Algorithms and Low Power Hardware for Keyword Spotting

by

Miaorong Wang

B.S., Microelectronics, Shanghai Jiao Tong University (2016)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . Signature redacted .....
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by ..... Signature redacted .....
Anantha P. Chandrakasan
Vannevar Bush Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ..... Signature redacted ..
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Algorithms and Low Power Hardware for Keyword Spotting

by

Miaorong Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Keyword spotting (KWS) is widely used in mobile devices to provide hands-free interface. It continuously listens to all sound signals, detects specific keywords and triggers the downstream system. The key design target of a KWS system is to achieve high classification accuracy of specified keywords and have low power consumption while doing real-time processing of speech data. The algorithm based on convolutional neural network (CNN) delivers high accuracy with small model size that can be stored in on-chip memory. However, the state-of-the-art NN accelerators either target at complex tasks using large CNN models, e.g. AlexNet, or support limited neural network (NN) architectures which delivers lower classification accuracy for KWS. This thesis takes an algorithm-and-hardware co-design approach to implement a low power NN accelerator for the KWS system that is able to process CNN with flexible structures.

On the algorithm side, we propose a weight tuning method that tweaks the bits of weights to lower the switching activity in the weight network-on-chip (NoC) and multipliers. The algorithm takes in 2's complement 8-bit original weights and outputs sign-magnitude 8-bit tuned weights. In our experiment, 60.96% reduction in the toggle count of weights is achieved with 0.75% loss in accuracy. On the hardware side, we implement a processing element (PE) to efficiently process the tuned weights. It takes in sign-magnitude weights and input activations, and multiplies them by an unsigned multiplier. An XOR gate is used to generate the sign bit of the product. The sign-magnitude product is converted back to 2's complement representation and accumulated using an adder-and-subtractor. The sign bit of the product is used as a carry bit to do the conversion. Comparing to the PE that processes original 2's complement weights, around 35% power reduction is observed. In the end, this thesis presents a CNN accelerator that consumes 1.2 mW when doing real-time processing of speech data with an accuracy of around 87.3% on Google speech command dataset [34].

Thesis Supervisor: Anantha P. Chandrakasan
Title: Vannevar Bush Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis would not have been possible without the help of a large group of people. First of all, I would like to thank my thesis supervisor, Professor Anantha Chandrakasan. Thank you for letting me into your group, offering me the opportunity to work on this interesting project and providing research directions along the way. Your encouragement, support and guidance are invaluable to me.

Next, I would like to thank Michael Price for his great help in this project. Although he has graduated before I joined this group, he is always willing to provide guidance and answer technical questions either in person or through emails.

I also owe a debt of gratitude to several of my colleagues. I am thankful to Utsav Banerjee, Priyanka Raina and Skanda Koppula for their technical assistance. And I would like to thank Chiraag Juvekar and Avishek Biswas for their helpful discussion and feedback on my research directions. Special thanks to Avishek Biswas for providing feedback on my thesis.

Besides, I want to thank all the Ananthagroup members. It has been a wonderful experience to work with those talented people from all over the world. I learned a lot from them.

I would also thank Cheng Wang for helpful discussions when I encountered problems, and great collaboration we had in the course project.

I would like to thank my sponsors for their generous supports. My first year is supported by Irwin Mark Jacobs and Joan Klein Jacobs Presidential Fellowship. And the second year is sponsored by Foxconn Technology Group. Thanks for making this project possible.

Aside from this project, I would like to thank the publicity co-chairs of Sydney-Pacific, Grace Liu, Chia-Jung Chang, Demi Fang and Li-Wen Wang, for their great collaboration in finishing the publicity work especially when I was busy.

And finally, I would like to express my gratitude to my parents for loving and supporting me along the way. They always encourage me to chase my dreams and have great confidence in me. That makes the person I am today.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 System Design Targets

Keyword spotting (KWS) is widely used in mobile devices or Internet-of-Things (IoT) applications to provide hands-free interface. It detects specific keywords and triggers the corresponding downstream system. For example, Apple's conversational assistant named Siri listens to 'Hey Siri' to initiate voice input for speech recognition. To enable efficient application, three design targets should be met for a KWS system, including low latency, low power consumption and high accuracy.

Low Latency

The KWS system needs to process real-time speech waveform and generates trigger signals as soon as the user utters the specified keywords. Typically a frame of speech features is extracted and needed to be processed every 10 ms to 20 ms for KWS. This requires low latency in the hardware implementation.

Low Power Consumption

The KWS system is always-on and serves as a trigger to some downstream systems. Since the battery life is important to mobile/IoT applications, power consumption of the KWS system should be minimized.

High Accuracy

When the KWS system detects specified keywords, it powers up the downstream system. The downstream system is usually much larger and consumes more power than the KWS system. One example of the downstream system is a speech recognizer. The work [26] consumes 1.8-7.8 mW on-chip power and the power hungry DRAM access is needed for typical speech recognition tasks, which leads to large overall system power. A false alarm generated by the KWS system will waste a large amount of energy on powering up the downstream system. Besides, large false rejection rate leads to frequent miss of keywords and thus has a negative impact on user experience. Given those two aspects, achieving high accuracy in the KWS system is critical.

1.1.2 KWS Algorithms

State-of-the-art algorithms for KWS are based on neural networks (NNs). A diagram which illustrates a classification of the word 'yes' in the KWS system is shown in Fig. 1-1. It takes in sampled speech data as the inputs and outputs a 1-D vector to indicate which keyword is detected [4]. The system is composed of three stages. First, a front-end unit extracts acoustic features from speech waveforms. Second, a NN computes the posterior probability of each specified keyword, given the acoustic features at each time step. In the last stage, a posterior handling unit smooths the outputs of the NN within a given time window to filter out classification noise. It then outputs a 1-D vector indicating which keyword is detected in this window. The accuracy and power consumption of the whole system are mainly determined by the

NN block.

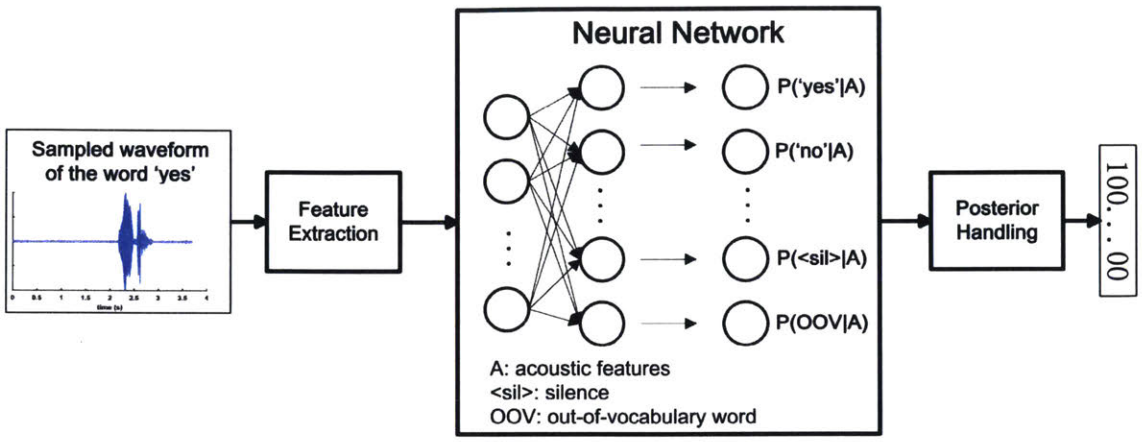

Figure 1-1: Top level diagram of a keyword spotting system.

A large variety of NNs have been employed in the KWS system. Multi-layer perceptron (MLP) is first implemented in the KWS system shown in Fig. 1-1 [4]. Then a paper, co-authored by one of the researchers in the previous work, shows that convolutional neural network (CNN) outperforms MLP in the KWS task [29]. Later on, NNs, such as recurrent neural network (RNN) [31], convolutional recurrent neural network (CRNN) [1], etc. were explored. Different works use different datasets to train the NNs and the criterion to choose the architectures varies. To give a direct comparison of various NNs, Y. Zhang trains different NNs on the same speech command dataset [34]. The architecture for each type of NN is chosen to have the best performance under certain memory and operation constraints. Under the constraint of 80kB memory and 6MOps, Fig. 1-2 illustrates the comparison of different types of NNs. As shown, MLP has much larger memory size over ops ratio compared to other NNs, and delivers relatively low accuracy.

To summarize, different NN architectures have different computation and memory requirements, and provide varied accuracies. MLP requires less computation, but has lower accuracy compared to other types of NNs for KWS tasks. Thus for a system requiring high accuracy, a NN accelerator designed only for MLP processing may not

fit.

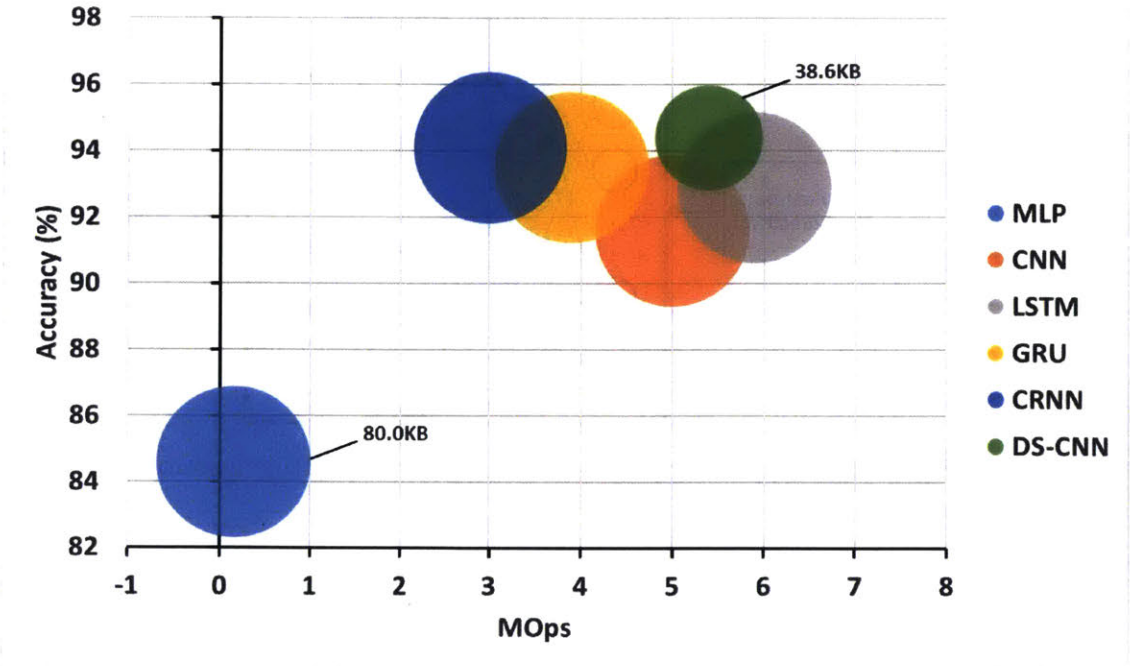

Figure 1-2: A comparison of different NNs on accuracy, the number of operations (ops) and memory size under the constraints of 80kB memory and 6MOps [37] on Google speech command dataset [34]. The memory size is depicted by the size of the bubble.

1.1.3 Hardware Design

This work focuses on a chip implementation of a full KWS system including the feature extraction unit, the NN accelerator and the posterior handling unit.

Recent research has presented many convolutional neural network (CNN) accelerators for high-complexity tasks using large networks requiring off-chip memory access as shown in Table 1.1. Dataflow and architecture of the designs are optimized to lower the off-chip memory access, since it consumes around 100x more energy than on-chip memory access and computation [5].

Several NN accelerators has been designed for moderate-complexity tasks with small NNs that can be stored on chip as shown in Table 1.2. But most of them

Table 1.1: Examples of CNN accelerators for high-complexity tasks.

Reference Paper	Supported Layers	Off-chip Memory Access	Power (mW)
Chen, ISSCC 17 [6]	CONV	off-chip DRAM	278
Lee, ISSCC 18 [22]	FC, CONV, RNN	off-chip DRAM	3.2 – 297†
Ueyoshi, ISSCC 18 [33]	FC, CONV, RNN	3D SRAM using inductive-coupling	3300

† Power for off-chip memory access is not included.

are designed for MLP, which has lower accuracy for KWS tasks, compared to other types of NNs. [3] can process convolutional layers, but only a fixed shape of filter is supported and the network should be binarized. This results in the loss of flexibility. And binary neural network (BNN) delivers lower accuracy compared to the floating point counterparts, e.g. 29.8% accuracy loss on ImageNet [9] dataset [32].

Table 1.2: Examples of NN accelerators for moderate-complexity tasks.

Reference Paper	Supported Layers	Off-chip Memory	Power (mW)
Whatmough, ISSCC 17 [35]	FC	none	22.4
Bang, ISSCC 17 [2]	FC	none	0.321
Shah, SiPS 15 [30]	FC	none	3.3
Bankman, ISSCC 18 [3]	FC, binarized CONV	none	0.889

1.1.4 Algorithm and Hardware Co-design

Although NNs deliver state-of-the-art performance in many applications, they are power hungry and thus, difficult to be implemented in IoT devices. To improve the energy efficiency, many algorithms are proposed to co-design the NN and its corresponding hardware accelerator. Those algorithms can be divided into two groups [32]:

- quantization
- model compression.

Quantization

Quantization is to map floating-point parameters in NN to a smaller set of quantized values. It reduces the complexity of the computation units and lowers the amount of storage since less bits are needed to represent the parameters. One popularly used quantizer is the linear quantizer. It is simply the fixed-point representation of numbers. Fixed point calculation units are used to process the NNs instead of the floating-point counterparts. Next, log2 quantizer is also explored [21]. After mapping data into the log2 domain, multiplication can be converted to shifting. But it increases the complexity of addition. Addition can be done either in the linear domain with additional domain conversion steps or in the log2 domain using an approximation algorithm. Another type of non-linear quantizer uses a quantization function learned from data. For example, data are clustered into groups using k-mean [14]. The mean of each group is used to represent the group of data. Since the quantization function is complicated, a look-up table and an index calculation unit to align weights and activations are needed to perform calculation on hardware.

Extreme cases of quantization are the tenary weight network [23, 38] and the binary weight network [8, 28]. The weights and/or activations in those networks are quantized into only one to two bits. It significantly reduces the storage, but has several drawbacks. If all layers of weights are quantized into one to two bits, the classification accuracy drops greatly. The accuracy may be maintained by keeping the first and the last layer of weights from this extreme way of quantization. However, it imposes a great challenge on the ASIC design to support both tenary/binary weight layers and normal layers with relatively large bitwidth.

Model Compression

Model compression is to reduce the model size and the number of operations with an acceptable level of accuracy loss. It can be divided into two categories.

Methods in the first category are pruning-based, which set low-magnitude weights and/or activations to zero. Related works can be found in [15], [36], etc. During hardware implementation, multiplication with zero is skipped to minimize the computation energy. And weights should be compressed to be zero-free before stored in the memories so that the memory power consumption can be lowered. Those require dedicated hardware to process the compressed data. Related works can be found in [13, 25], etc. For large NN with over 10M weights, weights cannot be fit in the on-chip SRAM. Thus off-chip memory access is needed, which takes up a large portion of the total power consumption. Using pruning-based method and dedicated hardware improves energy efficiency, because pruning greatly reduce the number of weights and/or activations, and thus reduces the number of off-chip memory accesses required. But for small NN that can be stored on chip, it is not clear whether using a more complex hardware architecture to handle pruned weights and/or activations provides higher energy efficiency.

The other category of methods does not leverage the zeros in weights and/or activations. It compacts the NN architecture. For example, tensor decomposition is applied to NN after training to approximate large weight tensors using a sequence of small tensors [17, 20]. It is reported to have similar compression rate and accuracy loss compared to the pruning based methods. On top of that, it eliminates the need for dedicated hardware architecture.

1.1.5 Research Focus

This thesis targets at the complete system implementation for KWS application, covering the functions of feature extraction, NN processing and posterior handling. A highly accurate, real-time and low-power KWS system imposes the need for a NN accelerator that falls in the gap between existing designs, as shown in Fig. 1-3. And the existing algorithm and hardware co-design techniques focus on reducing the memory

footprint and the number of operations. The influence of bit flips in the hardware on the energy consumption and how to utilize it have not analyzed. This thesis focuses on designing a NN accelerator using a novel algorithm-and-hardware co-design technique. The NN accelerator supports CNN in order to deliver high accuracy for keyword detection. It is optimized for medium size NNs with less than 80kB parameter size and around 5MOps to achieve low power consumption under the latency constraint for real-time processing.

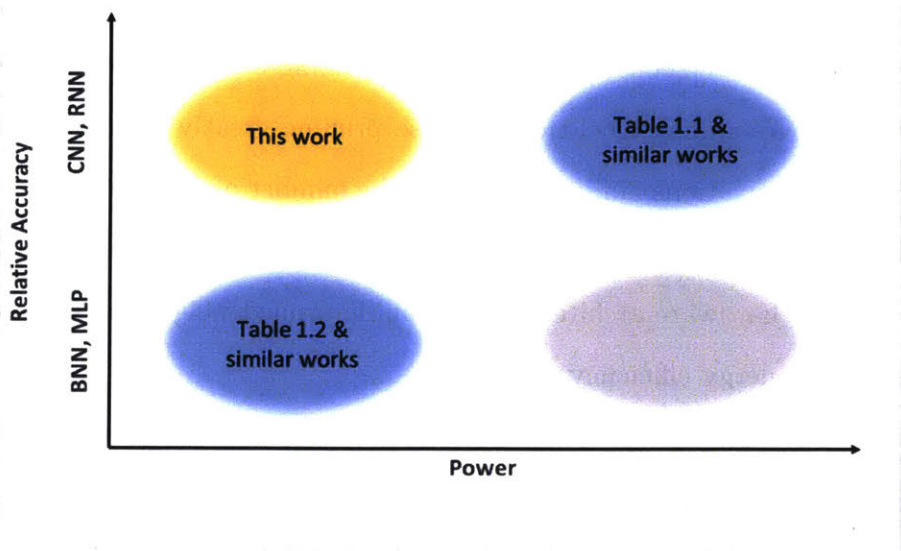

Figure 1-3: Research focus of designing a NN accelerator for highly accurate, real-time and low-power KWS system.

1.2 System Overview

We take a hardware and software co-design approach to implement the accurate, low-power and real-time KWS system, as illustrated in Fig. 1-4.

The raw model mainly determines the accuracy of the KWS system. Designing the hardware architecture to support existing CNN architectures guarantees the high accuracy of the system.

A large portion of the total power comes from accessing SRAMs. To reduce

the memory size, tensor decomposition [17] is applied and a compressed model is generated with little loss of accuracy. The dataflow of the NN accelerator mainly determines the number of memory accesses needed to process a NN. Different dataflows are analyzed and a weight stationary dataflow is found to work best for the NNs for our KWS system.

We also investigated into reducing the power consumption from the other parts of the NN accelerator. Dynamic power of a CMOS gate is determined by load capacitance (C_L), operation frequency (f), switching activity ($\alpha_{0->1}$), and supply voltage (V_{dd}), as shown in the following equation.

$$P_{dyn} = \alpha_{0->1} C_L V_{dd}^2 f \tag{1.1}$$

We tried to minimize each component. First, tensor decomposition reduces the amount of computation required in a NN. Thus lower clock frequency and supply voltage can be used to process the NN. Second, a spatial hardware architecture is designed to optimize the clock frequency and effective load capacitance of the system, and to meet the real-time requirement. Finally, a bit tuning algorithm is proposed to lower the switching activity on the weight bus and the multipliers with marginal loss of accuracy. It also provides the opportunity of using a custom memory proposed in [11], which is reported to reduce around 50% of the read access energy, given certain data statistics.

1.3 Thesis Overview and Contributions

This thesis takes a algorithm-and-hardware co-design approach to optimize the energy efficiency of the KWS system achieving high accuracy and real-time processing.

Chapter 2 proposes an algorithm to tune the bits of fixed-point weights, which lowers the toggle count with little loss in accuracy. Every step of the algorithm is

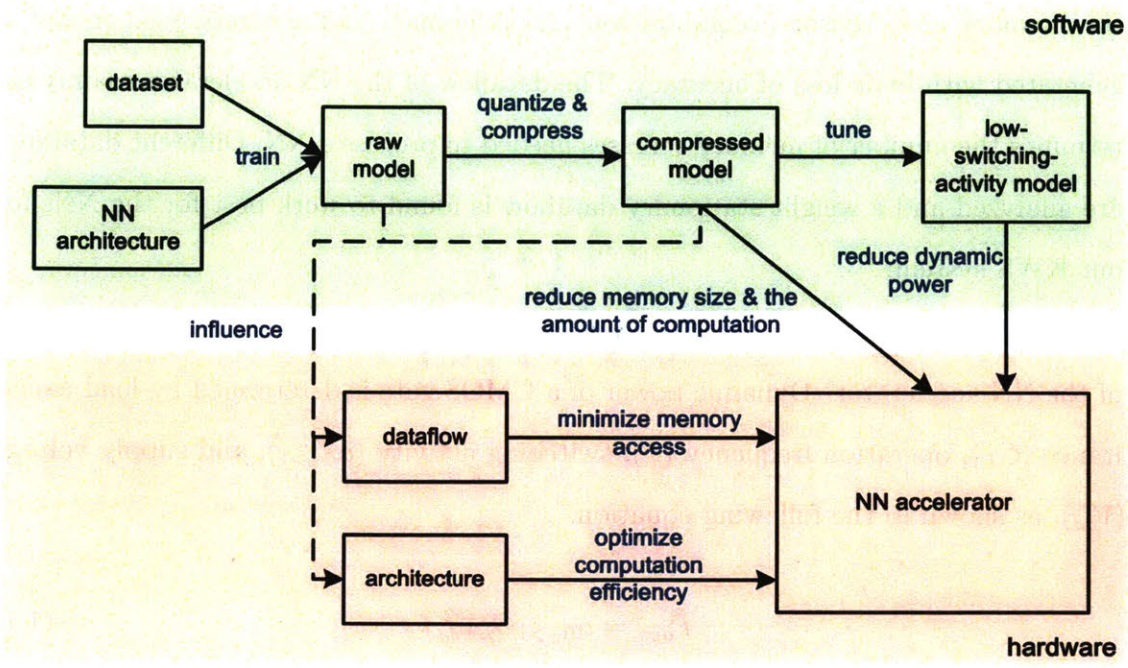

Figure 1-4: An illustration of the hardware and software co-design approach.

discussed in detail, along with training and testing results to show its influence on NN classification accuracy.

Chapter 3 provides a comprehensive analysis of the dataflows for the NN accelerator. The number of memory accesses using weight stationary (WS), output stationary (OS) and row stationary (RS) dataflow are analyzed under certain hardware constraints. Based on the analysis and comparison, WS dataflow is used for our NN accelerator for KWS.

Chapter 4 presents a NN accelerator supporting weight stationary dataflow. Hardware architectures and micro-architectures are illustrated. Processing elements (PEs) running the original NN model and the model tuned by the algorithm proposed in Chapter 2 are compared. It proves the effectiveness of the proposed algorithm in energy reduction. The simulation results of the whole NN accelerator are presented.

Chapter 5 summarizes the key contributions of the thesis and presents the future research directions.

Chapter 2

Bit Tuning Algorithm

2.1 Overview

The power consumption of a NN accelerator can be reduced by tuning the bit representation of weights. As shown in Eq. 1.1, dynamic power of a CMOS gate is linearly proportional to the switching activity $\alpha_{0->1}$. In a NN accelerator, weights are read from the buffer, delivered through network-on-chip (NoC) and then multiplied with input activations. The reading, delivery and processing sequence of weights are fixed by the designer. The bit toggling between consecutive weights affects the switching activity of weight buses and the multipliers. And for a memory that utilizes data statistics, e.g. [11], it also influences the switching activity of bit lines. Therefore minimizing the bit toggle count of weights reduces the power consumption of multipliers, the weight NoC and even the SRAM.

We propose an algorithm that tunes the bits of weights to lower the amount of bit toggling with insignificant loss in accuracy. A CNN is tuned by the following steps.

1. Tensor decomposition and retraining
2. Fixed-point sign-magnitude representation
3. Scaling

4. Bit perturbation

5. Fine tuning

This chapter will introduce each step and the experimental results will also be presented.

2.2 Tensor Decomposition and Retraining

Tensor decomposition is used to compressed the model size and the number of calculations in the NN with little loss of accuracy after retraining [17, 20]. The reason to include it as a step in our bit tuning algorithm for CNN is that, it magnifies the influence of weights on the switching activity of the multiplier. This will be explained in the following analysis.

We adopt the compression scheme proposed in [17]. The filter (W) with the shape of $M \times C \times R \times S$ can be decomposed into three smaller filters using Tucker-2 decomposition. It can be written as follows:

$$W_{r,s,c,m} = \sum_{r_3=1}^{R_3} \sum_{r_4=1}^{R_4} K_{r,s,r_3,r_4} X^{(3)}_{c,r_3} X^{(4)}_{m,r_4} \tag{2.1}$$

$$= \sum_{r_2=1}^{R_2} \sum_{r_4=1}^{R_4} K_{r,r_2,c,r_4} X^{(2)}_{s,r_2} X^{(4)}_{m,r_4} \tag{2.2}$$

where R_2, R_3 and R_4 are the rank of the mode-2, mode-3 and mode-4 matricization of the tensor W respectively. K is the core tensor and $X^{(2)}, X^{(3)}$ and $X^{(4)}$ are factor matrices of sizes $S \times R_2$, $C \times R_3$ and $M \times R_4$ respectively. For most CONV layers, R and S are relatively small. Thus Eq. 2.1 is used to decompose mode-3 and mode-4 that are associated with the input and output channel dimensions. It is depicted in Fig. 2-1. $X^{(3)}$ and $X^{(4)}$ are used as 1×1 convolution filters. The output activations generated by the lower part of the graph should be similar to the upper part. Activation functions,

such as ReLU, are only applied to the final outputs. Intermediate layers are just linear transformation. For the first layer of the CNN for KWS, where $C = 1$, Eq. 2.2 is used instead. Detailed explanantion of tensor decomposition and the tool chain can be found in [17].

Tensor decomposition generates a compressed NN architecture and also provides the values of every decomposed tensors. Because the reconstruction error of the weight tensors, instead of the output activations, are minimized during tensor decomposition, directly using the decomposed filters results in significant loss in accuracy (e.g. more than 50% in AlexNet [17]). Therefore, retraining is needed. The decomposed filter values are used as the starting point.

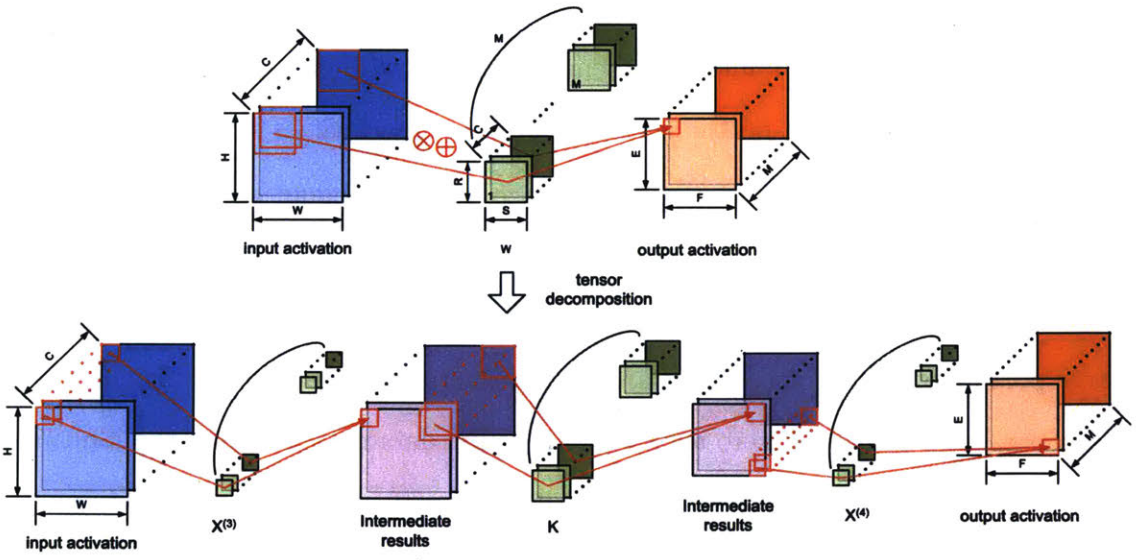

Figure 2-1: An illustration of tensor decomposition on CONV layers.

Table 2.1 shows the decomposition of a CNN trained for KWS on the Google speech command dataset [34] in [37]. The decomposed models are trained using the experimental setup in [37]. But batch normalization is not applied, because it is currently not supported in our NN accelerator. The classification accuracy evaluated on the test set is reported. The original and decomposed NN will be referred to as CNN-1 and CNN-1-decomp in the rest of this thesis. The first and second convolutional

layers (conv1, conv2) and the linear-low-rank layer (lin) take up a large portion of the total parameter size and the amount of computation, and thus are decomposed. In our experiments, decomposing the first fully-connected layer (fc1) and the output layer (fco) leads to great accuracy loss and thus they are kept unchanged. Although the original network is designed to be small-footprint, tensor decomposition further reduces the parameter size and the amount of computation.

We conducted experiments on other existing CNN architectures for KWS trained on the Google speech command dataset [34]. Table 2.2 summaries the effect of tensor decomposition on those CNN architectures. The architecture of CNN-2 is proposed in [37]. CNN-3 refers to the fstride4 architecture in [1]. The decomposed versions are referred to as CNN-2-decomp and CNN-3-decomp in the rest of this thesis. The detailed structure can be found in Appendix A. As shown, tensor decomposition with retraining has little effect on the accuracy for all different types of CNN for KWS.

Besides the reduction in the parameter size and the amount of computation, the reason why we apply tensor decomposition before tuning the weights of CNN are as follows. Most CNNs use ReLU as the activation function. In our experiments, a large portion of output activations are set to be zero after ReLU. An efficient way to process a large number of zero activations is to use data gating to skip the multiplier and weight read inside the PE [7]. Thus it reduces the influence of weights on the switching activity of the multiplier. Tensor decomposition generates several linear layers that are not followed by activation functions, thus improves our control of the switching activity of multipliers through bit tuning the weights.

2.3 Sign-Magnitude Representation

After we decomposed and retrained the CNN, we use linear quantizer to convert the model from floating point to fixed-point numbers for deployment on the NN

Table 2.1: The original and decomposed architecture of CNN-1. The input layer of CNN-1 has $H = 10$ and $W = 49$.

Original									Decomposed								
Layer	R	S	C	M	U	V	Wgt (K)	Mult (K)	Layer	R	S	C	M	U	V	Wgt (K)	Mult (K)
conv1	4	10	1	28	1	1	1.12	313.6	conv1-1	1	10	1	6	1	1	0.53	155.04
									conv1-2	4	1	6	9	1	1		
									conv1-3	1	1	9	28	1	1		
conv2	4	10	28	30	2	1	33.6	2150.4	conv2-1	1	1	28	18	1	1	16.25	1149.12
									conv2-2	4	10	18	21	2	1		
									conv2-3	1	1	21	30	1	1		
lin	1	1	1920	16	na	na	30.72	30.72	lin-1	1	1	1920	12	na	na	23.38	23.38
									lin-2	1	1	12	12	na	na		
									lin-3	1	1	12	16	na	na		
fc1	1	1	16	128	na	na	2.05	2.05	fc1	1	1	16	128	na	na	2.05	2.05
fco	1	1	128	12	na	na	1.54	1.54	fco	1	1	128	12	na	na	1.54	1.54

R, S: Length of filter in time and frequency axis
C, M: Number of input and output channels
U, V: Stride in time and frequency

Table 2.2: Comparison of accuracy, parameter size and the amount of computation before and after tensor decomposition and retraining.

CNN	Accuracy (%)			Parameters			Multiplications		
	Orig.	Decomp.	Loss	Orig	Decomp.	CR†	Orig	Decomp.	SR‡
CNN-1	89.78	88.54	1.24	69.0K	43.7K	1.58x	2.5M	1.3M	1.92x
CNN-2	90.94	90.65	0.29	178.2K	120.8K	1.48x	8.6M	6.0M	1.43x
CNN-3	81.90	81.20	0.70	949.4K	88.4K	10.74x	5.6M	0.5M	11.2x

† Compression rate
‡ Speed-up rate

accelerator. It is proved by many works, e.g. [12, 37], that storing weights in 8-bit is enough for maintaining good accuracy. Fig. 2-2 shows our experimental results on the decomposed CNN models mentioned in Table 2.2. Bit width of 8 delivers almost the same performance as the floating point NN. Weights have both positive and negative values. In most cases, 2's complement representation is used to handle signed numbers, due to its simplicity in computation. Thus we use 8-bit fixed-point 2's complement weights as the baseline of our bit tuning algorithm.

Compared to 2's complement representation, sign-magnitude weights have much lower toggle count. Table 2.3 presents the total number of 0-to-1 toggle given the processing sequence of our NN accelerator. Around 30%–40% of reduction can be made by converting weights from 2's complement to sign-magnitude. The reason is that weights of each layer roughly follows zero-mean Gaussian distribution as shown in Fig. 2-3. The majority of them are close to zero. 2's complement representation of two small numbers with opposite sign has large toggle count, because of the sign bits. Sign-magnitude representation does not have this issue.

The cost of using sign-magnitude representation is that the adder needed for it is more complex than the 2's complement adder. Two 2's complement numbers can be added up using a signed adder. But for the sign-magnitude numbers, we need to first check the sign to determine whether addition or subtraction is needed. Making use of the fixed computation pattern of NN, we propose a mixed-representation processing element (PE) that minimizes the hardware cost of using sign-magnitude weights. Section 4.2 discusses it in detail.

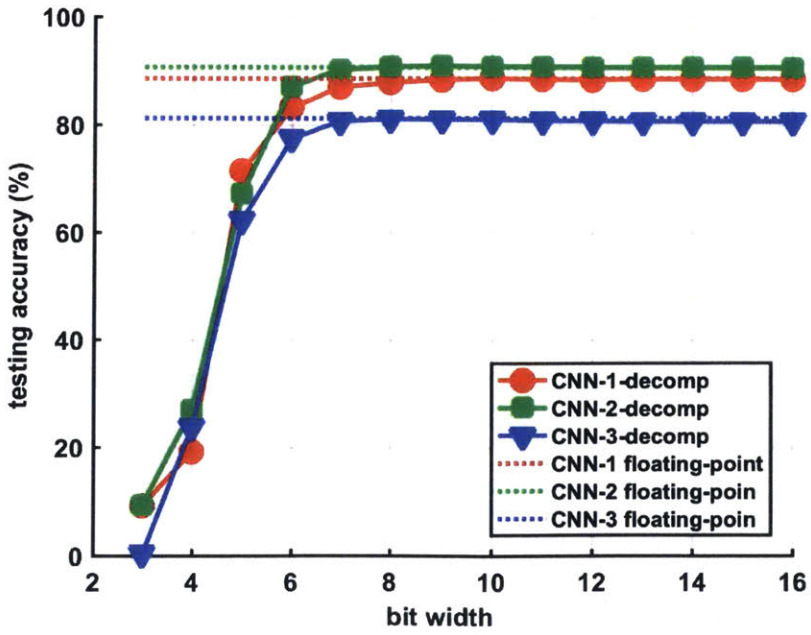

Figure 2-2: The influence of weight bitwidth on accuracy.

Table 2.3: The $0- > 1$ toggle count of 2's complement and sign-magnitude weights.

CNN	2's complement	sign-magnitude	reduction
CNN-1-decomp	79793	55623	30.3%
CNN-2-decomp	232752	125178	46.2%
CNN-3-decomp	172537	114717	33.5%

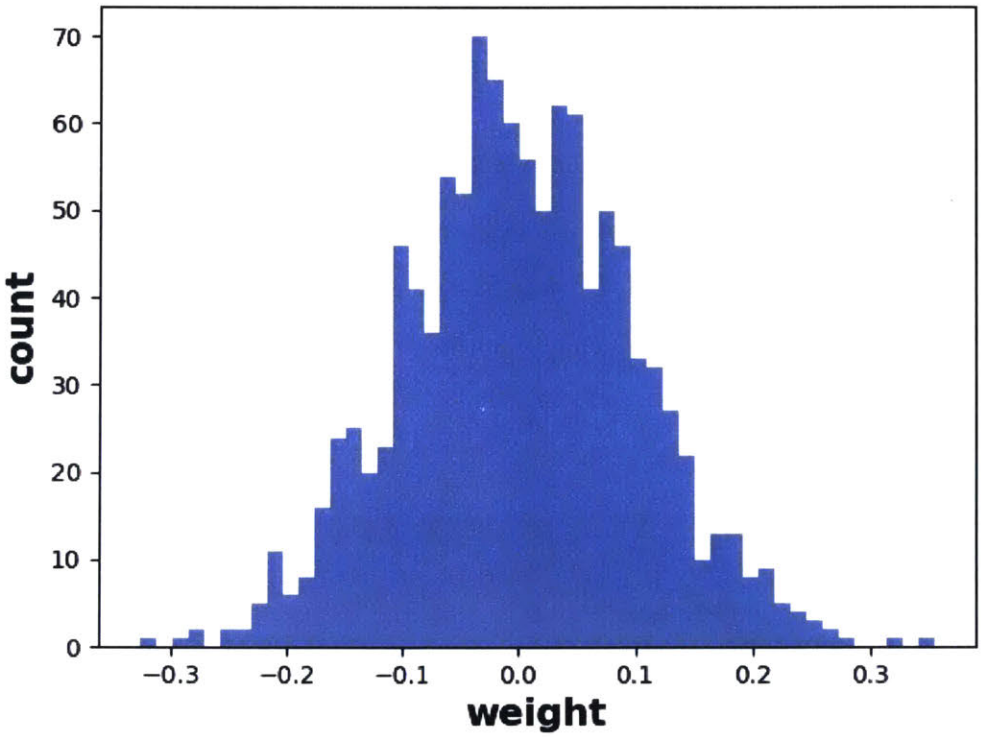

Figure 2-3: The histogram of weights in the first layer of CNN-1.

2.4 Weight Scaling

Weight scaling is to multiply the weights with a number k that is bigger than 0.5 and smaller than 2. The multiplication is done in floating-point and then the weights are converted back to 8-bit fixed-point. Since $0.5 < k < 2$, weight scaling does not change the location of the decimal point.

If we assume that the precision of weight and calculation is unlimited, scaling weight does not affect the accuracy of NNs that use ReLU as their activation function. The output activation can be calculated as follows:

$$\begin{aligned} O[m][e][f] &= ReLU(B[m] \\ &+ \sum_{r=0}^{r=R-1} \sum_{s=0}^{s=S-1} \sum_{c=0}^{c=C-1} I[c][Ue+f][Vf+s] \times W[m][c][r][s]), \\ &0 \leq m < M, 0 \leq e < E, 0 \leq f < F, \\ &E = (H - R + U)/U, F = (W - S + V)/V \end{aligned} \tag{2.3}$$

where O, I, B and W stand for output activations, input activations, biases and weights respectively. The explanation of indices are listed in Table 3.1. After weight scaling, we have $W' = kW$ and $B' = kB$. As shown in Fig. 2-4, $ReLU(kx) = kReLU(x)$. Thus we have the following equation:

$$\begin{aligned} O'[m][e][f] &= ReLU(B'[m] \\ &+ \sum_{r=0}^{r=R-1} \sum_{s=0}^{s=S-1} \sum_{c=0}^{c=C-1} I[c][Ue+f][Vf+s] \times W'[m][c][r][s]), \\ &= ReLU(kB[m] \\ &+ k \sum_{r=0}^{r=R-1} \sum_{s=0}^{s=S-1} \sum_{c=0}^{c=C-1} I[c][Ue+f][Vf+s] \times W[m][c][r][s]) \\ &= k \times O[m][e][f] \end{aligned} \tag{2.4}$$

Linear scaling of weights result in linear scaling of output activations. The outputs of the last layer are connected to softmax function, which is shown in Eq. 2.5.

$$\begin{aligned} \sigma(\mathbf{z})_m &= \frac{e^{z_m}}{\sum_{n=1}^{M} e^{z_n}} \\ m &= 1, ..., M \end{aligned} \tag{2.5}$$

where $\mathbf{z}$ is M-dimension output vector of the last layer. It can be proved that

$$\begin{aligned} \sigma(k\mathbf{z})_m &= \sigma(\mathbf{z})_m \\ m &= 1, ..., M. \end{aligned} \tag{2.6}$$

Thus linear scaling of weights does not affect the final output of a NN.

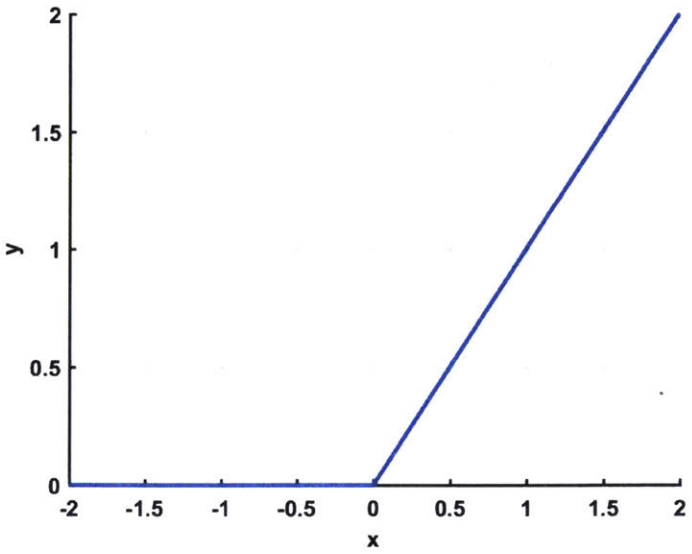

Figure 2-4: ReLU function.

In reality, weights are stored in 8-bit. When we scale the weights in floating point numbers and convert them back to fixed-point numbers, truncation of bit introduces nonlinearity. Thus the effective scales of different 8-bit numbers may be different. Table 2.4 shows an example. Original weights 0.15625 and 0.9921875 are multiplied with 0.8, and the products are 0.125 and 0.79375 respectively. We have weights stored in Q1.7 format. 0.125 can be directly represented in this format, while 0.79375 needs to be truncated. The Q1.7 fixed-point representation of 0.79375 has a decimal value of 0.7890625, which results in an effective scale of around 0.795 instead of 0.800. Although all the weights in one layer are multiplied with the same number in floating point, the effective scales for fixed point numbers may be different. Thus weight

scaling can lead to some accuracy loss of the overall system.

Table 2.4: An example of weight scaling.

Original Wgt		Scale	Scaled Wgt		Effective
Decimal	Fxpt		Decimal	Fxpt	Scale
0.15625	0.0010100	0.8	0.125	0.0010000	0.800
0.9921875	0.1111111		0.79375	0.1100101	0.795

Weight scaling is done following the Algorithm 1. We search for the scale that results in the smallest toggle count in a given range. Since the average magnitude of weights affects the average magnitude of activations at each layer, we need to scale the activations to prevent them from overflow or underflow for the fixed-point representation. After weights are tuned, we gather all the weight scale and determines the activation scale using Algorithm 2. Activations at each layer is kept unchanged or increased by a factor of 2 or 0.5. This can be done using a 1-bit shifter, which is implemented by simple rewiring.

Input: Weights of one layer W, scale range $[a, b]$ ($a > 0.5$, $b < 2$) and step s
Output: Scaled weights W_{best}, corresponding scale k_{best}

```
1  TC ← CountToggle(W);
2  minTC ← TC;
3  k ← a;
4  while k ≤ b do
5  |  W' = k × W;
6  |  newTC ← CountToggle(W');
7  |  if newTC < minTC then
8  |  |  minTC ← newTC;
9  |  |  W_best ← W' ;
10 |  |  k_best ← k;
11 |  end
12 |  k ← a + s;
13 end
```

Algorithm 1: Pseudo-code for weight scaling.

Input: Weight scale k_l of this layer, weight scale k_{l+1} of next layer, accumulated weight scale k_{acc} from the first layer to previous layer
Output: Output activation scale t_l of this layer, accumulated weight scale k'_{acc} from the first layer to this layer

$k'_{acc} \leftarrow k_{acc} \times k_l$;
if $k'_{acc} \times k_{l+1} <= 0.5$ **then** $t_l \leftarrow 2$;
else if $k'_{acc} \times k_{l+1} >= 2$ **then** $t_l \leftarrow 0.5$;
else $t_l \leftarrow 1$;

Algorithm 2: Pseudo-code for deriving activation scale for each layer.

2.5 Bit Perturbation

Inspired by the power reducing technique proposed for FIR filter [16], we perturb the bits of weights to further reduce the switching activity. Since we tweak the bits of weights, the weight values change after bit perturbation. Averaged relative error of all data is used to represent the tuning error. Relative error is defined as

$$e = \frac{|v - v_0|}{|v_0|} \tag{2.7}$$

where v_0 and v is the original value and the perturbed value respectively. Given the maximum relative error e_{max} the system can tolerate, bit perturbation of weights is done in the following steps.

1. Flatten the 4-D weight tensor into a 1-D vector following the sequence that weights are read, delivered and calculated in the NN accelerator
2. Split the large 1-D vector into smaller vectors with the length of L
3. Perturb every small 1-D vector using Algorithm 3, given the maximum relative error e_{max} the system can tolerate
4. Concatenate the perturbed vector and reshape them to the 4-D weight tensor

L and e_{max} are the parameters the designer can play with. They both affect the toggle count of the perturbed weights and the accuracy loss of the system. In our

experiments, L equal to the number of input channels and $e_{max} \in [0.1, 0.2]$ lead to good performance.

In Table 2.5, a 1×9 vector of weights picked from the fc1-3 layer of CNN-3 is used as an example to illustrate the basic idea of Algorithm 3. The orignal weights are shown in step 0. We set e_{max} to be 0.2. Since the length of the vector is 9, it can be split into a maximum of 5 trunks (4 trunks with 2 elements and 1 trunk with 1 element). Thus we have $nTrunk \in \{1, 2, 3, 4, 5\}$. In the first step, the least significant bit (LSB) of all the numbers (nTrunk = 1) are perturbed ($k = 1$). We calculate the average value of all the LSBs and round it to 1 bit, which equals to 1. LSBs of all the numbers are set to 1. We then check whether the relative error introduced by perturbation exceeds our limit and calculate the $0- > 1$ toggle count to see whether it is reduced. In step 1, e satisfies our requirement, but the toggle count remains the same. We then increase k to 2 and repeat the same process to tune the last 2 bits. It can be seen that the toggle count does not change either. In step 3, k is increased to 3. After we set the last 3 bits to be 011, the toggle count is reduced to 7 and the resulting relative error meets our requirement. This set of weights are kept in the memory until a better set is found in step 4. Step 5 sets the last 5 bit of all data to be the same. This results in an e that is bigger than our limit e_{max}, thus this step is invalid and it is crossed out in the table. We then increase nTrunk to split the weights into subsets. We tune the last 5 bits of different subsets of weights separately to see whether it results in smaller e. $nTrunk = 2, 3, 4$ are tried in step 6–8, but none of them leads to an e that is within the limit. In step 9 ($nTrunk = 5$), where the last 5 bits of $\{v0, v1\}$, $\{v2, v3\}$, $\{v4, v5\}$ and $\{v6, v7\}$ are tuned to be equal within the subset, e goes down to 0.159 and meets our requirement. But TC01 of this step is higher than what we got in step 4. Thus the tuning results of step 4 are still kept in the memory. We then increase k to 6 and set nTrunk to 1, and repeat the same process. The algorithm stops when all possible nTrunk values have been tried and

no valid e is found at a given k or all possible combinations of k and $nTrunk$ have been explored. Our example belongs to the latter case. The last step is step 19. The results kept in the memory is the set of perturbed weights with the smallest TC01. In this example, the best set of weights presents at step 4.

Pseudo code of bit perturbation algorithm is shown in Algorithm 3. Differences and improvements between our algorithm and the bit perturbation scheme in [16] (it is refered to as Kasturi1997) are as follows.

- Kasturi1997 is to tune the coefficients of FIR filters. Thus filter design specification such as passband ripple and stopband attenuation are used as criteria to choose between different sets of perturbed coefficient. Our algorithm is applied to the weights of NN. Considering that getting the accuracy of NN on dataset requires a large amount of computation, we use relative error as our criterion to speed up the perturbation.

- Since weights are learnable during training, we incorporate training into the bit tuning algorithm, which fine tuning the weights to minimize the accuracy loss during weight scaling and bit perturbation. Kasturi1997 does not have this step. It will be discussed in Section 2.6

- In Kasturi1997, line 23 of Algorithm 3 is done when the `If` condition in line 24 is true. This leads to inefficiency, because increasing nTrunk cannot bring us lower toggle count after relative error goes within the required range. For example, keeping $k = 1$ and further increasing nTrunk after step 1 in Table 2.5 will not provide better results. We fix this problem to improve the efficiency of the algorithm.

Table 2.5: An example of weight perturbation.

Step ($k^{\dagger}$, nTrunk‡)	0 original	1 (1, 1)	2 (2, 1)	3 (3, 1)
v_0	00011011	00011011	00011001	00011011
v_1	00010011	00010011	00010001	00010011
v_2	00010101	00010101	00010101	00010011
v_3	10100101	10100101	10100101	10100011
v_4	00010101	00010101	00010101	00010011
v_5	00011000	00011001	00011001	00011011
v_6	10011100	10011101	10011101	10011011
v_7	00010100	00010101	00010101	00010011
v_8	10001000	10001001	10001001	10001011
TC01††	9	9	9	7
e	na	0.028	0.048	0.092
Step (k, nTrunk)	4 (4, 1)	~~5 (5, 1)~~	9 (5, 5)	19 (7, 5)
v_0	00010111	~~00010011~~	00010111	00010111
v_1	00010111	~~00010011~~	00010111	00010111
v_2	00010111	~~00010011~~	00001101	00011101
v_3	10100111	~~10110011~~	10101101	10011101
v_4	00010111	~~00010011~~	00010111	00010111
v_5	00010111	~~00010011~~	00010111	00010111
v_6	10010111	~~10010011~~	10011000	10011000
v_7	00010111	~~00010011~~	00011000	00011000
v_8	10000111	~~10010011~~	10001000	10001000
TC01	5	~~4~~	8	6
e	0.122	~~0.313~~	0.159	0.160

† k is the number of LSBs that are perturbed
‡ $nTrunk$ is the number of trunks the vector is split into.
†† TC01 is the $0- > 1$ toggle count of this vector.

2.6 Fine Tuning

To restore the accuracy loss after weight scaling and bit perturbation, we fine tune the weights by incorporating training into those steps. Inspired by the training scheme of binarized neural network (BNN) [8], we keep both floating point weights and 8-bit scaled and perturbed fixed-point weights during training. The flowchart illustrating the fine tuning procedure is in Fig. 2-5. Starting from the pre-trained floating-point

weights of the compressed CNN, we applied the methods introduced in Section 2.3–2.5 sequentially to generate the scaled and tuned 8-bit sign-magnitude weights. This set of weights is used during the forward pass of NN training process to calculate the loss. Standard gradient descent algorithm is used to train the NN. During the back-propagation phase, the gradients are applied to the floating point weights. This process is repeated until loss converges.

Table 2.6 summarizes the experimental results of bit tuning. Three set of experiments on different NN models are conducted. NNs are trained using the experimental setup in [37]. The classification accuracy on the test set is reported. The fc1 and fco layers of CNN-1-decomp are kept unchanged. Other layers are tuned with $e_{max} = 0.1$ and weight scaling is skipped. The entire CNN-2-decomp model goes through all tuning steps and e_{max} is set to be 0.15 for bit perturbation. CNN-3-decomp is tuned similarly, but with e_{max} of 0.2. As shown, the bit tuning algorithm greatly reduces the switching activity with insignificant accuracy loss and the accuracy loss can be restored partly if fine tuning is applied. The user can control the accuracy loss by setting e_{max} and choosing which layer(s) to tune and which step(s) to use.

Table 2.6: A summary of classification accuracy and toggle count reduction of bit tuning algorithm.

CNN	raw 8-bit fixed-point	tuned fixed-point w/o fine tuning	tuned fixed-point w/ fine tuning	reduction in toggle count
CNN-1-decomp	87.8%	86.9%	87.3%	41.48%†
CNN-2-decomp	90.75%	89.5%	90.0%	60.95%
CNN-3-decomp	81.1%	77.4%	79.7%	59.50%

† Only tuned layers are counted.

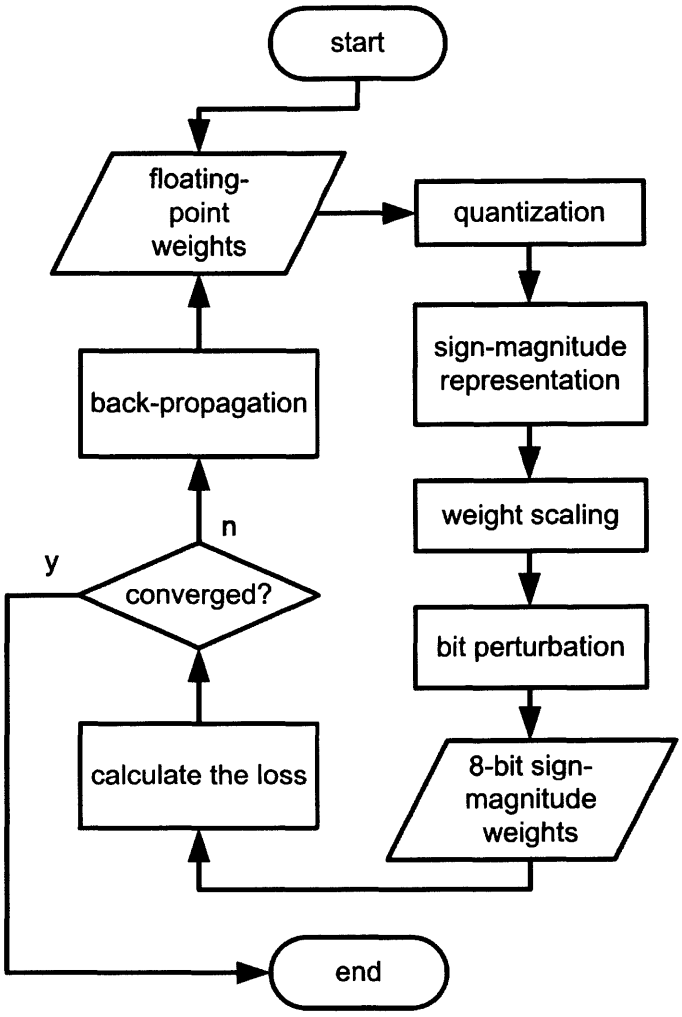

Figure 2-5: The flowchart of fine tuning.

Input: 1-D vector of data $\mathbf{D}$ with length of $1 \times L$ and bitwidth of K, maximum relative error e_{max}

Output: 1-D vector of data $\mathbf{D_{best}}$ with bit tuned for lowest toggle count

```
1  TC ← CountToggle(D);
2  minTC ← TC;
3  D_best ← D;
4  e_best ← ∞;
5  maxTrunk ← L/2;
6  for k = 1 to K do
7      contSplit ← True;
8      ntrunk ← 1;
9      while contSplit = True do
10         split the vector into ntrunk subsets;
11         for each subset do
12             avg ← AverageK(D, k);
13             /* average the value of last k bits of each original
                   coefficient in subset */;
14             a new subset of coefficients ← ReplaceK(subset, avg);
15             /* generate a new subset of coefficients by replacing
                   last k original bits of each coefficient in subset
                   with the average */;
16         end
17         e ← (D_new − D)/D;
18         e_ave ← sum(e)/L;
19         if e_ave ≤ e_max then goodPerf ← True;
20         else goodPerf ← False;
21         newTC ← CountToggle(D_new);
22         if goodPerf then
23             contSplit ← False;
24             if newTC < minTC OR (newTC = minTC AND e_ave < e_best) then
25                 minTC ← newTC;
26                 D_best ← D_new;
27                 e_best ← e_ave;
28             end
29         end
30         if ntrunk < maxTrunk then ntrunk ← ntrunk +1;
31         else break the while loop;
32     end
33     if ntrunk = maxTrunk AND goodPerf = False then
34         break the for loop;
35     end
36 end
```

Algorithm 3: Pseudo-code for bit perturbation.

Chapter 3

Dataflow Analysis

Various types of dataflows, including weight stationary (WS), output stationary (OS), row stationary (RS) and no-local-reuse (NLR) dataflows, have been proposed. This chapter presents a comprehensive analysis of the total number of memory accesses needed for these dataflows under certain hardware resource constraints. Based on the analysis, a comparison of different dataflows applied to the decomposed NNs, e.g. CNN-1-decomp, trained for KWS tasks is made.

3.1 Convolutional Neural Network (CNN)

A CNN is mainly composed of a series of convolutional (CONV) layers and fully-connected (FC) layers. CONV layers are basically tensor computation as illustrated in Fig. 3-1. It takes in 3-D input activations. A 4-D convolution filter, which does point-wise multiplication and summation, slides across the input activation map and generate a 3-D output activations. We use blue, green and orange to represent input activations, filter and output activations in this thesis. The mathematical expression

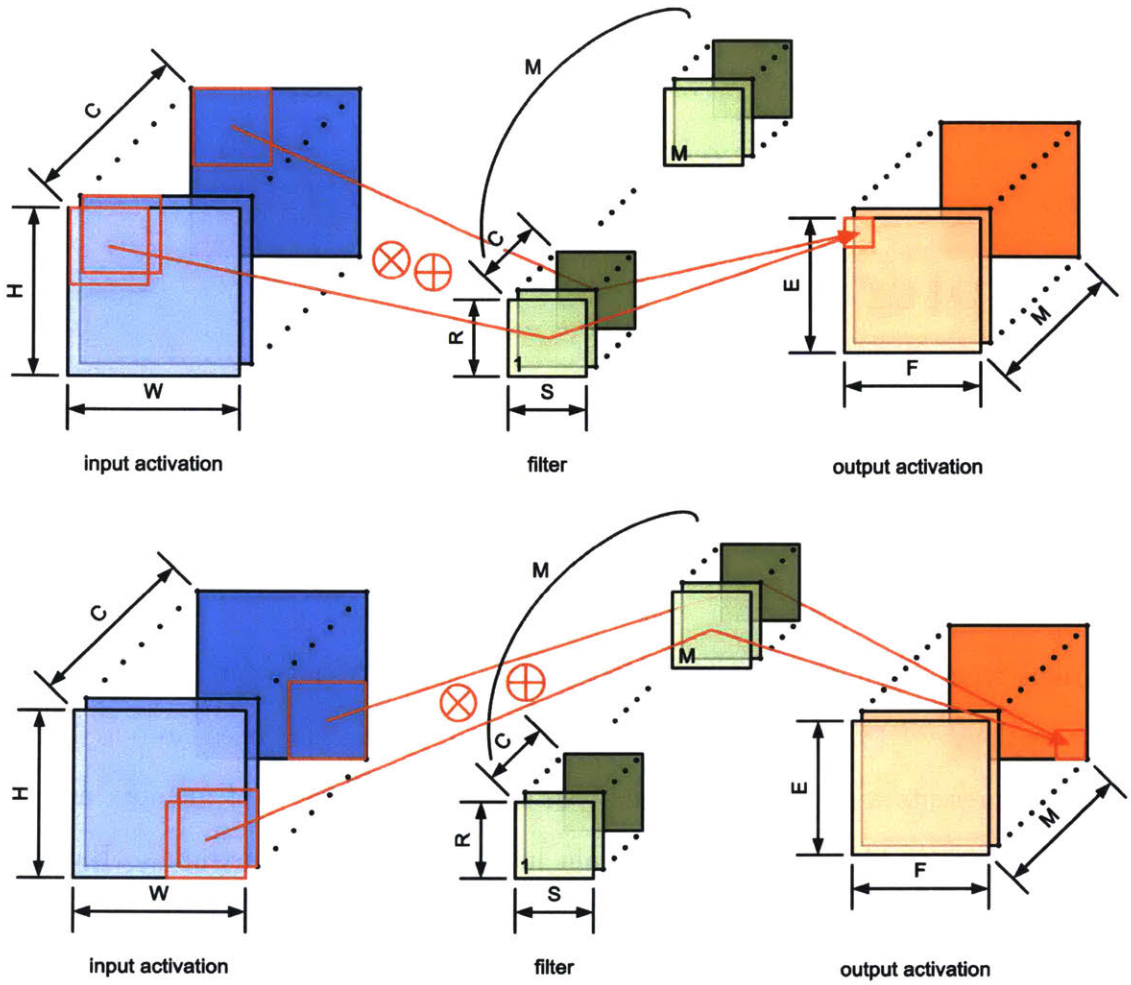

Figure 3-1: An illustration of calculation in a CONV/FC layer.

is shown in Eq. 3.1.

$$
\begin{aligned}
O[m][e][f] = Activations(&B[m] \\
&+ \sum_{r=0}^{r=R-1} \sum_{s=0}^{s=S-1} \sum_{c=0}^{c=C-1} I[c][Ue+f][Vf+s] \times W[m][c][r][s]), \\
&0 \leq m < M, 0 \leq e < E, 0 \leq f < F, \\
&E = (H - R + U)/U, F = (W - S + V)/V
\end{aligned}
\tag{3.1}
$$

where O, I, B and W stand for output activations (named partial sum if addition is partly finished), input activations, biases and weights respectively. *Activations*

represents the activation function, e.g. ReLU [24]. And the symbol representing different dimensions of data is summarized in Table 3.1 and it will be used in this thesis. During training or doing some image processing tasks, input and output activations have an additional dimension called batch size. It is to stack multiple input/output activations together and process at the time. For our KWS task, batch size is set to 1, because stacking multiple input activations increases the latency of the system and affect the real-time processing of speech signal. FC layer is a special case of CONV layer, where the height and width of input activations, output activations and the filter are all equal to 1.

Table 3.1: Symbols to represent different dimensions of data.

C	Number of channels of input activations
H	Height of input activations
W	Width of input activations
R	Height of the filter (weights)
S	Width of the filter (weights)
M	Number of channels of output activations
E	Height of output activations
F	Width of output activations
U	Vertical stride of convolution
V	Horizontal stride of convolution

3.2 Convolution Loop Nest

To facilitate our analysis of the number of memory accesses of different dataflow, a C-like loop representation is used. Corresponding to Eq. 3.1, the basic loop nest [25] of a CONV layer is shown in Fig. 3-2. There is no relationship between different loops and thus the sequence of loops can be arbitrarily permuted. A `for` loop represents temporal reuse of the same hardware structure. Spatial hardware structure for parallel

computation is represented by `parallel-for`. Fig. 3-2 corresponds to a hardware implementation with only one multiplier and one adder. If `for` in line 1 is replaced by `parallel-for`, the hardware implementation does parallel computation across M output channels and has M multiplier-and-adders.

```
1  for (m=0; m<M; m++) {
2    for (e=0; e<E; e++) {
3      for (f=0; f<F; f++) {
4        O[m][e][f] = B[m];
5        for (r=0; r<R; r++) {
6          for (s=0; s<S; s++) {
7            for (c=0; c<C; c++) {
8              O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
9            }
10         }
11       }
12       O[m][e][f] = Activation(O[m][e][f]);
13     }
14   }
15 }
```

Figure 3-2: Loop nest of a CONV layer.

3.3 Comparison of Different Dataflows

For naive implementation of Fig. 3-2 that reads data from SRAM for every computation, the number of memory accesses for input activations and weights is $M \times E \times F \times R \times S \times C$, equal to the number of loops. And the number of accesses for output activations is twice as many, since both read and write are needed for one calculation. For a typical NN for KWS as shown in Table 2.2, the total number of memory accesses needed for one classification is 8.6M.

To reduce the number of memory accesses, spatial and temporal reuse of data should be maximized. It can be done by loop sequence permutation, implementing spatial architecture and adding local storage. Although NN accelerators may have

different structures and storage hierarchy, the methods used to reduce the number of memory accesses can be grouped into different dataflows, including weight stationary (WS), output stationary (OS), row stationary (RS) and no-local-reuse (NLR) [5].

This section presents our analysis of different dataflows. Considering the decomposed CNN architecture shown in Table 2.2, our analysis is based on the following assumptions.

- Output channels (M) and input channels (C) is much larger than height and width of filter (R, S) for most layers in the CNN for KWS.
- The product of height and width ($R \times S$) of filter is smaller than the product of height and width ($E \times F$) of the output activation and roughly equal to E (to assume equal amount of total local storage of the PE array).
- The NN accelerator has an array of processing elements (PEs). Each PE has a multiplier, an adder and a fixed amount of local storage. Partial sum can be accumulated temporally inside a PE or spatially across multiple PEs if needed.
- The total local storage size of the PE array is limited by $max(R \times S \times C, R \times S \times M, E \times C, E \times M, C \times M)$
- No tiling is considered.

To focus on the critical part of CONV layer, biases and the activation function are ignored in the following loop nests.

3.3.1 Weight Stationary (WS)

WS dataflow is to maximize the reuse of weights. For a design with one multiplier and one adder, a loop nest representing WS is shown in Fig. 3-3. The loops of weights indices are kept in the outer part of the loop nest. Thus $W[m][c][r][s]$ can be hold

in the local storage and temporally reused during the loops of line 4–6. The number of weight accesses is minimized. However, I and O are fetched for every loop and no reuse opportunity is available.

The number of memory accesses of this simple case is shown in Table 3.2. The total number of W read accesses ($M \times C \times R \times S$) equals to the total number of weights, which means that weights are only accessed once. That is by design, since WS dataflow maximizes the reuse of weights. To do convolution, filters slide across the input activation with the strides of U and V. Thus the effective size of input activation used in convolution is $C \times R \times S \times E \times F$ instead of $C \times H \times W$. In this simple implementation of WS dataflow, the total number of I read is M times the effective size of I. The reason is that there is a `for` loop of M in the outer part of the loop nest. Similarly, $C \times R \times S$ times of accumulation are needed to produce the final output activation as shown the loops of line 2–4. Thus $(2 \times C \times R \times S - 1)$ times the total number of output activation $M \times E \times F$ is the number of O read/write. This simple implementation results in inefficient read/write pattern of I and O.

```
1  for (m=0; m<M; m++) {
2      for (r=0; r<R; r++) {
3          for (s=0; s<S; s++) {
4              for (c=0; c<C; c++) {
5                  for (e=0; e<E; e++) {
6                      for (f=0; f<F; f++) {
7                          O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
8                      }
9                  }
10             }
11         }
12     }
13 }
```

Figure 3-3: A simple case of WS.

To reduce the number of accesses to input and output activations and also increase throughput, parallel processing and larger local storage can be used. According to our assumptions, all unique design choices are listed in Table 3.3.

Table 3.2: The number of memory accesses of the simple case of WS.

W read	$M \times C \times R \times S$
I read	$M \times C \times R \times S \times E \times F$
O read & write	$(2 \times C \times R \times S - 1) \times M \times E \times F$

Table 3.3: A summary of different implementations of WS dataflow.

name	`parallel-for`	stored locally
WS-1	M	C
WS-2	M	R, S
WS-3	C	R, S
WS-4	R, S	C

WS-1

WS-1 is to compute all output channels in parallel using M multipliers and adders and keeping $W[m][0][r][s], W[m][1][r][s], ..., W[m][C-1][r][s]$ in the local storage of a PE. Fig. 3-4 illustrates the mapping of a CONV layer to a PE array. $1 \times 1 \times C$ weights from different output channels are delivered to different PEs. $1 \times 1 \times C$ input activations is broadcast to all PEs and spatially reused across M output channels. All the $1 \times 1 \times C$ input activations that needs to be convoluted by the loaded weights are broadcast sequentially into PEs. A new set of weights will not be delivered until the current set of weights has been multiplied with all the input activations. The loop nest representation of WS-1 is shown in Fig. 3-5. Line 6–8 represents the calculation inside a single PE and the local storage of $1 \times 1 \times C$ weights. Loop in line 4–5 is to multiply the current set of weights with all possible input activations. Different sets of weights are calculated in the loops of line 2–3. Line 1 indicates the spatial structure of the PE array.

The number of memory accesses of WS-1 is summarized in Table 3.4. Compared with Table 3.2, the number of read of input activations is reduced M times, because of the parallel computation across M output channels. Besides, the number of read and write to the output activation memory is roughly $1/C$ of that of the simple case,

since partial sum is accumulated C times inside each PE. When $C = 1$, e.g. the first layer of NN for KWS, WS-1 can be adjusted to keep a row of weights (corresponding to the s dimension) in the local storage. This brings around $1/S$ reduction in the total number of output activation read.

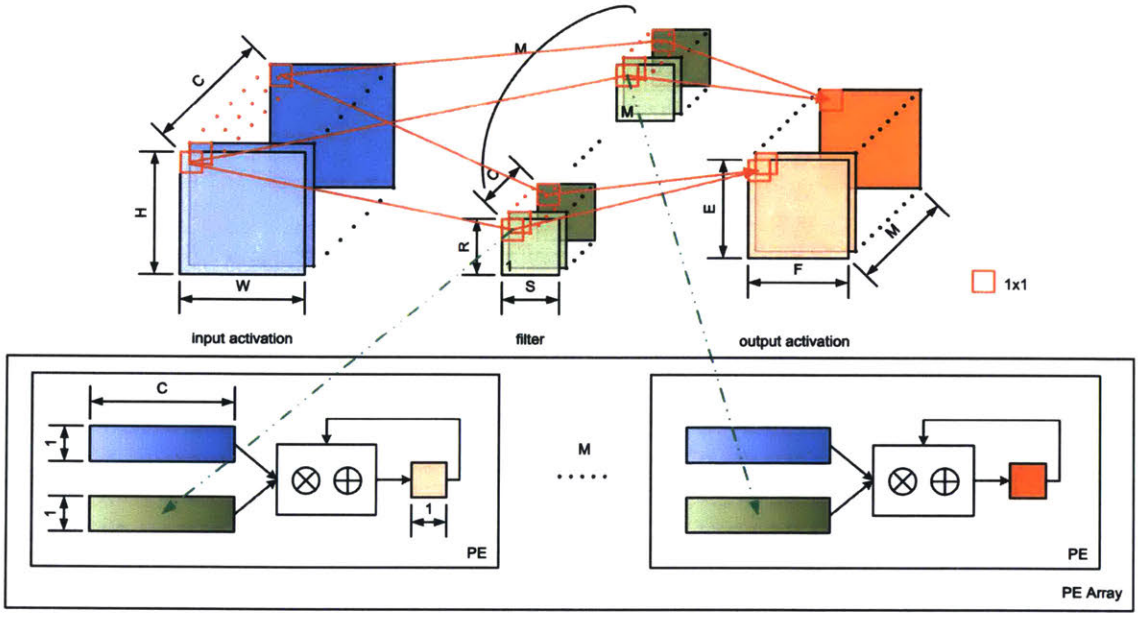

Figure 3-4: The illustration of WS-1. A register file with the size of $1 \times C$ for input activations is drawn in PE for illustration. Only one register is needed in actual implementation.

```
parallel-for (m=0; m<M; m++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (e=0; e<E; e++) {
        for (f=0; f<F; f++) {
          for (c=0; c<C; c++) {
            O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
          }
        }
      }
    }
  }
}
```

Figure 3-5: Loop nest of WS-1.

Table 3.4: The number of memory accesses of WS-1.

W read	$M \times C \times R \times S$
I read	$C \times R \times S \times E \times F$
O read & write	$(2 \times R \times S - 1) \times M \times E \times F$

WS-2

Similar to WS-1, WS-2 also computes the output channels in parallel. But instead of keeping the $1 \times 1 \times C$ weights in the local storage, WS-2 stores $W[m][c][0][0], W[m][c][0][1]$, $..., W[m][c][0][S-1], ..., W[m][c][R-1][S-1]$ in each PE. The mapping of a CONV layer is depicted in Fig. 3-6. Fig. 3-7 shows the loop nest of WS-2. Differ from WS-1, the loop of C is moved to the outer part, which represents the calculation of different sets of weights. And the loops of R and S are moved to the inner part, and thus are computed inside a PE.

From Fig. 3-7, we can calculate the number of memory accesses of WS-2 as shown in Table 3.5. The major difference between WS-1 and WS-2 is the accesses to output activation memory. For a CONV/FC layer that has $R \times S > C$, the number of memory accesses of WS-1 is bigger than that of WS-2, and vice versa. The number of I read may be slightly reduced, if we keep the overlapping parts of input activations (shown in Fig. 3-6) stored in the PE and only broadcast the rest when calculating the loop of line 4.

Table 3.5: The number of memory accesses of WS-2.

W read	$M \times C \times R \times S$
I read	$C \times E \times R \times W$
O read & write	$(2 \times C - 1) \times M \times E \times F$

WS-3

In the WS-3 dataflow, C input channels of input activations and weights are multiplied in parallel. Similar to WS-2, $R \times S$ weights are kept in the local storage of PEs. Data

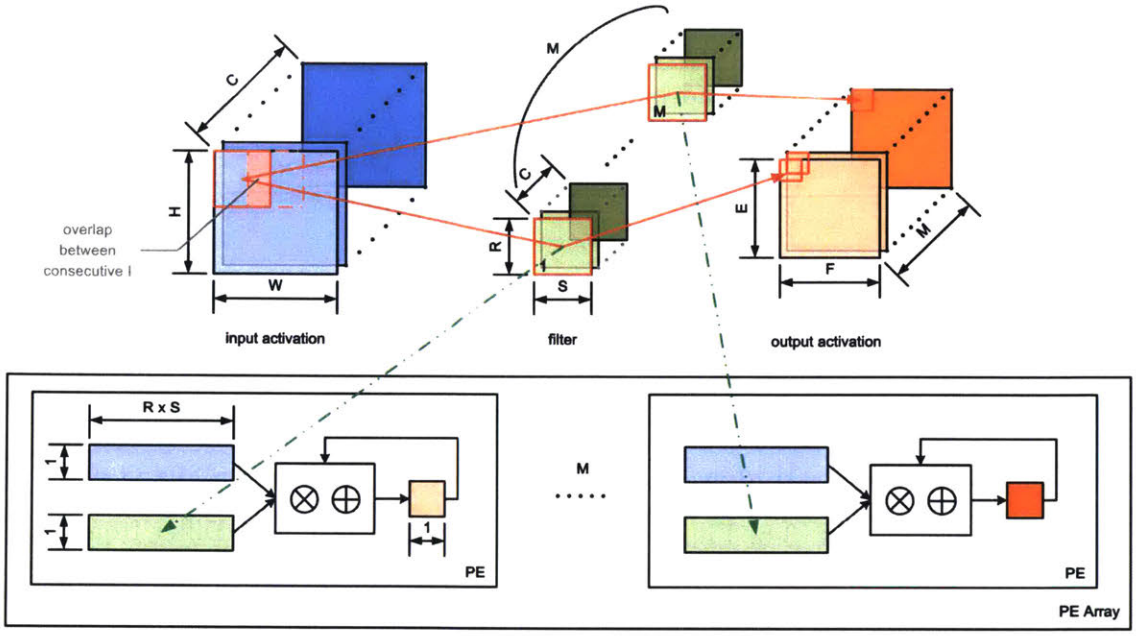

Figure 3-6: The illustration of WS-2. A register file with the size of $1 \times R \times S$ for input activations is drawn in PE for illustration. Only one register is needed in actual implementation.

```
1  parallel-for (m=0; m<M; m++) {
2    for (c=0; c<C; c++) {
3      for (e=0; e<E; e++) {
4        for (f=0; f<F; f++) {
5          for (s=0; s<S; s++) {
6            for (r=0; r<R; r++) {
7              O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
8            }
9          }
10       }
11     }
12   }
13 }
```

Figure 3-7: Loop nest of WS-2.

are mapped to the PE array as shown in Fig. 3-8. The weights in one output channel are stored in the PE array and are multiplied with all the input activations to generate $E \times F$ output activations in one channel. Then weights in next output channel will be loaded and the output activations in the next channel will be computed. This

computation flow is depicted in Fig. 3-9.

Table 3.6 concludes the number of memory accesses of WS-3 dataflow. It is shown that WS-3 minimizes the number of accesses to weights and output activations, but the reuse opportunity of input activations is little.

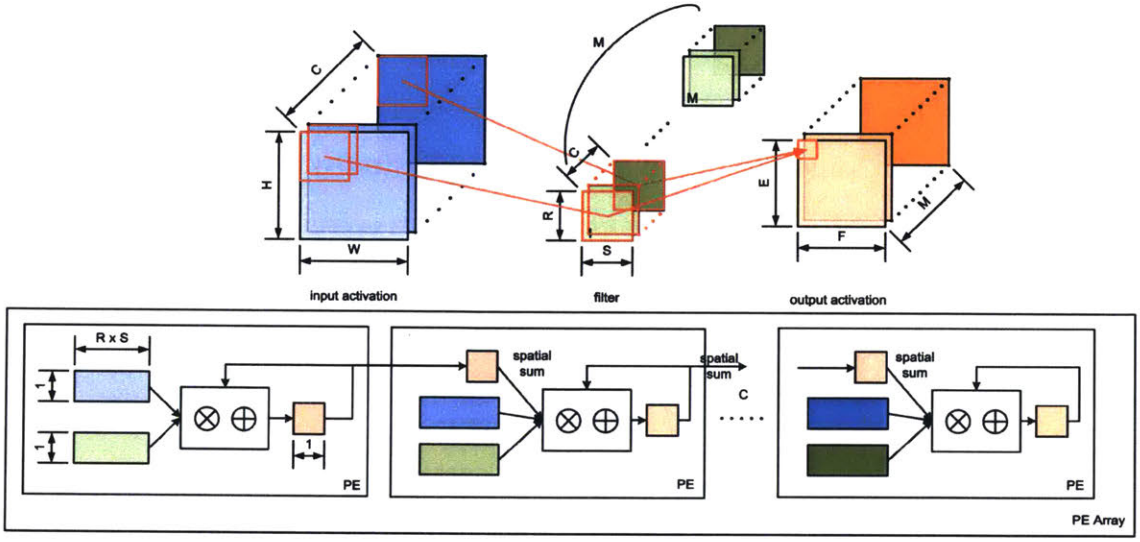

Figure 3-8: The illustration of WS-3. A register file with the size of $1 \times R \times S$ for input activations is drawn in PE for illustration. Only one register is needed in actual implementation.

```
1  parallel-for (c=0; c<C; c++) {
2    for (m=0; m<M; m++) {
3      for (e=0; e<E; e++) {
4        for (f=0; f<F; f++) {
5          for (s=0; s<S; s++) {
6            for (r=0; r<R; r++) {
7              O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
8            }
9          }
10       }
11     }
12   }
13 }
```

Figure 3-9: Loop nest of WS-3.

Table 3.6: The number of memory accesses of WS-3.

W read	$M \times C \times R \times S$
I read	$M \times C \times E \times R \times W$
O read & write	$M \times E \times F$

WS-4

WS-4 is to use `parallel-for` in the r and s loop and store C weights inside each PE. It has roughly the same effect as WS-3, which is illustrated in Fig. 3-10, Fig. 3-11 and Table 3.7.

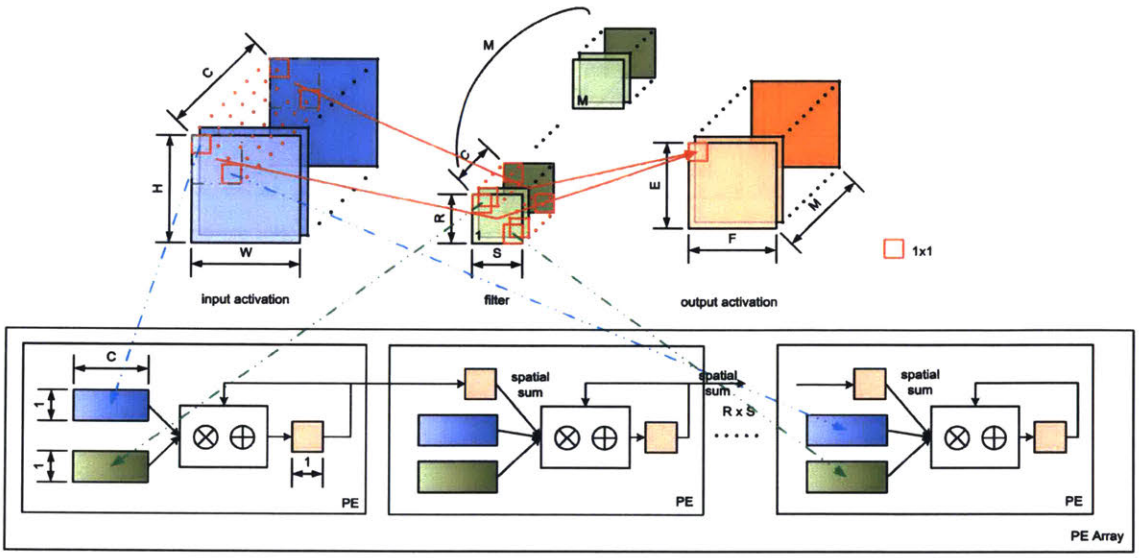

Figure 3-10: The illustration of WS-4. A register file with the size of $1 \times C$ for input activations is drawn in PE for illustration. Only one register is needed in actual implementation.

Table 3.7: The number of memory accesses of WS-4.

W read	$M \times C \times R \times S$
I read	$M \times C \times R \times S \times E \times F$
O read & write	$M \times E \times F$

```
1  parallel-for (r=0; r<R; r++) {
2    parallel-for (s=0; s<S; s++) {
3      for (m=0; m<M; m++) {
4        for (e=0; e<E; e++) {
5          for (f=0; f<F; f++) {
6            for (c=0; c<C; c++) {
7              O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
8            }
9          }
10       }
11     }
12   }
13 }
```

Figure 3-11: Loop nest of WS-4.

3.3.2 Output Stationary (OS)

OS dataflow is to maximize the reuse of output activations or partial sums. Fig. 3-2 is a simple example of OS dataflow with only one multiplier and one adder. As opposed to the simple WS dataflow, all indices related to O are kept in the outer loops. Therefore, no data movement between PE and O memory is needed before an output activation is computed. Table 3.8 summarizes the number of memory accesses of the simple implementation of OS. While O read & write is minimized, the access patterns to W and I are inefficient. All the weights are accessed $E \times F$ times and the total number of I read is M times the effective size of input activations.

Table 3.8: The number of memory accesses of the simple case of OS.

W read	$E \times F \times M \times C \times R \times S$
I read	$M \times C \times R \times S \times E \times F$
O read & write	$M \times E \times F$

The same approach introduced in WS section can be taken to optimize the implementation of OS. Limited by our assumptions, the possible implementations are summarized in Table 3.9.

Table 3.9: A summary of different implementations of OS dataflow.

name	`parallel-for`	stored locally
OS-Chn	M	E
OS-EF	E, F	none

OS-Chn

OS-Chn is to compute each columns of the output activation locally inside a PE and M output channels are computed in parallel, as shown in Fig. 3-12. All the input activations and weights used to compute the column of output activations are delivered sequentially into the PE, following the sequence specified by the loops of line 2–5.

The total number of memory accesses can be calculated as listed in Table 3.10. The number of W read is $1/E$ of that of the simple implementation, due to the local storage of E output activations. The number of I read is reduced by M, thanks to the spatial structure of the PE array.

Table 3.10: The number of memory accesses of OS-Chn.

W read	$F \times M \times C \times R \times S$
I read	$C \times E \times R \times F \times S$
O read & write	$M \times E \times F$

OS-EF

OS-EF is to compute the $E \times F$ output activations of one output channel in parallel. Since $E \times F$ output activations almost take up all the storage the PE array can obtain under our assumption, only one register for each type of data is allowed in a PE. The general structure of a PE array is illustrated in Fig. 3-14. The weights are multicast to all PEs in the PE array following the sequence specified in the loops of line 3–6 in Fig. 3-15. Different PE gets different input activations corresponding to the index of the output activation it is calculating.

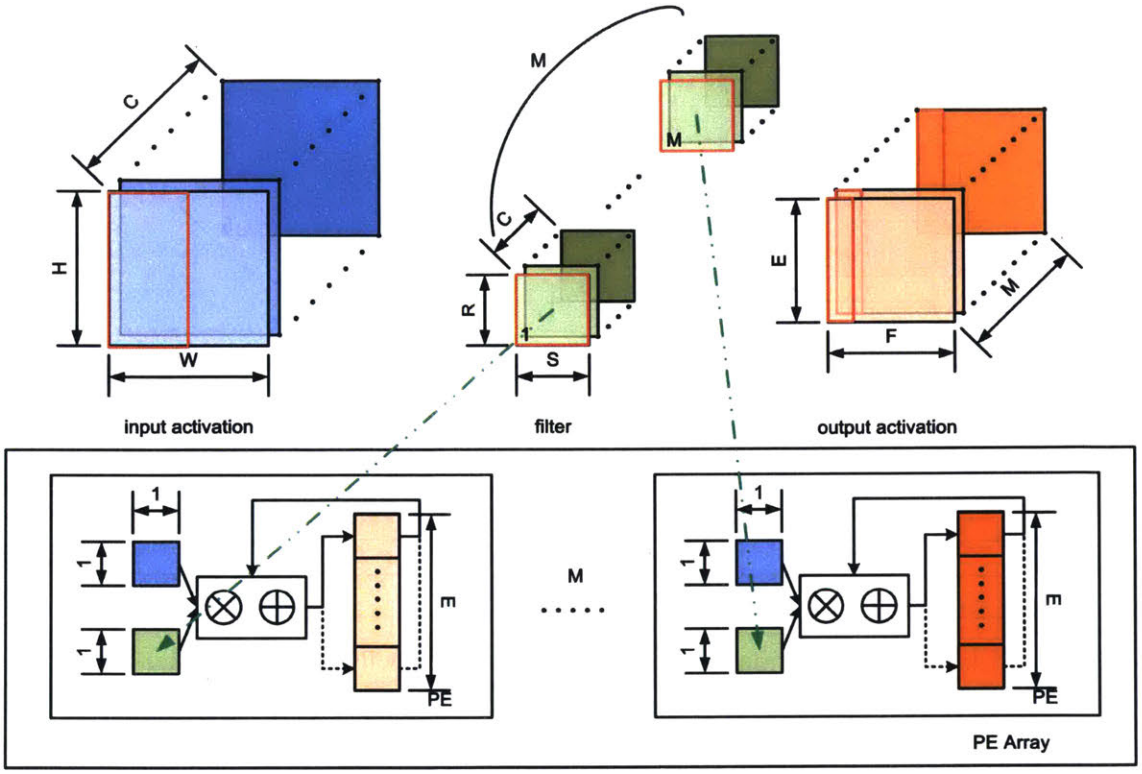

Figure 3-12: The illustration of OS-Chn.

```
parallel-for (m=0; m<M; m++) {
  for (f=0; f<F; f++) {
    for (c=0; c<C; c++) {
      for (r=0; r<R; r++) {
        for (s=0; s<S; s++) {
          for (e=0; e<E; e++) {
            O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
          }
        }
      }
    }
  }
}
```

Figure 3-13: Loop nest of OS-Chn.

Table 3.11 summarizes the number of memory accesses of this dataflow. Since the loops of E and F are done in parallel, the accesses to weights is also minimized in this dataflow.

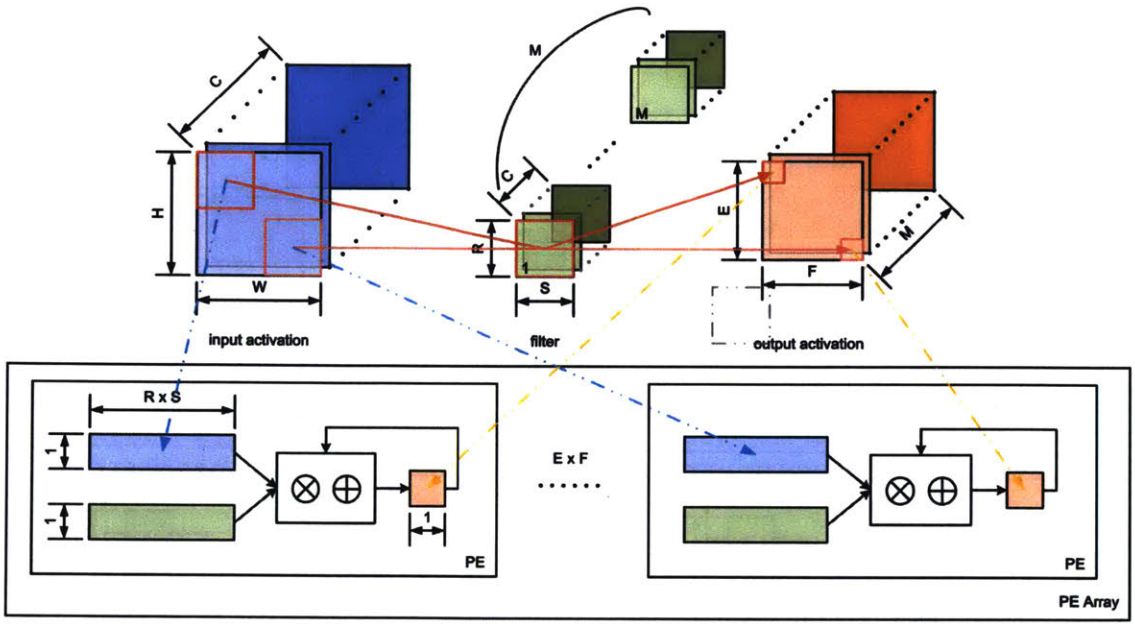

Figure 3-14: The illustration of OS-EF. Register files with the size of $1 \times R \times S$ for input activations and weights are drawn in PE for illustration. Only one register is needed for both data in actual implementation.

```
parallel-for (e=0; e<E; e++) {
  parallel-for (f=0; f<F; f++) {
    for (m=0; m<M; m++) {
      for (c=0; c<C; c++) {
        for (r=0; r<R; r++) {
          for (s=0; s<S; s++) {
            O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
          }
        }
      }
    }
  }
}
```

Figure 3-15: Loop nest of OS-EF.

Table 3.11: The number of memory accesses of OS-EF.

W read	$M \times C \times R \times S$
I read	$M \times H \times W \times C$
O read & write	$M \times E \times F$

3.3.3 Row Stationary (RS)

Different from all the previous dataflows, RS dataflow does not focus on minimizing the number of memory accesses to a single data type. It tries to maximize the overall data reuse. As shown in Fig. 3-16, each PE stores S weights and input activations and generates one partial sum. Different rows of weights are mapped to different rows of PEs. Hence the partial sums can be accumulated vertically and weights are reused spatially between different columns of PEs. Besides, the input activations are reused diagonally as shown in the diagram. The loop nest of RS dataflow is shown in Fig. 3-17. The loop of line 5 does not affect $W[m][c][r][s]$, thus W can be reused temporally. The consecutive execution of loops of F and S makes it easy for the temporal reuse of overlapping input activations.

The total number of memory accesses is shown in Table 3.12. The number of weight access is minimized, as a result of the `parallel-for` loop of E and the temporal reuse across the loop of F dimension.

Table 3.12: The number of memory accesses of RS.

W read	$M \times C \times R \times S$
I read	$M \times H \times W \times C$
O read & write	$(2 \times C - 1) \times M \times E \times F$

3.3.4 No Local Reuse (NLR)

NLR is proposed for large NN that cannot be stored on-chip. It minimizes the area of PE array by eliminating local storage. Chip area is saved for using a large global buffer which can minimize the off-chip memory access. Since NNs for KWS task can be stored on chip, NLR dataflow is not suitable in our case.

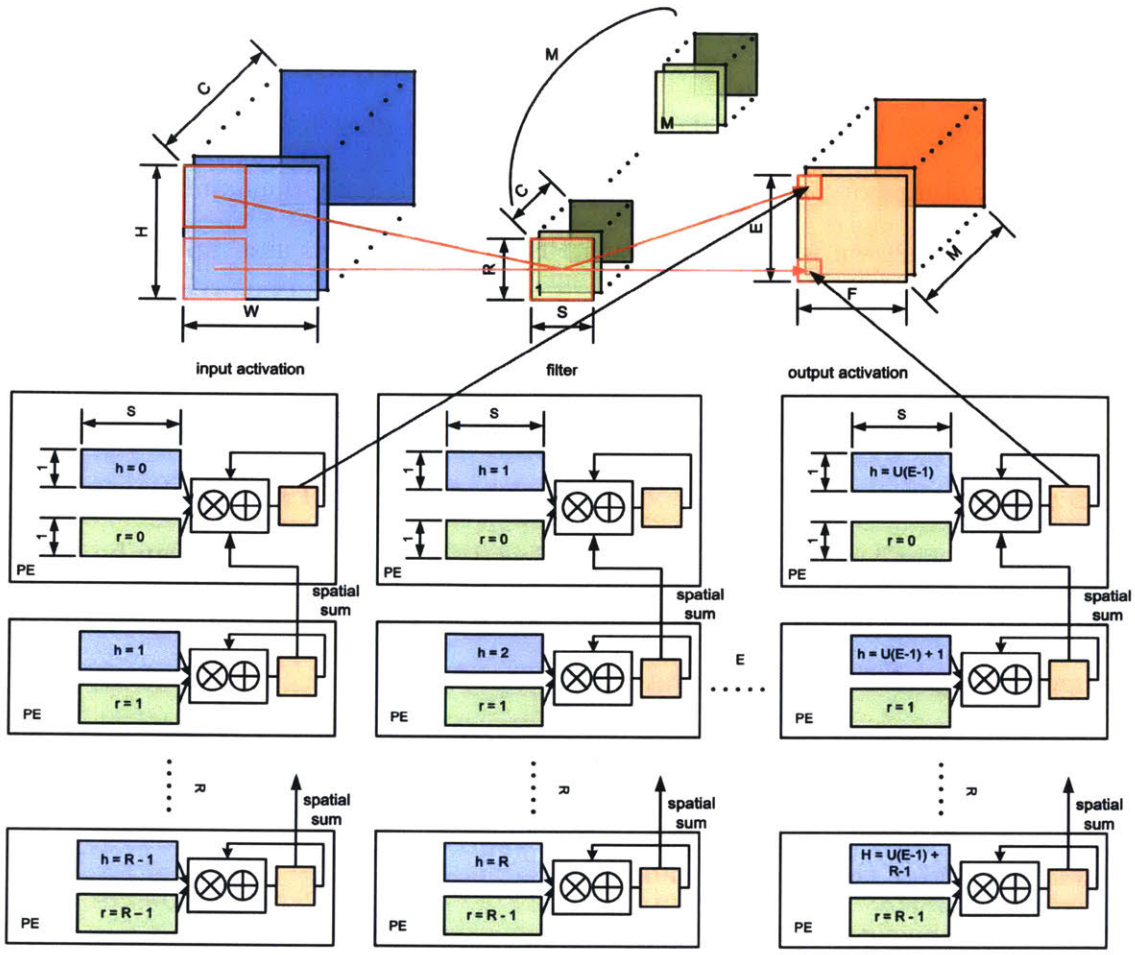

Figure 3-16: The illustration of RS.

```
for (m=0; m<M; m++) {
  for (c=0; c<C; c++) {
    parallel-for (e=0; e<E; e++) {
      parallel-for (r=0; r<R; r++) {
        for (f=0; f<F; f++) {
          for (s=0; s<S; s++) {
            O[m][e][f] += I[c][Ue+r][Vf+s] × W[m][c][r][s];
          }
        }
      }
    }
  }
}
```

Figure 3-17: Loop nest of RS.

3.4 Results

The comparison of different dataflows are made on our targeted decomposed NN structure shown in Table 2.2. The number of memory accesses of all layers are calculated using the formula in Table 3.4–3.7, 3.10–3.12. The energy cost per memory access depends on the size and shape of a memory bank. Although it may vary across different types of data, it is safe to assume that the difference is within an order of magnitude based on the datasheet of SRAMs. Thus the total number of memory accesses of all data is a good estimate of the normalized overall energy consumption. Although the implementation of the hardware, such as the number of PEs in the PE array and the size of local storage, will influence the actual number of memory accesses, this analysis gives a general guideline before doing actual hardware implementation.

As shown in Fig. 3-18, WS-1 dataflow have the smallest number of memory accesses on this compressed NN. After tensor decomposition, many filters have R and S both equal to 1. This leads to low utilization of PE array of WS-4 and RS. And since $R \times S$ is much less than C and M for most layers, WS-1 outperforms WS-2, WS-3 and WS-4. And CNNs for KWS usually have nearly half of the layers to be FC layers, where E, F, H and W equal to 1. As a result, OS-EF is not the most effective dataflow and the PE utilization is also impaired using RS. Based on our analysis, WS-1 is the most flexible dataflow in terms of handling any values of R, S, E, F, H and W. Hence we use it in our hardware implementation.

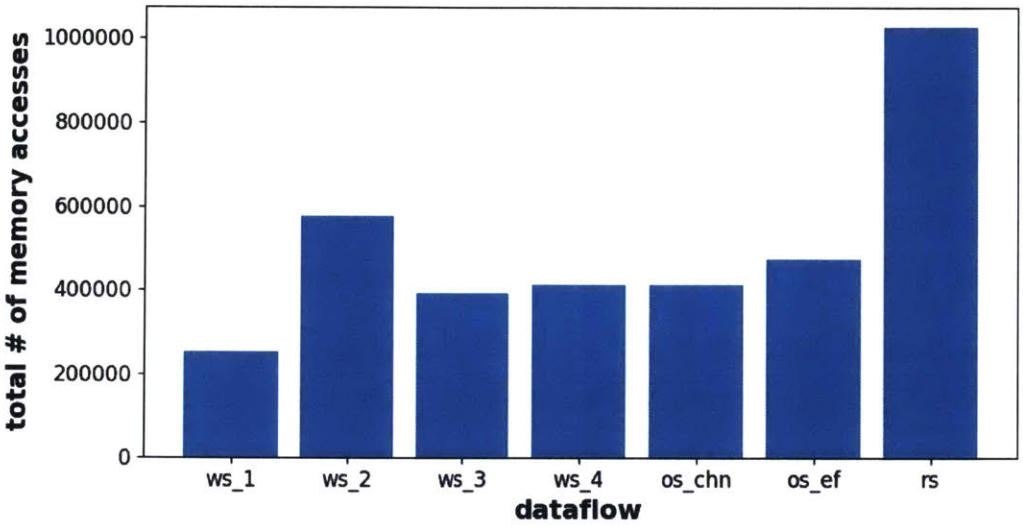

Figure 3-18: The comparison of different dataflows on CNN-1-decomp.

Chapter 4

Hardware Design

4.1 Overview

The system architecture of our NN accelerator for KWS is shown in Fig. 4-1. The accelerator has one clock domain and one power domain. The weights and configuration bits of the entire NN are loaded through the host interface at the start-up time. 80kB of weights and around 1kB of configuration bits (12 layers) are supported. The NN accelerator only needs to be setup once and can work independently to detect keywords in real-time. Extracted speech features are continuously sent to the feature buffer through the host interface. Layers are processed one-after-another by the PE array. The top-level controller configures the PE array according to the configuration bits of the layer before the processing of each layer. And then it generates the memory accessing sequence to read the weights and the input activations. Data are sent to PEs through network-on-chip (NoC) following the configuration of PE array. A tree-structured network is used. Spatial sum is supported in two consecutive PEs in a chain from PE0 to PE63 as shown in Fig. 4-1. After all layers have been processed, the classification results are stored in the activation memory and can be read through the host interface.

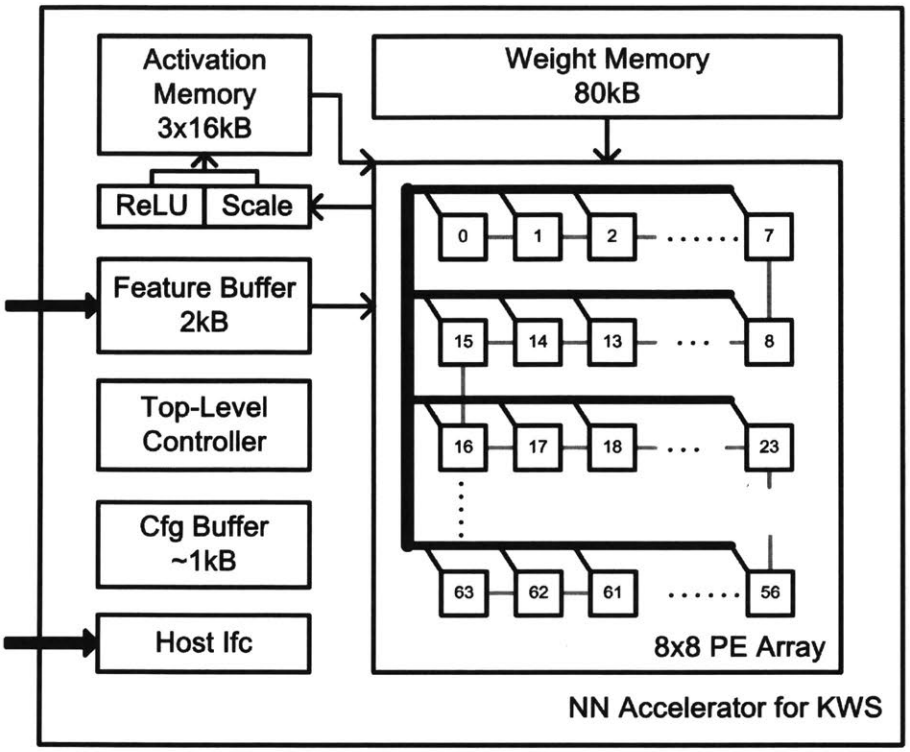

Figure 4-1: System architecture of the NN accelerator for KWS.

4.2 Processing Element (PE) Structure

Fig. 4-2 illustrates the structure of a PE. Data are received and sent out through input and output FIFOs to balance the delay during delivery. Each PE has its local controller. It is configured by the top-level controller at the beginning of processing every layer. Once all data needed for this PE has been loaded, the PE starts calculation immediately. Before multiplication and addition, the accumulation register can be initialized to store a bias, a partial sum or zero, according to the configuration. PE supports multiplication of 16-bit input activations and 8-bit weights and addition of two 16-bit numbers. The output of the multiplier and the adder are both truncated and rounded to 16-bit. In our experiment, that leads to little loss in the overall accuracy. PE also has an input port for spatial sum. Outputs of PEs can be passed and accumulated through the PE array.

Data are stored and calculated in two representations inside a PE. Input activa-

tions and weights are stored in sign-magnitude format. They are multiplied using an unsigned multiplier and an XOR gate that generates the sign. The sign-magnitude product is converted back to 2's complement representation and accumulated using an adder-and-subtractor. The sign bit of the product is used as a carry bit to do the conversion. In this way, the output activations are in 2's complement format after computation and need to be converted back to sign-magnitude format before they are used as the input activations of next layer. This is done when we apply activation function to the output activations. The reason for having the multiplication part in sign-magnitude is that using sign-magnitude format lowers the switching activity between weights and results in less accuracy loss during bit tuning. Besides an adder-and-subtractor, another option to handle to sign-magnitude addition is to have the positive numbers and negative numbers accumulate separately using two adders and merged at the final stage. It is not been used in our design, because more bits are needed to separately accumulate all the positive or negative numbers and all the relevant buses, input and output FIFOs also need to be doubled and enlarged.

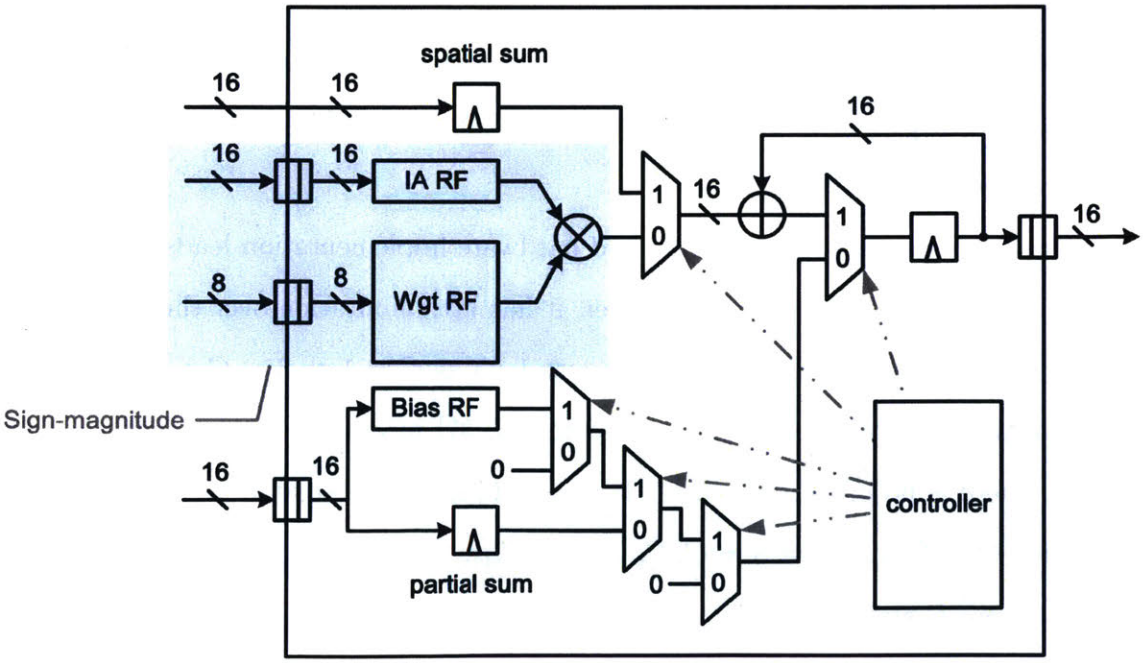

Figure 4-2: The PE structure.

4.3 Mapping Filters to PE Array

Based on previous analysis, WS-1 dataflow is implemented in our NN accelerator. Fig. 4-3 gives an example of mapping a filter with $M = 21$ and $C = 10$ to our PE array that contains 64 PEs. Only the weight storage is depicted in the figure. Input activations and biases are stored according to the mapping of weights.

C weights in one output channel will be split into multiple PEs if $C > 4$. Each PE can be configured to store either 3 or 4 elements in one output channel, so that a large variety of C can be mapped under balanced workload for different PEs. When $C = 10$, 3 PEs are used, one PE storing 4 elements and the other two PEs storing 3 elements. The results calculated by those 3 PEs will be accumulated spatially before stored to the output activation buffer. Each PE can also store weights from multiple output channels. Either 1, 2 or 3 output channels can be held in one PE. For $M = 21$, the first 3 PEs store 2 output channels of weights, and the rest store 1 channels.

To store the maximum number of weights under different configurations, the weight register file has 12 registers. The registers can be gated to save power if unused. Similarly, if the entire PE is unused for this layer, it can also be clock gated. As shown in Fig. 4-3, 63 PEs are enough for a filter with $M = 21$ and $C = 10$ and PE0 can be clock gated.

As shown, this mapping scheme and hardware implementation leads to balanced workload across different PEs. Moreover, it has little limitation over the filter shape. The PE array supports any CONV filter as long as $C = 1$ or $3 \leq C \leq 256$. All the state-of-the-art NNs for KWS proposed in [29, 37] meet this requirement. For FC layers, there is no constraint on C. If $C > 256$, it can be treated as a CONV layer of which $C = C_{map}$, $W = S = C/C_{map}$, $3 \leq C_{map} \leq 256$. Besides, the hardware can support any R and S thanks to the nature of WS-1 dataflow. As a results, the PE array provides large design space for NN architecture designers to look for the best architecture under different scenarios.

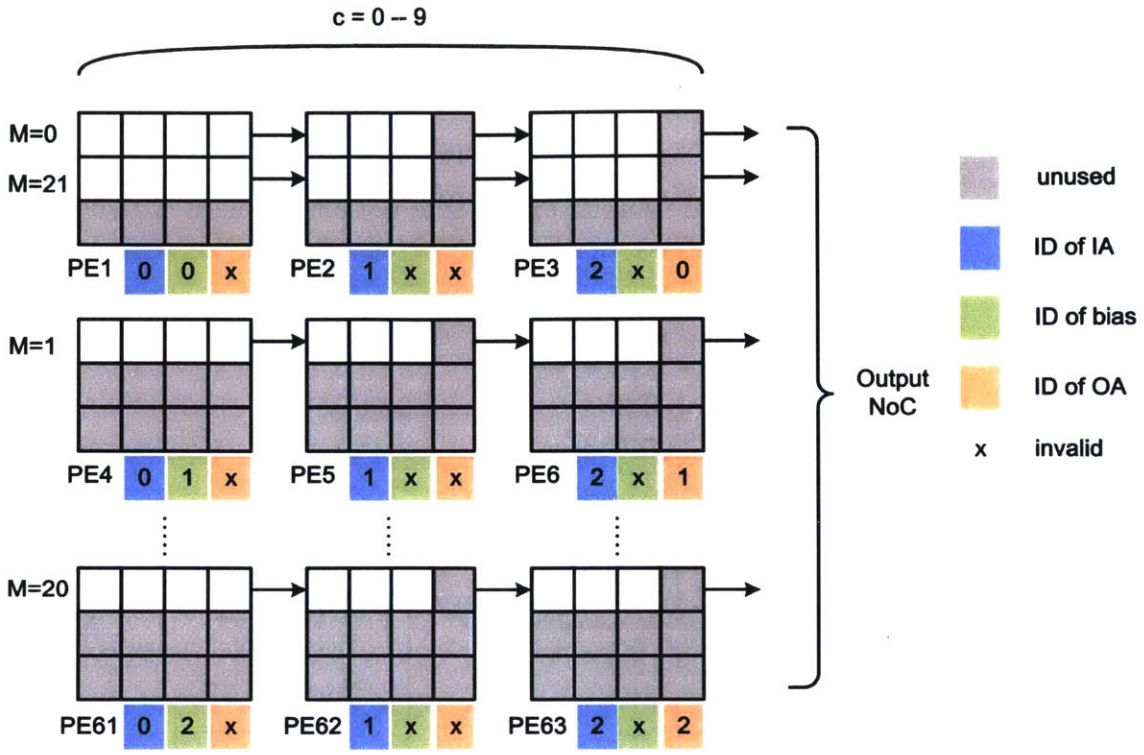

Figure 4-3: An illustration of mapping a filter with $M = 21$ and $C = 10$ to the PE array following the WS-1 dataflow.

4.4 Network-on-Chip

Data are delivered to PEs through network-on-chip. Different data type has different delivery patterns according to the WS-1 dataflow. Each PE needs to process different weights, thus weights are unicast. We adopt the global input network structure of Eyeriss [7] to deliver weights as shown in Fig. 4-4. The PE array are organized into 8 rows and 8 columns. Each rows of the PE array is assigned to a row ID. And PEs in different columns have different column IDs. For example, PE1 has a row ID of 0 and a column ID of 1, and PE14 has a row ID of 1 and a column ID of 7. Since weights are unicast in WS-1 dataflow, the weight IDs are fixed inside the unicast controller. Input activations are shared between multiple PEs and thus are multicast. Unlike the weights, IDs for input activations are configured at the beginning of each layer processing. Fig. 4-3 illustrates the ID configuration used to process a filter with

$C = 10, M = 21$. Unlike Eyeriss, IDs for input activations are not organized into row IDs and column IDs, because WS-1 requires input activations to be shared among all output channels of weights, which can be mapped to all rows of PE array. IDs for bias and output activations are similar except that not all PEs need data delivery. The unused controllers and FIFOs can be disabled.

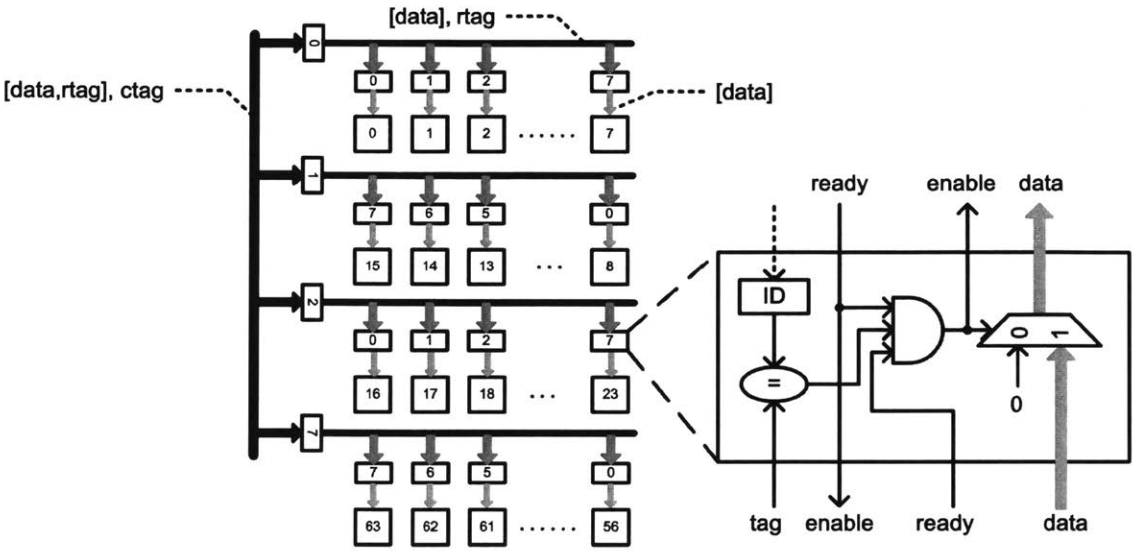

Figure 4-4: NoC for weights and the micro-architecture of the unicast/multicast controller [7].

4.5 Synthesis and Simulation Results

4.5.1 PE

Two PE structures, refer to as PE_a and PE_b, were synthesized at the same clock frequency (25MHz) using TSMC 40nm LP process to show the effectiveness of the proposed bit tuning algorithm. PE_a is introduced in Section 4.2. It takes in sign-magnitude weights and input activations, uses an unsigned multiplier and an adder-and-subtractor to do computation and outputs 2's complement output activations. PE_b has both inputs and outputs in 2's complement format and thus uses a signed

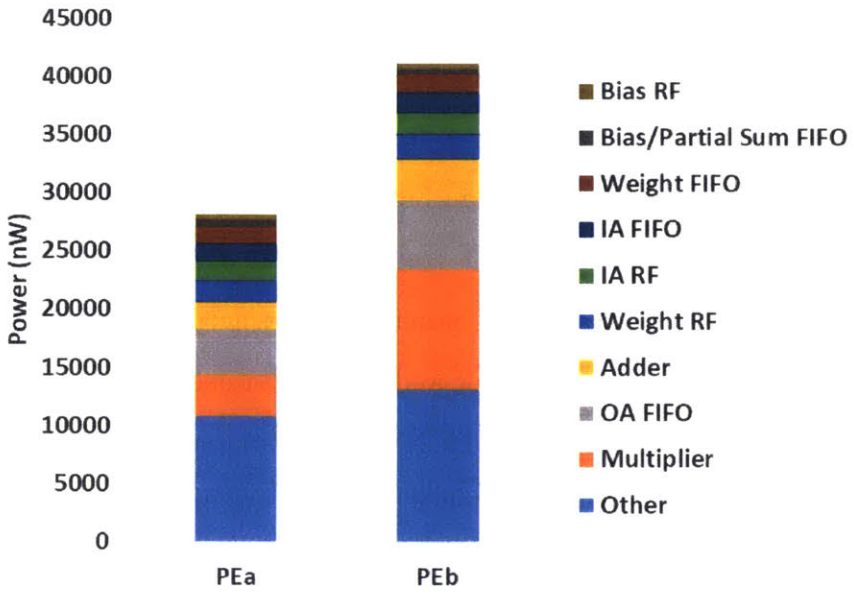

Figure 4-5: The power break down of PE_a and PE_b.

multiplier and an signed adder. Other blocks in those PEs are identical. During post-synthesis simulation, two PEs use the same set of input activations, but represents in different formats. PE_a takes in a set of weights from CNN-3-decomp tuned by the proposed bit tuning algorithm. PE_b uses the 2's complement unprocessed weights of CNN-3-decomp. Fig. 4-5 shows the power break down of two PEs. 35% overall power reduction is delivered by the application of bit tuning algorithm. For the multiplier, PE_a has around 70.0% power reduction compared to PE_b, which is attributed to toggle count reduction of inputs and a simple multiplier structure. PE_a also has lower power consumption of the output activation FIFO (OA FIFO), because most weights are scaled down during bit tuning. This makes the MSBs more stationary and thus less switching activity is observed.

4.5.2 NN Accelerator

The NN accelerator was synthesized at a clock frequency of 25MHz using TSMC 40nm LP process. The total logic area of the NN accelerator is around 166.61 KGates.

The total SRAM size is 133.12 kB. Post synthesis simulation is done by running the tuned CNN-1-decomp model in real-time with acoustic features extracted from Google speech command dataset [34] as inputs. The total power consumption is 1.2 mW. The power breakdown is shown in Fig. 4-6.

Table 4.1 compares our work with state-of-the-art accelerators. This work supports both FC and CONV layers. It provides higher accuracy for KWS tasks than [30, 18] which only supports FC layer and [3] that targets at binarized NN with a fixed shape. [7] is optimized for the CONV layers of large NNs and a batch size of 4 is used, which is not allowed for a KWS system due to the real-time processing requirement. The targeted run time of a KWS system is set by the frame rate of input features which is usually 10–20ms. The goal for a KWS system is to minimize the power consumption, given this run time requirement. In contrast, [18] targets at high throughput with a maximum clock frequency of 1.2 GHz.

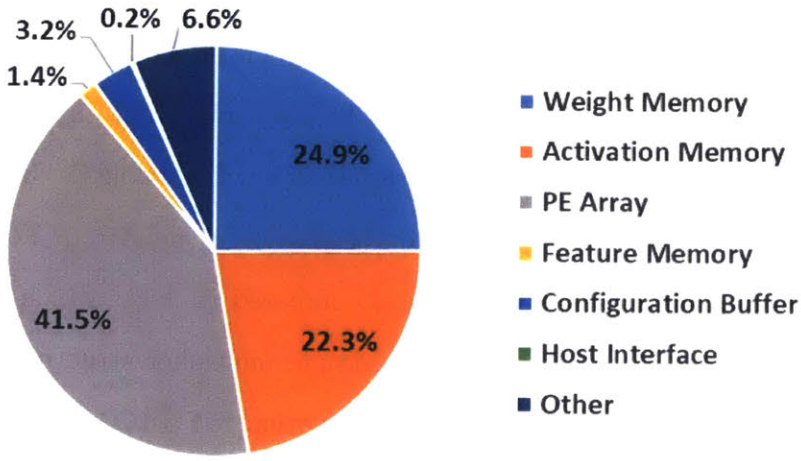

Figure 4-6: The power breakdown of the NN accelerator.

Table 4.1: NN accelerator performances.

	This Work	[30]	[3]	[35]	[5]
Application	KWS	KWS	Image classification	Image classification	Image classification
Dataset	Google [34]	RM [27]	CIFAR [19]	MNIST [10]	ImageNet [9]
Function	CONV, FC	FC	binarized CONV††	FC	CONV
MOps	2.6	0.9	2005.0	0.7	1332
Process (nm)	40	40	28	28	65
Voltage (V)	1.1	0.6	$V_{NEU} = 0.6$ $V_{COMP} = 0.8$ $V_{DD} = 0.8$ $V_{MEM} = 0.8$	0.715	1.0
Measured or Simulated	Sim.	Sim.	Meas.	Meas.	Meas.
Power (mW)	1.2	3.3	0.899	22.4	278
Batch Size	1	1	1	1	4
Run Time (msec)	16.5	10‡	4.2	0.06†	115.3
Memory (kB)	133.12	256	328	1158.5	192‡‡
Area (mm^2)	0.82	0.89	5.76	5.76	12.25

* It is the typical case corner (1.1V, 25C, TT).

† We estimated it assuming a full utilization of MACs.

‡ We estimated it by the latency requirement for real time processing.

‡‡ Off-chip DRAM access is needed.

†† It supports CONV layer with the filter shape of $2 \times 2 \times 256 \times 256$ and a squared input map at each channel.

Chapter 5

Conclusions and Future Work

5.1 Summary

The key contributions of this thesis are listed below.

- A bit tuning algorithm is proposed, which lowers the toggle rate of bits in the weight tensor, and thus reduce the power consumption of NoC and multipliers. In our experiments, up to 60.96% reduction in the toggle count of weights can be achieved with only 0.75% loss in accuracy. And the PE that processes tuned weights can achieve around 35% power reduction compared to the PE which handles the original 2's complement weights.

- Comprehensive analysis of various types of weight stationary, output stationary, row stationary and no-local-reuse dataflows are conducted. Weight stationary achieves the smallest number of memory accesses among all dataflows for the KWS NN architecture used in our accelerator.

- A CNN accelerator is implemented and optimized for processing CNN for KWS in real-time. It fills up the gap between the existing NN accelerator design shown in Fig. 1-3 and achieves comparable power consumption to the existing

NN accelerators target at similar model size.

We will continue optimizing the KWS system and realize a chip to validate our design.

5.2 Future Work

Some other work is not included in this project, but can be pursued in the future, including the following:

- Implement custom memory to utilize the low toggle rate of bits in weight tensors and lower read energy of weight SRAM
- Integrate KWS with speaker identification so that the system only detects the keyword uttered by a specified user
- Design the hardware to support the processing of recurrent neural network, convolutional recurrent neural network, etc. for KWS

Appendix A

NN Architectures Used In This Thesis

Table A.1: The original and decomposed architecture of CNN-2. The input layer of CNN-2 has $H = 10$ and $W = 49$.

Original									Decomposed								
Layer	R	S	C	M	U	V	Wgt (K)	Mult (K)	Layer	R	S	C	M	U	V	Wgt (K)	Mult (K)
conv1	4	10	1	64	1	1	2.56	716.8	conv1-1	1	10	1	6	1	1	1.38	393.6
									conv1-2	4	1	6	15	1	1		
									conv1-3	1	1	15	64	1	1		
conv2	4	10	64	48	2	1	122.88	7864.3	conv2-1	1	1	64	41	1	1	78.58	5596.2
									conv2-2	4	10	41	45	2	1		
									conv2-3	1	1	45	48	1	1		
lin	1	1	3072	16	na	na	49.15	49.15	lin-1	1	1	3072	12	na	na	37.2	37.2
									lin-2	1	1	12	12	na	na		
									lin-3	1	1	12	16	na	na		
fc1	1	1	16	128	na	na	2.05	2.05	fc1	1	1	16	128	na	na	2.05	2.05
fco	1	1	128	12	na	na	1.54	1.54	fco	1	1	128	12	na	na	1.54	1.54

R, S: Length of filter in time and frequency axis
C, M: Number of input and output channels
U, V: Stride in time and frequency

Table A.2: The original and decomposed architecture of CNN-3. The input layer of CNN-3 has $H = 40$ and $W = 98$.

Original									Decomposed								
Layer	R	S	C	M	U	V	Wgt (K)	Mult (K)	Layer	R	S	C	M	U	V	Wgt (K)	Mult (K)
conv1	8	98	1	186	1	1	145.8	4812.19	conv1-1	1	98	1	13	1	1	14.03	472.04
									conv1-2	8	1	13	44	1	1		
									conv1-3	1	1	44	186	1	1		
lin1	1	1	6138	128	na	na	785.66	785.66	lin1-1	1	1	6138	9	na	na	56.48	56.48
									lin1-2	1	1	9	9	na	na		
									lin1-3	1	1	9	128	na	na		
lin2	1	1	128	128	na	na	16.38	16.38	lin2	1	1	128	128	na	na	16.38	16.38
fco	1	1	128	12	na	na	1.54	1.54	fco	1	1	128	12	na	na	1.54	1.54

R, S: Length of filter in time and frequency axis
C, M: Number of input and output channels
U, V: Stride in time and frequency

Bibliography

[1] Sercan O. Arik, Markus Kliegl, Rewon Child, Joel Hestness, Andrew Gibiansky, Chris Fougner, Ryan Prenger, and Adam Coates. Convolutional recurrent neural networks for small-footprint keyword spotting. *arXiv:1703.05390 [cs]*, March 2017. arXiv: 1703.05390.

[2] S. Bang, J. Wang, Z. Li, C. Gao, Y. Kim, Q. Dong, Y. P. Chen, L. Fick, X. Sun, R. Dreslinski, T. Mudge, H. S. Kim, D. Blaauw, and D. Sylvester. 14.7 A 288 uW programmable deep-learning processor with 270kb on-chip weight storage using non-uniform memory hierarchy for mobile intelligence. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 250–251, February 2017.

[3] D. Bankman, L. Yang, B. Moons, M. Verhelst, and B. Murmann. An always-on 3.8 uJ/86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28nm CMOS. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 222–224, February 2018.

[4] 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)*, pages 4087–4091, May 2014.

[5] Y. H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, June 2016.

[6] Y. H. Chen, T. Krishna, J. Emer, and V. Sze. 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, January 2016.

[7] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, January 2017.

[8] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[10] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[11] C. Duan, A. Gotterba, M. E. Sinangil, and A. P. Chandrakasan. Reconfigurable, conditional pre-charge SRAM: Lowering read power by leveraging data statistics. In *2016 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, pages 177–180, November 2016.

[12] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv:1604.03168 [cs]*, April 2016. arXiv: 1604.03168.

[13] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254, June 2016.

[14] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *arXiv:1510.00149 [cs]*, October 2015. arXiv: 1510.00149.

[15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 1135–1143. Curran Associates, Inc., 2015.

[16] Nitin Kasturi. Power reducing algorithms in fir filters. Master's thesis, Massachusetts Institute of Technology, 1997.

[17] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv:1511.06530 [cs]*, November 2015. arXiv: 1511.06530.

[18] S. Kodali, P. Hansen, N. Mulholland, P. Whatmough, D. Brooks, and G. Y. Wei. Applications of Deep Neural Networks for Ultra Low Power IoT. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 589–592, November 2017.

[19] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[20] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. *ArXiv e-prints*, 1412:arXiv:1412.6553, December 2014.

[21] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong. LogNet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904, March 2017.

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

[23] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[24] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[25] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 27–40, New York, NY, USA, 2017. ACM.

[26] Michael Price, James Glass, and Anantha P Chandrakasan. 14.4 a scalable speech recognizer with deep-neural-network acoustic models and voice-activated power gating. In *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*, pages 244–245. IEEE, 2017.

[27] Patti Price, William M Fisher, Jared Bernstein, and David S Pallett. The darpa 1000-word resource management database for continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1988. ICASSP-88., 1988 International Conference on*, pages 651–654. IEEE, 1988.

[28] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[29] T Sainath and Carolina Parada. Convolutional neural networks for small-footprint keyword spotting. In *Proc. Interspeech*, 2015.

[30] M. Shah, J. Wang, D. Blaauw, D. Sylvester, H. S. Kim, and C. Chakrabarti. A fixed-point neural network for keyword detection on resource constrained hardware. In *2015 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, October 2015.

[31] M. Sun, A. Raju, G. Tucker, S. Panchapagesan, G. Fu, A. Mandal, S. Matsoukas, N. Strom, and S. Vitaladevuni. Max-pooling loss training of long short-term memory networks for small-footprint keyword spotting. In *2016 IEEE Spoken Language Technology Workshop (SLT)*, pages 474–480, December 2016.

[32] V. Sze, Y. H. Chen, T. J. Yang, and J. S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, December 2017.

[33] K. Ueyoshi, K. Ando, K. Hirose, S. Takamaeda-Yamazaki, J. Kadomoto, T. Miyata, M. Hamada, T. Kuroda, and M. Motomura. QUEST: A 7.49tops multi-purpose log-quantized DNN inference engine stacked on 96mb 3d SRAM using inductive-coupling technology in 40nm CMOS. In *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, pages 216–218, February 2018.

[34] P. Warden. Speech command: A public dataset for single-word speech recognition. 2017.

[35] P. N. Whatmough, S. K. Lee, H. Lee, S. Rama, D. Brooks, and G. Y. Wei. 14.3 A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with >0.1 timing error rate tolerance for IoT applications. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 242–243, February 2017.

[36] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *arXiv:1611.05128 [cs]*, November 2016. arXiv: 1611.05128.

[37] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. Hello Edge: Keyword spotting on microcontrollers. *arXiv:1711.07128 [cs, eess]*, November 2017. arXiv: 1711.07128.

[38] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.