&= \sum_{0\leqslant i,j,s,t\leqslant m-1} [(\boldsymbol{A}_{i,x}\boldsymbol{A}_{j,y}\boldsymbol{B}_{s,i}\boldsymbol{B}_{t,j}) \cdot \boldsymbol{Q}_{i,j} \cdot \boldsymbol{I}_{s,t}]
\end{aligned} \tag{5.18}$$

Therefore, compared with Equation 5.12, each element in the coefficient tensor $\boldsymbol{H}$ is calculated as

$$\boldsymbol{H}_{x,y,i,j,s,t} = \boldsymbol{A}_{i,x}\boldsymbol{A}_{j,y}\boldsymbol{B}_{s,i}\boldsymbol{B}_{t,j} \tag{5.19}$$

where $0 \leqslant x, y \leqslant m - n, 0 \leqslant i, j, s, t \leqslant m - 1$.

With tensor $\boldsymbol{H}$ calculated, by removing one weight $\boldsymbol{Q}_{i,j}$, the change on each output $\boldsymbol{O}_{x,y}$ is

$$\Delta \boldsymbol{O}_{x,y}|_{\boldsymbol{Q}_{i,j}} = -1 \cdot \sum_{0\leqslant s,t\leqslant m-1} (\boldsymbol{H}_{x,y,i,j,s,t} \cdot \boldsymbol{Q}_{i,j} \cdot \boldsymbol{I}_{s,t}) \qquad 0 \leqslant x, y \leqslant m - n \tag{5.20}$$

In Winograd pruning, we need to remove a certain amount of weights while minimizing the change of the output activations $||\Delta \boldsymbol{O}||_2$. Removing an important weight will lead to a larger change in output activations. Therefore, we propose to measure the importance of each weight $\boldsymbol{Q}_{i,j}$ by the expected value of $||\Delta \boldsymbol{O}|_{\boldsymbol{Q}_{i,j}}||_2^2$. In this case, we have

$$\begin{aligned}
E(||\Delta \boldsymbol{O}|_{\boldsymbol{Q}_{i,j}}||_2^2) &= E\Big(\sum_{0\leqslant x,y\leqslant m-n} \Big[\sum_{0\leqslant s,t\leqslant m-1} (\boldsymbol{H}_{x,y,i,j,s,t} \cdot \boldsymbol{Q}_{i,j} \cdot \boldsymbol{I}_{s,t})\Big]^2\Big) \\
&= \boldsymbol{Q}_{i,j}^2 \cdot \Big\{ \sum_{0\leqslant x,y\leqslant m-n, 0\leqslant s,t\leqslant m-1} \big[\boldsymbol{H}_{x,y,i,j,s,t}^2 \cdot E(\boldsymbol{I}_{s,t}^2)\big] + \\
&\qquad \sum_{\substack{0\leqslant x,y\leqslant m-n \\ 0\leqslant s,t,s',t'\leqslant m-1 \\ (s,t)\neq(s',t')}} \big[\boldsymbol{H}_{x,y,i,j,s,t} \cdot \boldsymbol{H}_{x,y,i,j,s',t'} \cdot E(\boldsymbol{I}_{s,t} \cdot \boldsymbol{I}_{s',t'})\big] \Big\}
\end{aligned} \tag{5.21}$$

For simplicity, we can assume input values are independent and identically distributed (i.i.d.), and have expected values of 0. With this assumption, we have

$$E(\boldsymbol{I}_{s,t} \cdot \boldsymbol{I}_{s',t'}) = E(\boldsymbol{I}_{s,t}) \cdot E(\boldsymbol{I}_{s',t'}) = 0 \qquad (s,t) \neq (s',t') \tag{5.22}$$

Since the importance of weights are relative numbers, we can assume $E(\boldsymbol{I}_{s,t}^2) = 1$. In this case,

$$E(||\Delta \boldsymbol{O}|_{\boldsymbol{Q}_{i,j}}||_2^2) = \boldsymbol{Q}_{i,j}^2 \cdot \sum_{0 \leqslant x,y \leqslant m-n, 0 \leqslant s,t \leqslant m-1} \boldsymbol{H}_{x,y,i,j,s,t}^2 \tag{5.23}$$

Based on Equation 5.23, we can generate an importance factor matrix $\boldsymbol{F}$, where

$$\boldsymbol{F}_{i,j} = \sqrt{\frac{E(||\Delta \boldsymbol{O}|_{\boldsymbol{Q}_{i,j}}||_2^2)}{\boldsymbol{Q}_{i,j}^2}} = \sqrt{\sum_{0 \leqslant x,y \leqslant m-n, 0 \leqslant s,t \leqslant m-1} \boldsymbol{H}_{x,y,i,j,s,t}^2} \qquad 0 \leqslant i,j \leqslant m-1 \tag{5.24}$$

Therefore, $\boldsymbol{F}$ is only determined by m and n, and keeps the same for all 2D Winograd-domain filters $\boldsymbol{Q}$ in a specific layer. Then Equation. 5.23 can be simplified to

$$E(||\Delta \boldsymbol{O}|_{\boldsymbol{Q}_{i,j}}||_2^2) = \boldsymbol{Q}_{i,j}^2 \cdot \boldsymbol{F}_{i,j}^2 \tag{5.25}$$

In this case, with a specific threshold $t^{Winograd}$, we can generate the mask matrix $\boldsymbol{M}^{Winograd}$ as

$$\boldsymbol{M}_{i,j}^{Winograd} = \begin{cases} 0 & \boldsymbol{Q}_{i,j}^2 \cdot \boldsymbol{F}_{i,j}^2 < t^{Winograd} \\ 1 & \boldsymbol{Q}_{i,j}^2 \cdot \boldsymbol{F}_{i,j}^2 \geqslant t^{Winograd} \end{cases} \qquad 0 \leqslant i,j \leqslant m-1 \tag{5.26}$$

For a specific weight $\boldsymbol{Q}_{i,j}$, conventional pruning algorithms [18, 19] use its absolute value $|\boldsymbol{Q}_{i,j}|$ as the weight importance, which is equivalent to using $\boldsymbol{Q}_{i,j}^2$. Therefore, in Equation 5.26, the employed weight importance, $\boldsymbol{Q}_{i,j}^2 \cdot \boldsymbol{F}_{i,j}^2$, can be considered as using the importance factor matrix $\boldsymbol{F}$ to adjust the conventional weight importance $\boldsymbol{Q}_{i,j}^2$.

Winograd Retraining As the same with the spatial retraining, we fix the removed Winograd-domain weights to 0 by applying $\boldsymbol{Q} = \boldsymbol{Q} \odot \boldsymbol{M}^{Winograd}$ after each training iteration.

However, using conventional SGD to retrain the Winograd-domain parameters will lead to divergence. This is because, as shown in Equation. 5.24, different locations of Winograd-domain weights have different importance and, therefore, require different learning speeds. Using an extremely small learning rate can avoid the divergence but makes the retraining much slower.

To address this problem, in Winograd retraining, we propose to adjust the gradients of Winograd-domain weights with the importance factor matrix $\boldsymbol{F}$. Assume $loss$ to be the loss value. At the training step k, after the backward computation, the gradients of $\boldsymbol{Q}$, $\frac{\partial\, loss}{\partial \boldsymbol{Q}}\big|_k$, will be adjusted by

$$\left(\frac{\partial\, loss}{\partial \boldsymbol{Q}}\bigg|_k\right)^{adjusted} = \frac{\partial\, loss}{\partial \boldsymbol{Q}}\bigg|_k \oslash \boldsymbol{F}^{\circ\alpha} \tag{5.27}$$

where $\oslash$ and $\circ\alpha$ are the Hadamard (element-wise) division and power functions, respectively. In this chapter, based on empirical results, α is fixed to 1.5. In this case, with the learning rate of η, the SGD update for the Winograd-domain weights $\boldsymbol{Q}$ at the training step k becomes

$$\boldsymbol{Q}|_{k+1} = \boldsymbol{Q}|_k - \eta \cdot \left(\frac{\partial\, loss}{\partial \boldsymbol{Q}}\bigg|_k \oslash \boldsymbol{F}^{\circ\alpha}\right) \tag{5.28}$$

5.4 Experiments

To evaluate the spatial-Winograd pruning, we perform the experiments on three datasets: CIFAR-10, CIFAR-100 [65] and ImageNet (ILSVRC-2012) [7]. PyTorch [90] is used to implement the pruning framework.

We use the Winograd-ReLU pruning [55] as the baseline pruning technique. To show the effectiveness of our proposed method, we test the same models as in Winograd-ReLU pruning: VGG-nagadomi [91], ConvPool-CNN-C [92] and ResNet-18 [72] on the three datasets tested, respectively. Those models are chosen since the majority of the included convolutional layers use 3×3 kernels.

For 3×3 kernels, we set the input tile size m to 6 instead of 4. A larger input tile size can help achieve higher computation speedup. With our proposed method, we expect that lower input

tile sizes can lead to a similar or higher sparsity. This is because, with lower input tile sizes, fewer spatial-domain weights are correlated with each other and the spatial structured pruning can achieve a higher sparsity.

5.4.1 CIFAR-10 and CIFAR-100

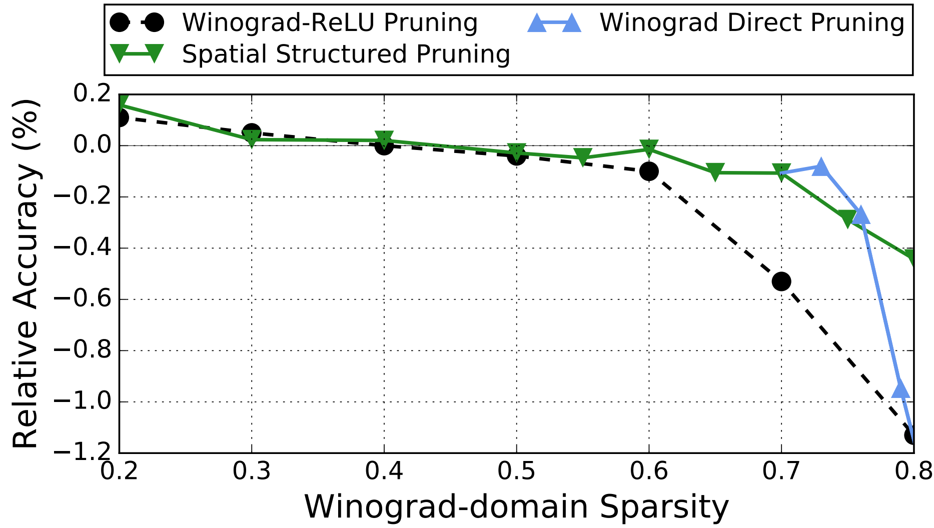
For the CIFAR-10 and CIFAR-100 datasets, we test VGG-nagadomi and ConvPool-CNN-C, respectively. VGG-nagadomi contains 8 convolutional layers with 3×3 kernels, and has an original validation accuracy of 93.96%. ConvPool-CNN-C contains 9 convolutional layers, in which 7 layers use 3×3 kernels. It achieves a validation accuracy of 69.88%. For both models, we prune the first convolutional layers with a fixed Winograd-domain sparsity of 20%. For the remaining convolutional layers with 3×3 kernels, we incur a uniform Winograd-domain sparsity, increasing from 20% to 80%, for simplicity.

Figure 5.3 shows the pruning results. The baseline results reported in [55] are shown as the dashed lines. For VGG-nagadomi and ConvPool-CNN-C , with $<0.1\%$ accuracy loss, the baseline Winograd-ReLU pruning achieves the Winograd-domain sparsities of 60% and 40%, respectively. With spatial-Winograd pruning, we can now increase the Winograd-domain sparsities to 73% and 69%, respectively. Meanwhile, spatial-Winograd pruning does not require changing the network structure.

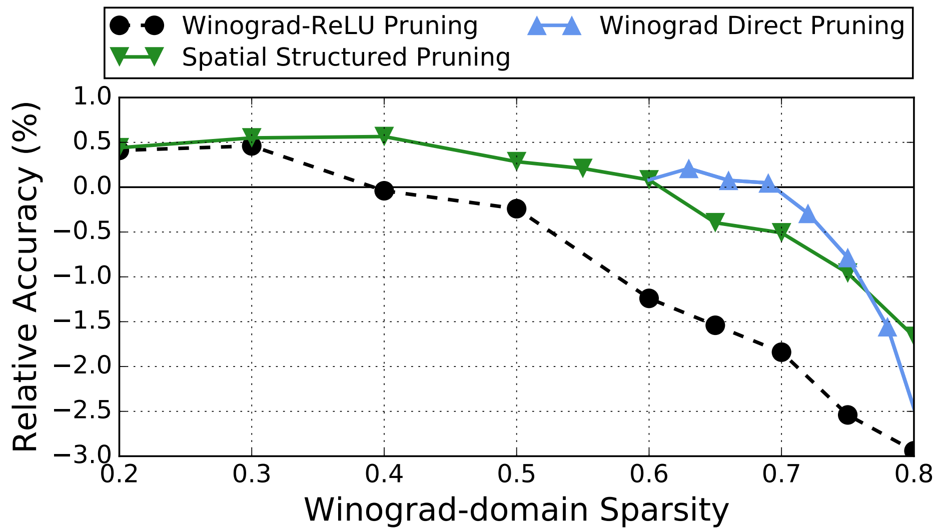
5.4.2 ImageNet

We test the ResNet-18 model on the ImageNet (ILSVRC-2012) dataset. As the same with Winograd-ReLU pruning, we replace each 2×2-stride 3×3 convolutional layer with a 2×2-stride max-pooling layer followed by a 1×1-stride 3×3 convolutional layer. This modification makes it easier to apply Winograd convolution on most of the convolutional layers.

The original model has a top-1/top-5 prediction accuracy of 69.82%/89.55%. However, for Winograd-ReLU pruning, Liu et al. [55] use the model with the original top-1/top-5 accuracy of only 66.67%/87.42%. Despite this, we still use the accuracy values and achieved sparsities



(a) VGG-nagadomi on CIFAR-10.

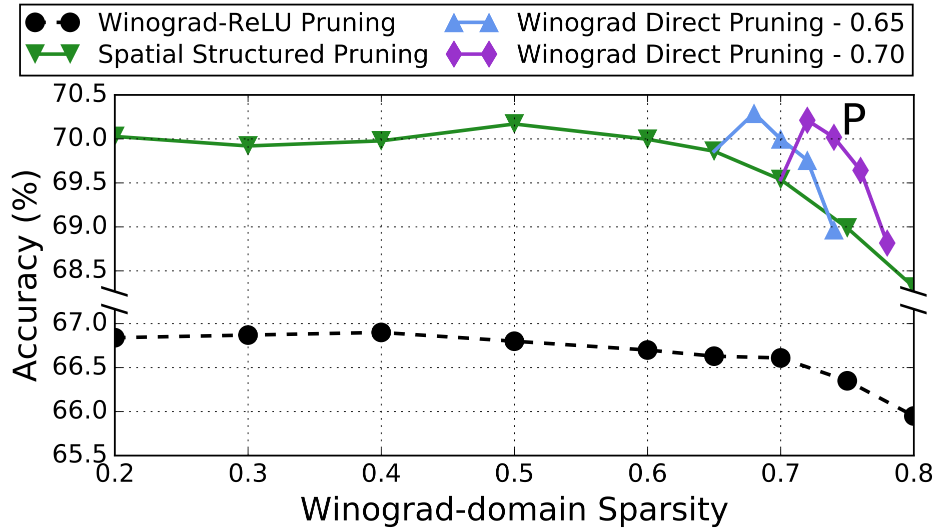


(b) ConvPool-CNN-C on CIFAR-100.

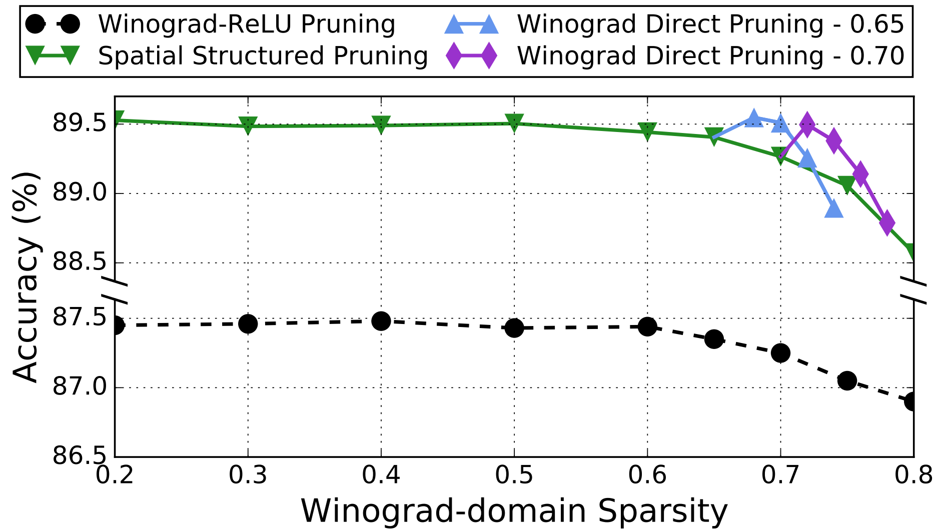
Figure 5.3: Pruning of (a) VGG-nagadomi on CIFAR-10 (b) ConvPool-CNN-C on CIFAR-100 with uniform sparsity across the pruned layers.

reported in [55] as the baseline. We prune the 16 convolutional layers in the residual blocks with the same Winograd-domain sparsity. The first convolutional layer and the downsample layers are kept intact. Figure 5.4 shows the results of the top-1/top-5 accuracy against the Winograd-domain sparsity. As the dashed lines show, Winograd-ReLU pruning achieves a sparsity of 70%/65% with $<0.1\%$ top-1/top-5 accuracy loss.

We apply Winograd direct pruning to the models pruned by spatial structured pruning with



(a) Top-1 accuracy against sparsity.



(b) Top-5 accuracy against sparsity.

Figure 5.4: Pruning of ResNet-18 on ImageNet with uniform sparsity across the pruned layers.

65% and 70% Winograd-domain sparsity, annotated as Winograd direct pruning - 0.65 and 0.70, respectively. As shown in the figure, with $<0.1\%$ top-1/top-5 accuracy loss, applying Winograd direct pruning to the model with 70% Winograd-domain sparsity can achieve a higher sparsity of 74%/72%. This is because, with the sparsity increasing, Winograd direct pruning makes the prediction accuracy drop much faster than spatial structured pruning. Although we can use the importance factor matrix $\boldsymbol{F}$ to adjust the weight gradients to accelerate the Winograd-domain retraining, the learning rate still needs to be much lower than in the spatial retraining. In this case,

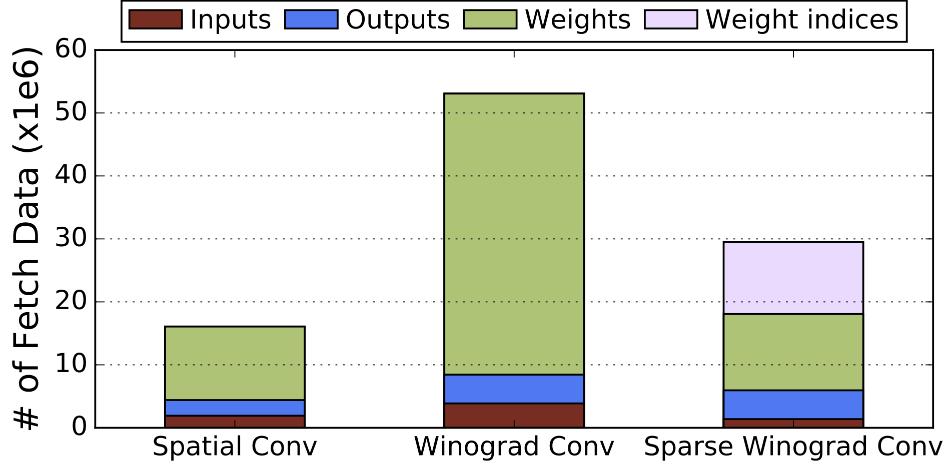


Figure 5.5: Number of data values that need to be accessed, including weights, inputs and outputs for ResNet-18 computation.

the accuracy loss recovered through Winograd retraining is limited, which makes the accuracy drop much faster when applying Winograd direct pruning.

For ResNet-18, we also calculate the numbers of data values that need to be accessed, including weights, inputs and outputs for different convolution algorithms. Figure 5.5 shows the comparison. For Winograd convolution, we only include the data accessed for the element-wise multiplications. As shown in the figure, although Winograd convolution help reduce the computation, the data access is increased by 3.30x. With spatial-Winograd pruning, the data access is reduced by 44.4% to 1.83x.

5.4.3 Sparsity Distribution

For the pruned ResNet-18 model, we analyze more detailed sparsity distribution across and inside 2D weight filters. Here each Winograd-domain weight matrix $\boldsymbol{Q}$ is considered as a 2D filter. We use the last convolutional layer (7-b) as an example. The model with a uniform sparsity of 74% across all pruned layers, which corresponds to point P in Figure 5.4a, is tested. Figure 5.6 shows the sparsity distribution across the filters. As shown in the figure, more than half (62%) of the filters have all weights removed. An interesting observation is that a large portion of the filters have exact 20 weights removed.

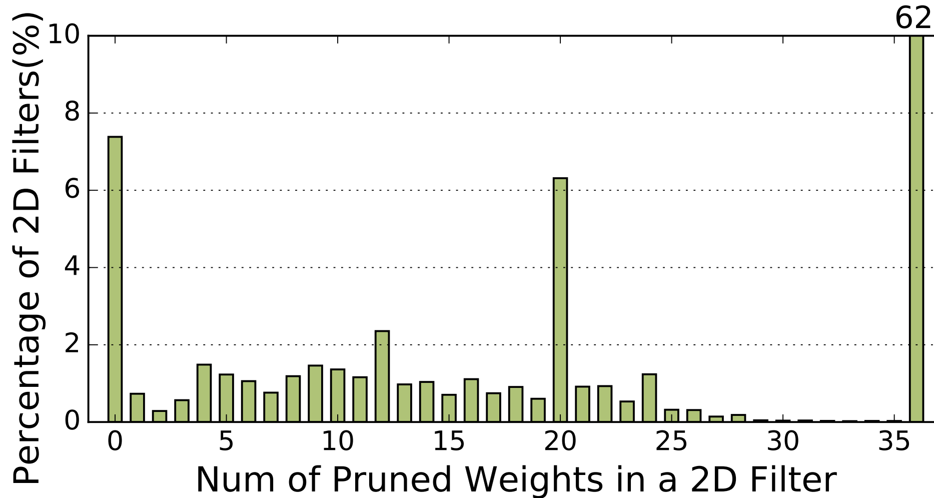


Figure 5.6: Distribution of 2D filters with different numbers of weights pruned.

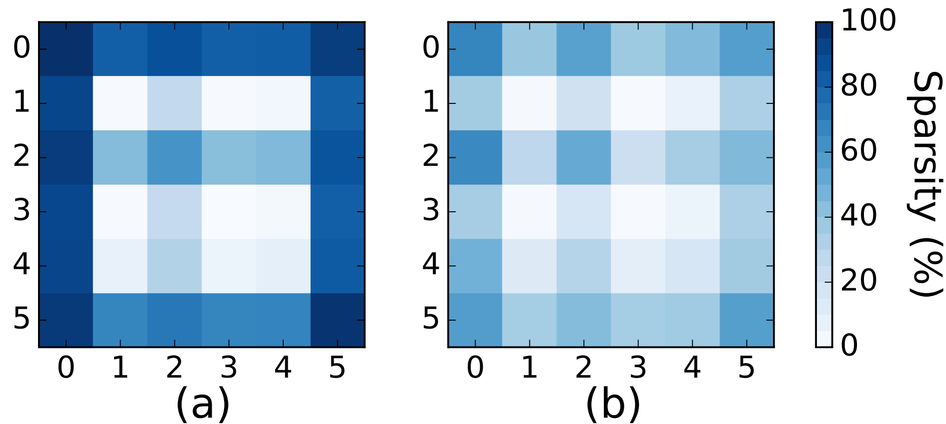


Figure 5.7: Sparsity of different locations of Winograd-domain weights for: (a) filters with 20 weights pruned; (b) filters with at least one weight remaining. Darker locations have higher sparsities.

To explain why many filters have exact 20 weights removed, we visualize the sparsity distribution inside the filters. Figure 5.7a shows the sparsity of different locations for the filters with 20 weights removed. Darker locations have higher sparsities where more weights are removed. The border part of the 6×6 filter, which includes 20 weights, has much higher sparsity than the central part. It means the border part of the Winograd-domain weights is much less important than weights in the central part. A potential reason is that the weights in the central part are correlated to more spatial-domain weights and, therefore, removing them will lead to a larger difference in the output activations. In Figure 5.7b, we also visualize the sparsity distribution inside the filters

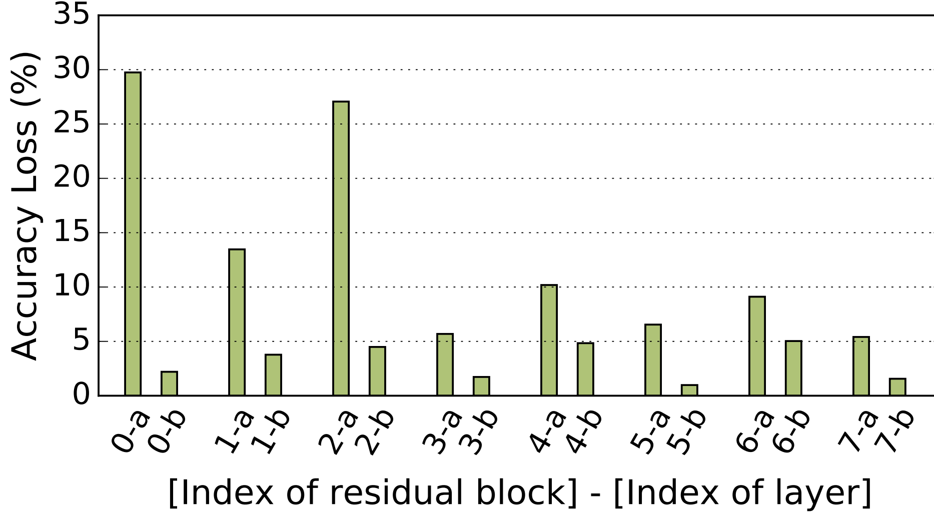


Figure 5.8: Accuracy loss of ResNet-18 when incurring 60% Winograd-domain sparsity into different layers. Spatial structured pruning is applied with no retraining.

with at least one weight remaining, and it shows a similar pattern.

5.4.4 ResNet-18 Pruning with Varied Sparsities across Layers

In addition to pruning ResNet-18 with the same sparsity across all targeting layers, we experiment incurring different sparsities into different layers with spatial-Winograd pruning.

For spatial structured pruning, we first test the pruning sensitivity of each convolutional layer to decide which layers need to be pruned and the corresponding thresholds. To choose the targeting layers, we measure the accuracy loss when 60% of Winograd-domain weights are pruned for each layer. Only one layer is pruned at one time, and other layers are kept intact. Figure 5.8 shows the results. In ResNet-18, the i-th residual block contains two convolutional layers, i-a and i-b. As shown in the figure, the first layer in each residual block is much more sensitive to pruning than the second layer. Therefore, we will only prune the second convolutional layer, i-b, in each residual block.

For each targeting layer i-b, we determine the corresponding pruning threshold $t_i^{spatial}$ based on its pruning sensitivity. We gradually increase the threshold until the validation accuracy drops by 2% and the threshold is recorded as $t_i^{spatial,\,2\%\,loss}$. Then in spatial structured pruning, we can

Table 5.1: The sparsity for the pruned convolutional layers when pruning the second convolutional layer in each residual block of ResNet-18.

Layer	Spatial Structured Pruning		Winograd Direct Pruning
	Winograd Sparsity	Corr. Spatial Sparsity	Winograd Sparsity
0-b	78.0 %	88.8 %	85.0 %
1-b	77.1 %	87.9 %	84.0 %
2-b	76.4 %	88.5 %	84.4 %
3-b	85.2 %	93.1 %	90.4 %
4-b	76.9 %	88.9 %	84.7 %
5-b	88.6 %	95.1 %	93.0 %
6-b	74.4 %	88.3 %	82.4 %
7-b	82.6 %	87.8 %	92.3 %
Average	79.4 %	88.9 %	87.6 %
Top-1 Acc.	69.92 %		69.94 %
Top-5 Acc.	89.34 %		89.51 %

calculate the threshold used for layer i-b as $t_i^{spatial} = \beta \cdot t_i^{spatial,\, 2\%\, loss}$ where β is a multiplier shared across all targeting layers. With a larger β, the threshold and, therefore, the sparsity will be higher. Also, in Winograd direct pruning, we use the same strategy to choose the thresholds used for different layers.

Table 5.1 lists the pruning results. After spatial structured pruning, we can reach an average Winograd-domain sparsity of 79.4% for the pruned layers. The corresponding spatial-domain sparsity is 88.9% which is 9.5% higher. Winograd direct pruning can further improve the Winograd-domain sparsity to 87.6% and layer 5-b has the highest sparsity of 93.0%.

5.4.5 Computation Performance

We first compare the arithmetic operations (OPs) required for the tested models with conventional spatial convolution, Winograd convolution, and sparse Winograd convolution. Figure 5.9 shows the relative results. Here the corresponding models with the conventional spatial convolution are considered as the baseline. For sparse Winograd convolution, we use the models pruned by spatial-Winograd pruning with $< 0.1\%$ loss in top-1 accuracy. The OPs for the transformation/inverse-

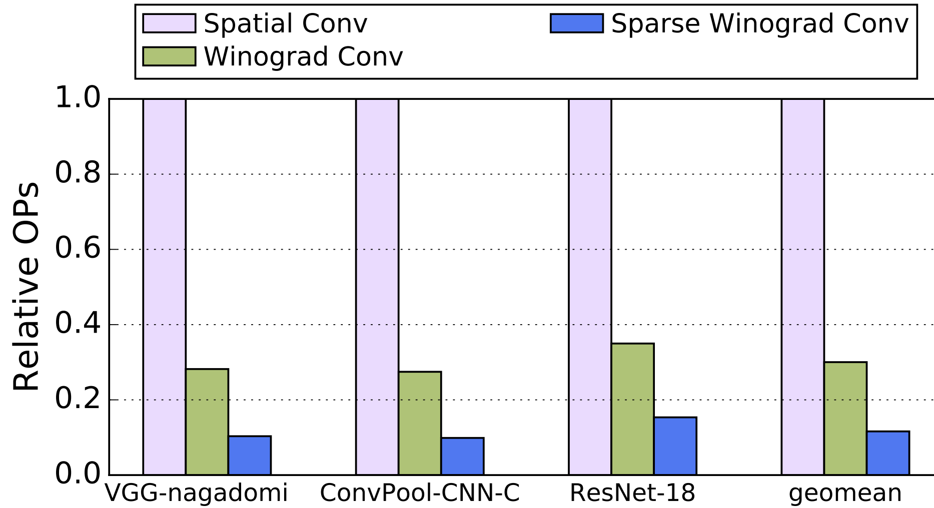


Figure 5.9: Relative arithmetic operations (OPs) required for different models.

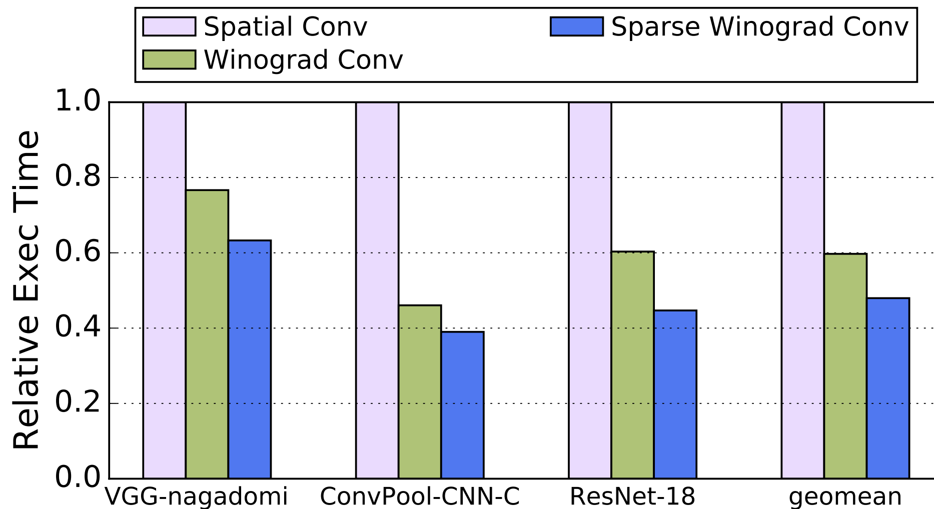


Figure 5.10: Relative execution time for different models.

transformation of the input/output tiles are also included. As the figure shows, using Winograd convolution can reduce the required OPs to 30.0% on average. With spatial-Winograd pruning, the required OPs is further reduced to 11.6%.

To test the computation performance on real hardware, we implement sparse Winograd convolution with NNPACK [93], and profile execution time on Raspberry Pi 3 B+ which includes a quad-core Arm Cortex-A53 processor. Figure 5.10 shows the relative execution time for the tested models with different convolution algorithms. Here we still use spatial convolution as the baseline, and the models pruned with spatial-Winograd pruning for sparse Winograd convolution. The only

difference is that the first layers of the pruned VGG-nagadomi and ConvPool-CNN-C models are filled with zeros to retain the dense format. As shown in the figure, the relative execution time is lowered down to 59.7% on average with conventional Winograd convolution. Sparse Winograd convolution can then reduce the execution time to only 48.0%. VGG-nagadomi benefits much less than the other two models since it contains three fully-connected layers.

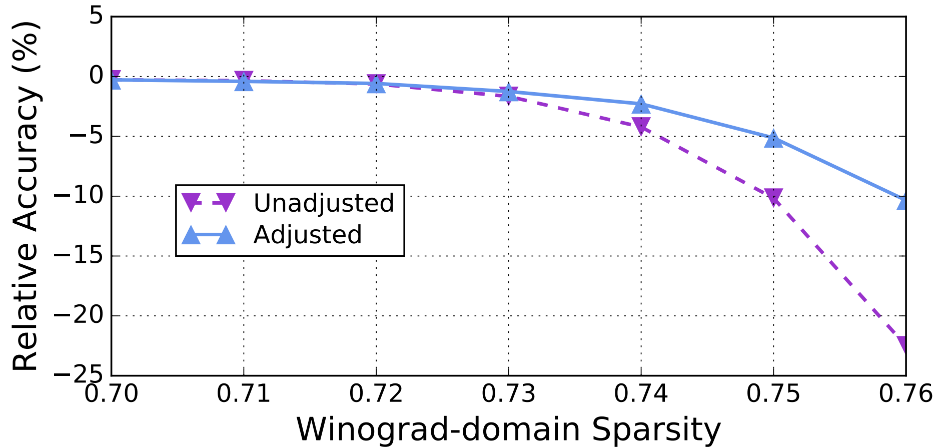
As the same with conventional spatial convolution, the main computation of Winograd convolution can be constructed as matrix multiplication. Therefore, the structured pruning techniques [89, 73] can be combined with spatial-Winograd convolution to improve the computation performance on general-purpose processors. Besides, various hardware designs [94, 95] have been proposed to support sparse Winograd convolution. Spatial-Winograd pruning achieves a higher Winograd-domain sparsity without changing the network structure, and can directly benefit the computation performance on the proposed hardware.

5.4.6 Effectiveness of Importance Factor Matrix

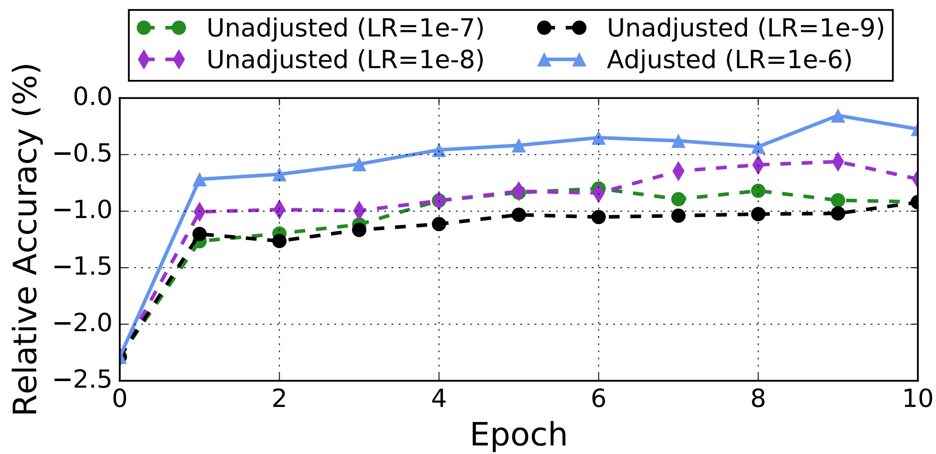
In Winograd direct pruning, we use the importance factor matrix $\boldsymbol{F}$ to adjust the weight importance and gradients for different locations of Winograd-domain weights. Here we test the effectiveness of employing the importance factor matrix in both the Winograd pruning and retraining.

We first test how the importance factor matrix $\boldsymbol{F}$ helps in Winograd pruning. Winograd pruning without retraining is applied to the model pruned by spatial structured pruning with 70% sparsity. Figure 5.11a shows the relative top-1 accuracy, with respect to the original model (69.82%), against the sparsity when pruning with weight importance unadjusted or adjusted with $\boldsymbol{F}$. Adjusting the weight importance with the importance factor matrix here can dramatically reduce the accuracy loss when performing Winograd pruning. When pruning the model to 76% sparsity, using the absolute value as the weight importance will cause a 22% accuracy loss. In comparison, using the importance factor matrix to adjust the weight importance can help reduce the accuracy loss to 10%.

We also test the effectiveness of the importance factor matrix in Winograd retraining. For the



(a) Winograd pruning: unadjusted or adjusted weight importance for different locations.



(b) Winograd retraining: unadjusted or adjusted gradients for different locations..

Figure 5.11: Effectiveness of employing importance factor matrix $\boldsymbol{F}$ in (a) Winograd pruning and (b) Winograd retraining.

model pruned with spatial structured pruning (70% sparsity), Winograd pruning is applied to increase the sparsity to 74%. We then perform Winograd retraining for 10 epochs. Figure 5.11b shows the relative top-1 accuracy against the retraining epochs with unadjusted and adjusted gradients. For unadjusted gradients, we try three learning rates (LR) of 1e-7, 1e-8 and 1e-9. Higher learning rates, e.g., 1e-6, will lead to an accuracy drop through retraining. As shown in the figure, adjusting the gradients with the importance factor matrix can substantially accelerate the convergence. With retraining of only 10 epochs, it reduces the accuracy loss to 0.2% while retraining without gradient adjustment only reduces the accuracy loss to 0.6%.

5.5 Summary

In this chapter, we present a new pruning method, spatial-Winograd pruning, to improve the Winograd-domain weight sparsity without changing network structures. It includes two parts: spatial structured pruning and Winograd direct pruning. In spatial structured pruning, we prune the spatial-domain weights based on the internal structure in the Winograd transformation. It can help efficiently transfer the spatial-domain sparsity into the Winograd domain. For Winograd direct pruning, we perform both pruning and retraining in the Winograd domain. An importance factor matrix is proposed to adjust the weight gradients in Winograd retraining, which makes it possible to effectively retrain the Winograd-domain network to regain the original accuracy without changing the network structure. We evaluate spatial-Winograd pruning on three datasets, CIFAR-10, CIFAR-100, ImageNet, and it can achieve the Winograd-domain sparsities of 73%, 69%, and 74%, respectively.

CHAPTER 6

Conclusion and Future Directions

6.1 Conclusion

As more deeper and larger DNN models are proposed and deployed in various applications, the required computation resources keep increasing. To address this problem, different network compression techniques are introduced to remove the unnecessary computation in DNN models. Besides, domain transformation methods can replace the spatial convolution computation with less computation in other domains. These two techniques can both cut down the required computation in theory. However, these techniques are designed ignoring the characteristics of the target hardware. Directly applying these techniques cannot fully utilize the computation reduction and may even lead to degradation in computation performance.

In this thesis, I propose three solutions to customize network compression and domain transformation techniques for the target processors.

First, in Chapter 3, I introduce a new pruning algorithm, Scalpel. It includes two techniques: SIMD-aware weight pruning and node pruning. SIMD-aware weight pruning ensures the remaining weights to be in groups with the size of SIMD width. In this case, the SIMD units in low-parallelism hardware can be fully utilized. For high-parallelism hardware, the sparsity incurred by conventional pruning techniques hurt the computation efficiency. Node pruning is introduced to prune redundant nodes instead of individual weights. It helps avoid bringing sparsity into DNN models through the pruning process. SIMD-aware weight pruning and node pruning customize

the conventional pruning techniques to provide enough regularity in the computation of the pruned networks. Across the microcontroller, CPU, and GPU, Scalpel can accelerate the DNN inference by 3.54x, 2.61x, and 1.25x on average, respectively.

In Chapter 4, I provide a new pipeline to deploy sub-byte DNN models on low-power microcontrollers. A new convolution algorithm, bitset self-loop convolution, is introduced. It uses bitwise logic operations to perform the multiply-accumulate computation between sub-byte weights and inputs, which dramatically reduces the overhead of unpacking sub-byte data. To further accelerate the inference, I argue for extending the existing ISA instead of adding extra accelerators. It can help save the design and manufacturing cost, but, at the same time, significantly improve computation and energy efficiency. With the proposed pipeline, using 4-bit inputs and ternary weights achieves a 3.37x performance improvement.

As the third part, Chapter 5 introduces a two-step pruning technique to improve the network sparsity in the Winograd domain. In the first step, spatial weights correlated with the same Winograd weight are grouped together. Then I do spatial pruning by removing redundant weight groups instead of individual weights. This strategy helps transfer the spatial sparsity efficiently into the Winograd domain. After this, I also do the pruning directly in the Winograd domain to boost the sparsity further. With the proposed spatial-Winograd pruning technique, the performance of the tested models is improved by 2.08x on average.

The techniques proposed in this thesis help customize and combine the network compression and domain transformation algorithms for target processors. By taking the hardware characteristics into the consideration, the computation and model size reduction can be fully utilized to improve the computation performance and energy efficiency of DNN inferences.

6.2 Future Directions

Based on the work completed in this thesis, there are many new directions that can be explored in the future.

First, Scalpel is designed for existing processors. However, there are more and more ASIC and FPGA designs proposed for accelerating DNN applications. Also, existing processors are being extended to provide better computation support [96]. These new hardware designs and supports expose more opportunities for how to customize the DNN pruning techniques for an efficient software-hardware co-design solution.

Second, an efficient strategy to choose the convolution algorithm for each layer is still required. Including the Winograd algorithm, there are four choices for performing the convolution computation: spatial dense convolution, spatial sparse convolution, Winograd dense convolution and Winograd sparse convolution. If including FFT convolution algorithms, there will be even more design choices. All these convolution algorithms achieve different computation performance for different layers. Also, the design choice of a specific layer will affect the design choice of other layers. Therefore, to achieve a better computation performance, it is of great importance to develop an efficient strategy to choose the appropriate convolution algorithm for each layer.

Third, although this thesis provides a solution for combining network pruning and domain transformation techniques, it is still difficult to use low-precision computation for domain transformation methods. Using Winograd convolution as an example, a high precision, e.g., 8-bit integers, for weights and inputs is still required. Lowering the computation precision in the Winograd domain further will lead to a large drop in prediction accuracy. Combining pruning, precision reducing and domain transformation techniques can provide more opportunities for DNN acceleration.

Fourth, the proposed spatial-Winograd pruning can be further improved. I adopt a conventional thresholding and retraining strategy for pruning in the spatial and Winograd domains. However, recent advanced spatial weight pruning techniques, e.g., ADMM [97], can be combined with our framework to transfer more spatial sparsity into the Winograd domain. Also, another opportunity is to explore using adaptive learning rates algorithms for efficient stochastic optimization. In spatial-Winograd pruning, an importance factor matrix is used to adjust the learning rates for efficient Winograd-domain retraining. It adopts a similar idea as adaptive learning rate algorithms, but the learning rate is still manually adjusted. Therefore, it is expected that a better adaptive learning rate

algorithm can make the retraining in the Winograd domain more efficient, which may help achieve a higher Winograd sparsity.

This thesis discusses the problems in conventional DNN computation reduction techniques and provides corresponding solutions. However, more improvement and extensions can be made for our proposed designs.

BIBLIOGRAPHY

[1] R. Girshick, "Fast r-cnn," in *International Conference on Computer Vision (ICCV)*, 2015.

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

[3] G. Chen, C. Parada, and G. Heigold, "Small-footprint keyword spotting using deep neural networks," in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4087–4091, IEEE, 2014.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[5] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv preprint arXiv:1512.03385*, 2015.

[7] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

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

[9] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *arXiv preprint arXiv:1602.01528*, 2016.

[10] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84, ACM, 2017.

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

[12] M. Courbariaux and Y. Bengio, “Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1,” *arXiv preprint arXiv:1602.02830*, 2016.

[13] C. Leng, H. Li, S. Zhu, and R. Jin, “Extremely low bit neural network: Squeeze the last bit out with admm,” *arXiv preprint arXiv:1707.09870*, 2017.

[14] A. Lavin and S. Gray, “Fast algorithms for convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[15] M. Denil, B. Shakibi, L. Dinh, N. de Freitas, *et al.*, “Predicting parameters in deep learning,” in *Advances in Neural Information Processing Systems*, pp. 2148–2156, 2013.

[16] Y. Le Cun, J. S. Denker, and S. A. Solla, “Optimal brain damage.,” in *NIPS*, vol. 89, 1989.

[17] B. Hassibi, D. G. Stork, and G. J. Wolff, “Optimal brain surgeon and general network pruning,” in *Neural Networks, 1993., IEEE International Conference on*, pp. 293–299, IEEE, 1993.

[18] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both weights and connections for efficient neural networks,” *arXiv preprint arXiv:1506.02626*, 2015.

[19] Y. Guo, A. Yao, and Y. Chen, “Dynamic network surgery for efficient dnns,” *arXiv preprint arXiv:1608.04493*, 2016.

[20] T. He, Y. Fan, Y. Qian, T. Tan, and K. Yu, “Reshaping deep neural network for fast decoding by node-pruning,” in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 245–249, IEEE, 2014.

[21] T. Miconi, “Neural networks with differentiable structure,” *arXiv preprint arXiv:1606.06216*, 2016.

[22] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,”

[23] V. Vanhoucke, A. Senior, and M. Z. Mao, “Improving the speed of neural networks on cpus,” 2011.

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

[25] F. Li, B. Zhang, and B. Liu, “Ternary weight networks,” *arXiv preprint arXiv:1605.04711*, 2016.

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

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

[28] J. Faraone, N. Fraser, M. Blott, and P. H. Leong, "Syq: Learning symmetric quantization for efficient deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4300–4309, 2018.

[29] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[30] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O'Leary, R. Genov, and A. Moshovos, "Bit-pragmatic deep neural network computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 382–394, ACM, 2017.

[31] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan, *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 395–408, ACM, 2017.

[32] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," *ArXiv e-prints*, Dec. 2017.

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

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

[35] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[36] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network computing," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 1–13, IEEE, 2016.

[37] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. ISCA*, 2016.

[38] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 255–266, IEEE Press, 2016.

[39] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 267–278, IEEE Press, 2016.

[40] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pp. 541–552, IEEE, 2017.

[41] P. Chi, S. Li, Z. Qi, P. Gu, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of ISCA*, vol. 43, 2016.

[42] Y. Shen, M. Ferdman, and P. Milder, "Maximizing cnn accelerator efficiency through resource partitioning," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 535–547, ACM, 2017.

[43] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2017.

[44] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 65–74, ACM, 2017.

[45] J. Park, H. Sharma, D. Mahajan, J. K. Kim, P. Olds, and H. Esmaeilzadeh, "Scale-out acceleration for machine learning," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 367–381, ACM, 2017.

[46] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 13–26, ACM, 2017.

[47] P. Hill, A. Jain, M. Hill, B. Zamirai, C.-H. Hsu, M. A. Laurenzano, S. Mahlke, L. Tang, and J. Mars, "Deftnn: addressing bottlenecks for dnn execution on gpus via synapse vector elimination and near-compute data fission," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 786–799, ACM, 2017.

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

[49] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li, "In-situ ai: Towards autonomous and incremental deep learning for iot systems," in *2018 IEEE InternatiOnal SympOsium On High PerfOrmance COmputer Architecture (HPCA)*, pp. 92–103, IEEE, 2018.

[50] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE Computer Architecture Letters*, 2016.

[51] S. Xu and D. Gregg, "Bitslice vectors: A software approach to customizable data precision on processors with simd extensions," in *2017 46th International Conference on Parallel Processing (ICPP)*, pp. 442–451, IEEE, 2017.

[52] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through ffts," *arXiv preprint arXiv:1312.5851*, 2013.

[53] S. Li, J. Park, and P. T. P. Tang, "Enabling sparse winograd convolution by native pruning," *arXiv preprint arXiv:1702.08597*, 2017.

[54] X. Liu and Y. Turakhia, "Pruning of winograd and fft based convolution algorithm," 2016.

[55] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *arXiv preprint arXiv:1802.06367*, 2018.

[56] S. Han, H. Mao, and W. J. Dally, "A deep neural network compression pipeline: Pruning, quantization, huffman encoding," *arXiv preprint arXiv:1510.00149*, 2015.

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

[58] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[59] "NIVDIA DIGITS DevBox," 2016. https://developer.nvidia.com/devbox.

[60] A. Krizhevsky, "cuda-convnet," 2012. https://code.google.com/p/cuda-convnet/.

[61] M. Lin, Q. Chen, and S. Yan, "Network in network," *arXiv preprint arXiv:1312.4400*, 2013.

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

[63] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–9, 2015.

[64] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, ACM, 2014.

[65] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[66] "Stm32 nucleo-f411re development board." https://os.mbed.com/platforms/ST-Nucleo-F411RE/.

[67] "Stm32 nucleo-f401re." https://msdn.microsoft.com/en-us/library/bb514083(v=vs.120).aspx.

[68] "Stm32 nucleo-f302r8." https://msdn.microsoft.com/en-us/library/bb514083(v=vs.120).aspx.

[69] "Stm32 nucleo-l476rg." https://msdn.microsoft.com/en-us/library/bb514083(v=vs.120).aspx.

[70] "Stm32 nucleo-f303ze." https://msdn.microsoft.com/en-us/library/bb514083(v=vs.120).aspx.

[71] "Armv7-m architecture reference manual." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0403e.b/index.html.

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

[73] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 548–560, ACM, 2017.

[74] D. Zhang, J. Yang, D. Ye, and G. Hua, "Lq-nets: Learned quantization for highly accurate and compact deep neural networks," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 365–382, 2018.

[75] "Cortex-m4 devices generic user guide." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CHDHCCJG.html.

[76] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[77] "Two's complement." http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch04s13.html.

[78] "Parabix transform." http://parabix.costar.sfu.ca/wiki/ParabixTransform.

[79] "Lattice." http://www.latticesemi.com/Products/FPGAandCPLD/iCE40.

[80] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H.-J. Yoo, "Unpu: A 50.6 tops/w unified deep neural network accelerator with 1b-to-16b fully-variable weight bit-precision," in *Solid-State Circuits Conference-(ISSCC), 2018 IEEE International*, pp. 218–220, IEEE, 2018.

[81] "Cortex-m4 arm developer." https://developer.arm.com/products/processors/cortex-m/cortex-m4.

[82] "_mm_popcnt_u32." https://msdn.microsoft.com/en-us/library/bb514083(v=vs.120).aspx.

[83] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.

[84] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.

[85] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches,"

[86] "Stm32 power shield datasheet." http://www.st.com/content/ccc/resource/technical/document/data_brief/group1/1d/46/2a/b9/60/98/47/13/DM00417848/files/DM00417848.pdf/jcr:content/translations/en.DM00417848.pdf.

[87] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," 2016.

[88] D.-s. Byeon, S.-s. Lee, Y.-h. Lim, D. Kang, W.-k. Han, D.-h. Kim, and K.-d. Suh, "A comparison between 63nm 8gb and 90nm 4gb multi-level cell nand flash memory for mass storage application," in *Asian Solid-State Circuits Conference, 2005*, pp. 13–16, IEEE, 2005.

[89] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 2074–2082, 2016.

[90] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," in *NIPS-W*, 2017.

[91] Nagadomi, "Code for kaggle-cifar10 competition. 5th place." https://github.com/nagadomi/kaggle-cifar10-torch7, 2014.

[92] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[93] M. Dukhan, "Nnpack." https://github.com/Maratyszcza/NNPACK, 2016.

[94] F. Shi, H. Li, Y. Gao, B. Kuschner, and S.-C. Zhu, "Sparse winograd convolutional neural networks on small-scale systolic arrays," *arXiv preprint arXiv:1810.01973*, 2018.

[95] L. Lu and Y. Liang, "Spwa: an efficient sparse winograd convolutional neural networks accelerator on fpgas," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[96] Intel, "Lower numerical precision deep learning inference and training." https://software.intel.com/en-us/articles/lower-numerical-precision-deep-learning-inference-and-training, 2016.

[97] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 184–199, 2018.