ABSTRACT

LI, WEIFU. Design of Hardware Accelerators for Hierarchical Temporal Memory and Convolutional Neural Network (Under the direction of Dr. Paul D Franzon).

In recent years, the artificial neural network (ANN) achieved incredible successes in numerous application areas. Among these ANNs, hierarchical temporal memory (HTM) and convolutional neural network (CNN) has been considered as one of most representative networks in the cortical learning algorithm and machine learning algorithm respectively. In this paper, we propose a multi-level hierarchical ASIC implementation to support full-scale HTM and an ASIP implementation of CNN to leverage the zero input features and weights in convolutional layers. To improve the unbalanced workload in HTM, the propose design provides different mapping methods for the spatial and temporal pooling respectively. Also, we implement a distributed memory system to improve the efficiency of memory bandwidth. Finally, the hot-spot operations are optimized using a series of customized units. Regarding to scalability, we propose a ring-based network consisting of multiple processor cores to support a larger HTM network. To evaluate the performance of our proposed design, we map an HTM network that includes 2048 columns and 65536 cells on both the proposed design and NVIDIA Tesla K40c GPU using the KTH database as input. The latency and power of the proposed design is 6.04ms and 4.1W using GF 65nm technology. Compared to equivalent GPU implementation, the latency and power is improved 12.4x and 70.2x respectively. For CNN, the proposed ASIP can support a novel dataflow for the 1-D primitive that allows us to skip the computations with zero input features and weights by processing the data encoded in run-length compression (RLC) format directly. For computations beyond 1-D primitives, the proposed ASIP can perform multiple 1-D primitives in parallel, all of which share the input feature row but with various weight rows. By dividing an array of 96 PEs into multiple groups and performing the weight exchange within each group, the proposed design

reduces the total off-chip memory accesses by N, where N is the PE number per group. The proposed design can achieve a frame rate of 56.6f/s for the AlexNet at 1.65W, and a frame rate of 2.6f/s for the VGG16 at 1.58W. Compared to Eyeriss, the proposed design provides a 2.0x and 4.65x processing latency improvement for AlexNet and VGG16 respectively with the similar hardware resource and same clock rate.

Design of Hardware Accelerators for Hierarchical Temporal Memory and Convolutional Neural Network

by
Weifu Li

A dissertation or submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Electrical Engineering

Raleigh, North Carolina
2019

APPROVED BY:

_____	_____
Paul D. Franzon Committee Chair	James Tuck
_____	_____
David Ricketts	Xipeng Shen

DEDICATION

To my family.

BIOGRAPHY

Weifu Li was born in June 1987, Harbin, China. He received his bachelor's degree from Harbin Institute of Technology in July 2010, and his master's degree from Northeastern University in Dec 2012. In January 2013, he started his graduate research work with Dr. Paul D. Franzon in the area of digital hardware design in North Carolina State University. His major research interests include the custom accelerator design for machine learning algorithms.

ACKNOWLEDGMENTS

Most of all, I would like to thank my wife, Zijian Chen, for her unconditional support, love and understanding all these years. I will never be able to achieve this without her encouragement. I also want to thank my parents for their endless love during my entire life.

I want to express my sincere gratitude to my research advisor, Dr. Paul Franzon, for his guideline, understanding and support. I also want to thank him for introducing me to the machine learning related research topic and providing me the opportunity to join the research areas that are changing the our world. In addition, I want to thank my committee member, Dr. James Tuck, Dr. David Ricketts and Dr. Xipeng Shen for insightful advice and understanding during the research work.

I would like to thank my colleagues in NCSU, Steve Lipa, Lee Baker, Josh Schabel, Sumon Dey, Josh Stevens, Zachary Johnston and Tse-Han Pan for their help, advice and support on the physical design follows.

Finally, I would like to express my endless love to my kids Kyrie and Kyle, you are the best things ever happen to me. I love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Artificial neural networks (ANNs) are biologically inspired system designed to emulate the way in which the human brain processes information by detecting the patterns and relationships in data and learn through experience [1]. From the aspect of modeling neocortex, we can divide most of these ANN-based algorithms into two categories, the machine-learning algorithm (MLA) and the cortical learning algorithm (CLA). Most MLAs nowadays model each synapse as a continuous weight value and learn the knowledge through repeatedly updating these weight value during the training. On the other hand, the CLAs model each synapse as one or multiple binary bits and use the summation of all bits belonged to one neuron to indicate corresponding status. The CLAs learn knowledge by generating new synapses or updating the existing ones to emulate the way a human brain learns. In this work, we focus on one of the most representative algorithms in each category, the Convolutional Neural Network (CNN) in MLA and Hierarchical Temporal Memory (HTM) in CLA. Then, we implement an application-specific integrated circuit (ASIC) and an application-specific instruction set processor (ASIP) for HTM and CNN respectively to explore the potential performance improvement from the custom hardware designs.

1.1. Motivation

In the past few years, HTM has been demonstrated as successful in anomaly detection for identifying unusual changes in network servers [2], and to detecting anomalous data sequences in the network bus of vehicles [3]. In addition, several applications of HTM, such as geospatial tracking, nature language predictions and stock volume anomalies is under commercial test [4]. In 2008, [5] proposed an HTM based network to learn and recognize 11 words from TIDIGIS corpus.

In the area of pattern recognition, the HTM network in [6] can achieve an accuracy of 95.65% on traffic sign recognition and HTM network in [7] outperforms several other neural network on automatic license plate recognition. The integration of HTM on top of a Support Vector Machine (SVM) achieved a 96% classification rate in an object recognition task [8].

Unlike most neural networks in MLAs, HTM is not a matrix-based network and the data dataflow within each neural column and cell is un-deterministic. Therefore, though there are large amount of parallelisms existing at each step of the entire algorithm, the steps in each column and cell can be various depending on the local input data, column state and cell state, which results in an unbalanced workload among the processing engines (PEs). Meanwhile, each column and cell in HTM requires intensive memory accesses, which are narrow, discrete and asynchronous due to the sparse distributed representation (SDR) of active columns and the workload difference among columns/cells. Consequently, mapping HTM network onto a conventional hardware platform like GPU and CPU fails to achieve a satisfactory performance. Fortunately, we efficiently handle these issues by providing proper design of ASIC-based hardware accelerator. Furthermore, the custom hardware design allows us to further improve the performance of HTM by optimizing these critical steps.

Regarding to MLA, CNNs have achieved an unprecedented success across a board range of applications including the face detection [9], image classification [10], speech recognition [11], advertisement recommendation [12], and even the game playing [13]. Particularly, CNN achieve a near-human performance in video [14] and audio recognition [15]. However, given the increasing complexity of potential applications, the size of state-of-arts CNNs is also dramatically increased. For instance, AlexNet [16], a state-of-art CNN proposed in 2012, has about 2.3M parameters in convolutional layers. In 2014, the total parameter amount of VGG16 [17] has to be increased to

15.3M to beat the accuracy of AlexNet with same testbench. For most of these state-of-arts CNNs, the convolution operation in each layer dominates the total execution time [18] and requires up to hundreds of megabytes (MB) of parameters for a single pass in inference. Therefore, optimizing the latency and power of convolutional layers can significantly improve those of entire CNN-based system.

Though there are many research focusing on optimizing the performance of convolutional layers by allocating more arithmetic units, increasing the size of on-chip SRAMs and developing better dataflow, the performance improvement from these methods highly depends on the available hardware resource and the total number of required computations stays constant. This work tries to improve the performance of convolutional layers from a different angle and is motivated by the observation that the immediate convolutional layers can have up to 79% and 89% of zero input features in AlexNet and VGG16 due to the rectified linear unit layers (ReLUs). Meanwhile, using the pruning technology proposed in [19] can reduce the average percentage of none-zero weight down to 36.7% and 32.6% in these CNNs. As a result, skipping operations with zero input features or zero weights could significantly reduce the total number of computations in the convolutional layers, which allows us to improve the latency and power of CNNs with same hardware resource and memory bandwidth.

1.2. Contribution

In this work, we implement a custom ASIC for HTM to explore the performance benefits from the proposed optimization methods. The details of our contributions for HTM is:

- Design and implement a hierarchical architecture and control flow to improve the unbalanced workload and synchronization issues among the columns and cells while maintaining the computation parallel.

- Design and implement a distributed memory organization, which assigns a dedicated memory bank to each level of the proposed hierarchy, to improve the utilization efficiency of memory bandwidth. The simulation results have demonstrated that the proposed design can outperform the GPU implementation with 8.04x smaller memory bandwidth.
- Design and implement a series of custom hardware modules to improvement the performance of critical operations in HTM, such as the global inhibition, selecting learning/active cells and counting distal segment activity.
- Design and implement an ASIC-based accelerator to perform an HTM network including 2048 columns and 65536 cells. Compared to the equivalent implementation on NVIDIA Tesla 40Kc, the proposed design achieves a 12.45x speedup, 2.15x silicon area deduction and 137x power efficiency.

For CNN, we implement a custom ASIP that can support various network sizes and the concrete contributions on this work include,

- Design and implement a novel dataflow for 1-D convolution primitive that allows us to directly process the data encoded in run-length compression (RLC) format. As a result, the proposed design can skip the operations with zero input features and weights to improve the processing latency.
- Design and implement a dataflow beyond 1-D convolution primitive that makes all the PEs in proposed design share the same input features in parallel and minimize the data movement between on-chip and off-chi memory.
- Design and implement an ASIP-based hardware accelerator to support the proposed dataflow and outperform state-of-arts designs with similar hardware resource while benchmarking with the AlexNet and VGG16.

- Investigate the influence of input feature and weight sparsity, weight filter size and operation mode on the processing latency in proposed design to achieve the most optimized performance for various CNNs.

1.3. Organization

We organize the rest of this dissertation as follow. Chapter 2 introduces the fundamentals of HTM and CNN including the network architecture and operations in both algorithms. In chapter 3, we survey the related state-of-arts design of CNN and HTM on various hardware platforms and briefly compare them with the proposed design.

Chapter 4 presents the custom ASIC design of HTM. This chapter starts from the overview of the proposed processor core. Then, we introduce the network mapping method, optimization for critical operations and design scalability applied to overcome the performance bottleneck in the existing implementations. In the result section, we describe the functionality validation for both software and hardware implementations, then discuss the result of their performance comparison from the aspects of processing latency, silicon area and power consumption.

Chapter 5 presents the proposed zero-skipping technology and corresponding ASIP design of CNN. We start from the technology introduction applied within each 1-D convolution primitive that allows us to skip the computations with zero input features and weights. Then, we discuss the dataflow among multiple 1-D primitives. In subsequence section, we elaborate the design of ASIP-based processor core. In the result section, we investigate the influence of sparsity and weight filter size on processing latency and compare the performance of proposed design against the start-of-art custom hardware implementation.

In Chapter 6, we summarize the design methodology and achieved improvement for both algorithms and discuss the potential future work.

1.4. Abbreviations

ANN Artificial Neural Network

MLA Machine Learning Algorithm

CLA Cortical Learning Algorithm

CNN Convolutional Neural Network

HTM Hierarchical Temporal Memory

DCNN Deep Convolutional Neural Network

DNN Deep Neural Network

ASIC Application-Specific Integrated Circuit

ASIP Application-Specific Instruction Set Processor

SDR Sparse Distributed Representation

CPU Central Processing Unit

GPU Graphics Processing Unit

FPGA Field-Programmable Gate Array

PE Processing Element

DSP Digital Signal Processing

NoC Network-on-Chip

RLC Run Length Compression

MB Mega Bytes

KB Kilo Bytes

eDRAM Enhanced Dynamic Random-Access Memory

SRAM Static Random-Access Memory

DRAM Dynamic Random-Access Memory

1/2-D One/Two Dimension

Psum Partial Summation

MAC Multiply and Accumulation Computation

FIFO First-in, First-out Buffer

CHAPTER 2

BACKGROUND

In this chapter, we describe the fundamental properties of both HTM and CNN including the network structure, the operations in neurons and the data storage format. Considering the nature of online learning algorithm, we introduce both learning and inference mode of HTM in the section 2.1. For CNN, we focus on the inference mode since most of applications nowadays can directly utilize the pre-trained weight filters.

2.1. Fundamental of HTM

HTM is an on-line machine learning algorithm inspired by the structural and algorithmic properties of neocortex in the human brains [20]. By combining and extending approaches in the spatial and temporal clustering algorithms and the sparse distributed representation (SDR), HTM has the capacity to learn and recognize streaming data, and then makes a prediction for the next possible input data based on the learned knowledge. Meanwhile, we can consider HTM as an unsupervised learning algorithm since it can learn the unlabeled data. Normally, a typical HTM network consists of multiple layers organized as a hierarchy, each layer of which shares the same operations but has different network sizes. In general, HTM network use the binary data received from the practical world as input data at bottom level, and then recombines the output data from lower levels at higher layer to memorize a complicated pattern as shown in Figure 2.1. In addition, HTM has the potential to combine multiple networks together, each of which works on the data from different resources as shown in Figure 2.2. Jeff Hawkins firstly proposed the HTM algorithm in his book [21] in 2004. In 2011, Jeff Hawkins and his colleagues published the 2nd generation of HTM in [20]. In this section, we only introduce the latest version of HTM.

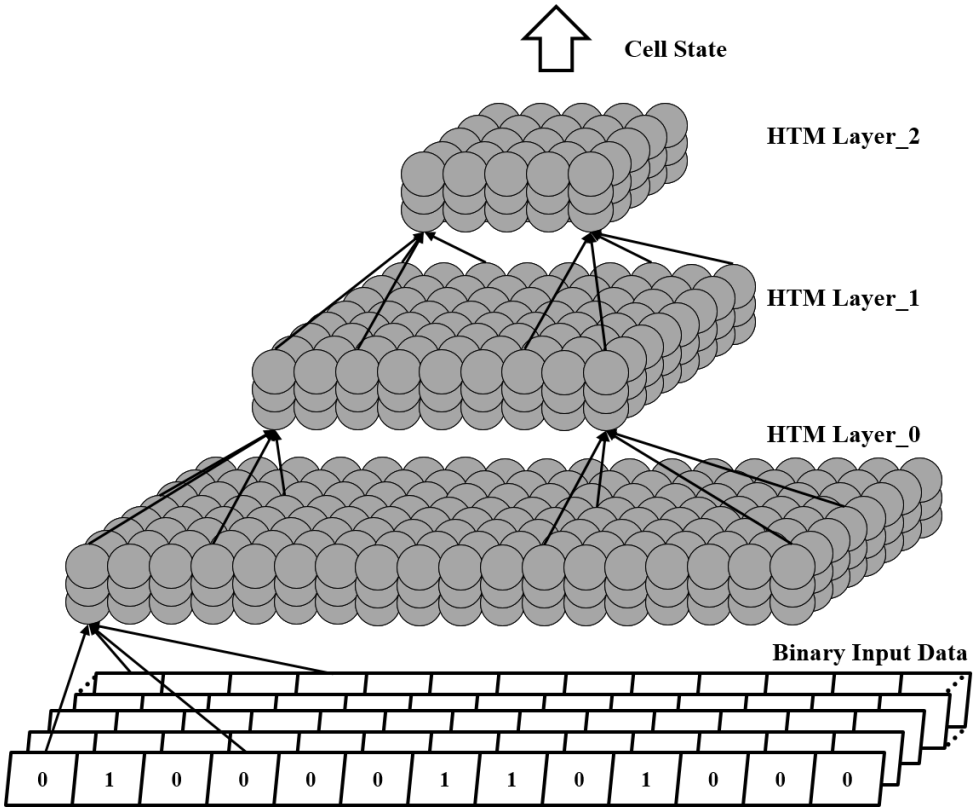

Figure 2.1 A Three-layer HTM Network

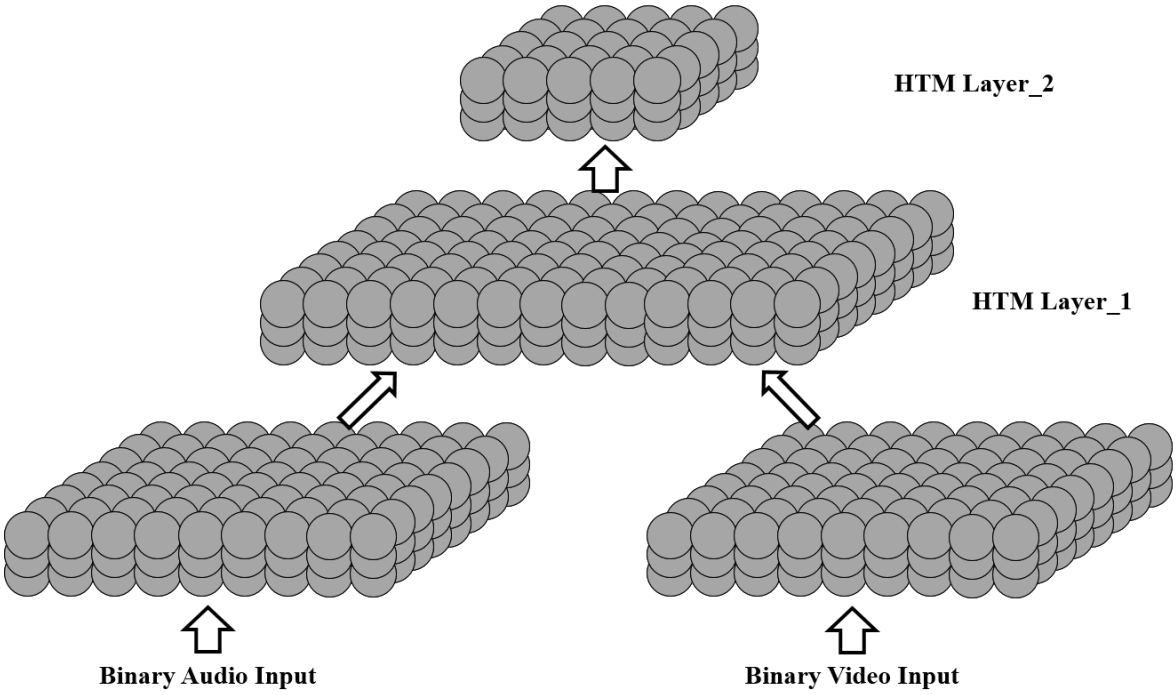

Figure 2.2 HTM Network Processing Data from Multiple Resource

In general, we can model each layer of HTM network as a 2-D matrix of columns, each of which contains multiple identical cells. For each input data, HTM network extracts both spatial and temporal information in the format of column and cell state respectively. During processing, we can perform HTM algorithm in two operation modes: learning and inference. The inference in HTM is a process of matching current input pattern with the ones learned. Each input data uses a list of active columns to represent its spatial pattern, and then turns corresponding cells into active or predict state to represent the temporal pattern.

In learning mode, HTM performs all operations in inference mode. In addition, operations used to update parameters related to column activity, such as column boost value and permanence of proximal synapses, are performed in spatial pooling to create a unique SDR for each input. In temporal pooling, each learning cell in active columns either generates a series of connections or updates the strength of existing connections to memorize the temporal pattern of input data.

2.1.1. Spatial Pooling

Motived by how our human brain always represents the outside world information in only a small percentage of active neurons, the major purpose of spatial pooling is to generate the SDR for each input pattern in the format of active columns. By using the SDR, we integrate several desirable properties into HTM network.

1) In HTM, we can represent the complicated input patterns, in which most input bits are active, by only a small percentage of columns without losing any information. Since many operations in HTM are only performed in active columns, the memory space, processing time, and power consumption for single input pattern can be improved over the dense network.

2) Generally, input data received from the practical world is always accompanied with various noise. Since flipping just a few bits in the input data might not affect the SDR generated by the spatial pooling at all, HTM is more robust to deal with input data containing spatial noise or missing parts.

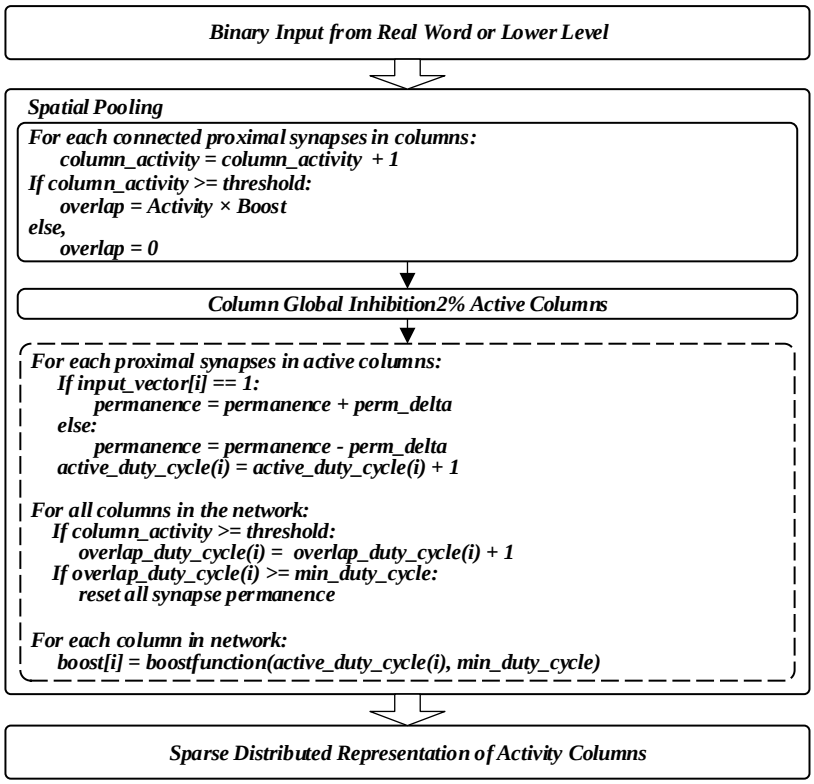

Figure 2.3 Operation Details of Spatial Pooling

In spatial pooling, each proximal synapse in all the columns randomly connect to 1 bit of corresponding binary input vector. In cast that a proximal synapse is connected to an input bit of "1" and has a permanence value above the threshold, we consider it as a "connected synapse" and we use the summation of connected synapses in a column as column activity as shown in Figire.2.3. The operations in dot block only perform under learning mode. During global inhibition, we sort all columns within the same region in descending order of overlap value. Then it turns the columns with top n% overlap value (usually 2%) into active state and use the

combination of active columns as output of spatial pooling. In learning mode, it increases the boost value of columns that are less active to increase its possibility of being active for next input data. Meanwhile, the synapse permanence value increases if it connects to a “1”. Otherwise, the value decreases.

2.1.2. Temporal Pooling

In temporal pooling, HTM combines the output from the spatial pooling, stored as an SDR, and the previous cell state at t-1 to generate the temporal representation of each input, which is the combination of active cells at t. The details of temporal pooling are described in Figure 2.4 The definition of several of the terms used in describing HTM are as follow,

1) Distal synapse: the connection between any two cells from different columns refers to the distal synapse and a synapse is considered as “connected” if its permanence value is above threshold.
2) Distal segment: a group of distal synapses within the same cell connected to cells from same or similar combinations of active columns is distal segments.
3) Segment activity: the total number of connected synapses to active/learning cells at t refers as the segment activity of active/learning cells at t.
4) Best matching cell: a cell has largest number of synapses to active cells at t -1 or fewest number of existing segments in a column is the best matching cell.

In inference mode, cells in each active column are turned into active state if they were at predict state at t-1. In case that there is not any predict cells at t-1, all cells in that active column are turned into active state to indicate an unexpected input data. To make a prediction for the next input, we turn a cell into predict state if its maximum segment activity of active cells at t is above threshold.

In learning mode, for each active column, we turn a cell into learning state if it was at predict state at t-1 and has a segment activity of learning cells at t-1 above threshold. Otherwise, we select the "best matching cell" as learning cell. In each learning cell, we generate a new segment if the largest segment activity of active cells at t-1 is below threshold. Otherwise, we increase the permanence value of synapse connected to active cell at t-1 and decrease that of all the other synapses in the segment with largest activity. In addition, if a cell was in predict state for the past two input data but never turns into learning state, the permanence value of all connected synapses marked in prediction phase is decreased to reduce "incorrect" predictions.

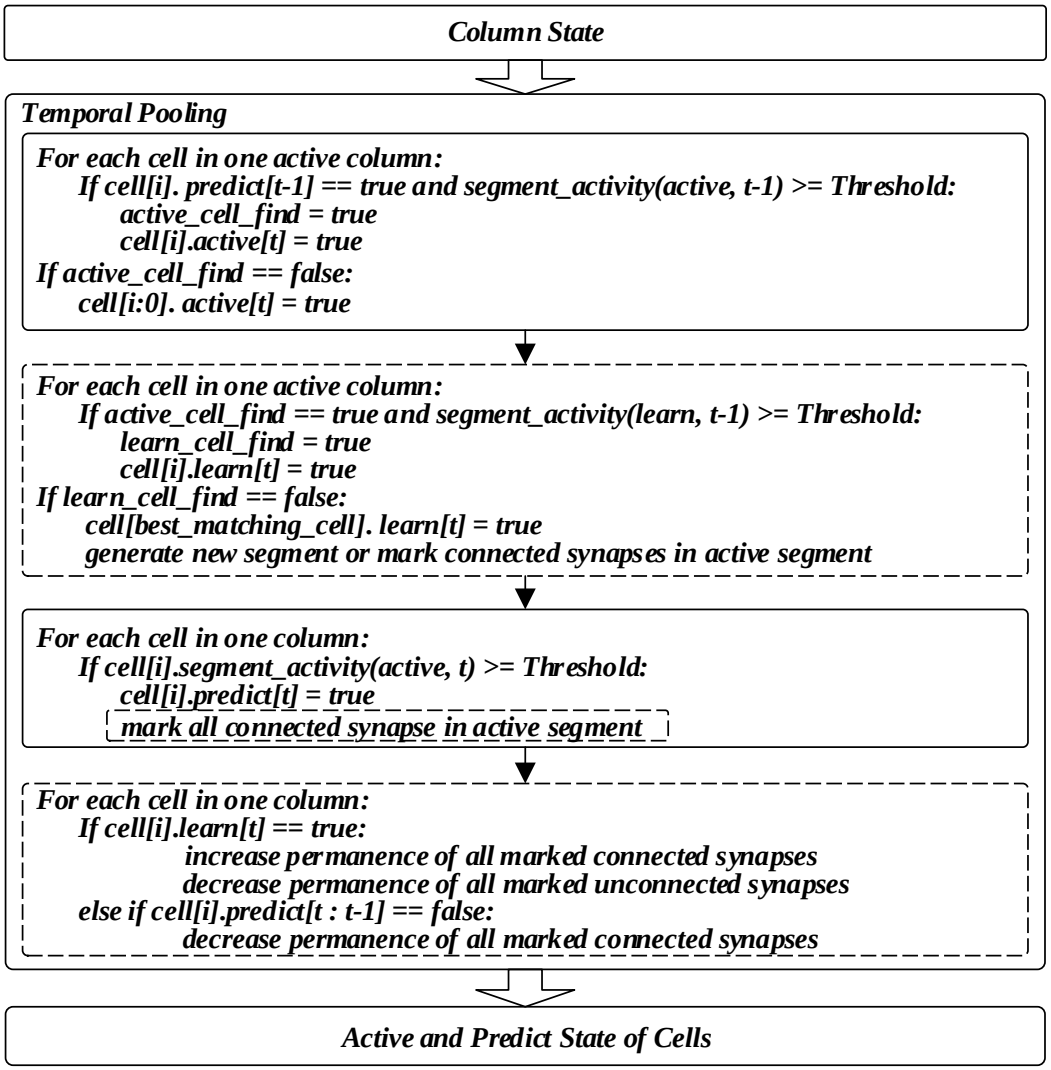

Figure 2.4 Operation Details of Temporal Pooling

Though spatial pooling can generate different SDR for each input, temporal pooling allows HTM network to memorize the input data as an ordered sequence rather than a sequence of non-related data. In addition, temporal information is necessary for distinguishing identical input data at different locations in the same sequence. Taking the input sequences “a, b, d, b, e” as an example, both letter “b” in spatial pooling should have the same combination of active columns due to the identical input vectors as shown in Figure 2.5 (a). During the temporal pooling, since HTM use both column state and cell state at t-1 as input data, the combination of active cells at t can be various following different precedents. Since the precedents of two “b” are different in this sequence, the combinations of active cells at t is also different as shown in Figure 2.5 (b), where the blue indicates the active cells for the first “b” and the red ones are for the second one.

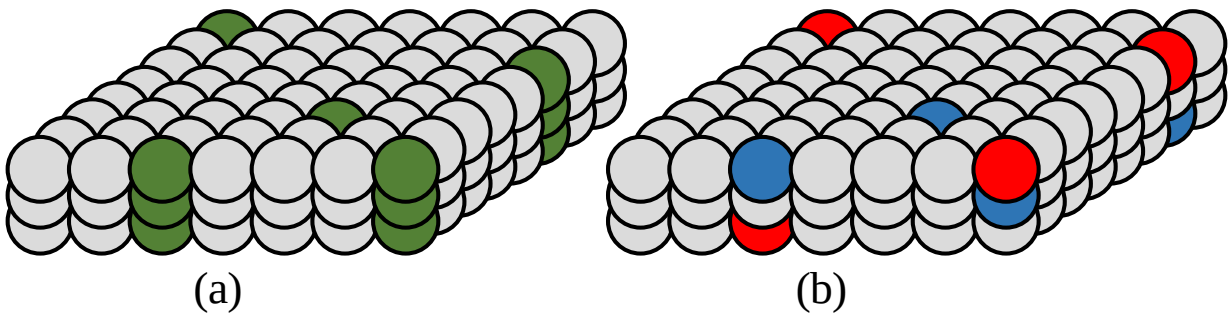

Figure 2.5 (a) Combination of active columns for letter "b". (b) Combination of active cells for letter “b”

2.2. Fundamental of CNN

CNN is a class of supervised matrix-based machine learning algorithm and usually consists of multiple computation layers including convolutional layer, max-pooling layer, non-linear layer and the fully connected layer as shown in Figure 2.6. The state-of-art CNNs nowadays can have a total layer number up to 192. By sweeping multiple weight filters across the entire input data, the convolutional layers in CNN can extract different features from each input data and store them in

the format of feature maps, referred as output features. Then, we can downsize the output features of by performing max pooling, which only leaves the maximum value within a pooling window. The non-linear layers in CNNs are used to increase the network non-linearity, so that adding more layers can provide a better approximation power. The most successful non-linear function for CNN is the Rectified Non-Linear unit (ReLU), which force all the negative values in the output features to zero. Compared the other non-linear functions, the ReLU can speed up the training progress and increase the overall sparsity. The same group of convolutional layers, max-pooling layers and the ReLU can be repeated multiple times to extract the high-level feature for each input data. Finally, we flat the output features and use them as the input of the fully connected layers for classification. Though there can be several fully connected layers, the neuron number of the final fully connected layer should equal to the category number of target dataset. Meanwhile, the output value from each neuron indicates the possibility of corresponding category.

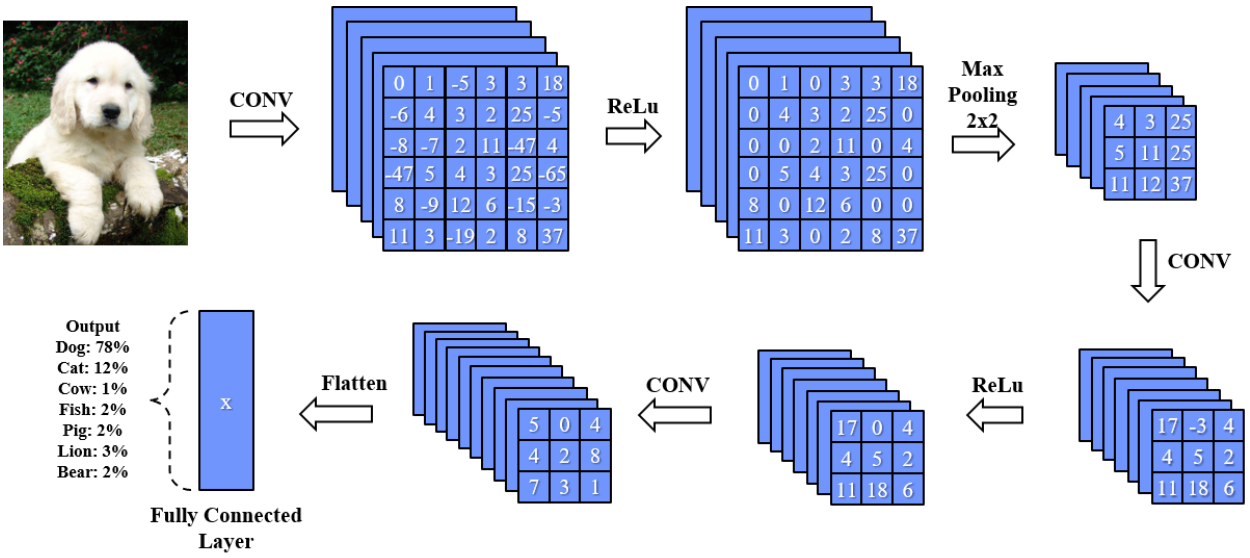

Figure 2.6 Example of Deep Convolutional Neural Network

The computation in CNNs mainly come from the high-dimensional convolution operations between the input features and the corresponding weight filters, each of which can produce one element of the output feature map. More specifically, it includes the element-wise multiplications between one filter and corresponding receptive field in the input feature, and the summation of all these products as shown in Figure 2.7.

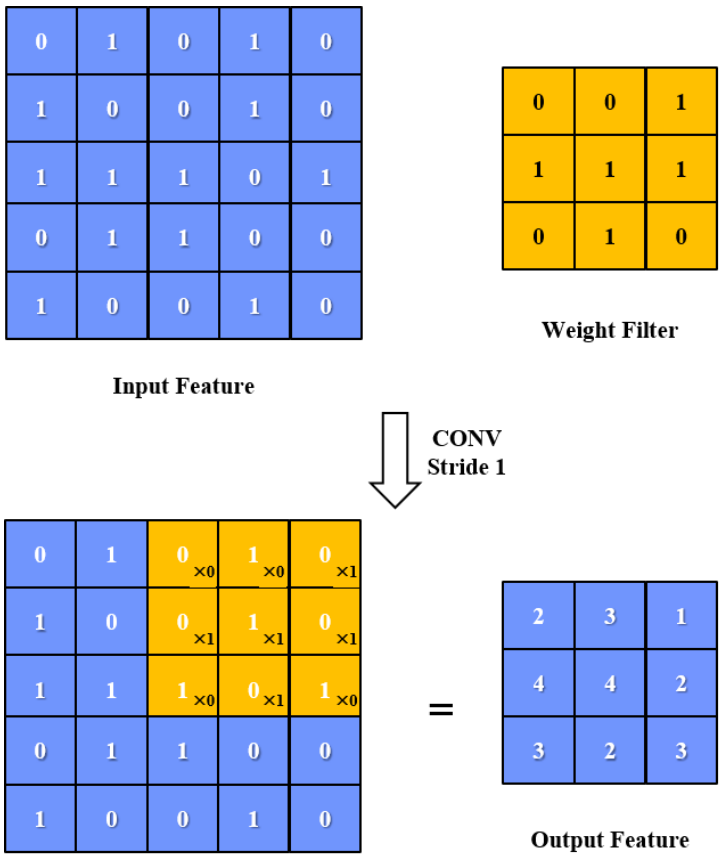

Figure 2.7 Example of Convolution Operation

Usually, each convolutional layer in CNN consists of multiple weight filters, each of which includes various channels. During inference, we sweep each channel of a weight filter across the corresponding channel of target input feature with a given interval, referred as stride, to generate one partial summation (psum). Then, we perform an element-wise accumulation among the psums

resulted from all channels of a filter to generate one complete output feature. We repeat the same iteration with all input features and weight filters of a given convolutional layer. Since each output feature from the previous layer is used as one input channel of the next layer, the filter channel number of a convolutional layer is usually related to the total filter number of its previous layer. A processing example between one input feature and one weight filter is shown in Figure 2.8, where the “C” stands for the channel number of target filter.

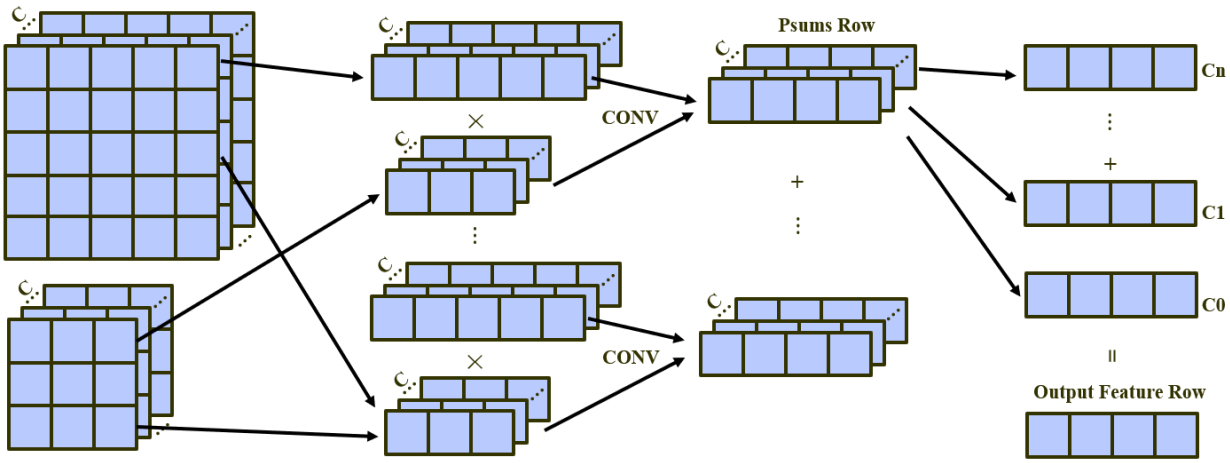

Figure 2.8 Processing Example Between One Input Feature and One weight Filter

CHAPTER 3

STATE-OF-THE-ART

During the past few years, there has been numerous works about HTM and CNN published due to their successes in various application areas. In this chapter, we present the state-of-the-art hardware implementations of both networks including the key features, performance and potential improvement achieved in our design.

3.1. HTM Implementation

To improve the performance of HTM on the traditional hardware platforms, HTM has been implemented using multi-core CPU and GPU. However, due to the nature of parallel computation, intensive memory access and dynamic workload distribution, mapping HTM onto these platforms fails to achieve a satisfactory performance. In [22], two latency hotpot operations in the sequential CPU implementation were identified and re-implemented using OpenMP library in parallel. Then, the author mapped the parallelized code onto the multiple-core processor and evaluated the latency vs number of cores occupied. Though adding more cores can improve the processing latency, the parallel efficiency decreases from 100% down to about 48% while the number of cores increases from 1 to 6. In addition, they predicted the efficiency can deteriorate if using more cores. For the GPU implementation, we proposed an optimized CUDA-based implementation of HTM on GPU as our performance baseline. Though the GPU implementation allows us to explore the parallelism existing at each operation of HTM, it could not improve the unbalanced workload distribution and memory bandwidth utilization efficiency, which limits the performance improvement from GPU implementation for large network. This is also demonstrated by the simulation results discussed in the result section of chapter 4.

Regarding the customized hardware design, Lennard and his team implemented a scalable hardware accelerator optimizes the memory organization, power and area by leveraging the large-scale solid-state flash memory in 2016[23]. The proposed design is validated by the MNIST dataset and achieved a classification accuracy of 91.98%. The power and area of proposed spatial pooler is 30.538mm^2 and 64.394mW respectively under TSMC180nm Technology. In the same year, Abdullah's group proposed a reconfigurable and scalable architecture of HTM in [24]. They ported the design for a 100-column matrix onto a Xilinx Virtex-IV FPGA fabric, and then verified it with both MNIST and EU number plate dataset, which achieved a 91% and 90% accuracy respectively. In addition, the proposed hardware offers a speedup of 4817x over the software implementation. Both works, however, only implemented spatial pooling in HTM and cannot explore the temporal patterns of input data, which is one of most important features of HTM compared to other neural algorithms.

In [25], we proposed a fully flattened hardware design for the complete HTM algorithm, in which we mapped each column and its corresponding cells onto one processing element (PE), and connected all PEs through the mesh network. To verify the proposed design, we implemented a 400-column PE matrix, each of which can include up to 2 neural cells, to perform a series of test cases using MNIST dataset. Compared to the software implementation in [22], the proposed design can achieve a speedup of 329.6 with a power and area of 1.29mW and 17511μm^2 for single PE. In [26], Abdullah's group again presented a FPGA-based design for both spatial pooler and temporal memory. They developed a synthetic synapse concept to address the dynamic interconnections of synapses during learning. For a 100-column matrix with 3 cells per column, the proposed design provides a processing latency of 5.75us for one image in MNIST dataset with a power of 1.39mW for a single cell. Though both works implemented successfully implemented entire algorithm, the

network size in their testbench is relatively small compared to the typical one of 2048 columns [27] and did not discuss about the potential challenges and solution to handle the scalability, which is extremely important for real-world applications.

In [28] and [29], they proposed two analog implementations of HTM using the memristor circuits. The [28] implemented the 2-D column array using parallel memristor crossbar arrays and validated the design on both face recognition and speech recognition dataset. In [29], the authors implemented a modified version of spatial and temporal pooling using the memristor logic circuits and demonstrated the propose design with multiple face recognition dataset. Though the analog implementations allow us to remove the analog-to-digital converter between the input sensor and processing engineer, these implementations are not configurable and are power and area hungry. For instance, each image pixel requires a memory cell in [29], which has an area of $23.85um^2$ and a power of 0.44mW using the TSMC 0.18um Technology. Therefore, the total power and area of memory cell only can be huge even processing a small input image. To the best of our knowledge, the work presented in this paper is the first hardware implementation that can support the both spatial and temporal pooling with a large network size.

3.2. CNN Implementation

In recent years, there has been numerous papers about custom hardware accelerators aimed at skipping the computations with zero weights or input features published. For skipping zero input features, the author of [18] encoded each row of the input features into the run length compression (RLC) format, which can efficiently reduce the total DRAM accesses for sparsity input features. It also proposed a dataflow named row stationary to improve the energy efficiency. The proposed design can achieve a frame rate of 34.7fps with a power of 278mW for AlexNet and a frame rate of 0.7fps with a power of 236mW for VGG16. However, since the weights and input features are

stored in the on-chip memory in the dense format, it cannot fully eliminate the energy resulted from moving zero value among on-chip memory banks. In addition, though gating the multiplier when the zero input features are being processed can save computation energy, it cannot really improve execution time due to idle cycles in the pipeline arithmetic units. Meanwhile, Eyeriss in [18] did not skip the operations with zero weights. In [30], the author proposed the DaDianNao[31] based custom hardware accelerator to skip the computations with zero input features. The proposed design decoupled the groups of execution lanes in original architecture, which allows each execution lane to access memory independently to asynchronization among execution lanes. They encoded the input features stored in eDRAM into the zero-free neuron array format (ZFNAf), which provides an offset for each none-zero value, to skip the computations with zero input features. The proposed claimed a latency improvement of 1.38x and 1.39x compared to the original design. However, the ZFNAf format cannot efficiently skip the zero-related computations while the whole data bricks are zero and is not capable to handle the zero weights either. In addition, the proposed data format can result in a significant increase in memory space and memory access energy regardless the density of input feature.

Regarding skipping zero weights, the proposed design in [32] only stored and none-zero weights of each neuron, so that the inefficiency computations can be skipped. It also allowed each PE to access the memory asynchronously to improve the utilization efficiency of PEs. They also implemented an index module using step indexing method to calculate the index of corresponding input features for each none-zero weight. The proposed design provided a 7.23x speedup compared against the DadianNao. However, this work cannot skip the computations with input features and is less efficiency while processing the convolutional layers compared to that of fully connected layer. In addition, it required a 2.11x increase in area and a 1.98x increase in power. The author of

[33] proposed an energy efficiency neural processing unit that supports both convolutional layer and fully connected layer. They integrated 1024 MAC units, which are connected using a bufferfly structure and divided into two cores. By applying the feature-map selection circuitry, the proposed design can skip the computations with zero weights. The NPU achieved a 6.9TOPs and 3.5TOPs for the 5x5 and 3x3 weight filters while there are 75% zero values. The power of proposed design is 39mW and 1553mW under 0.5V and 0.8V respectively. Though this work can skip zero-weight, it cannot skip those computations with zero input features. In addition, they did not benchmark the proposed design with any state-of-the-arts CNNs to evaluate the actual performance.

In [34], the author compressed both input features and weights using a variant of the sparse matrix representation to reduce memory accesses. They also proposed a novel dataflow named as PlanarTiled-InputStationary-CartesianProduct (PT-IS-CP), in which each PE reads multiple none-zero weights and input features within a given region, and then delivers them into a multiplier array along with their coordinates. Then, it passes all these products into a series of accumulation buffer to calculate the partial sums. Compared to a dense CNN accelerator, the proposed design can provide an average latency improvement of 2.37x and 3.52x for the AlexNet and VGGNet respectively. However, the proposed dataflow assumes a filter stride value equal to 1. Otherwise, there are many unnecessary none-zero multiplications, since each input feature is only covered by a part of the filter when the stride is larger than 1. As a result, the efficiency of proposed accelerator will be decreased while supporting the convolutional layers with a stride value larger than one, such as AlexNet. In addition, it requires a large SRAM to store the input/output features on-chip and a high DRAM bandwidth to maintain the performance compared to other designs. In [35], they generated a 1-bit flag for each input feature and weight to indicate if its value is zero, and then by performing an “AND” operation between the flags of stored input features and weights in the fetch

controller, the proposed design can identify the none-zero operand pairs, so that the inefficient computations can be skipped. They also proposed zero-aware kernel allocation method to improve the unbalanced workload resulted from filter sparsity. Overall, the proposed design achieved a 4.4x/5.6x speedup using the pruned AlexNet/VGG16 as input data compared to that of their own implementation of Eyeriss. However, since the fetch controller still need to scan through the result of all "AND" operations to find the none-zero pairs, it is difficult to keep the pipelined arithmetic units busy, which can result in a decrease in the computation efficiency. In addition, this proposed design dose not employ any data compression technology for input features or weights to improve the memory access energy.

It is also worth to mention that none of these state-of-the-arts provide an absolute latency per frame as performance metrics except [18], which makes it difficult for us to perform apple-to-apple comparison.

In addition to paper focusing on sparse convolution layer, there are also some state-of-the-arts focus on improving performance of hardware accelerator without any zero-aware technology, such as [36] and [37]. In [36], the author proposed a DCNN processor consisting of multiple DSP clusters, a co-processor and around 5.6MB on-chip SRAM. The proposed accelerator is fabricated using 28nm FD-SOI technology and can provide a frame rate of 58frame/s for AlexNet. In [37], the author presented an energy-scalable processor, which supported the dynamic voltage accuracy frequency scaling technology to improve the energy efficiency. After quantizing the input features and weights down to 4-bit, the proposed design can achieve a frame rate of 47frame/s and a power of 44mw under 200MHz clock.

CHAPTER 4

DESIGN OF HTM ACCELERATOR

In this chapter, we present the design details of proposed HTM hardware accelerator, which includes the architecture overview, network mapping method, operation implementation and the design scalability. We perform a series of test cases to validate the functionality our hardware and software implementation and robustness against noise data. In the result section, we compare the performance of our proposed accelerator with that of equivalent software implementation.

4.1. Overview of Hardware Accelerator

In the proposed design, a processor core is organized as a 2-level hierarchical architecture as shown in Figure 4.1. The first level of the hierarchy consists of 1 central processor and 8 PEs, each of which connects to the central processor through two 32-bit data ports and two 2-bit control ports. The second level of hierarchy exists in each PE and consists of 1 PE controller and 8 identical execution lanes.

The central processor consists of four parts, a control module, a series of customized units, an interface module and an on-chip memory bank as shown in Figure 4.1 (a). The control module is responsible for system configuration, synchronizing the PEs and collecting data from all PEs. Coordinated by the control module, the customized design units can perform operations requiring data from all PEs, such as the global inhibition, without any inter-PE communication. The interface module is designed to implement the inter-core data communication described in the later section. The memory bank in central processor stores packages received from other processor cores or local PEs.

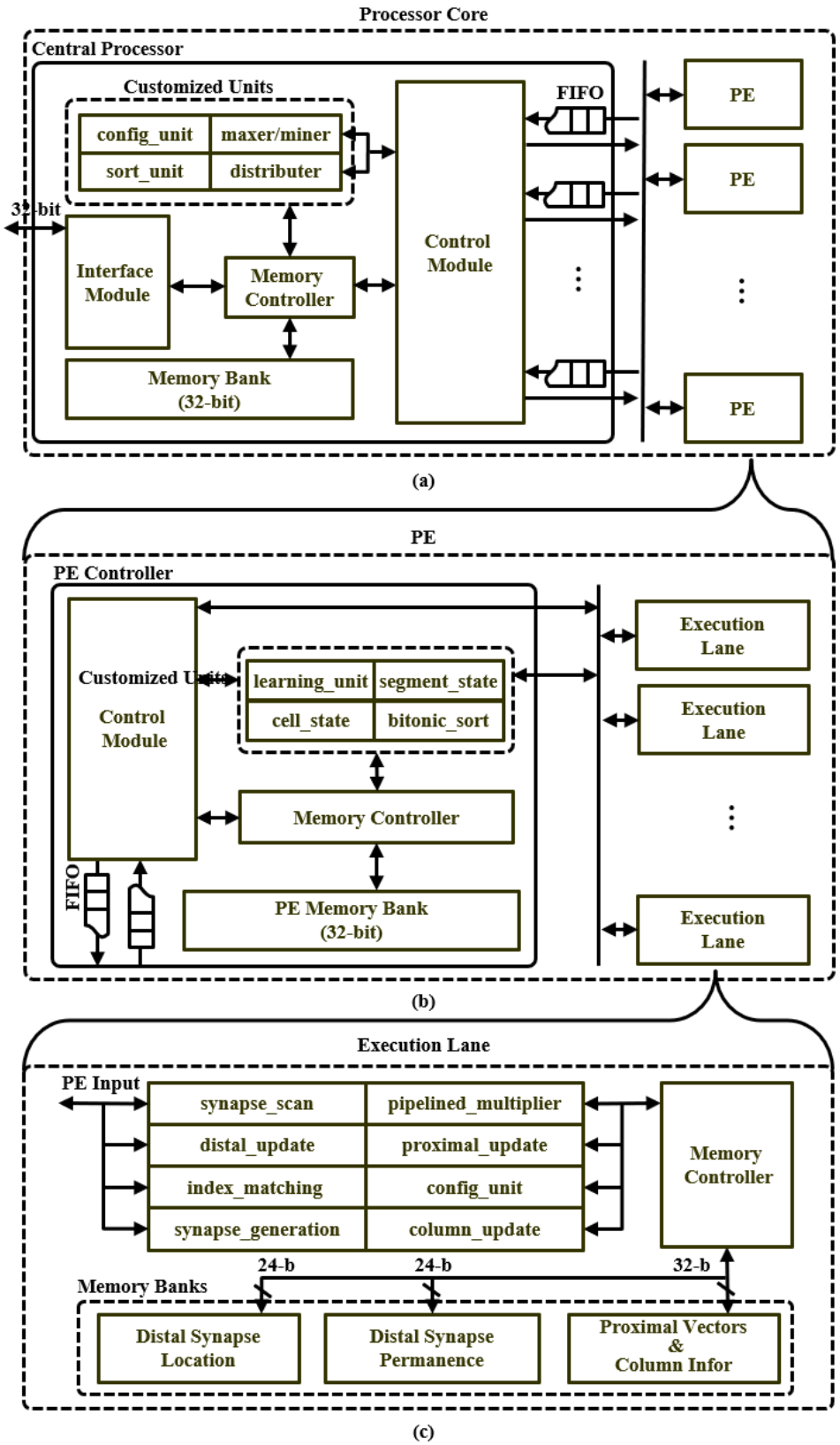

Figure 4.1 (a) Schematic of Processor Core. (b) Schematic of PE. (c) Schematic of Execution Lane.

Similar to the central processor, there is a control module in each PE controller designed to implement the inter-lane data communication and synchronize all execution lanes as shown in Figure 4.1 (b). Each PE controller includes a series of customized units used to perform the operations requiring data from all execution lanes, such as counting distal segment activity. The memory bank in PE controller is used to store information for local cells, such as existing segment number per cell, cell states, and the list of active/learning cells. We summarize the information stored in PE memory bank in Table 2.1. The lists of active/learning cells at t and t-1 in each processor core can have up to 40 elements, since each processor core can support up to 2048 columns and only 2% of the total column should become active for each input data.

Table 2.1 Information in PE Controller Memory

Information	Bit Number	Vector Details
Local Cell Status	15	[14 : 7] index of stored cell [6 : 3] count of existing segments [2] if cell is in predict state at t-1 [1] if cell is in predict state at t [0] if cell is in learning state at t
Element in Cell List	32	[31 : 16] column location [15 : 8] learning cell location [7 : 0] active cell location

The primitive module in our proposed design, referred as the execution lane, implements most of the operations in HTM and execution lanes in all the PEs are always performing in parallel.

Each execution lane includes multiple individual custom units as shown in Figure 4.1 (c), each of which implements one specific operation in the spatial and temporal pooling. Each execution lane has a dedicated memory bank used to store the information of proximal and distal synapses mapped on local PE. In case that all memory space for the distal synapses is occupied, the newly generated synapses will overwrite the existing ones. The details of stored information for one column and one distal synapse is summarized in Table 2.2 and 2.3 respectively.

Table 2.2 Information Stored For Each Column

Information	Bit Number Per Vector	Vector Number Per Column
Synapse Mapping Vector	32	2
Synapse Status Vector	32	2
Synapse Permanence	32	40
Column Value Vector	48	1

Each word for the columns is 32-bit.

Table 2.3 Information Stored For Each Distal Synapse

Information	Bit Number Per Vector	Vector Details
Synapse Index Vector	24	[23 : 8] location of connected column [7 : 0] location of connected cell
Synapse Permanence	16	[15 : 0] permanence value of distal synapse
Synapse Status Vector	8	[7 : 4] times of being predict [3 : 0] synapse status vector

In this work, we combined the synapse permanence and synapse status vector together as a 24-bit word.

Each bit in synapse mapping vector indicates the connection status between one synapse and 1-bit data in input vector, while the corresponding bit in synapse status vector indicates if that synapse is active based on the current permanence value. The column value vector records column boost value and times of being active for each column. HTM network only requires and updates the column information during spatial pooling and update them after the processing of each input in learning mode.

The data format for each distal synapse includes two kinds of vector, location of connected cells (X and Y index) and corresponding connection strength referred as permanence value. The lower 4-bit of each synapse status vector records information such as, if it is a valid synapse, and if the synapse is just generated during the learning of current input.

Instead of putting one large and wide memory in the central processor, we distribute the memory across each level of the proposed hierarchical design. Compared to a centralized memory system, the distributed one can improve the memory access overhead and power efficiency with a given bandwidth due to the following reasons,

1) Unlike the matrix-based algorithms, such as convolutional neural network and long short-term memory, the memory access in HTM is not always continuous, since some of the operations are only performed in the active columns/cells, which are usually discretely stored in memory. In addition, the data size of a single column/cell is usually small (16-32 bits). As a result, the total number of memory access using one 256-bit memory will be similar as that using eight 32-bit memories while reading this kind of data. However, due to the higher power and sequential access of that 256-bit memory, it is more efficient from the aspect of power to use 8 distributed memory banks in HTM implementations as proposed.

2) For operations on the continuously stored data, each access to the centralized memory can get data for multiple PEs or execution lanes. However, the processing latency of each PE or execution lane can vary depending on the states of target columns or cells. For instance, the execution lane for active column requires more cycles to update proximal synapses than others. As a result, using centralized memory can resulted in duplicated accesses to same location from different lanes. In the proposed distributed memory, since each execution lane has a dedicated memory, we can fully eliminate this situation.

4.2. Network Mapping & Implementation

In the proposed design, the network mapping onto hardware is different in spatial pooling and temporal pooling. In this section, we will discuss the details of hardware implementation and network mapping

4.2.1. Spatial Pooling

During spatial pooling, the proposed design divides the entire network into multiple sub-regions of columns, each of which is mapped onto one PE. In Figure 4.2, we present a network example of spatial pooling, in which a 16x12-column matrix is mapped onto 8 PEs and each PE is responsible for the computation of a sub-region with 8 x 2 columns. In the practical world, it is impossible to process all these columns within the same sub-region simultaneously due to the limitation of hardware resource and memory bandwidth for input data. Therefore, we further divide the columns mapped on one PE into multiple batches, each of which consists of 8 columns equaling to the number of execution lanes in each PE. During one iteration, we map each column from the same batch onto one execution lane within the PE and each PE only processes one batch at one

time. The PEs in the proposed design repeat the same iteration until all columns in network are processed.

In the proposed design, we implement spatial pooling as two individual operations, (1) computation of the overlap values (2) sorting of the overlap values from all PEs to find the active columns in inference mode. To compute the overlap value, each execution lane in all PEs reads both mapping and status vector of the proximal synapses, which are randomly generated and downloaded into each memory bank during the initial phase. Meanwhile, execution lanes start to accumulate the bitwise “and” operation result of these two vectors and the input vector to calculate the activity of one column in the “synapse_scan” unit. After processing all vectors connected to one column, each execution lane multiplies the local column activity by a corresponding boost value to obtain the overlap value of current column in case that its column activity is larger than the threshold.

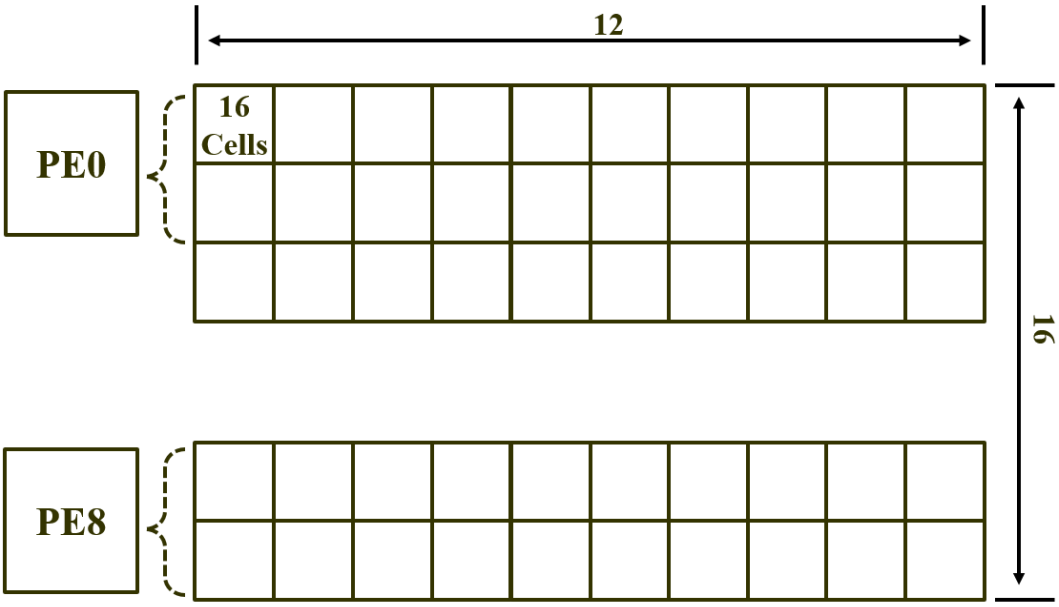

Figure 4.2 Mapping Example of Spatial Pooling.

For each iteration, there are a total of 64 overlap values from all the PEs. Along with the value from previous batch, we need to sort 104 overlap value to find the top 40 active columns. In this work, we divide the entire sorting process in two phases as shown in Figure 4.3. In phase one, the overlap value in each PE is re-ordered by an 8 element bitonic sorter in the PE controller. Then, the sorted arrays from all PEs are sent to the central processor at the same time, in which there is a 4-level pipeline comparator used to find the top 40 columns from all PEs by always moving the columns with larger overlap values to the next level. Compared to the implementation in [25], the proposed design only requires 39 comparators, which is 52x fewer. In addition, the max number of active columns in the proposed design is configurable up to the network size in this work, but is limited by the number of comparators and buffers in [26].

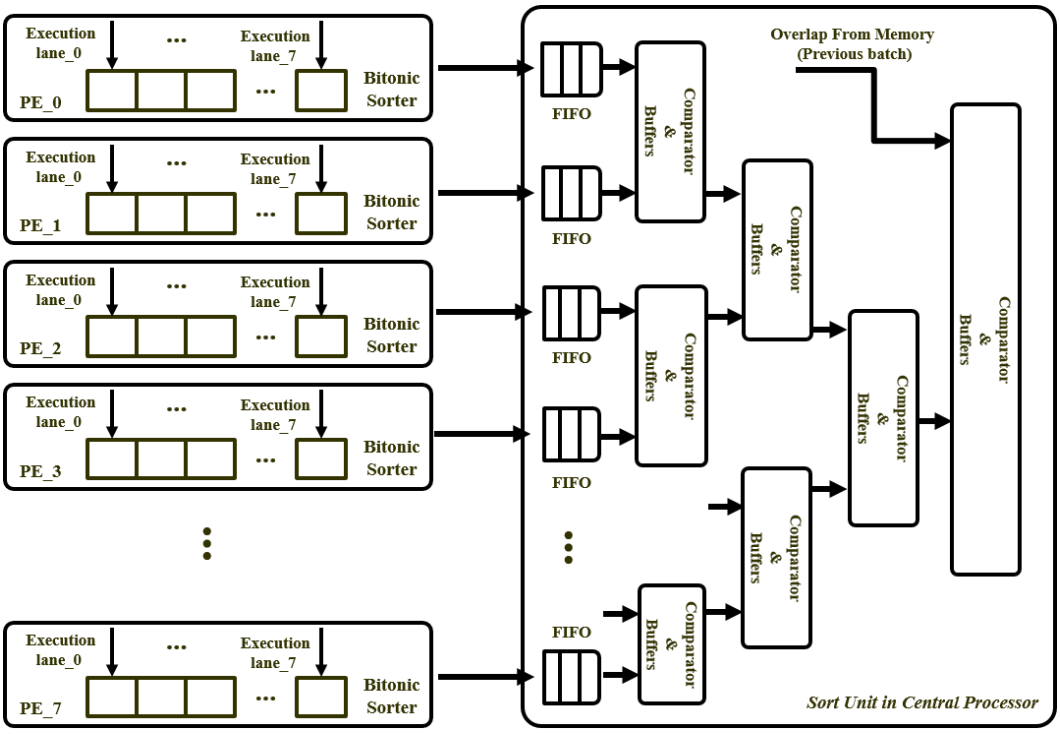

Figure 4.3 Implementation of Global Inhibition In Proposed Design

The hierarchical architecture in the proposed design allows us to implement the two-phase sorting as a pipeline, in which we sort the overlap of the n and n+1 column batches in the PEs and the central processor simultaneously. Since the sorting performed in PEs is faster than that in the central processor, we combine the sorting in PEs with the operations used to calculate the overlap value for one batch. As a result, we re-organize the operations of spatial pooling in inference mode as a two-state pipeline, which allows us to bury the latency of the faster stages (overlap calculation + PE sorting) into the slower one (sorting in central processor) to improve the overall performance of spatial pooling. In the learning mode, we still perform the operations used to find these active columns in pipeline, and perform the learning operations after all the columns processed.

In the learning mode, we perform three more operations in the “column_update” unit and the “proximal_update” unit in all execution lanes and the “column_state” unit in PE controllers to update the local boost value, times of being active column, and the permanence value of proximal synapses. Then, each bit in the synapse status vector is re-evaluated based on the latest permanence value, “1” indicates proximal synapse with value above threshold.

4.2.2. Temporal Pooling

Parallelism in temporal pooling exists at both column level and cell level. However, due to the mechanism of finding active columns in spatial pooling, using the same mapping method in Figure 4.2 can result in a dynamic distribution of active columns among all the PEs. Since some operations in temporal pooling are only performed in active columns, the PEs with more active columns have to suffer a much longer processing latency, while the rest PEs stay idle, which can eventually result in a significant increase of the total processing latency. In addition, this situation can get worse while the number of distal segments increase.

To optimize the unbalanced workload, the proposed design divides cells in the network into 8 layers, each of which is mapped onto one PE. In Figure 4.4, we present a mapping example of temporal pooling, where we reuse the same network dimension in spatial pooling. Since there are 16 cells in each column, each PE in the proposed design is responsible for the computation of 384 cells now. Consequently, the number of active columns mapped onto each PE is always same regardless of their locations, which can result in a more balanced workload at PE level. In temporal pooling, most operations target on the distal synapses within certain cells, which can be performed on all synapses from the same segment in parallel. To exploit this parallelism, the proposed design evenly distributes all the distal synapses from one segment among all execution lanes within each PE during temporal pooling. Compared to mapping cells onto the execution lanes, the proposed synapse-level mapping can significantly improve the unbalanced workload resulted from the various existing segment number and cell state at execution lane level.

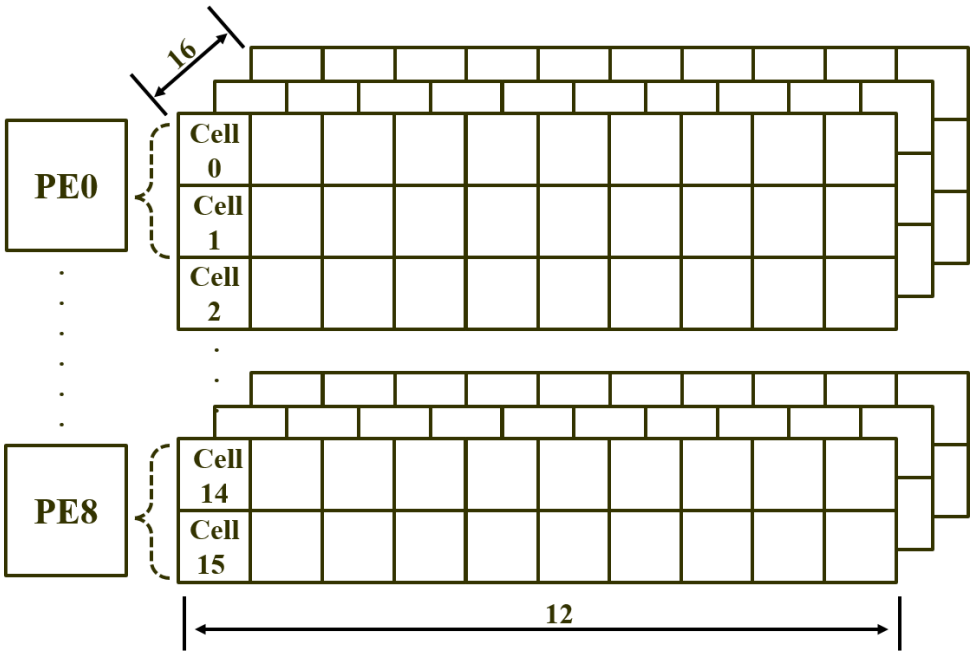

Figure 4.4 Mapping Example of Temporal Pooling.

The temporal pooling majorly consists of two operations in inference mode, deciding active cells and deciding predict cells. The state of each cell in both operations depends on the activity of distal segments and its previous state at t-1. Generally, we obtain the segment activity by checking the state of cells connected by the target distal synapses and counting those at learning/active state. However, due to the sparse distribution of connected cells, obtaining segment activity of multiple cells in parallel requires intensive cross-PE memory accesses, which can result in significant inter-PE communication overhead.

In the proposed design, we count the target segment activity by counting the number of matching pairs between the cells in current active list and those stored in target segment. More specifically, the processing starts from each execution lane reading out one synapse index vector belonging to the target segment. Then the PE controller begins to broadcast all elements in current active list to all local execution lanes one by one, while execution lanes are matching the index of local synapse with the received ones. Since the cell index in both active list and distal synapses is stored in the descending order of column index, we only need to broadcast the active list one time for each segment regardless of the synapse number per segment. Once the PE controller broadcasts all elements in that active list, we can count the activity of target segment by summing the number of matching pairs in all execution lanes up in the "segment_state" unit located in the PE controller. An example of index matching for one segment is shown in Figure 4.5. Different from the previous implementation, execution lanes in the proposed one only accesses the local memory banks, which can dramatically improve the latency for obtaining one segment activity.

In additional to the network mapping method, the number of existing segments in each PE can influence the workload as well. In the proposed design, there is a unit named "distributor" in the central processor used to balance the segment number among all the PEs. For any given active

column, the “distributor” collects the maximum segment activity and existing segment number of learning cell candidates from all PEs and then assigns the learning cell into the PE with the best matching cell. In case that the cells from multiple PEs have the same segment activity and existing segments number, the “distributor” unit finds the PE with least total segment through customized comparator name “miner” and assigns the learning cell into that PE to balance the total number of segments among all PEs.

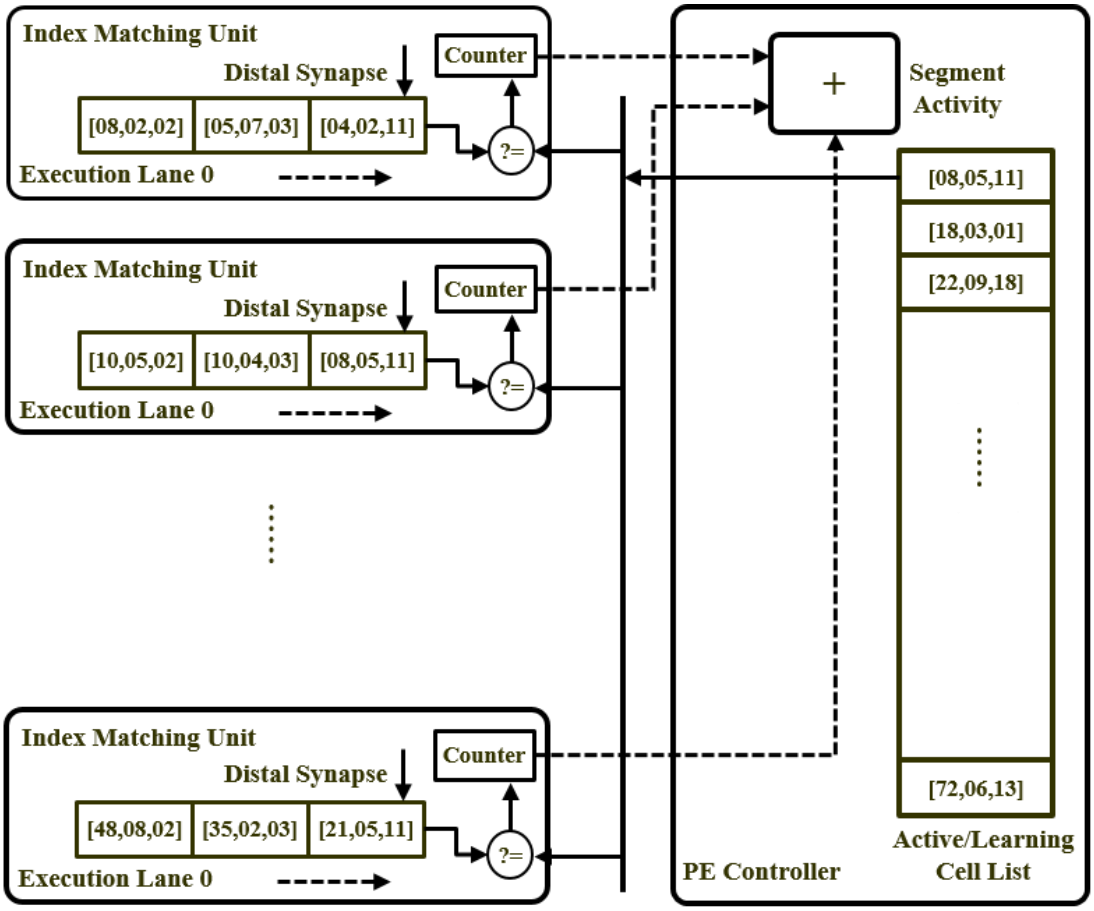

Figure 4.5 Example of Index Matching for One Segment.

In learning mode, there is one more operation to perform to update the permanence value of distal synapses. In the proposed design, we implement that operation in both PE controllers and

execution lanes. The “cell_state” unit in PE controller loops through the state vectors of all local cells. In case of finding a “dirty” cell, the PE controller enables “distal_update” unit in all execution lanes, each of which is responsible for the update of one synapse. Execution lanes repeat the same operation until all distal synapses in that “dirty” cell are updated. The concrete update value for each synapse depends on both synapse status vectors in local memory bank and the cell/segment status received from PE controller.

4.2.3. Inter-Core Communication

The total memory size of one single processor core limits the maximum size of the network mapped onto it. However, simply increasing the size of memory bank to process a larger network can result in a dramatic decrease in performance, since we are mapping more columns/cells onto each PE now. To efficiently process a larger network or improve the performance of a given network, the proposed design supports a ring-based inter-core network includes multiple identical processor cores as shown in Figure 4.6. Each core within the network connects to two neighbor cores through two 32-bits single direction data ports and two 2-bits control ports. The bandwidth of inter-core communication in the proposed design is 3.2Gbs assuming a 100MHz system clock frequency.

The proposed design provides a three-step topology for the inter-core communication, synchronization, sending local data, and forwarding received data. When a processor core comes to the communication stage, it sends a “Done” message to the core next to it, in where the “Done” message is forwarded to the rest of cores within the same network. Meanwhile, there is a counter in each interface module used to records the number of received “Done” message. Once the counter reaches the number that is equal to the total number of processor core, it indicates all cores synchronize to the communication stage and are ready to send data. In the second stage, each

processor starts to send local data to their neighbor cores and store the received data into memory bank in central processor until all local data are send. Then, each processor core starts to read the previously stored data and send them to their neighbor cores until it moves all the received data as in stage two. By the end of our inter-core communication, each core will have a local copy of data from all the other cores stored in central processor.

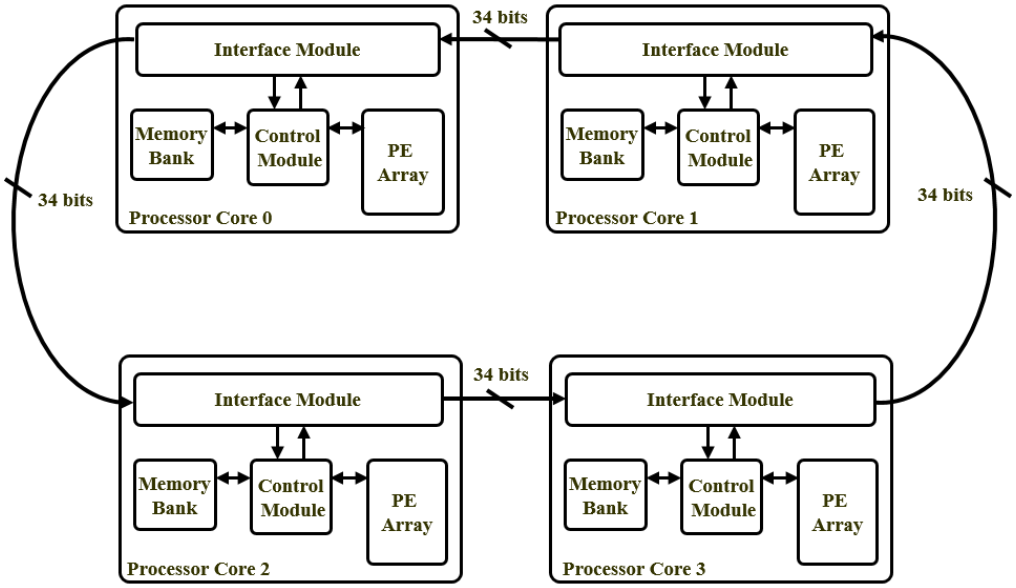

Figure 4.6 Example of Proposed Ring Network.

Compared to a network-on-chip (NOC) providing higher bandwidth and lower latency, like butterfly and mesh, the ring network in the proposed design can provide the following desirable features:

1) For each input, the number of packets requiring transfer is small in HTM. For instance, each processor core in current testbench only has 40 local packages, each of which is 32-bit wide. As a result, the total inter-core communication latency for a ring network with 8 processor cores is about 3.8 us, which is less than 1% of the average processing time.

2) From the aspect of hardware resource, the proposed design only requires 2 data ports per processor core, which is only 1/4 and 1/8 of the ports number in mesh and butterfly network respectively. As a result, the proposed design requires less control logic and smaller buffer size to implement the inter-core communication.

4.3. Functionality Validation and Performance Evaluation

HTM has been successfully applied in numerous areas [3]-[9]. Therefore, the functionally validation, in this work, is focused on the validation of our software and hardware implementation. We compare the performance of the proposed design with that of the software implementation on GPU with the chosen benchmark for evaluation.

4.3.1. Network Configuration

To verify the functionally of proposed hardware and evaluate its performance against a GPU implementation, we map a 64×32 columns matrix onto one processor core, each of which includes 32 cells. In each column, there are totally 40 proximal synapses covered a 64-bits input region. To indicate a valid connection between one input bit and synapse, we set the corresponding bit in synapse mapping vector to “1”. For each cell, the proposed design can generate up to 12 segments, each of which includes up to 16 distal synapses. The configuration of target network in this work is summarized in Table 2.4.

During the initialization phase, we set all bits in proximal synapse status vector to “1” and use an initial permanence value of 120, while the threshold is 85. For each input in spatial pooling, we turn up to 40 columns into active state, which is 2% of the total column number. Meanwhile, we set the threshold of column activity to 2, so that the number of columns with a none-zero overlap value is larger than 40 in most cases. In temporal pooling, each new generated distal synapse has a permanence value of 120 and the threshold is 85.

Table 2.4 Configuration of Target Network

Network Features	Value	Total Proximal/Distal Synapse
Column Matrix Dimension	32×64	79040
Proximal Synapse Per Column	40	
Neural Cell Per Column	32	
Segment Per Cell	12	1.25M
Distal Synapse Per Segment	16	

4.3.2. Verification Dataset

In order to validate our implementation, we randomly select images from the KTH action dataset as input data. The KTH database includes 6 human action, such as the walking, boxing and joggling [18]. To apply these images to the HTM network, we binarize and convert each of them into a 160×120 binary matrix, each bit of which is used as the input of one proximal synapse in the spatial pooling. Several image examples from the KTH database before conversion are shown in Figure 4.7.

Figure 4.7 Examples of Images from KTH Database

4.3.3. Functionality Validation

For any given input in functional validation, the recognition is considered as successful if the combination of active cells in inference mode is the same as that of learning cells in learning mode. Prediction is considered as correct if the combination of active cells at t is the same as that of predict cells at t-1. Each digit in Table 2.5-2.8 stands for one image randomly picked from the KTH benchmark.

The first order network refers to a network that includes only one cell within each column. Since the same input image should always have the same or similar combination of active columns in spatial pooling, the combination of active/learning cells for 2 identical input images should be the same as well in first-order network regardless their location in input sequence.

In our first test, we randomly pick 9 images from the KTH database and then present them to the processor core one by one in learning mode. Then we present the same input images again in inference mode. The simulation result summarized in Table 2.5 shows that the proposed design could recognize all the images successfully.

Table 2.5 Test One of First Order Network

Network Data	Time Step								
	t1	t2	t3	t4	t5	t6	t7	t8	t9
HTM Input Images	1	2	3	4	5	6	7	8	9
List of Active Cells	✓	✓	✓	✓	✓	✓	✓	✓	✓
List of Predict Cells	☑	☑	☑	☑	☑	☑	☑	☑	

✓: indicates the list of active cells for input data at t in inference mode is same as the list of learning cells for input data at t in learning mode (successful inference).

☑: indicates the list of predict cells at t-1 is the same as list of active cells at t (correct prediction).

In the second test of first order network, we repeat the fourth image twice at different location of the input sequence in both learning and inference mode. During inference, the first order network can recognize every input image. Since there is one cell in each column, the learnings cell of both the 5th image and the 7th image connect to the same learning cells of the 4th image in learning mode. As a result, it is not surprising to see the active cells of both the 5th and 7th image are turned into predict state as shown in Table 2.6 every time we present the 4th image to the processor core in inference mode. In summary, the first-order network cannot distinguish the identical input data within one sequence while making predictions.

Table 2.6 Test Two of First Order Network

Network Data	Time Step								
	t1	t2	t3	t4	t5	t6	t7	t8	t9
HTM Input Images	1	2	3	4	5	4	7	8	9
List of Active Cells	✓	✓	✓	✓	✓	✓	✓	✓	✓
List of Predict Cells	☑	☑	☑	5,7	☑	5,7	☑	☑	

✓: indicates the list of active cells for input data at t in inference mode is same as the list of learning cells for input data at t in learning mode (successful inference).

☑: indicates the list of predict cells at t-1 is the same as list of active cells at t (correct prediction)

The high order network refers to a network, which has more than one cell per column. In the high order network, the same input pattern, which has the same SDR, can generate different combinations of active cells and predict cells following various precedents as we discussed in Chapter 2.1.2.

In this test, we map a high order network that has 32 cells per column onto the proposed design and use the second test case in Table 2.6 as input data. Different from previous test, the

high order network can make a correct prediction for each 4th image as shown in Table 2.7 due to the following reasons,

1) For the two 4th images presented at different time steps, the combination of active columns is same due to the identical input data, which are different from those of the rest input images.
2) In learning mode, the cell #31 in all active columns of the 4th image at t4 are turned into learning state, and then each learning cell generates a series distal synapse connected to the learning cell of 3rd image. However, since the learning cells for the 5th image are different from those for the 3rd image, the segment activity of cell #31 in active columns of the 4th image at t6 is below threshold. As a result, we use the cell #30 which has fewer segments as the learning cell of the 4th image at t6 and generate new segments connected to the learning cells of the 5th image. Consequently, when we present the 3rd image again in inference mode, only cell #31 in active columns of the 4th image can turn into predict state. In the same way, only cell #30 in active columns of the 4th image should turn into predict state when we present the 5th image in inference mode.
3) In learning mode, since the learning cells of the 4th image at t4 and t6 are different as mentioned above, the learning cells of the 5th image and the 7th image are connected to the cell #31 and cell #30 in the active columns of the 4th image respectively. In inference mode, due to the prediction made by the 3rd image, only the cell #31 in active columns of the 4th image is in active state at t4. As a result, only the cells in active columns of the 5th image have a segment activity above the threshold, which only turns

these cells to predict state at t4 and only predicts the 5th image as the next input data correctly.

In this paragraph, we test the robustness of both spatial pooling and temporal pooling using the proposed design. For the spatial pooling, we randomly pick 100 images from the KTH database and then distribute around 2% of total input bits into each input image in inference mode. For each input image with noise, we compare the active column list with that of the corresponding clean one and refers number of different elements as error count. We consider the inference for a given input image as failure if the error count exceeds 10% of the total active column number. The simulation result shows the failure rate of our 100 images is 3%.

Table 2.7 Test One of High Order Network

Network Data	Time Step								
	t1	t2	t3	t4	t5	t6	t7	t8	t9
HTM Input Images	1	2	3	4	5	6	7	8	9
List of Active Cells	✓	✓	✓	✓	✓	✓	✓	✓	✓
List of Predict Cells	☑	☑	☑	☑	☑	☑	☑	☑	

✓: indicates the list of active cells for input data at t in inference mode is same as the list of learning cells for input data at t in learning mode (successful inference).
☑: indicates the list of predict cells at t-1 is the same as list of active cells at t (correct prediction).

For temporal pooling, we replace the original 5th image with a totally different image in inference mode as the noise data. In inference mode, when we present the 4th image to our network, there is one cell in each active column of the original 5th image turning into the predict state as being taught in learning mode. However, since active column list of the noisy 5th is absolutely different from that of the original one, all cells in active columns of the noisy 5th image

are turned into active state to indicate the “unexpected” input data. Furthermore, we are not able to make any prediction for the next input since this noisy 5th image has never been seen in learning mode as shown in Table 2.8. As a result, all cells in these active columns are turned into the active state, when we present the 6th image. However, we can recognize and predict the 7th image correctly, since the input sequence is back to the learned one after then. This simulation result shows that, as we expect, the HTM network is capable to identify a noisy input data in inference mode but cannot make a correct prediction based on it. In addition, the noisy input only influences the recognition of the following input in inference mode.

Table 2.8 Noise Test of High Order Network

Network Data	Time Step								
	t1	t2	t3	t4	t5	t6	t7	t8	t9
Input Images Learning	1	2	3	4	5	6	7	8	9
Input Images Inference	1	2	3	4	10	6	7	8	9
List of Active Cells	✓	✓	✓	✓	×	⊗	✓	✓	✓
List of Predict Cells	☑	☑	☑	☑	☒	☑	☑	☑	

✓: indicates the list of active cells for input data at t in inference mode is same as the list of learning cells for input data at t in learning mode (successful inference).
☑: indicates the list of predict cells at t-1 is the same as list of active cells at t (correct prediction).
×: indicates the list of active columns or cells for input data at t in inference mode is different from those for input data at t in learning mode.
☒: indicates the list of predict cells at t-1 is different from the list of active cells at t (wrong prediction).
⊗: indicates the list of active columns for input data at t in inference mode is same as that for input data at t in learning mode, but all cells in the active columns are in active state in inference mode.

4.3.4. Performance Evaluation

To evaluate the performance of our proposed processor core, we implement the network in Table 2.4 on both CPU and GPU as our performance baseline. The GPU implementation of HTM uses CUDA C programming language with NVCC complier on Linux environment and is executed

on the NVIDIA K40c GPU, which includes 2880 CUDA cores. During spatial pooling, each kernel is responsible for the overlap computation and parameter update of one column in learning mode. The global inhabitation is implemented as a separately function using parallel bubble-sorting algorithm to take advantage of the computational power of GPU. For temporal pooling, we implement the operations in each cell as a kernel and perform the kernel on multiple threads in parallel. In the initial phase, we move the network parameters of all columns and cells to the GPU memory and optimize the program to minimize data movement between the GPU and host during processing, which can be time consuming. In addition, the dimensions of launched threads are optimized to maximize coalesce memory access that improves the memory utilization efficiency of GPU implementation.

We synthesis and route the processor core using the GF 65nm standard cell library, while analysis the timing using the Synopsys Primetime. The post-route area and power of each module in one processor core is summarized in Table 2.9.

Table 2.9 Performance Comparison with GPU Baseline

Design Features	GPU	This Work	Scale Down
Technology	TSMC 28nm	GF 65nm	32nm
Clock Frequency	745 MHz	100 MHz	100 MHz
Latency[2] (ms)	75.2	6.04	6.04
Area(mm^2)	561	886.7	260.6
Bandwidth (GB/s)	288	35.8	35.8
Power (W)	235	4.1	4.1
Total Improvement[1]	-	451x	1543x

1: The total improvement refers to the product of area, latency and power improvement.
2: The processing latency assumes a 10% existing distal synapse.

The performance comparison between the GPU baseline and our proposed design is listed in Table 2.10. The simulation results show that the processing latency and power in proposed design is improved 12.45x and 137x respectively. In addition, the proposed design can achieve the reported speedup with an 8.04x smaller memory bandwidth, which can demonstrate the efficiency of our proposed distrusted memory system. It is not surprised to see that area of proposed design is 1.58x larger than GPU, since the Tesla K40c GPU is fabricated by a more advanced technology. To be a fair comparison, we scale the area of the proposed design down to 32nm with the scaling factor in [38], which can achieve a 2.1x improvement in area.

Table 2.10 Post-Route Power and Area of the Proposed Design

Feature Name	Execution Lane	PE Controller	Central Processor
Clock Frequency		100	
Technology		GP 65nm	
Total Area (mm^2)	11.50	1.95	2.10
SRAM Size (Mb)	9.05	0.26	0.26
Power (mW)	51.3	92.2	61.2
Number of Unit	64	8	1

In both learning and inference mode, the processing latency per input data increases as more input patterns are learned. This is because learning new input patterns in temporal pooling may result in generating new distal segments in learning cells. Since we select the learning/active cells and predict cells based on the segment activity of cells in the active columns and all columns respectively, increasing the total number of existing segments can result in a longer overall latency

to process one input. In fact, selecting predict cells can contribute up to 90% of the total processing latency when most of the cells reach the maximum number of distal segments. On the other hand, the latency almost remains the same even learning new input data after that. The processing latency vs percentage of existing segments for both GPU and our proposed design in learning mode is shown in Figure 4.8. The 110% of existing segments in this figure indicates the number of existing segments reaches the max number of that in proposed design.

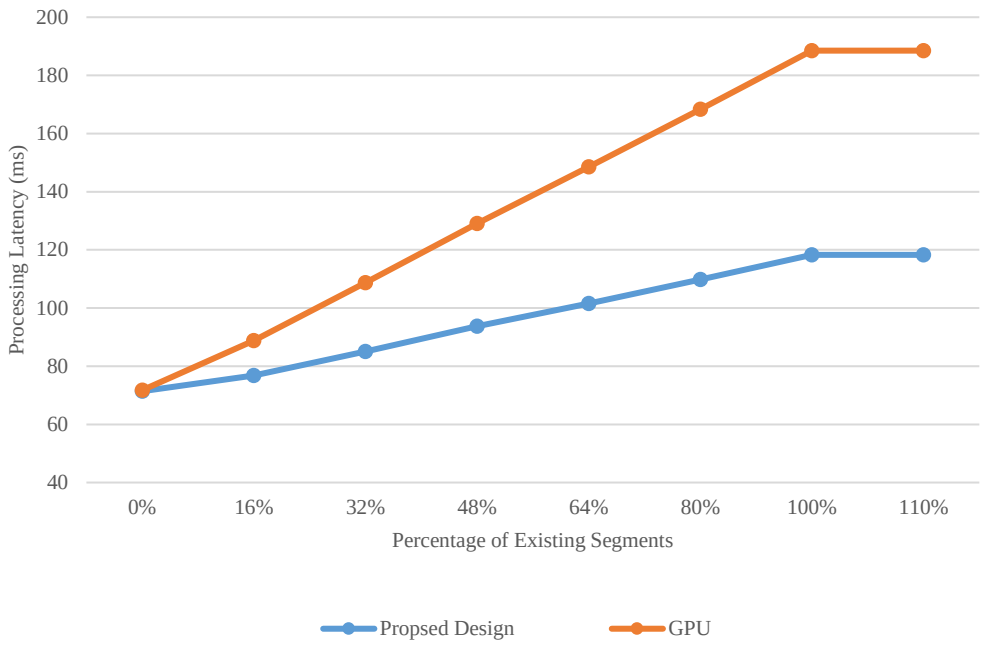

Figure 4.8 Processing Latency vs Percentage of Existing Segments.

Due to the lack of designs supporting the complete algorithm and a similar network size in the literature, it is difficult to do a peer comparison. In this work, we estimate the performance of the proposed design while mapping with 128 columns, each of which includes 8 cells and compare it with the designs in [25] and [26] as summarized in Table 2.11. The proposed design can beat the performance of [25] even with 3.3x more cells. For the design in [26], the latency of our proposed

design is 13% longer than that of [25] with 21.8% more cells. In addition, it must be noticed that we can simplify most of those customized units in proposed design to achieve a better speed if we only target on the tiny network like [25] and [26].

Table 2.11 Performance Comparison with State-of-the-arts

Design Features	[25]	[26]	This Work
Total Cell Number	800	300	1024
Technology	Nangate 45nm	TSMC 65nm	GF 65nm
Clock Rate	100 MHz	100 MHz	100 MHz
Latency Per Input	4.52 us	5.75 us	5.2 us
Max Cell Per Column	2	3	32
Configurable Design	N	N	Y

4.4. Conclusion

The main contribution of this work is the development of a customized hardware design of HTM that can support both the spatial pooling and temporal pooling. The proposed design can support up to 65,536 cells, which makes it capable to be applied to real-world applications, such as pattern classification for high resolution images and speech recognition in the real scenes. By applying multiple design optimization, such as the hierarchical architecture, customized units for hot-spot operations, flexible network mapping, and distributed memory the proposed design provides a total improvement of 451x over the optimized GPU implementation. In addition, the functionality validation in this work demonstrates the robustness of HTM against both spatial and temporal noise.

CHAPTER 5

DESIGN OF CNN ACCELERATOR

In this chapter, we present the design details of proposed CNN hardware accelerator, which includes the zero-skipping technology applied in each 1-D convolution primitive, dataflow beyond the 1-D primitive and overview of the hardware accelerator. To evaluate the performance of the proposed design, we benchmark the convolutional layers in two state-of-the-art CNNs on an array of 96 PEs and compare the performance against Eyeriss.

5.1. Zero-Skipping Technology

For any convolutional layer, we can divide the computation of entire layer into multiple 1-D convolution primitives running in parallel, each of which refers to the operations between one row of the input features and weights. Each 1-D primitive can generate one row of the partial sums, referred as the psums row. To compute one complete row of the output feature, we perform the element-wise accumulation on all psums rows generated by all channels and rows of the same filter. Since the total processing latency of a CNN equals to the latency sums of all 1-D primitives, improving the latency of 1-D primitive by skipping the operations with zero input features and weights can provide us a significant performance improvement on the entire CNN system.

5.1.1. Existing Challenges

For the computations within each 1-D primitive, we can send one pair of input feature and weight to a multiplier at each cycle and accumulate all products within the same window to get one element in that psums row. Then, we slide the weight row with a given stride and repeat previous operations until it covers the entire input row as shown in Figure 5.1. Therefore, the latency required for 1-D primitive relates to the total number of input feature and weight pair.

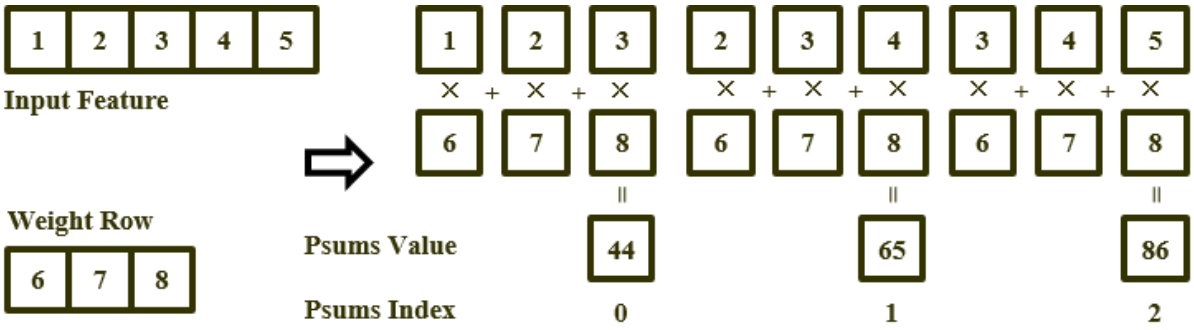

Figure 5.1 Example of 1-D Primitive with a Stride of One

In theory, since the none-zero instances of input features and weights may only account for a small portion of the input pairs, skipping operations with zero value should significantly reduce the total latency of convolutional layers. However, we have to face the following challenges to implement the zero skipping technology in real world.

1) Existing work cannot identify the zero input features before reading out them from the memory. For a given none-zero input feature, it cannot skip the corresponding zero weights before reading them out. As a result, this implementation cannot improve the memory access energy for either input features or weights.
2) Skipping computation with zero input features and weights as shown in Figure 5.2 (a) can improve the power consumption in arithmetic units. This method, however, cannot improve the memory access power and processing latency due to the idle cycles in pipelined multiplier.
3) Due to the random distribution of these none-zero input features, extra operations are necessary to locate the corresponding weight and product for each of them instead of simply incrementing on those of the previous ones. In addition, optimization of these operations becomes important to minimize the idle cycles in the pipelined multiplier.

4) For most state-of-arts CNNs, the first convolutional layer usually has relatively dense input features and contributes a nonnegligible percentage of total processing latency. As a result, the proposed design should be able to support dense input features with minimal additional hardware resource and processing latency.

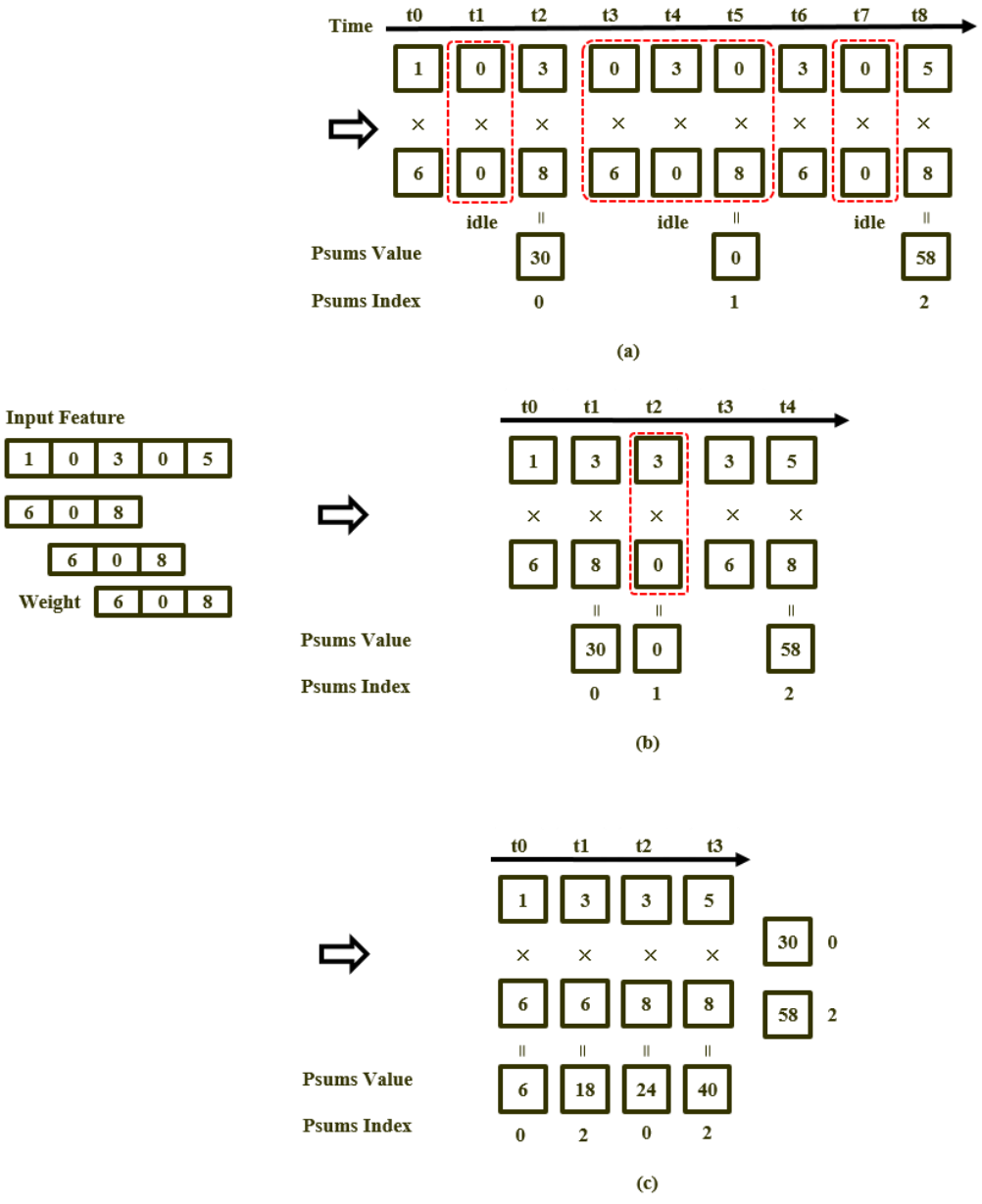

Figure 5.2 (a) 1-D Primitive without Any Zero-Skip Technology. (b) 1-D Primitive with Skipping Zero Input Features. (c) 1-D Primitive with Skipping Zero Input Features and Weights

5.1.2. Proposed Solutions

To skip the zero input features in a row, we encode each input feature row into the RLC format as shown in Figure 5.3. Every three none-zero value are packed into a 64-bit data chunk, where each of them is followed by a 5-bit offset. The 5-bit offset indicates the distance from previous value. The last bit of each data chunk indicates the end of current row. In case that there are more than 31 zeros between two adjacent none-zero value, we insert a zero into the data chunk to have a larger offset. The RLC allows us to only deliver none-zero input features to the PEs in succession to keep the pipelined arithmetic occupied.

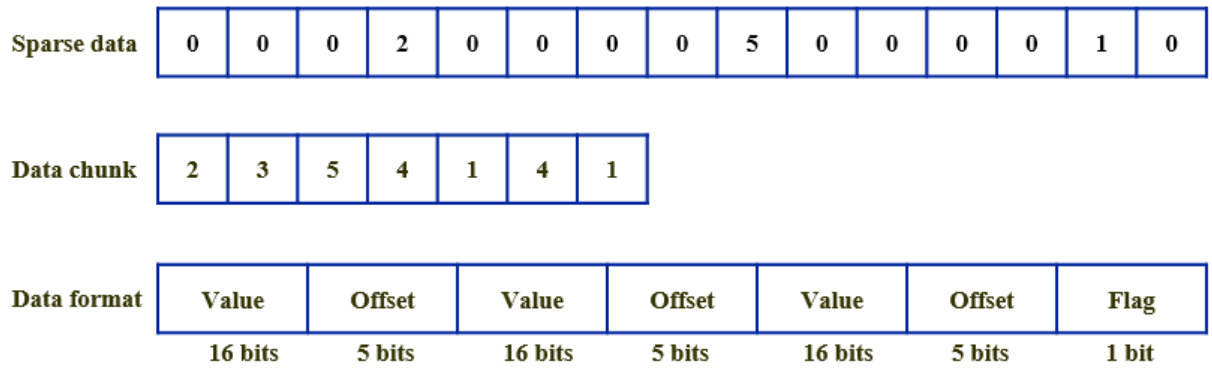

Figure 5.3 Example of Input Row in RLC format

For data encoded in RLC format, the required memory space and bandwidth depends on the percentage and locations of zero value due to the 5-bit overhead for each offset. In this work, we randomly generated 100 rows for each sparsity level and encode them into RLC format. We varied the row size from 15 to 1024 to simulate convolutional layers of different sizes as shown in Figure 5.4. Each line in the figure represents a different word number per row ranging from 14 to 1024. We compute the y-axis by dividing the total bits number required for one row by the word

number. In the dense format, each word always takes 16-bit. The simulation result shows that encoding data into RLC format can lead to savings in memory space and bandwidth when the sparsity is above 30%. In general, the input data to the first layer of CNNs are directly from the input images and can have and density up to 100%. For these data, we packed every four 16-bit value into a 64-bit data chunk as input for the proposed design.

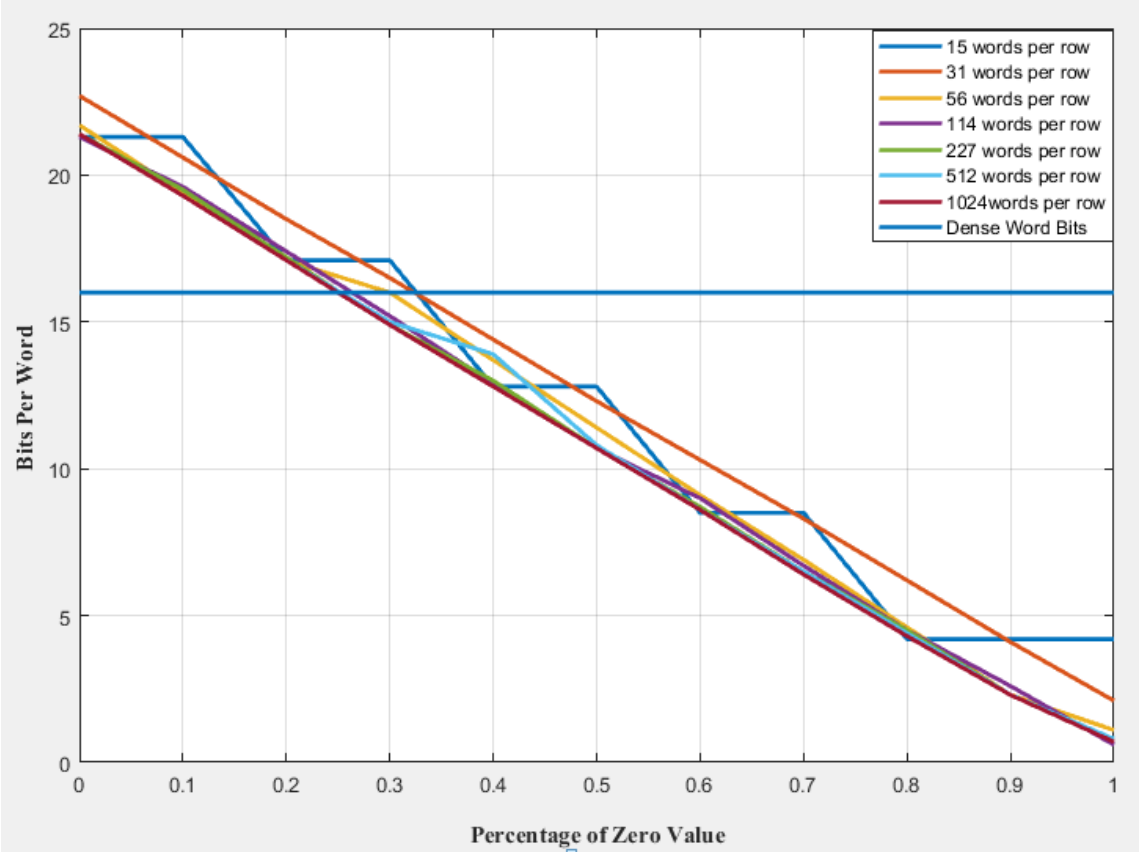

Figure 5.4 Bits per Word vs Data Sparsity.

Encoding input feature into RLC format allows the proposed design to deliver none-zero values to the arithmetic unit at every cycle. Now we need collect the corresponding weights for each none-zero input feature within the same window to perform the element-wise multiplication and summation, which refers to the horizontal processing sequence. However, this sequence is not

efficient to process the RLC encoded input feature row due to the discrete distribution of none-zero values as mentioned in the previous section.

In this work, we propose a vertical processing sequence that allows the out-of-order product summation for different psums. More specifically, since each input feature in the 1-D primitive can be reused for multiple times with all weights belonging to the same "column", the proposed design generates one product with each of these weights but the same input feature. We repeat the same iteration for all none-zero input features within the 1-D primitive and accumulates the products within same window to obtain the complete psums. To locate the weights for each of these "column", we calculate the index of each none-zero input feature by incrementing the offset in RLC format. Then, for any given input feature, we can obtain the index of "bottom" weight using (5.1). For the rest weights in the same column, we calculate their index through incrementing the "bottom" index by the weight stride value.

$$Index_{Weight} = Index_{Feature} \% \, Stride_{Weight} \tag{5.1}$$

Since products for one psum are generated out-of-order in the vertical processing sequence, we provide a "label" for each product that indicates the address of corresponding psum in the psum memory bank, which allows us to accumulate them out-of-order later. For any input feature, we can calculate the label of "bottom" product using (5.2) and decrement it to get the psums label of rest products.

$$Label_{psum} = Index_{Feature} \div Stride_{Weight} \tag{5.2}$$

In Figure 5.5, we present an example of converting the horizontal processing sequence in Figure 5.1 into the vertical one. Since the index of the input feature in yellow is 1, we can find that the weight index and psum label for the "bottom" product is 0 and 1 respectively using the (5.1) and (5.2). Now we can figure out the weight index and psum label for the next product in column as 1 and 0 by increment/decrement that by one. Since we can implement both (5.1) and (5.2) using the customized pipelined modules, using proposed sequence allows us to deliver none-zero input features and corresponding weights continuously to the multiplier. As a result, we can reduce the total number of required cycles in Figure 5.2 (a) down to 5 with the encoded input feature as shown in Figure 5.2 (b), which also demonstrates the vertical processing sequence can significantly improve the total number of required cycles for the sparse 1-D primitive.

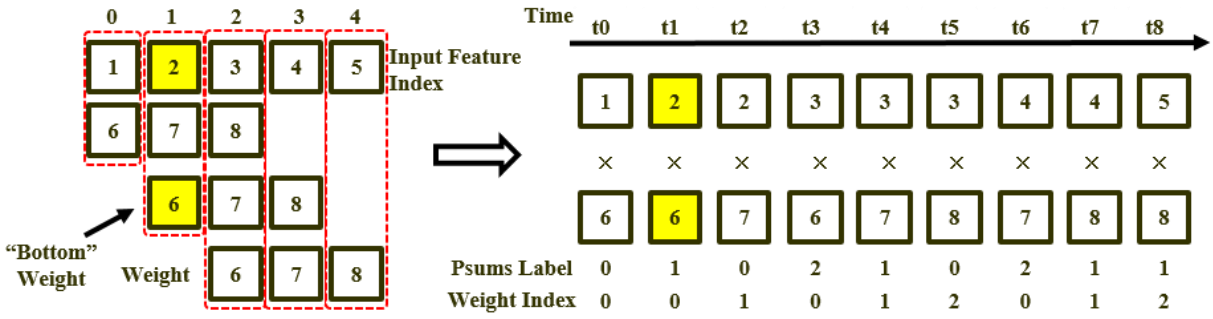

Figure 5.5 Example of Vertical Processing Sequence.

In addition to the performance improvement from none-zero input features, the vertical processing sequence also provides us an opportunity to exploit the zero weights. The proposed design encodes each "column" of the weight row in 1-D primitive into the RLC format as shown in Figure 5.6. Considering the weight row is usually smaller than that of the input feature row, we reduce the bit number of offset to two. The 1-bit flag is used to indicate if it is the last weight in current column. We can still use (5.2) to calculate the psum label of the "bottom" product, but the

labels of rest products now are obtained by decrementing the “bottom” label by the offset plus one now.

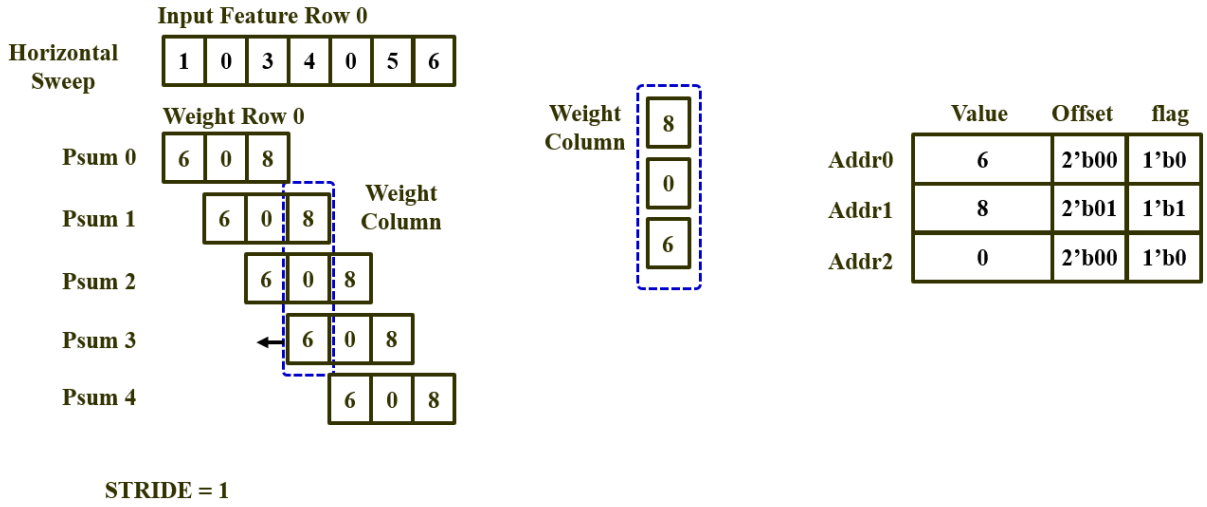

Figure 5.6 Example of Weight Row in RLC Format.

Encoding columns of the weight row into RLC format allows us to skip most of the zero weight and delivery one pair of none-zero value to the multiplier without any idle cycles as shown in Figure 5.2 (c). As a result, the proposed zero skipping technology can reduce the total required cycle for the example in Fig.5.2 (a) from 9 to 4, while the sparsity of input feature and weight is 40% and 33% respectively. We can also apply the zero skipping technology to the input feature in dense format without any additional latency and hardware resource. The only difference is that we calculate the index of each input feature by always incrementing that by one until we reach the end of an input feature row.

5.2. Proposed Dataflow

Most of these state-of-art implementations exploit parallelism by processing numerous 1-D primitives with various input feature and weight rows in parallel, each of which is mapped on one PE [30]-[37]. However, due to the irregularity of the sparse input feature encoded in RLC format, it is difficult to locate a particular input feature row among all the data without decoding. However, the input feature rows in dense format require more memory space and cannot skip the reading of zero values. In addition, the performance of existing dataflow highly depends on the bandwidth of memory used to store the input feature due to the simultaneous read request from various PEs.

In the proposed design, we always broadcast the same input feature row to all PEs and perform the 1-D primitives in parallel with different weight rows in all PEs. The main advantages of proposed dataflow are as follow.

1) Since all PEs share the input feature row now, the proposed dataflow allows us to read each input feature from the first row to the last one in order. As a result, we can directly store the encoded input features on-chip, which requires less memory space for sparse data and allows us to skip the zero input features with the zero skipping technology described in section II.
2) The proposed dataflow reduces the bandwidth requirement for memory used to store input features and simplify the control logic, since there is always only one read request at the same time. Along with the vertical sequence, we only need read each input feature once for the same weight filter, which provides us the opportunity to directly streaming the input feature from off-chip memory.

3) For most of the state-of-arts CNNs, the zero values usually come from the input features only before pruning. With the proposed dataflow, all PEs now receive the same number of none-zero input features, which can result in a similar number of operations in all PEs. As a result, inefficiency of PE utilization resulted from the unbalanced workload can be significantly improved.

To improve the performance, the proposed design performs calculations on multiple 1-D primitives simultaneously with same input feature row. In addition, since it is usually not possible to store all input features on-chip at one time, maximizing the reuse of each input feature row can reduce the times reading it from memory, which significantly improves the memory access energy and latency. For a given input feature row, we have two forms of data reuse that can be performed in parallel as shown in Figure 5.7 (a) and (b) respectively.

1) Within each filter, we can reuse one input feature row for L_w/S_w times with various weight rows. The L_w is the length of weight filter and S_w is the stride value. This reuse is from the overlap of weight rows, which results from the vertical sliding of filter across the entire input feature.
2) Beyond one filter, we can reuse one input feature row for m times with the same row in various filters to generate psum rows belonging to different output features. The m is the number of weight filters in a given layer.

To exploit the parallelism within each weight filter, we group several PEs together, referred as a PE group. We store the same weight filter on all PEs in the same group. The number of PEs per group is equal to the row number in that overlap and can be calculated using (5.3).

$$N = floor(L_w \div S_w) \tag{5.3}$$

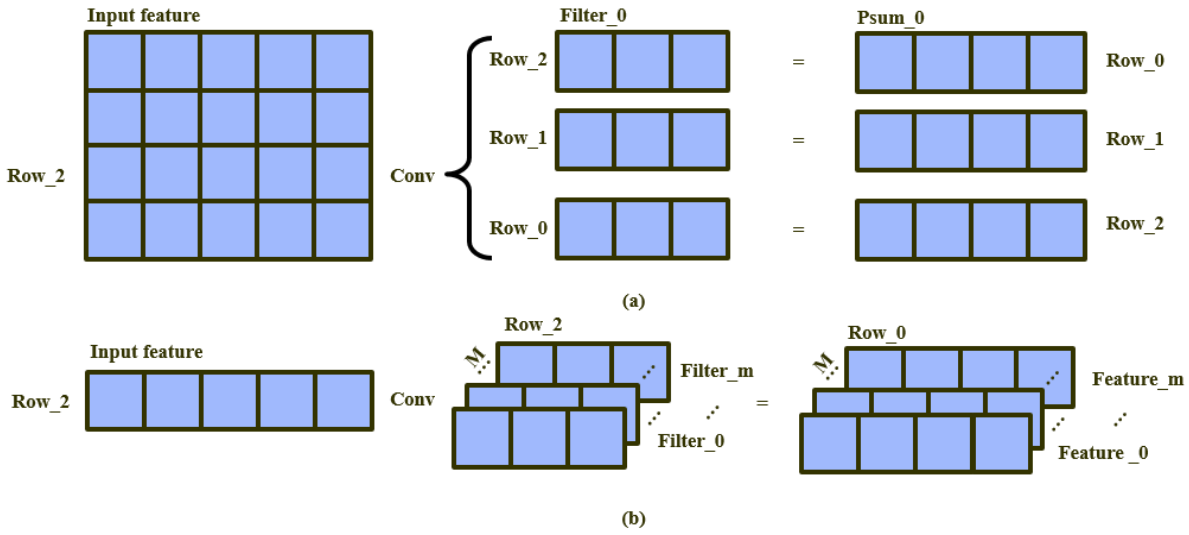

Figure 5.7 Example of Input Feature Row Reuse Using 3 x 3 Filter.

To process one complete input feature, the PEs from the same group are enabled one by one with an interval of "stride" rows of input feature, since there is not any overlap until we start to vertically slide the weight filter. During the processing in Figure 5.8, each enabled PE marked in red performs the1-D primitive with the received input feature and corresponding weight row, and then accumulates current psum row with previous results that have the same psum row index. The "w_r_x" and "f_r_x" indicate the xth row of weight and input feature. This example assumes a weight length of 3 and a stride value of 1. As a result, there are 3 PEs in each group and interval for PE enabling is 1, which means we should enable one more PE after the processing of one input feature row until all the PEs in that group are active. After the processing of input feature row 3, the weight index in PE0 is reset since it completes the processing for psum row 0 and will start the processing for psum row 4 from next input feature row, which requires PE0 not to accumulate the results of input feature row 4 with previous psum rows.

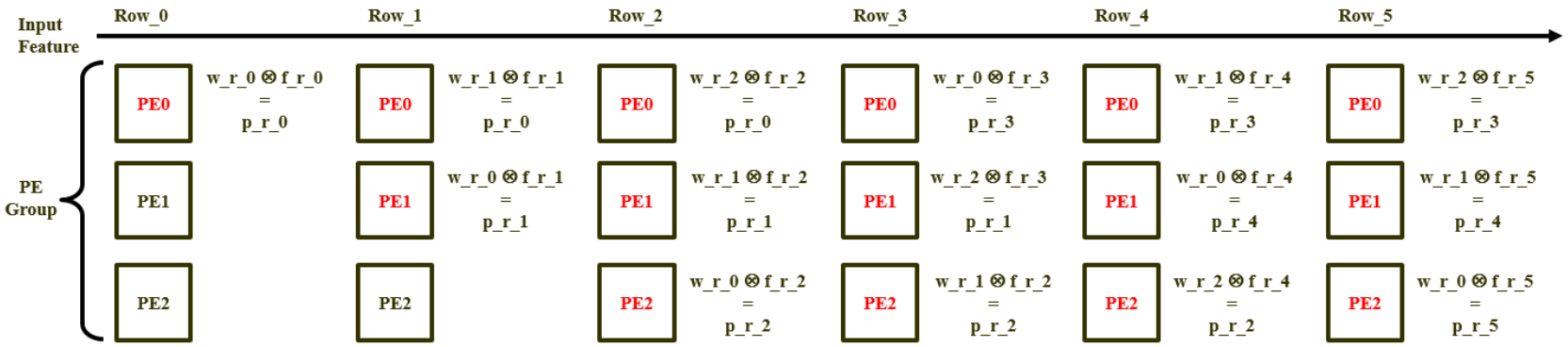

Figure 5.8 Example of Processing Within One PE Group

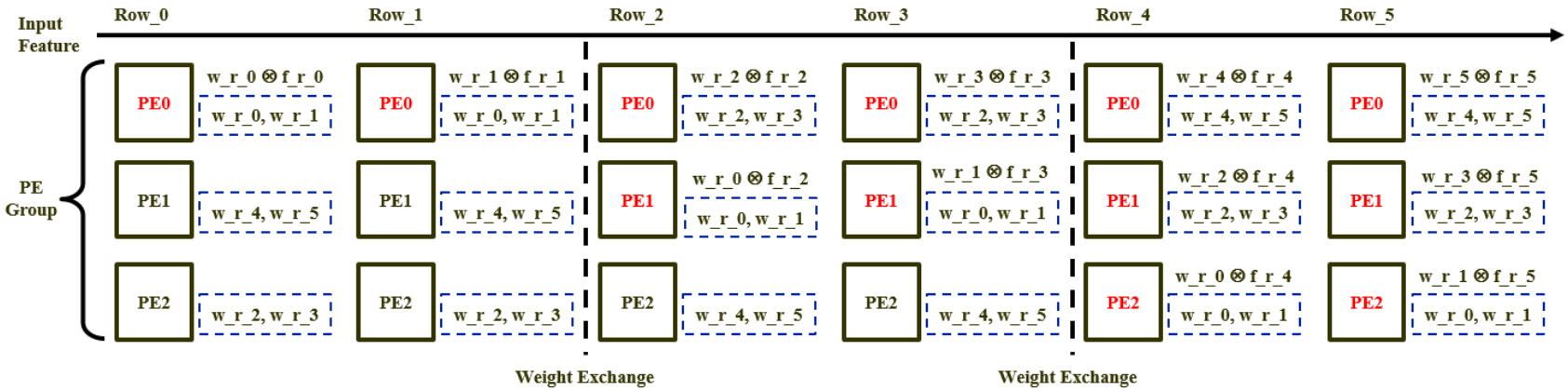

Figure 5.9 Example of Processing with Weight Exchange

To implement the vertical filter sliding, the proposed dataflow increments the index of weight row in all enabled PEs after the processing of each input feature row unit until the whole weight filter is covered. In that case, we need to reset the weight row index and increment the psum row index to indicate the completeness of current psum row. All enabled PEs repeat the same iteration until entire input feature is processed.

For a given convolutional layer, the total number of required 1-D primitives is constant regardless the processing order. Though the dataflow in [30]-[37] allows us to operate on multiple input feature rows at the same time, the proposed design can perform the 1-D primitives between weight rows of numerous filters and one input feature row simultaneously, which results in a similar throughout with a given number of PEs. In addition, since the psum rows required accumulation always stays in the same PE in the propose dataflow, we can perform the psum accumulation without any inter-PE communication and bury the latency into the processor pipeline to further improve the performance.

From the aspect of PE utilization, since the number of PEs per group relates to the stride of weight filter and is usually small in the proposed dataflow, we can maintain a high percentage of active PEs regardless of the shape of input feature and weight filter. In Table 5.1, we summarize the percentage of active PEs for AlexNet in [13] and this work.

Table 5.1 PE Utilization Rate of AlexNet

Layer	Eyeriss [13]	This Work
CONV-1	92%	100%
CONV-2	80%	98.9%
CONV-3	93%	100%
CONV-4	93%	100%
CONV-5	93%	100%
Average	88%	99.8%

To obtain one complete output feature row, we need to accumulate the corresponding psum rows resulted from all the channels of that filter. However, due to the limited on-chip memory, we cannot always store all the channels of target filter or the intermediate psum rows of entire output feature on-chip. As a result, we have to move the intermediate all the psum rows stored in each PE between the on-chip and off-chip memory for multiple times, which can result a significant amount of memory access energy and latency. The required data transaction times for psum rows depends on the on-chip storage capacity of filters. For instance, we need to move the psum rows of an input feature between the on-chip and off-chip memory for 16 times, if the total number of weight filter channel and the maximum channel number stored on-chip is 256 and 32 respectively.

To reduce the data movement between the on-chip and off-chip memory, the proposed dataflow supports a weight exchange among all PEs from the same group. Instead of storing the entire weight filter in each PE, we distribute the rows of target weight among all PEs and perform the weight exchange every time we complete the processing of all local weight rows as shown in Figure 5.9, where the "w_r_x" and "f_r_x" indicates the xth row of weight and input feature. This example assumes a weight filter length of 6 and a stride value of 2. As a result, there are initially 2 weight rows in each PE as shown in the blue dot block and we perform the weight exchange after the processing of every two input feature rows. With the weight exchange, we are capable to improve the maximum channel number stored in one PE by N. Therefore, it can reduce the total required times of data movement to $(W_c/P_c - 1) \times 2 / N$, where N is the PE number in each PE group, W_c is the channel number per filter, and P_c is the maximum channel number stored in one PE for each processing pass.

In most cases, processing with the weight exchange offers a better latency and power due to the improvement of off-chip data movement. However, the proposed weight exchange requires

an additional latency for each weight row and the maximum cycle number of that latency is close to $N + W_S$, where W_s is number of weights per row. As a result, storing the entire weight filter in each PE can achieve a better performance in case that all psum rows of one input feature can be stored on-chip, such as the CONV3-5 in AlexNet.

Regarding the parallelism among various weight filters, the proposed design divides the PE array into multiple groups, each of which stores multiple channels of one filter and performs the operations mentioned above. With all these groups running in parallel, we are able to generate multiple psum rows belonging to various output features simultaneously. We refer the required computations for these filters as a processing pass. In a processing pass, we only need to broadcast the corresponding channels of one input feature to the PE array once and the channel number per pass is larger while using weight exchange as described above. Usually, the convolutional layers in state-of-arts CNNs require multiple passes to complete the processing. The schedule of processing passes for the "conv2-1" in VGG16 without and with the weight exchange are shown in Figure 5.10 (a) and (b) respectively. In this example, we assume a batch size of 1. For the batch size larger than 1, the processing passes can stay the same. However, since there are more input features in each pass, it requires more on-chip memory to store the intermediate psum rows. In Figure 5.10 (a), we totally need 8 processing passes, every two of which work on the same filters but different channels of them, to process all weight filters without weight exchange. On the other hand, we can store all the weight channels on-chip due to the weight exchange in Figure 5.10 (b). Though the number of processing passes is smaller in (b), we need process more channels of the input feature during each processing pass. Therefore, the total number of computations in (a) and (b) is the same.

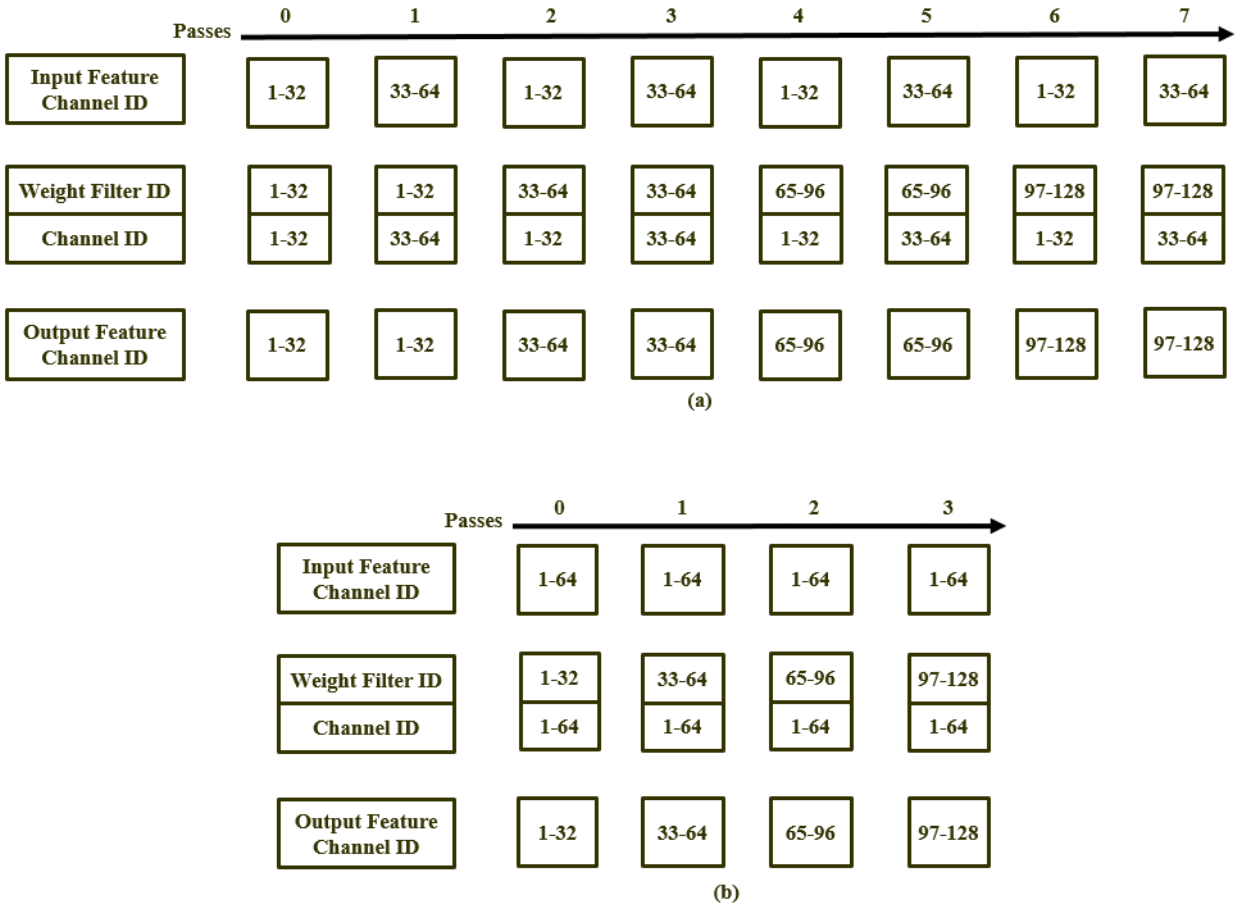

Figure 5.10 (a) Schedule of Processing Passes for “conv2-1” in VGG16 without Weight Exchange. (b) Schedule of Processing Passes for “conv2-1” in VGG16 with Weight Exchange.

5.3. Design of Proposed Hardware

To support the zero value skipping technology and dataflow described in this paper, we propose an AISP implementation for CNN, which consists of a central processor and an array of 96 PEs as shown in Fig.5.11. The proposed processor connects to the off-chip DRAM through two 64-bit signal direction data bus and organizes the PE array as an 8×12 matrix. Each PE in that matrix connects to the 4 neighbor PEs through a mesh network. Meanwhile, we connect each PE row of the matrix to the central processor through a dedicated port to implement the global data communication.

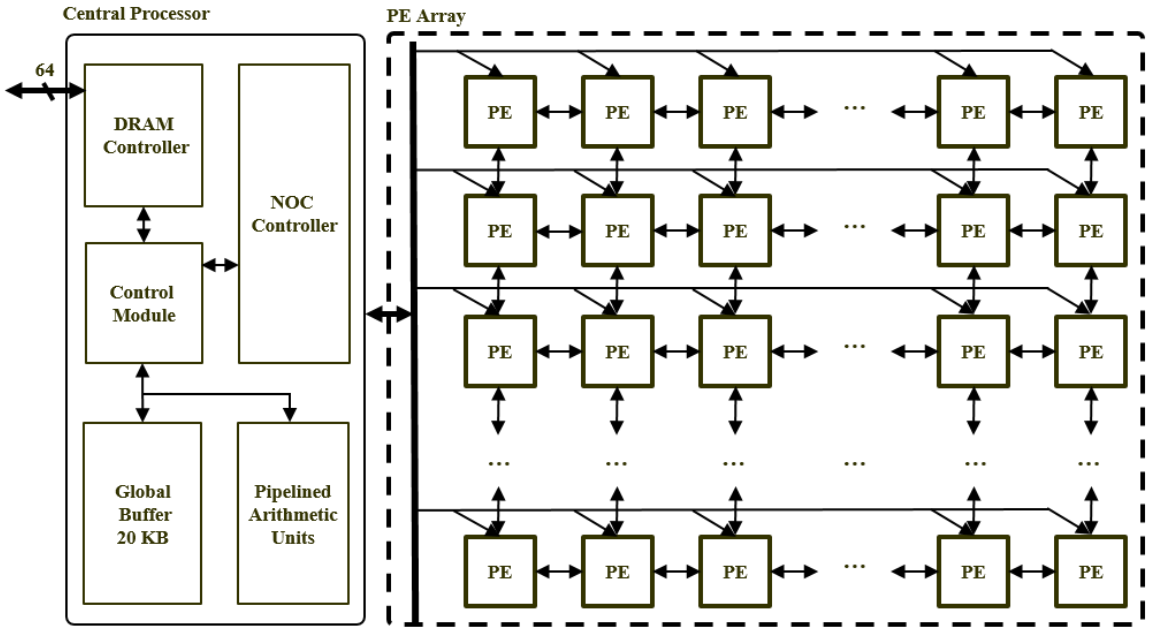

Figure 5.11 Schematic of Proposed Processor Core.

5.3.1. Central Processor

The central processor consists of 5 parts, a control module, a NOC controller, a group of arithmetic units, a DRAM interface and a 20 KB global buffer. The function of these modules is as follow.

1) The control module is the “brain” of our proposed design, which fetches and executes the customized instructions. It is also responsible for the configuration of entire processor core to handle various convolutional layer shapes and PE synchronization during processing.
2) The NOC controller is responsible for performing the data communication between central processor and PE array, so that the central process can broadcast these none-zero input features to all the PEs. In addition, by using the customized instructions, the NOC controller can access the memory in any PE to send the corresponding weights and psums from off-chip memory and read the results back when processing completes. We will discuss more implementation details in the later section.
3) The DRAM controller implements the data communication between on-chip SRAMs and off-chip DRAMs. By using the customized instructions, we can deliver the data from DRAMs to two locations, the SRAMs in central processor and SRAMs in PE. Meanwhile, the instructions allow us to configure the size and initial address of data transaction during processing. In addition, the output feature read from PE can be encoded into the RLC format in DRAM interface module.
4) The arithmetic units in the central processor consists of two parts, the RLC decoder and pipelined modulo/divide unit. The RLC decoder is designed to calculate the index of each none-zero input feature by incrementing the offset in RLC data chunk. To

calculate the “bottom” weight index and psum label using (5.1) and (5.2), we modify the hardware design of x mod z in [38] as shown in Figure 5.12, where “I” is the index of none-zero input feature and “S” is the stride value of filter. The “L” in this figure is the initial label of product that is usually equal to 0 and “C” is the increment for product label that starts from 12’h800 in this work. The pipelined modulo/divide unit use output from decoder as input data and provides one weight index and one psum label together at every cycle.

5) The size of global buffer in the central processor is 20KB (5 banks of 512×64-bit). Since we directly write/read the weight and psums to/from the SRAM in each PE from/to the DRAM, we only need store none-zero input features encoded RLC format and instructions in this global buffer. For the VGG16 and AlexNet, the size of instruction is less than 1% of that of the data.

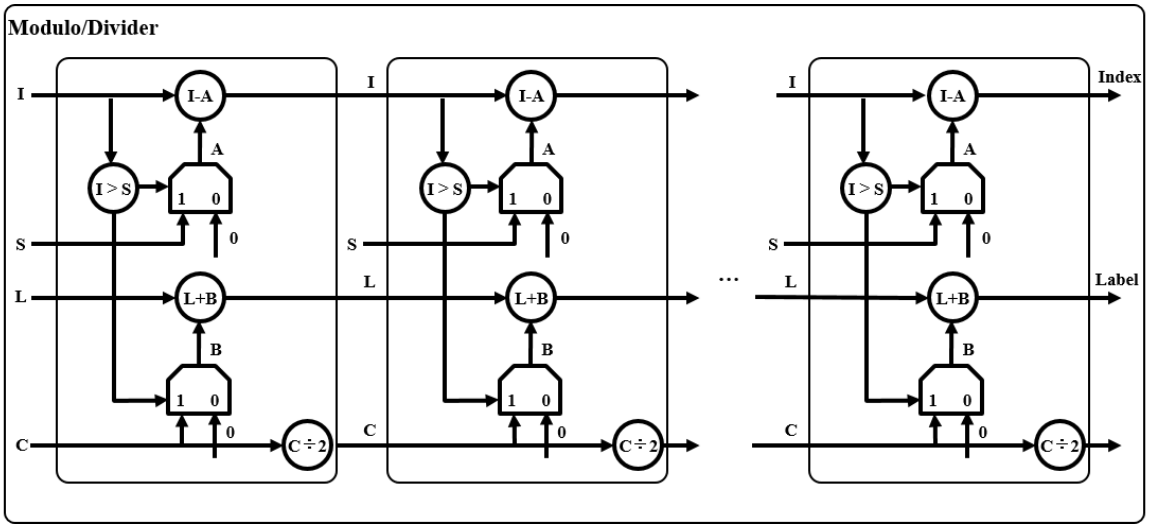

Figure 5.12 Schematic of Modulo/Divider Unit.

5.3.2. Processing Element

The PE in the proposed design consists of 3 parts, a control module, a 288×19-bit SRAM and two identical execution lanes, each of which includes an unroll unit, a pipelined multiplier, an accumulation unit and a 256×16-bit register file as shown in Figure 5.13.

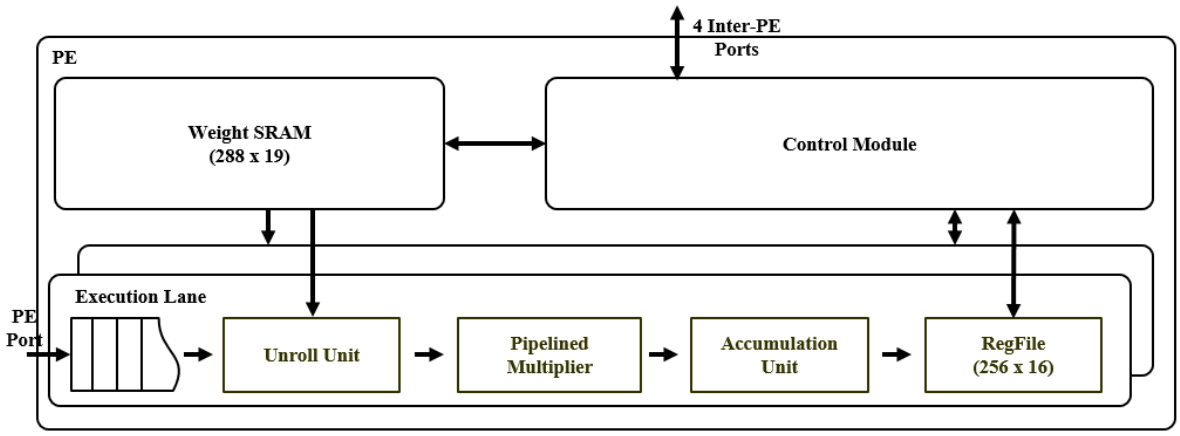

Figure 5.13 Schematic of Proposed Processing Element

The control module in each PE is designed to implement the inter-PE communication, data movement with central processor and monitor the local processing status. The 288×19-bit SRAM in each PE is used to store the weight and has 2 read/write ports, each of which is assigned to one of the execution lanes. Due to the 2-bits offset and 1- bit terminal flag in RLC format, we have to the increase the word size of each weight from 16-bit to 19-bit, which results in a memory space increase of 15.7%. Instead of storing the psums in centralized global buffer, we distribute the on-chip SRAM among PEs. The distributed memory system allows us to avoid the overhead resulted from the moving data between the central processor and PEs.

The execution lane in PEs is designed to implement the zero skipping technology described in section II. More specifically, the FIFO buffer in each execution lane receives a series of none-

zero values along with their “bottom” weight indexes and psum labels from central processor during processing. The purpose of “unroll_unit” is to read out these values in order and calculate the weight address and psum label for each weight and product within the same “column” as shown in Fig.7. Since the address of next none-zero weight can be calculated using the 2-bit offset stored in current one, the “unroll_unit” is able to continuously send none-zero pairs of input feature and weight to the pipelined multiplier. Based on received the psum label, the accumulation unit accumulates the product from multiplier with the previous result stored in register file.

Controlled by the customized instruction, the execution lanes in each PE can work on the same weight filter together, referred as cooperation mode or process two filters in parallel, referred as parallel mode. For the cooperation mode, the control module divides received none-zero input feature into two groups based on the parity of their “bottom” psum labels and each execution lane is responsible for the computation of one group. For the parallel mode, we can divide the weight SRAM in each PE into two parts, each of which stores multiple channels of one weight filter, so that these execution lanes can process the same input feature with various filters in parallel.

The cooperation mode can improve the latency without any additional request for on-chip memory space to store the weight or psums. However, due to the random distribution of none-zero input features in each row, the workload of each execution lane can be unbalanced under the cooperation mode, which results in a decrease in the computation efficiency compared to that of execution lanes under parallel mode. Unfortunately, since there are more weight filters under processing, it requires more space to store the intermediate psums and reduces the channel number on-chip in each pass.

By supporting flexible utilization of the execution lanes, our proposed design can provide the optimized solution for various CNNs. For instance, since we cannot store all the channels of a

filter or the entire output feature on-chip for most of the layers in VGG16, the cooperation mode improves the latency without increasing the amount of data movement between on-chip and off-chip memory compared to processing by only one execution lane. On the other hand, for the convolutional layers such as the CONV3-5, since we can store the entire output feature on-chip, the parallel mode allows us to reduce the processing latency to about half by doubling the number of filters in each pass.

5.3.3. Network-on-Chip

The propose design supports two kinds of NOC to implement the data movement between the central processor and PE array, and the data movement among PEs. In the proposed design, we connect the central processor and each PE through two single direction data buses, which is 64-bit and 32-bit respectively. We assign an "active" flag to each PEs and configure these "active" flags through the customized instruction. Every time the central processor brings the weights or psums onto the bus, all the PEs can see these packages but only the "active" PEs can grab and store them into local memory. As a result, the proposed design can support various network topologies including broadcast, multicast and point-to-point and switch among them during the processing.

To support the weight exchange described in Chapter 5.2, we implement a mesh-based topology that allows us to group any number of PEs together and perform the data exchange within each group. To perform the data exchange, we assign a source and a destination to each PE within a group. During the weight exchange, each PE sends the local weight to the destination PE while stores the weight received from the source PE into local memory. Meanwhile, we directly forward the weight received from none-source PE to the neighbor on the correct path.

For hardware design, we implement the proposed topology by configuring the data move direction for each port in the PEs using customized instruction. More specifically, during the data exchange, one of these output ports in each PE is configured to send the local data to its neighbor while there is another output port assigned to forward all the data received from the input port connected to the none-source PE. Meanwhile, the PEs store the data received from the input port connected to the source PE into local memory. In Figure 5.14 (a), we divide a 4 x 4 PE matrix into 5 groups, each of which includes 3 PEs. Within each group, we assign one weight row initially. The weight filter in each PE group is different, but all the groups share the same allocation of weight rows. In Figure 5.14 (b) and (c), we present the data movement within different PE groups. For each data exchange, the Red PE store the weight received from blue PE and send the weight to yellow PE. The yellow PE send the local weights to blue and forward the PE received from blue PE to the read PE.

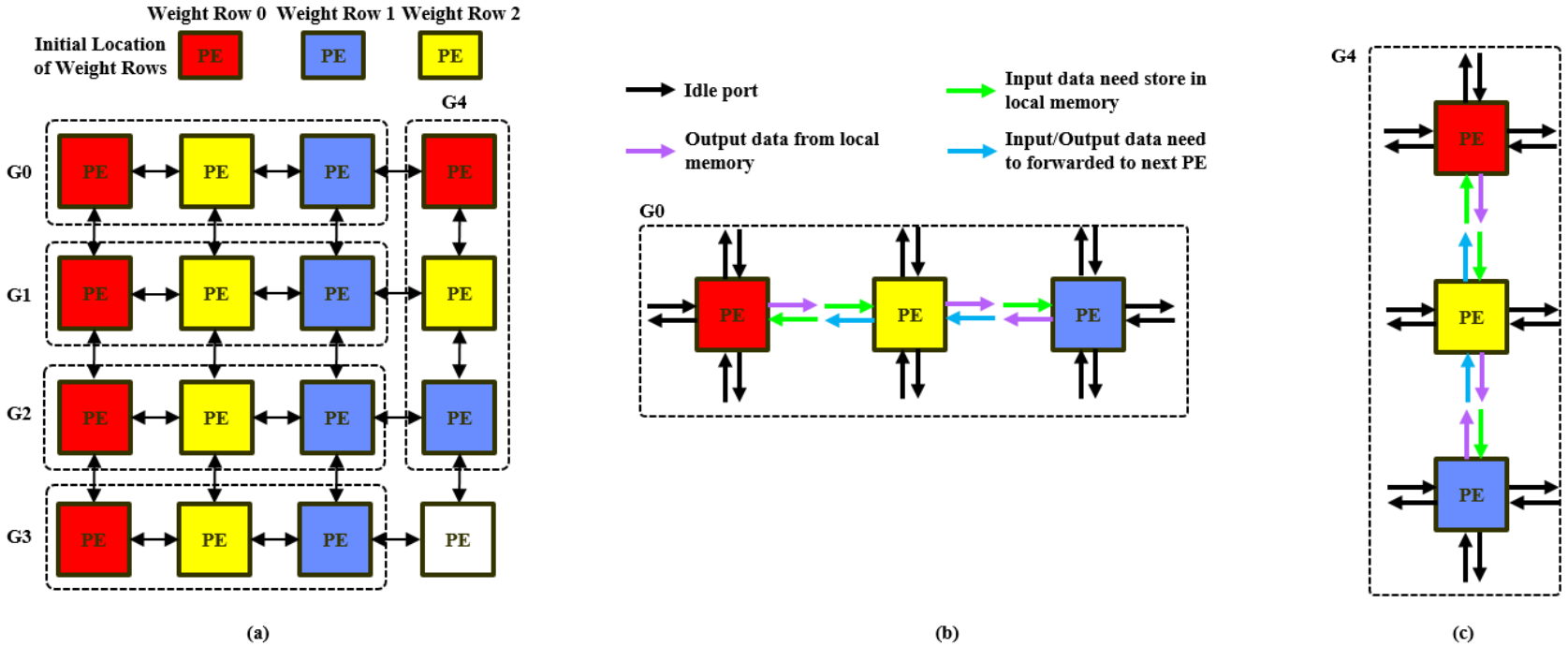

Figure 5.14 Example of PE Grouping and Port Configuration.

5.3.4. Customized Instruction

In the propose design, we manage the processing progress for each convolution layer through a set of customized instructions as summarized in Table 5.2-5.5. Most of the customized instructions is 32-bit wide. For DRAM related instructions, we use the first 32-bits to store the control information, such as the size of memory access and the initial address of target on-chip memory. Meanwhile, we store the initial address of the DRAM in the second 32-bits of these instructions. In Table 5.3, we elaborate the instructions used to set the configuration of each port in proposed PEs.

Table 5.2 Configuration Instructions in Proposed Design

Instruction Type [31: 29]	Instruction Function [28 : 26]	Instruction Details [25 : 0]
001	001	[25 : 14] : input feature row width [13 : 02] : weight filter row width
	010	[25 : 14] : partial summation row width [13 : 02] : address offset between filters
	011	[25 : 14] : # of input feature rows per psum row [13 : 02] : # of weight filter rows per psum row
	100	[25 : 14] : initial address of psums in PEs [13 : 02] : initial address of weight in PEs
	101	[25 : 18] : set active PEs row [17 : 06] : set active PEs in active rows [05]: reset PEs in inactive rows if set to 1
	110	[25 : 14] : set move direction for received data during PE data exchange. [13 : 11] : set move direction for local PE data during PE data exchange.
	111	[25 : 14] : value of filter stride [13 : 08] : configure data processing precision

Table 5.3 Details of Port Configuration Instructions

Instruction Function [28 : 26]	Target Ports	Instruction Definition
110	[25 : 23]: North Output Port	000: Stay idle, dose not forward any data 001: forward data from north input port 010: forward data from south input port 011: forward data from east input port 100: forward data from west input port 101: send data in local SRAM
	[22 : 20] : South Output Port	000: Stay idle, dose not forward any data 001: forward data from north input port 010: forward data from south input port 011: forward data from east input port 100: forward data from west input port 101: send data in local SRAM
	[19 : 17] : East Output Port	000: Stay idle, dose not forward any data 001: forward data from north input port 010: forward data from south input port 011: forward data from east input port 100: forward data from west input port 101: send data in local SRAM
	[16 : 14] : West Output Port	000: Stay idle, dose not forward any data 001: forward data from north input port 010: forward data from south input port 011: forward data from east input port 100: forward data from west input port 101: send data in local SRAM
	[15 : 13] : Input Ports	000: Stay idle, dose not write any data 001: write data from north input port into memory 010: write data from south input port into memory 011: write data from east input port into memory 100: write data from west input port into memory

Table 5.4 Processing Instructions in Proposed Design

Instruction Type [31: 29]	Instruction Function [28 : 26]	Instruction Details [25 : 0]
101	001	[25 : 12] : # of input feature rows to be processed [11 : 05] : initial address of input feature in central SRAM. [04 : 00] : # of cycles between two adjunct SRAM read
	101	[25 : 14] : target initial address of weight in PE [13 : 02] : # of weight to be transferred during data exchange

Table 5.5 DRAM Access Instructions

Instruction Type [31: 29]	Instruction Function [28 : 26]	Instruction Details [25 : 0]	Instruction Details [25 : 0]
010	101	[25 : 14] : target initial address of weight in PE [13 : 02] : # of weight value to be read from DRAM to local PE	Read weight filter from DRAM to local PE SRAM (19-bit each value)
001	000/100	[25 : 14] : target initial address of psums in PE [13 : 02] : # of value to be read/write from DRAM to both local PE SRAM	Read/Write partial summation from the DRAM to both local PE SRAM(16-bit each value)
	001/101	[25 : 14] : target initial address of psums in PE [13 : 02] : # of value to be read/write from DRAM to local PE SRAM #1	Read/Write partial summation from the DRAM to local PE SRAM #1 (16-bit each value)
	010/110	[25 : 14] : target initial address of psums in PE [13 : 02] : # of value to be read/write from DRAM to local PE SRAM #2	Read/Write partial summation from the DRAM to local PE SRAM #2 (16-bit each value)
	111	[25 : 14] : target initial address of input feature in central memory [13 : 02] : # of value to be read/write from DRAM to central memory	Read the input feature from the DRAM to central memory (64-bit each value)
	000 ~ 010	[1] : Write constant into local PE SRAM without accessing DRAM [0] : Indicate if the data read from DRAM is in “RLE” format	
	100 ~ 110	[1] : Write constant back to SRAM after read the data out [0] : Indicate if encode the data into “RLE” format before writing into DRAM	

For each convolutional layer, the processing in the proposed design starts from the instructions used to set the CNN network parameters such as the size of input feature row, weight row and psum row. Then, we divide the PE array into numerous groups by configuring the ports in each PE and set the operation mode of all PEs. For each processing pass, we write the corresponding weight filters of each PE into the local SRAM and initialize the psum register file. After all the PEs are loaded, we start to issue the instructions used to perform convolution operation in active PEs following by the weight exchange instructions if required. We write the psums stored in each PE back to the DRAM while the psum register file is full or the current pass completes. We repeat the same instruction set for each processing pass until the entire input feature is processed.

5.4. Performance Evaluation

5.4.1. Sensitivity to Network Density

To evaluate the influence of network density on performance of the proposed design, we randomly generate numerous input features for each weight size and sweep both density from 100% down to 10%. Then, we distribute the weight rows on multiple PEs as described in section III to perform the 1-D primitive. In dense format, the ideal number of cycles required for one 1-D primitive with one multiplier and one adder is fixed and can be calculated using (5.4),

$$N_c = \left(W_p \times W_w\right) + 1 \tag{5.4}$$

where Wp is the row width of psum and Ww is the row width of weight filter.

For each network density, we process the same set of input feature and weight under both cooperation and parallel mode with a stride of 1. The normalized cycle number vs various density for the 7x7, 5x5 and 3x3 filter is shown in Figure 5.15 - 5.17 respectively. The x-axis indicates the density of input feature and weight. For example, the 0.4/0.4 represents 40% of the input features and weights is none-zero value. We calculate the y-axis through dividing the cycle number for one 1-D primitive from simulations by the computation results from (5.4). Ideally, the minimal cycle number required for the 1-D primitive should be linear to the density. For instance, if the input feature and filter have a 40% density, the corresponding minimal cycle number is 16% of the ideal cycle number.

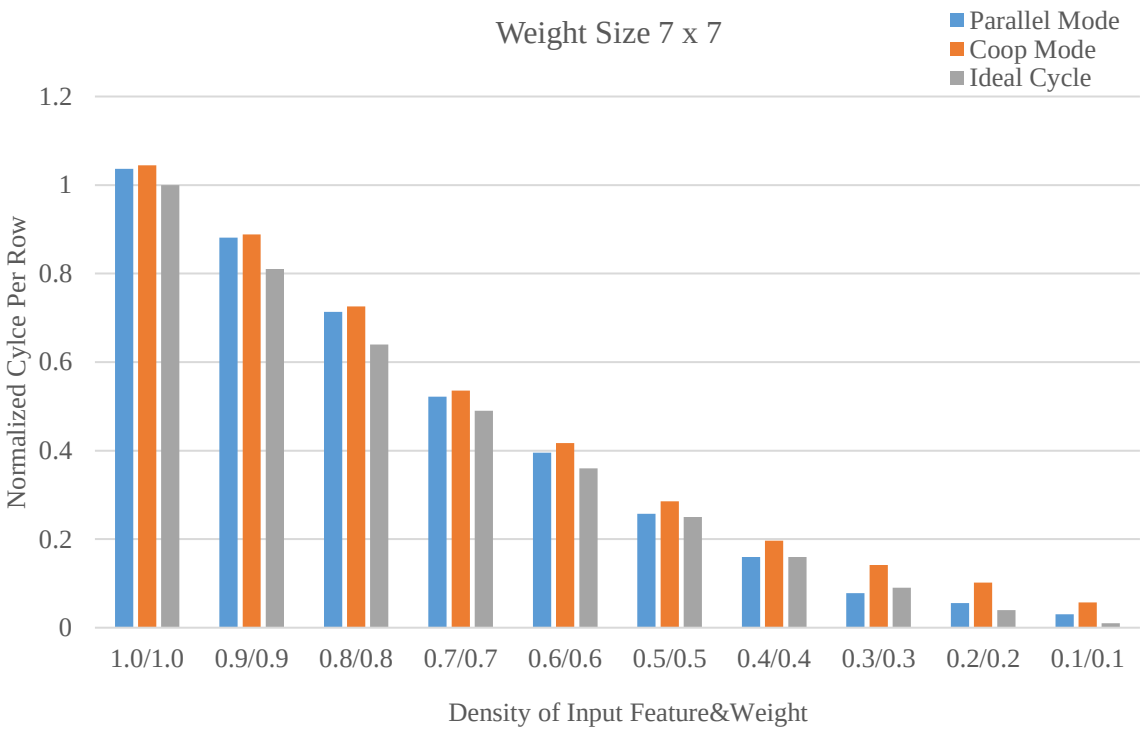

Figure 5.15 Normalized Cycler Per vs. Network Density of 7x7 Filter

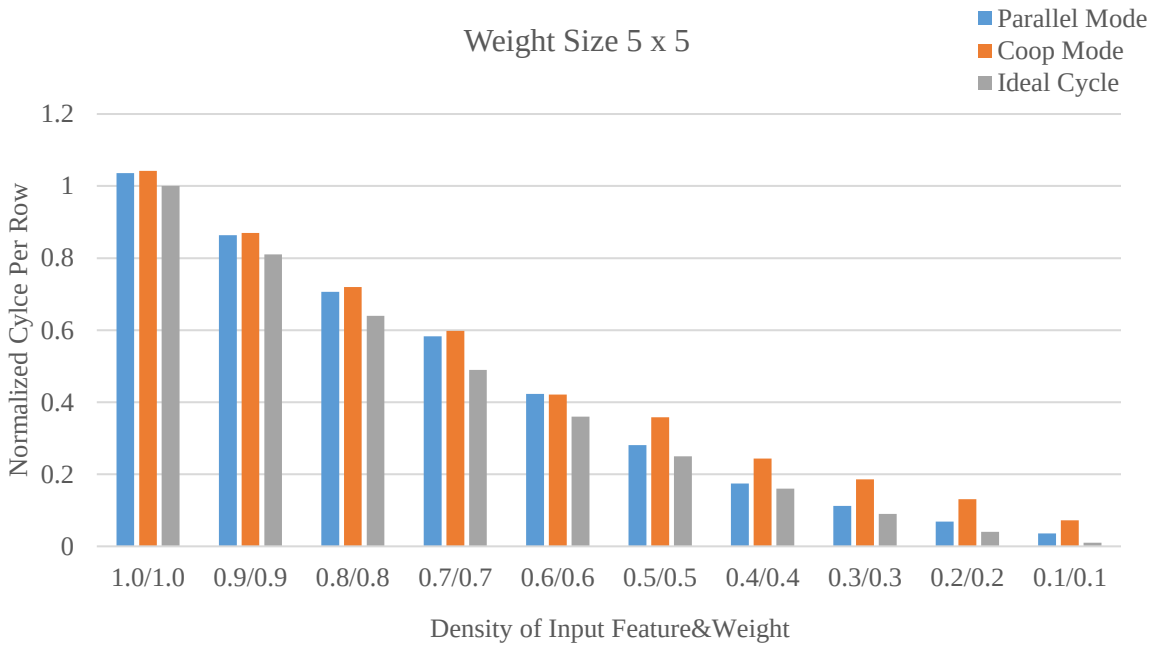

Figure 5.16 Normalized Cycler Per vs. Network Density of 5x5 Filter

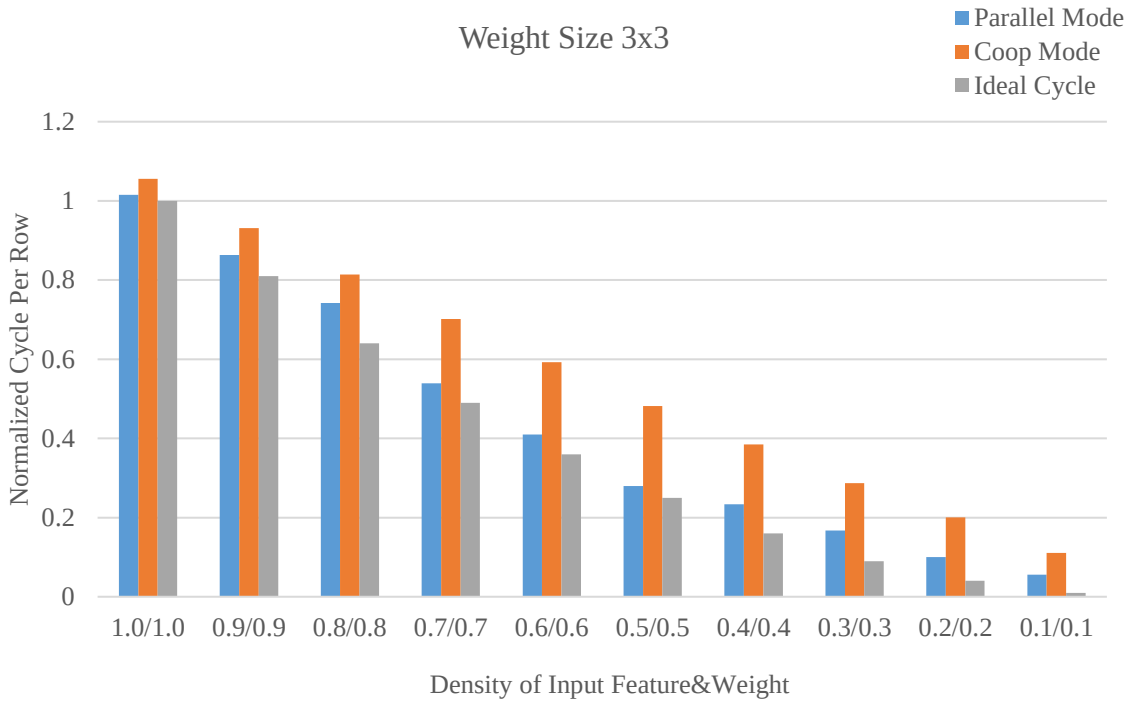

Figure 5.17 Normalized Cycler Per vs. Network Density of 3x3 Filter

From the simulation results above, we can make the following conclusion.

1) For the 100% dense network, the proposed design requires a 1%-5% increase compared to the ideal result in (5.4). This increase majorly comes from the overhead from the control follow and the initial latency of pipelined arithmetic units, which is unavoidable in all real-world implementations. As a result, we can conclude that the proposed design can also efficiently process the fully dense network. For sparse network, the proposed design can reduce cycle number for all filter size and the improvement is increased from 1.07x to 22.12x as the network become sparser.
2) In the proposed design, the parallel mode can achieve almost the minimal cycle number required for one 1-D primitive with all weight filter size until the density decreases to about 40%. This is because the number of weight rows filled with only zero increases when the sparsity is above 40%, especially in those small filters, like 3x3. Therefore, it increases the idle cycles in arithmetic units and deteriorate the workload balancing among PEs. Meanwhile, this situation gets worse as density decreases. Even though, the proposed design can achieve an average improvement from 1.34x to 22.12x across the entire density range.
3) The cooperation mode in the proposed design takes more cycles for each 1-D primitive compared to the equivalent processing under parallel mode. As described in section III, this is majorly due to the dynamic distribution of none-zero input features, which can result in an unbalanced workload distribution between the 2 execution lanes. Meanwhile, the weight rows filled with all zeros can cause more idle cycle under cooperation mode since both execution lanes process the same filter now. As a result, the performance difference between two operation modes gets larger while processing

the smaller filters and sparser network. For the 7x7 and 5x5 weight filter, the cooperation mode can achieve the similar improvement as the parallel one until the sparsity increases to 50%. The average improvement of the cooperation mode is from 1.07x to 17.54x.

The frame rate of CNN depends on two parts, the processing latency and memory access latency. Although the cooperation mode requests longer processing latency while targeting on the small filters and high sparsity network, it doubles the maximum weight channel number stored on-chip, which can improve the memory access latency and power. The same situation happens to the weight exchange as well, which requires extra processing latency but increases the number of stored channels. Taking the "CONV2-1" in VGG16 and "CONV3" in AlexNet as examples, the processing and memory access latency under both modes is summarized in Table 5.6. Since we cannot store all the channels of one filter in "CONV2-1" on-chip, we have to move the psum twice under parallel mode, which results in a 2.76MB additional DRAM accesses and 2.3ms latency. Therefore, the cooperation mode is more efficient for processing the "CONV2-1", even it requires a slightly longer processing latency. On the other hand, since we can store one complete output feature of the "CONV3"

Table 5.6 Performance Summary of Cooperation and Parallel Mode

Performance	CONV2-1		CONV3	
	Coop	Parallel	Coop	Parallel
Processing Latency (ms)	21.16	19.78	2.41	1.75
DRAM Latency (ms)	11.75	14.05	1.64	1.64
DRAM Accesses (MB)	9.4	12.16	1.31	1.31
Total Latency (ms)	32.91	33.83	4.05	3.39

Assuming a DRAM bandwidth of 1.2GB/s.

Above all, supporting multiple processing modes allows the proposed design to optimize the performance of each individual convolutional layer based on their shape and the density. More specifically, the cooperation mode can be more efficient when we do not have enough on-chip memory to store all the channels of one filter or one complete output feature within a single PE. Otherwise, the parallel mode can achieve a better performance without additional memory accesses, which is even close to the theoretic minimal latency.

5.4.2. Design Specification

The proposed design was implemented using the FUJITSU 55nm technology. The system specifications are summarized in Table 5.7. We apply the 16-bit fixed-point data system wide and achieve a clock frequency of 250MHz at 1.2V. The proposed design can support the shapes of all known states-of-arts CNNs and does not have limitation on maximum number of filters and maximum channel number per filter.

Table 5.7 Specification of the Proposed Design

Design Feature	Value
Technology	FUJITSU 55nm
Clock Frequency	200 ~ 250 MHz
Supply Voltage	1.2
Number of PEs	96
Global Buffer	20 KB
On-chip Memory per PE	Filter weight:288×19-bit Partial sums: 512×16-bit FIFO: 16×35bit
Total On-chip Memory	186.7 KB
Arithmetic Precision	16-bit fixed-point
Support CNN Shape	Filter size: 1-32 Stride: 1-32 Number of channels: N/A Number of filters: N/A
Total Synthesized Area	9.82 mm^2

The area breakdown of a single PE and the central processor is shown in Figure 5.18 and Figure 5.19 respectively. In the proposed design, the on-chip memory used to store the psums and weights accounts about 34% of the PE area, while the global buffer accounts for 82% of the area in central processor. Overall, the on-chip memory devices account for 36.4% of the total system area.

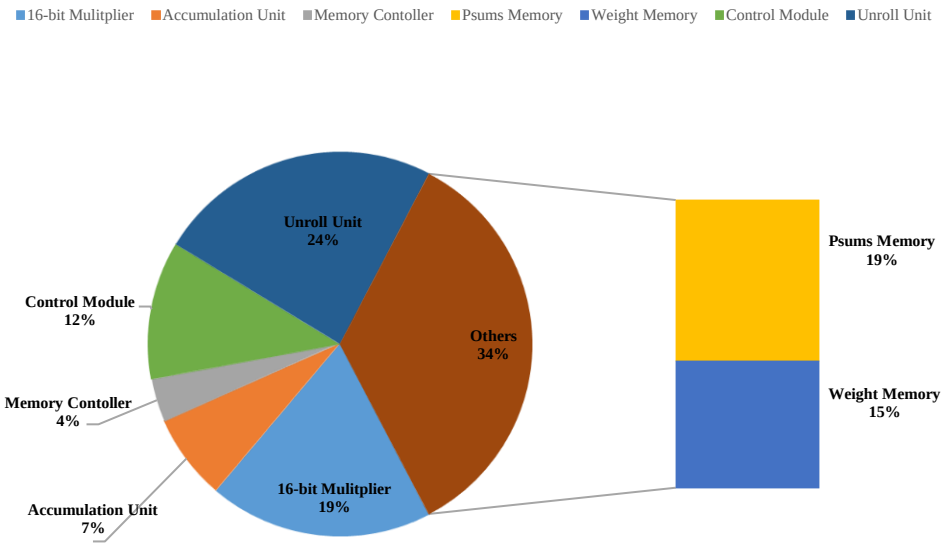

Figure 5.18 Area Breakdown of PE

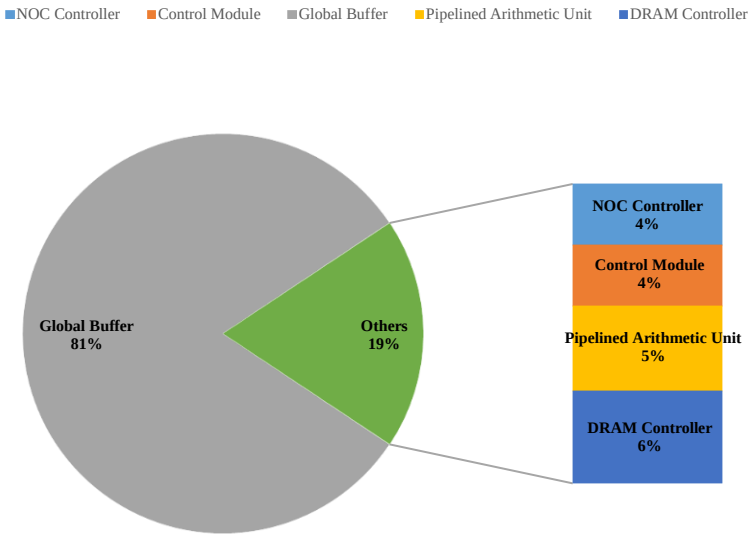

Figure 5.19 Area Breakdown of Central Processor

5.4.3. Benchmark Performance

To evaluate the performance of the proposed design, we pick two publicly available and widely used state-of-the-arts CNNs, VGG16 and AlexNet, as benchmarks, and then compare the performance with our baseline implementation, Eyeriss. Since the sparsity of input feature and weight can influence the performance of both implementations, we use the input features and weight filters those have the similar sparsity as the ones in Eyeriss as input data. Meanwhile, we limit the total size of on-chip memory in the proposed design close to that of Eyeriss as well to perform a fair comparison. Except the input features for the first layers of both networks, we encode the rest input features and all the output features into RLC format. Since there is rarely zero in the intermediate psums during processing, we read/write these psums from/to off-chip in the dense format.

The performance breakdown and processing mode of each convolutional layer in AlexNet is summarized in Table 5.8. We measure the processing latency with a 200MHz clock rate while estimate the memory access latency assuming an off-chip bandwidth of 1.2GB/s with a batch size of 3. On average, the proposed design can achieve a frame rate of 56.6frames/s with a total power of 1.65W. Compared to Eyeriss, the proposed design improves the total processing latency by 2.0x with the same clock frequency, which includes a 2.3x improvement for the sparse layers.

For the dense layer (CONV1), the processing latency of the proposed design is improved about 1.78x compared to Eyeriss. This is majorly because the proposed design includes 192 MAC units and can fully utilized them during processing while Eyeriss can only use 154 of 168 MAC units to process CONV1. For the sparse layers (CONV2-CONV5), the performance improvement majorly comes from skipping the computations with zero input features and weights.

Table 5.8 Performance Summary of Convolutional Layers in AlexNet

Layer	This Work						Eyeriss		Improvement
	Processing Latency Per Image (ms)	Power (W)	Memory Access (MB)	Operation Mode	Weight Exchange	Sparsity	Processing Latency Per Image (ms)	Sparsity	
CONV1	3.51	1.765	4.062	Coop	Y	0.01%	4.125	0.01%	1.18x
CONV2	5.64	1.686	4.855	Coop	Y	38.56%	9.80	38.7%	1.74x
CONV3	1.75	1.524	3.858	Parallel	N	71.11%	5.45	72.5%	3.11x
CONV4	1.08	1.508	2.518	Parallel	N	77.29%	4.00	79.3%	3.70x
CONV5	0.98	1.348	1.633	Parallel	Y	76.58%	2.50	77.6%	2.55x
Total	12.966	1.646	16.926	N/A		56.43%	25.875	57.53%	2.0x

The DRAM access per input in the proposed design is higher than that of Eyeriss due to two reasons: 1) the 2-bit offset and 1-bit termination flag increases the size of filters from 4.45MB to 5.93MB; 2) the size of input and output features per image is 1.07MB larger in this work after using same encoding method. For the first point, we can eliminate the DRAM access increase by encoding the original filters on fly using customized module while being read from the off-chip memory. Like encoding the output features into RLC format on fly, it should only require a minor overhead for each processing pass. Meanwhile, we can reduce the DRAM accesses to 13.58MB assuming the same input and output feature size as Eyeriss, which can increase the frame rate of the proposed design to 59.7frames/s.

We summarize the performance breakdown and operation settings of all the convolutional layers in VGG16 in Table 5.9. With a batch size of 3, the proposed design achieves a frame rate of 2.62frames/s with a power of 1.58W. Compared to Eyeriss, we can provide an improvement of 4.65x on the total processing latency with a 200MHz clock frequency.

In theory, it should request about 138016 and 151425 cycles to process the first layers in AlexNet and VGG16 respectively. In Eyeriss, however, it takes 43.0us and 197.9us to process one filter in these layers, which demonstrates that the shapes of filter can significantly decrease the performance for a given number of computations. In the proposed design, since the processing latency only depends on the total computation required for each layer, it takes 36.5us and 33.5us to process one filter in target layers, which provides a more robust performance against the filter shapes.

Table 5.9 Performance of Convolutional Layers in VGG16

Layer	This Work						Eyeriss		Improvement
	Processing Latency Per Image (ms)	Power (W)	Memory Access (MB)	Operation Mode	Weight Exchange	Sparsity	Processing Latency Per Image (ms)	Sparsity	
CONV1-1	2.15	1.94	14.64	Parallel	Y	1.6%	12.67	1.6%	5.89x
CONV1-2	28.62	1.65	42.66	Coop	Y	45.8%	270.2	47.7%	9.43x
CONV2-1	21.16	1.66	28.61	Coop	Y	21.3%	135.1	24.8%	6.38x
CONV2-2	40.41	1.66	56.96	Coop	Y	41.3%	270.3	38.7%	6.68x
CONV3-1	19.99	1.73	32.69	Coop	Y	38.0%	68.0	39.7%	3.40x
CONV3-2	30.13	1.61	51.58	Coop	Y	60.1%	130.0	58.1%	4.31x
CONV3-3	31.91	1.60	50.37	Coop	Y	57.3%	130.0	58.7%	4.07x
CONV4-1	15.54	1.44	20.21	Coop	Y	66.5%	35.0	64.3%	2.25x
CONV4-2	26.48	1.59	37.39	Coop	Y	71.2%	70.0	74.7%	2.64x
CONV4-3	30.58	1.44	35.81	Coop	Y	74.7%	70.0	85.4%	2.28x
CONV5-1	7.00	1.40	10.74	Coop	Y	80.1%	16.1	79.4%	2.30x
CONV5-2	7.77	1.33	10.85	Coop	Y	81.6%	16.2	87.4%	2.08x
CONV5-3	7.38	1.20	10.27	Coop	Y	87.3%	16.2	88.5%	2.20x
Total	269.12	1.58	402.78	N/A		57.4%	1251.67	58.6%	4.65x

5.5. Conclusion

The main contribution of this work is the development of a custom ASIP of CNN that can efficiently skip the computations with zero input features and weights in 1-D primitive and support a novel dataflow to explore parallelism and minimize DRAM memory access. We benchmark two most popular state-of-the-arts CNNs on the proposed accelerator for performance evaluation. The simulation results show that the proposed design can achieve a frame rate of 56.6fps and 2.62fps with a power of 1.65W and 1.58W respectively at 200MHz. Compared to the Eyeriss, one state-of-the-art accelerator, the proposed design provides a 2.0x and 4.65x improvement of processing latency.

CHAPTER 6

CONCLUTION AND FUTURE WORK

6.1. Conclusion

As the complexity of potential application areas increase rapidly, the network size of ANNs grows significantly, which makes a higher requirement for the hardware platforms used to perform these networks. Though the conventional hardware platforms such as the cloud sever and GPU can provide competitive performance, the nature of large silicon area and high power makes it hard to utilize these platforms in certain scenarios, such as mobile devices. On the other hand, the custom design hardware can improve the performance of neural networks by optimizing critical operations, exploring parallelism and taking advantage of unique network character with lower area and power cost.

In this work, we pick HTM and CNN, the most representative algorithm in CLA and MLA, as our target and propose an ASIC and ASIP accelerator respectively to explore the performance benefits from custom hardware. By applying multiple optimization methods in the custom designs, the proposed design of HTM achieves a 12.45x and 137x improvement in processing latency and power compared to the equivalent CUDA-based implementation on NVIDIA Tesla K40c. For the CNN, the performance of the proposed design is evaluated using AlexNet and VGG16. Compared to one of the most successful state-of-the-art custom accelerators, we can provide a 2.0x and 4.65x improvement in processing latency of convolutional layers by skipping the computations with zero input features and weighs.

6.2. Future Work

For HTM, the network size mapped on a hardware platform is limited by the memory size. In current version, we store all the network parameters in the on-chip SRAM, which can be area hungry for larger network. However, since only a small portion of network parameters is required during each phase of entire processing, it provides us an opportunity to replace these SRAMs with one or several off-chip DRAMs. On the other hand, as the 3DIC technology and the processor-in-memory (PIM) advance rapidly, the performance of our proposed design can be further improved cooperating with these concepts.

For CNN, the proposed can efficiently eliminate the computations with zero input features and weights. However, we cannot fully avoid the idles cycles resulted from the input feature rows and weight rows filling with zero value, which can be solved by including more information in the encoded data. On the other hand, there is a potential opportunity to leverage the proposed design to skip the unnecessary computation in the fully-connected layers that has even larger percentage of zero input feature and weights.

REFERENCES

[1] Agatonovic-Kustrin S, Beresford R. "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research," *J Pharm Biomed Anal*. 2000, pp.717-727.

[2] Numenta, "Machine Intelligence Technology: Find Real-Time Anomalies in your Streaming Data," Jun 2016. [Online]. Available: https://www.numenta.com/machine-intelligence-technology/htm-studio/. [Accessed: May 2018].

[3] C.Wang, Z. Zhao, L. Gong, L. Zhu, Z. Liu and X. Cheng, "A Distributed Anomaly Detection System for In-Vehicle Network Using HTM, " in *IEEE Access*, vol. 6, pp. 9091-9098, 2018.

[4] Tavish Srivastava, "An Alternative to Deep Learning? Guide to Hierarchical Temporal Memory (HTM) for Unsupervised Learning," May 2018. [Online]. Available: https://www.analyticsvidhya.com/blog/2018/05/alternative-deep-learning-hierarchical-temporal-memory-htm-unsupervised-learning/. [Accessed: Jan 2019].

[5] J. V. Doremalen and L. Boves, "Spoken Digit Recognition using a Hierarchical Temporal Memory, " *9th Annual Conference of the International Speech Communication Association (INTERSPEECH)*, Brisbane, Australia, September 22-26, 2008

[6] W.Melis and M.Kameyama, "A study of the different uses of colour channels for trafficsign recognition on hierarchical temporal memory," *Fourth International Conferenceon Innovative Computing, Information and Control (ICICIC)*, pp. 111–114, 2009.

[7] P. P. Afeefa and P. P. Thulasidharan, "Automatic License Plate Recognition using HTM cortical learning algorithm," *2017 International Conference on Intelligent Computing and Control (I2C2)*, Coimbatore, 2017, pp. 1-4.

[8] Kostavelis, I., Nalpantidis, L. and Gasteratos, A., "Object recognition using saliency maps and HTM learning," *Imaging Systems and Techniques (IST), 2012 IEEE International Conference*, vol., no., pp.528-532.

[9] S. S. Farfade, M. J. Saberian, L. J. Li, "Multi-view face detection using deep convolutional neural networks", *ACM Int. Conf. Multimedia Retrieval*, pp. 643-650, 2015.

[10] W. Rawat, Z. Wang, "Deep convolutional neural networks for image classification: a comprehensive review," *Neural Comput.*, vol.29, pp.2352–2449, 2017.

[11] Y. Zhang, W. Chan and N. Jaitly, "Very deep convolutional networks for end-to-end speech recognition," *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, New Orleans, LA, 2017, pp. 4845-4849.

[12] P. Huang, X. He, J. Gao, L. Deng, "Learning deep structured semantic models for web search using clickthrough data", *International Conference on Information and Knowledge Management*, 2013.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no.7540, pp 529–533, 2015.

[14] Y. LeCun, et al., "Deep Learning," *Nature*, vol. 512, no. 7553, pp. 436-444, 2015.

[15] K. He, X. Zhang, S. Ren, J. Sun, "Deep residual learning for image recognition", *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 770-778, Jun. 2016.

[16] A. Krizhevsky, I. Sutskever, G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks", *Neural Information Processing Systems*, pp. 1097-1105, 2012.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, pp. 1–14, Sep. 2014.

[18] Y. Chen, T. Krishna, J. S. Emer and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127-138, Jan. 2017.

[19] S. Han, J. Pool, J. Tran, W. J. Dally, "Learning both weights and connections for efficient neural networks", *Proceedings of Advances in Neural Information Processing Systems*, 2015.

[20] J. Hawkins, S. Ahmad and D. Dubinsky, "Hierarchical Temporal Memory including HTM Cortical Learning Algorithms," Numenta, December 2011.

[21] J. Hawkins and S. Blakeslee, "On Intelligence," Henry Holt, New York, 2004.

[22] R. W. Price, "Hierarchical temporal memory cortical learning algorithm for pattern recognition on multi-core architectures," Thesis of Portland State University, 2011.

[23] L. Streat, D. Kudithipudi, K. Gomez, "Non-volatile hierarchical temporal memory: Hardware for spatial pooling," *arXiv preprint arXiv:1611.02792*, 2016.

[24] A. M. Zyarah, D. Kudithipudi, "Reconfigurable hardware architecture of the spatial pooler for hierarchical temporal memory," *28th IEEE International System-on-Chip Conference (SOCC)*, Beijing, 2015, pp. 143-153.009.

[25] W. Li and P. Franzon, "Hardware implementation of Hierarchical Temporal Memory Algorithm," 2*9th IEEE International System-on-Chip Conference (SOCC)*, Seattle, WA, 2016, pp. 133-138.

[26] A. M. Zyarah and D. Kudithipudi, "Neuromorphic Architecture for the Hierarchical Temporal Memory," in *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 3, no. 1, pp. 4-14, Feb. 2019.

[27] Numenta, "HTM Forum: HTM Cheat Sheet," Jun 2016. [Online]. Avaliable: https:// discourse.numenta.org/t/htm-cheat-sheet/828. [Accessed: Aug 2018].

[28] A. P. James, I. Fedorova, T. Ibrayev and D. Kudithipudi, "HTM Spatial Pooler With Memristor Crossbar Circuits for Sparse Biometric Recognition," in *IEEE Transactions on Biomedical Circuits and Systems*, vol. 11, no. 3, pp. 640-651, June 2017.

[29] Krestinskaya, T. Ibrayev and A. P. James, "Hierarchical Temporal Memory Features with Memristor Logic Circuits for Pattern Recognition," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 6, pp. 1143-1156, June 2018.

[30] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," *2016 ACM/IEEE*

43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 1-13.

[31] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, O. Temam, "Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," *ASPLOS*, 2014.

[32] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, 2016, pp. 1-12.

[33] J. Song *et al.*, "7.1 An 11.5TOPS/W 1024-MAC Butterfly Structure Dual-Core Sparsity-Aware Neural Processing Unit in 8nm Flagship Mobile SoC," *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, San Francisco, CA, USA, 2019, pp. 130-132.

[34] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, Toronto, ON, 2017, pp. 27-40.

[35] D. Kim, J. Ahn and S. Yoo, "ZeNA: Zero-Aware Neural Network Accelerator," in *IEEE Design & Test*, vol. 35, no. 1, pp. 39-46, Feb. 2018.

[36] G. Desoli *et al.*, "14.1 A 2.9TOPS/W deep convolutional neural network SoC in FD-SOI 28nm for intelligent embedded systems," *2017 IEEE International Solid- State Circuits Conference - (ISSCC)*, San Francisco, CA, USA, 2017, pp. 238-239.

[37] B. Moons, R. Uytterhoeven, W. Dehaene and M. Verhelst, "Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI," *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, CA, 2017, pp. 246-247.

[38] A. Stillmaker, B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration the VLSI Journal*, vol. 58, pp. 74-81, 2017.

[39] J. T. Butler and T. Sasao, "Fast Hardware Computation of x Mod z," *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, Shanghai, 2011, pp. 294-297.