December 2018

Efficient machine learning: models and accelerations

Zhe Li
Syracuse University

ABSTRACT

One of the key enablers of the recent unprecedented success of machine learning is the adoption of very large models with millions of parameters (i.e., weights). The larger-scale model tend to enable the extraction of more complex high-level features, and therefore, lead to a significant improvement of the overall accuracy. On the other side, the layered deep structure and large model sizes also demand to increase computational capability and memory requirements. In order to achieve higher scalability, performance, and energy efficiency for deep learning systems, two orthogonal research and development trends have attracted enormous interests. The first trend is the acceleration while the second is the model compression. The underlying goal of these two trends is the high quality of the models to provides accurate predictions. In this thesis, we address these two problems and utilize different computing paradigms to solve real-life deep learning problems.

To explore in these two domains, this thesis first presents the cogent confabulation network for sentence completion problem. We use Chinese language as a case study to describe our exploration of the cogent confabulation based text recognition models. The exploration and optimization of the cogent confabulation based models have been conducted through various comparisons. The optimized network offered a better accuracy performance for the sentence completion. To accelerate the sentence completion problem in a multi-processing system, we propose a parallel framework for the confabulation recall algorithm. The parallel implementation reduces runtime, improves the recall accuracy by breaking the fixed evaluation order and introducing more generalization, and

maintains a balanced progress in status update among all neurons. A lexicon scheduling algorithm is presented to further improve the model performance.

As deep neural networks have been proven effective to solve many real-life applications, and they are deployed on low-power devices, we then investigated the acceleration for the neural network inference using a hardware friendly computing paradigm, stochastic computing. It is an approximate computing paradigm which requires small hardware footprint and achieves high energy efficiency. Applying this stochastic computing to deep convolutional neural networks, we design the functional hardware blocks and optimize them jointly to minimize the accuracy loss due to the approximation. The synthesis results show that the proposed design achieves the remarkable low hardware cost and power/energy consumption.

Modern neural networks usually imply a huge amount of parameters which can not be fit into embedded devices. Compression of the deep learning models together with acceleration attracts our attention. We introduce the structured matrices based neural network to address this problem. Circulant matrix is one of the structured matrices, where a matrix can be represented using a single vector, so that the matrix is compressed. We further investigate a more flexible structure based on circulant matrix, called block-circulant matrix. It partitions a matrix into several smaller blocks and makes each submatrix to be circulant. The compression ratio is controllable. With the help of Fourier Transform based equivalent computation, the inference of the deep neural network can be accelerated with high energy efficiency on the FPGAs. We also offer the optimization for the training algorithm for block circulant matrices based neural networks to obtain a high accuracy after compression.

EFFICIENT MACHINE LEARNING: MODELS AND ACCELERATIONS

by

Zhe Li

B.S., Beijing University of Posts and Telecommunications, 2012

M.S., Syracuse University, 2014

Dissertation

Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Electrical and Computer Engineering

Syracuse University
December 2018

ACKNOWLEDGEMENTS

During my graduate career, I received help from many people, without whom this dissertation could not have been finished.

First of all, I would like to express my most sincere gratitude to my advisor, Dr. Qinru Qiu, who recognized me and offered me the chance of doctorate study on December, 2012 when I hesitated whether to move a step forward in the academia and needed a guidance. She provided comprehensive support all the way through the completion of this degree. Secondly, I am also thankful to Dr. Roger Chen, who recognized me and recommended me to seek the doctorate degree when I was a master student in his class. He offered me such opportunity to re-consider my career path. My sincere gratitude also goes to my close collaborator, Dr. Yanzhi Wang, without whom I could not have achieved so much. I learned a lot from his advice and enjoyed the journey we went on together. I would like also to show my appreciation to my committee members, Dr. Sucheta Soundarajan, Dr. Mustafa C. , Dr. Lixin Shen for their valuable feedbacks. I would like to extend my deepest thanks to all my collaborators, Dr. Bo Yuan, Dr. Ji Li, Dr. Jian Tang, Dr. Xue Lin, Siyu Liao, Caiwen Ding, Ao Ren, Ruizhe Cai, Shuo Wang. I would like to thank all my labmates: Dr. Qiuwen Chen, Dr. Khadeer Ahmed, Wei Liu, Jianwei Cui, Amar Shrestha, Yilan Li, Haowen Fang and Ziyi Zhao. Your support, company and friendship are the treasures during my years in Syracuse.

Finally, I would like to express my everlasting appreciation to my family, who accompanied, encouraged, and supported me in this difficult journey. Without their trusts and constant love, I would never be able to achieve this much. This thesis is dedicated to my wife Lei Zhang, my baby girl Vivian Li, and my parents, Shengzhang Li and Honglei Zou.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

From the end of the first decade of the 21st century, neural networks have been experiencing a phenomenal resurgence thanks to the big data and the significant advances in processing speeds. Large-scale deep neural networks (DNNs) have been able to deliver impressive results in many challenging problems. For instance, DNNs have led to breakthroughs in object recognition accuracy on the ImageNet dataset [26], even achieving human-level performance for face recognition [117]. Such promising results triggered the revolution of several traditional and emerging real-world applications, such as self-driving systems [51], automatic machine translations [22], drug discovery and toxicology [13]. As a result, both academia and industry show the rising interests with significant resources devoted to investigation, improvement, and promotion of deep learning methods and systems.

In this chapter, we discuss the motivation of the study. Then we introduce the basics of three computing paradigms for machine learning models especially deep learning models. Applications of different learning systems are introduced. Finally, the contributions of the thesis are reviewed.

1.1 Motivation

Machine learning technology benefits many aspects of modern life: web searches, e-commerce recommendations, social network content filtering, *etc.* [67]. Unfortunately, the conventional machine learning techniques were restricted due to the lack of ability to automatically extract high-level features.

These features traditionally have been extracted by well-engineered manual feature extractors. *Deep learning* methods have taken advantage of the architecture of multi-level representations to learn very complex functions [67]. Here, each representation is obtained from a slightly less abstract level through the transformation based on a simple non-linear module. Deep learning significantly enhances the machine learning capability using learning from data by these multiple layers for different features without human involvement.

Due to the deep structure, the performance of deep learning model highly relies on the capability of hardware resources. From high performance server clusters [25, 14] to *General-Purpose Graphics Processing Units (GPGPUs)* [54, 9], parallel accelerations of *deep neural networks (DNNs)* are widely used in both the academic and industry. Recently, hardware acceleration for DNNs has attracted enormous research interests on *Field-Programmable Gate Arrays (FPGAs)* [138, 89, 93]. Nevertheless, there is a trend of embedding DNNs into lightweight embedded and portable systems, such as surveillance monitoring systems [52], self-driving systems [51], unmanned aerial systems [87], and robotic systems [62]. These scenarios require very low power & energy consumptions and small hardware footprints. Besides, cell phones [67] and wearable devices [41] equipped with hardware-level neural network computation capability require the radical reduction in power & energy consumptions and footprints.

This thesis mainly addresses the inference acceleration and model compression for modern machine learning models. The acceleration also implies an energy-efficient system that enables the models to run on low-power devices. Meanwhile, effective modeling and training optimization for specific applications are proposed to maintain a good accuracy. We investigate three com-

puting paradigms to explore the efficient machine learning model acceleration. The first one, Cogent Confabulation is an application oriented acceleration research, which is introduced in Chapter 2 and Chapter 3. The second computing paradigm in this thesis is Stochastic Computing, which is more general to neural networks, is described in Chapter 4. The last computing paradigm is the structured matrices based neural network acceleration and model compression, which is presented in Chapter 5.

1.2 Cogent Confabulation

Inspired by human cognitive process, cogent confabulation [47] mimics human information processing including Hebbian learning, correlation of conceptual symbols and recall action of brain. Based on the theory, the cognitive information process consists of two steps: learning and recall. The confabulation model represents the observation using a set of features. These features construct the basic dimensions that describe the world of applications. Different observed attributes of a feature are referred as symbols. The set of symbols used to describe the same feature forms a lexicon and the symbols in a lexicon are exclusive to each other. In learning process, matrices storing posterior probabilities between neurons of two features are captured and referred as the *knowledge links (KL)*.A KL stores weighted directed edges from symbols in source lexicon to symbols in target lexicon. The $(i, j)^{th}$ entry of a KL, quantified as the conditional probability $P(s_i|t_j)$, represents the Hebbian plasticity of the synapse between i^{th} symbol in source lexicon s and j^{th} symbol in target lexicon t. The knowledge links are constructed during learning process by extracting and associating features from the inputs and collection of all knowledge links in the model forms its *knowl-*

edge base (KB). During recall, the input is a noisy observation of the target. In this observation, certain features are observed with great ambiguity, therefore multiple symbols are assigned to the corresponding lexicons. The goal of the recall process is to resolve the ambiguity and select the set of symbols for maximum likelihood using the statistical information obtained during the learning process. This is achieved using a procedure similar to the integrate-and-fire mechanism in biological neural system. Each neuron in a target lexicon receives an excitation from neurons of other lexicons through KLs, which is the weighted sum of its incoming excitatory synapses. Among neurons in the same lexicon, those that are least excited will be suppressed and the rest will fire and become excitatory input of other neurons. Their firing strengths are normalized and proportional to their excitation levels. As neurons gradually being suppressed, eventually only the neuron that has the highest excitation remains firing in each lexicon and the ambiguity is thus resolved. Let l denote a lexicon, F_l denote the set of lexicons that have knowledge links going into lexicon l, and S_l denote the set of symbols that belong to lexicon l. The excitation of a symbol t in lexicon l is calculated by summing up all incoming knowledge links:

$$el(t) = \sum_{k \in F_l} \{ \sum_{s \in S_k} [el(s) \ln(\frac{P(s|t)}{p_0})] + B \}, t \in S_l \tag{1.1}$$

where the function $el(s)$ is the excitation level of the source symbol s. The parameter p_0 is the smallest meaningful value of $P(s_i|t_j)$. The parameter B is a positive global constant called the bandgap. The purpose of introducing B in the function is to ensure that a symbol receiving N active knowledge links will always have a higher excitation level than a symbol receiving $(N-1)$ active knowledge links, regardless of their strength. As we can see, the excitation level of a symbol is actually its log-likelihood given the observed attributes in other lexicons.

1.3 Stochastic Computing

Deviated from the conventional binary computing (referred as *conventional computing*), *stochastic computing (SC)* represents any number using a stream of bits. Here the value of real number x in the unit interval is interpreted by the ratio of bit-1 in the entire bit-stream, i.e., $P(X = 1)$. For instance, the 8-bit sequence 00100101 containing three 1s denotes $x = P(X = 1) = \frac{3}{8} = 0.375$. Since each bit has the same weight, number representation in stochastic computing is unary and hence enables different interpretations for the same value. Besides this unipolar coding format [34], bipolar coding format [34] is another popular number representation scheme in stochastic computing. In the scenario of bipolar coding, the relationship between x and $P(X = 1)$ becomes $P(X = 1) = \frac{x+1}{2}$, which enables the stochastic representation for negative number. Notice that for either unipolar or bipolar coding format, the represented number ranges in $[0, 1]$ or $[-1, 1]$. To represent a number beyond this range, a pre-scaling operation [135] or integer bit-stream based representation [6] can be used to relax this constraint.

A major advantage of stochastic computing is its ultra-low hardware cost: Many complicated arithmetic functions can now be implemented with very simple logic circuits. For instance, as shown in Figure 1.1, the real multiplication can be performed with an **AND** gate in the unipolar coding form since

$$\begin{aligned} c &= P(C = 1) \\ &= P(A = 1)P(B = 1) \\ &= ab \end{aligned}$$

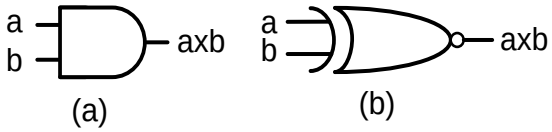


Figure 1.1: Stochastic multiplication using: (a) unipolar encoding (b) bipolar encoding.

or with an **XNOR** gate in bipolar coding form since

$$\begin{aligned} c &= 2P(C=1)-1 \\ &= 2[P(A=1)P(B=1)+P(A=0)P(B=0)]-1 \\ &= 2[P(A=1)P(B=1)+(1-P(A=1))(1-P(B=1))]-1 \\ &= (2P(A=1)-1)(2P(B=1)-1) \\ &= ab. \end{aligned}$$

Another example is regarding the adder, which can be simply implemented with a multiplexer (see Figure 1.2) in the scenario of stochastic computing, for

$$\begin{aligned} c &= P(C=1) \\ &= \frac{1}{2}(P(A=1)+\frac{1}{2}P(B=1) \\ &= \frac{1}{2}(a+b). \end{aligned}$$

Additionally, the addition in the bipolar form uses this multiplexer as well, since

$$\begin{aligned} c &= 2P(C=1)-1 \\ &= 2[\frac{1}{2}(P(A=1)+\frac{1}{2}P(B=1)]-1 \\ &= \frac{1}{2}[2P(A=1)-1)+(2P(B=1)-1)] \\ &= \frac{1}{2}(a+b). \end{aligned}$$

In general, such significant saving in hardware resource makes stochastic computing circuits well-suited for the area-constrained applications, such as

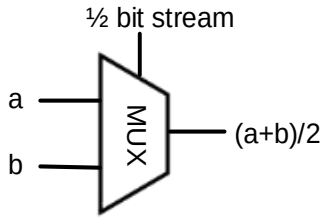


Figure 1.2: Scaled addition in stochastic computing.

signal sensing and processing in wearable devices. Besides, the abundant budget on area offers immense design space in optimizing hardware performance in terms of power, latency and speed via efficient trade-offs between area and those metrics, thereby implying the potential application of stochastic computing in large-scale systems that requires massive parallelism for basic computing units.

Another advantage of stochastic computing is its inherent error-resilience. By nature, the redundant representation of stochastic computing translates to the strong capability for tolerating transient error and soft error (bit-flipping) since each bit has the same weight in bit-stream. For instance, as reported in [97, 85, 134], compared to their conventional computing counterparts, the stochastic digital signal processing component shows much better error-resilience capability, which is attractive for the emerging noise-rich deep nanoscale CMOS era.

1.4 Structured Matrices based Neural Networks

In general, a circulant matrix $\mathbf{W} \in \mathbb{R}^{n\times n}$ [94] is defined by a vector $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ as the following:

$$\mathbf{W} = \begin{bmatrix} w_1 & w_n & \ldots & w_3 & w_2 \\ w_2 & w_1 & w_n & & w_3 \\ \vdots & w_2 & w_1 & \ddots & \vdots \\ w_{n-1} & & \ddots & \ddots & w_n \\ w_n & w_{n-1} & \ldots & w_2 & w_1 \end{bmatrix}. \tag{1.2}$$

From equation (1.2) it is seen that an n-by-n circulant matrix only has n parameters because of its strong structure. Clearly, when such structure is imposed to the weight matrices of DNNs, the required space cost for storing the weights is immediately reduced from $O(n^2)$ to $O(n)$.

Besides the advantage on low space cost, the use of circulant matrices as weight matrices can also lead to low computational complexity for both inference and training, which are described as below:

Inference: The dominating computation during the forward propagation in the inference is the matrix-vector multiplication ($\mathbf{Wx}$). According to [94], when $\mathbf{W}$ is a circulant matrix, $\mathbf{Wx}$ can be performed as below:

$$\mathbf{a} = \mathbf{Wx} = \text{IDFT}(\text{DFT}(\mathbf{w}) \circ \text{DFT}(\mathbf{x})), \tag{1.3}$$

where $\circ$ denotes the element-wise multiplication; DFT$(\cdot)$ denotes the Discrete Fourier transform; and IDFT$(\cdot)$ denotes the inverse Discrete Fourier transform. Notice that since the computational complexity of n-point DFT/IDFT is only $O(n \log n)$, the computational complexity of DNN inference can achieve order-of-magnitude reduction (from $O(n^2)$ to $O(n \log n)$).

Training: For backward propagation in the training, recall that its key procedure is to perform the chain rule-based calculation for the gradient of loss function L with respect to the weight vector $\mathbf{w}$ as below:

$$\frac{\partial L}{\partial \mathbf{w}} = \frac{\partial L}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{w}}, \tag{1.4}$$

where $\frac{\partial L}{\partial \mathbf{a}}$ is the gradient back-propagated from the subsequent layer. Notice that in the scenario that $\mathbf{W}$ is a square circulant matrix, as indicated in [18], $\frac{\partial \mathbf{a}}{\partial \mathbf{w}}$ is a circulant matrix defined by the vector $\mathbf{x}' = (x_1, x_n, x_{n-1}, \ldots, x_2)$. Therefore, according to [94], equation (1.4) can be simplified as below:

$$\frac{\partial L}{\partial \mathbf{w}} = \text{IDFT}(\text{DFT}(\frac{\partial L}{\partial \mathbf{a}}) \circ \text{DFT}(\mathbf{x}')), \tag{1.5}$$

where $\mathbf{1}$ is a column vector full of ones. In addition, the gradient of input $\mathbf{x}$ which is back-propagated to the previous layer, should be calculated as:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial \mathbf{x}}. \tag{1.6}$$

Notice that here $\frac{\partial \mathbf{a}}{\partial \mathbf{x}}$ is also a circulant matrix that is defined as $\mathbf{w}' = (w_1, w_n, w_{n-1}, \ldots, w_2)$. Hence equation (1.6) can also be simplified as below:

$$\frac{\partial L}{\partial \mathbf{x}} = \text{IDFT}(\text{DFT}(\frac{\partial L}{\partial \mathbf{a}}) \circ \text{DFT}(\mathbf{w}')). \tag{1.7}$$

From equation (1.5) and (1.7) it is seen that, when $\mathbf{W}$ is a circulant matrix, the updating scheme for the gradients of $\mathbf{w}$ and $\mathbf{x}$, as the key part of DNN training, can also be calculated using DFT/IDFT, thereby rendering order-of-magnitude reduction in computational cost for training (from $O(n^2)$ to $O(n \log n)$).

1.5 Applications

We applied the above three computing paradigms on three applications as below. To be specific, cogent confabulation is applied on the module for the *In-*

telligent Text Recognition System (ITRS); Stochastic Computing is applied on the *deep convolutional neural networks (DCNNs)* and structured matrices based neural network is used to train the *Recurrent Neural Networks (RNNs)* for *automatic speech recognition (ASR)*.

1.5.1 Intelligent Text Recognition System

In a recent development, the cogent confabulation model was used for sentence completion [47, 101]. Trained using a large amount of literature, the confabulation algorithm has demonstrated the capability of completing a sentence (given a few starting words) based on conditional probabilities among the words and phrases. We refer these algorithms as the "association" models. The brain inspired signal processing flow could be applied to many applications. A proof-of-concept prototype of context-aware Intelligence Text Recognition system (ITRS) is developed on high performance computing cluster [100]. As shown in Figure 1.3, the lower layer of the ITRS performs pattern matching of the input image using a simple non-linear auto-associative neural network model called Brain-State-in-a-Box (BSB) [4]. It matches the input image with the stored alphabet. A race model is introduced that gives fuzzy results of pattern matching. Multiple matching patterns will be found for one input character image, which is referred as ambiguity. The upper layer of the ITRS performs information association using the cogent confabulation model [47]. It enhances those BSB outputs that have strong correlations in the context of word and sentence and suppresses those BSB outputs that are weakly related. In this way, it selects characters that form the most meaningful words and sentences.

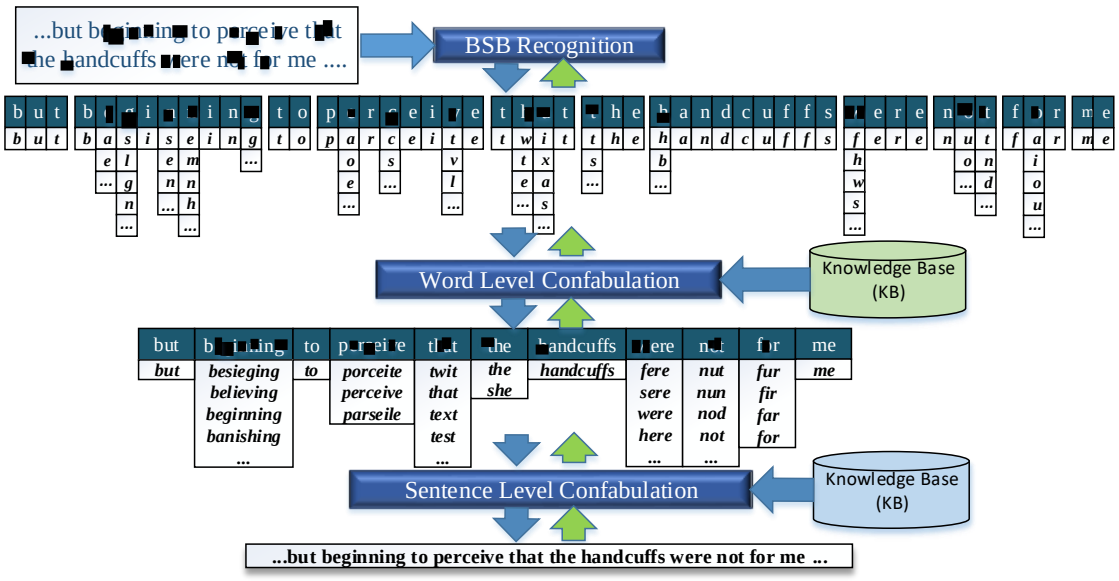


Figure 1.3: Overall architecture of the ITRS models and algorithmic flow.

1.5.2 Deep Convolutional Neural Network

Deep Convolutional Neural Networks (DCNN) are biologically inspired variants of *multi-layer perceptrons (MLPs)* by mimicking the animal visual mechanism [50]. Thus, a DCNN has special sets of neurons only connected to a small receptive field of its previous layer rather than fully connected. Besides an input layer and an output layer, a general DCNN architecture consists of a stack of *convolutional layers*, *pooling layers*, and *fully connected layers* shown in Figure 1.4. Please note that some special layers like normalization or regularization are not the focus in this thesis.

1) A convolutional layer is associated with a set of learnable filters (or kernels) [68], which are activated when specific types of features are found at some spatial positions in the inputs. Filter-sized moving windows are applied to the inputs to obtain a set of feature maps by calculating the convolution of the filter and inputs in the moving window. Each *convolutional neuron*, representing one

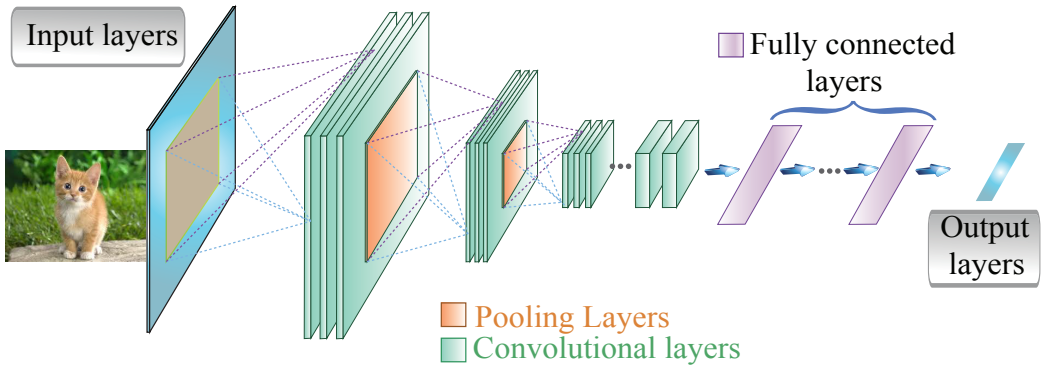


Figure 1.4: General DCNN architecture.

pixel in a feature map, takes a set of inputs and corresponding filter weights to calculate their inner-products.

2) After extracting features using convolution, a subsampling step can be applied to aggregate statistics of these features to reduce the dimensions of data and mitigate over-fitting issues. This subsampling operation is realized by a *pooling neuron* in pooling layers, where different non-linear functions can be applied, such as max pooling, average pooling, and L2-norm pooling. Among them, max pooling is the dominating type of pooling in state-of-the-art DCNNs due to the higher overall accuracy and convergence speed. The activation functions are non-linear transformation functions, such as Rectified Linear Units (ReLU) $f(x) = max(0, x)$, hyperbolic tangent (tanh) $f(x) = tanh(x)$ or $f(x) = |tanh(x)|$, and sigmoid function $f(x) = \frac{1}{1+e^{-x}}$. Among them, the ReLU function is the dominating type in the (large-scale) DCNNs due to *i)* the lower complexity for software implementation; and *ii)* the reduced vanishing gradient problem [37]. These non-linear transformations are conducted somewhere before the inputs of the next layer, ensuring that they are within the range of $[-1, 1]$. Usually, a combination of convolutional neurons, pooling neurons and activation functions forms a *feature extraction block (FEB)* to extract high-level

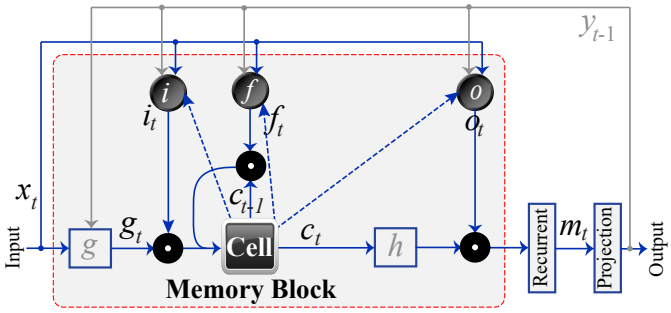


Figure 1.5: An LSTM based RNN architecture.

abstraction from the input images or previous low-level features.

3) A fully connected layer is a normal neural network layer with its inputs fully connected with its previous layer. Each *fully connected neuron* calculates the inner-product of its inputs and corresponding weights.

1.5.3 Recurrent Neural Networks based Automatic Speech Recognition

Long short-term memory (LSTM)

Modern large scale Automatic Speech Recognition (ASR) systems take advantage of LSTM-based RNNs as their acoustic models. An LSTM model consists of large matrices which is the most computational intensive part among all the steps of the ASR procedure. We focus on a representative LSTM model presented in [109] whose architecture is shown in Figure 1.5. An LSTM-based RNN accepts an input vector sequence $\mathbb{X} = (\mathbf{x}_1; \mathbf{x}_2; \mathbf{x}_3; ...; \mathbf{x}_T)$ (each of $\mathbf{x}_t$ is a vector corresponding to time t) with the output sequence from last step $\mathbb{Y}^{T-1} = (\mathbf{y}_0; \mathbf{y}_1; \mathbf{y}_2; ...; \mathbf{y}_{T-1})$ (each of $\mathbf{y}_t$ is a vector). It computes an output sequence

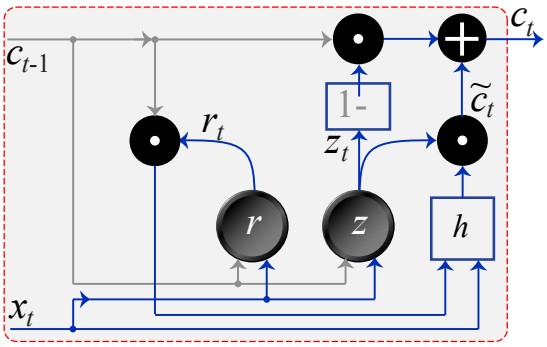


Figure 1.6: A GRU based RNN architecture.

$\mathbb{Y} = (\mathbf{y}_1; \mathbf{y}_2; \mathbf{y}_3; ...; \mathbf{y}_T)$ by using the following equations iteratively from $t = 1$ to T:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ir}\mathbf{y}_{t-1} + \mathbf{W}_{ic}\mathbf{c}_{t-1} + \mathbf{b}_i), \tag{1.8a}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fr}\mathbf{y}_{t-1} + \mathbf{W}_{fc}\mathbf{c}_{t-1} + \mathbf{b}_f), \tag{1.8b}$$

$$\mathbf{g}_t = \sigma(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{cr}\mathbf{y}_{t-1} + \mathbf{b}_c), \tag{1.8c}$$

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{g}_t \odot \mathbf{i}_t, \tag{1.8d}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{or}\mathbf{y}_{t-1} + \mathbf{W}_{oc}\mathbf{c}_t + \mathbf{b}_o), \tag{1.8e}$$

$$\mathbf{m}_t = \mathbf{o}_t \odot \mathbf{h}(\mathbf{c}_t), \tag{1.8f}$$

$$\mathbf{y}_t = \mathbf{W}_{ym}\mathbf{m}_t, \tag{1.8g}$$

where symbols $\mathbf{i}$, $\mathbf{f}$, $\mathbf{o}$, $\mathbf{c}$, $\mathbf{m}$, and $\mathbf{y}$ are respectively the input gate, forget gate, output gate, cell state, cell output, and projected output [109]; the $\odot$ operation denotes the point-wise multiplication, and the + operation denotes the point-wise addition. The $\mathbf{W}$ terms denote weight matrices (e.g. $\mathbf{W}_{ix}$ is the matrix of weights from the input vector $\mathbf{x}_t$ to the input gate), and the $\mathbf{b}$ terms denote bias vectors. Please note $\mathbf{W}_{ic}$, $\mathbf{W}_{fc}$, and $\mathbf{W}_{oc}$ are diagonal matrices for peephole connections [36], thus they are essentially a vector. As a result, the matrix-vector multiplication like $\mathbf{W}_{ic}\mathbf{c}_{t-1}$ can be calculated by the $\odot$ operation. σ is the logistic activation function and $\mathbf{h}$ is a user defined activation function. Here we use

hyperpolic tangent (tanh) activation function as **h**.

In the above equations, we have nine matrix-vector multiplications (excluding peephole connections which can be calculated by $\odot$). In one gate/cell, $\mathbf{W}_{*x}\mathbf{x}_t + \mathbf{W}_{*r}\mathbf{y}_{t-1}$ can be combined/fused in one matrix-vector multiplication by concatenating the matrix and vector as $\mathbf{W}_{*(xr)}[\mathbf{x}_t^T, \mathbf{y}_{t-1}^T]^T$. Furthermore, the four gate/cell matrices can also be concatenated and calculated through one matrix-vector multiplication as $\mathbf{W}_{(ifco)(xr)}[\mathbf{x}_t^T, \mathbf{y}_{t-1}^T]^T$. In this way, we can compute the above equations with only two matrix-vector multiplications, i.e. $\mathbf{W}_{(ifco)(xr)}$ $[\mathbf{x}_t^T, \mathbf{y}_{t-1}^T]^T$ and $\mathbf{W}_{ym}\mathbf{m}_t$.

Gated recurrent units (GRU)

The GRU is a variation of the LSTM as introduced in [20]. It combines the forget and input gates into a single "update gate". It also merges the cell state and hidden state, and makes some other changes. The architecture is shown in Figure 1.6. Similarly, it follows equations iteratively from $t = 1$ to T:

$$\mathbf{z}_t = \sigma(\mathbf{W}_{zx}\mathbf{x}_t + \mathbf{W}_{zc}\mathbf{c}_{t-1} + \mathbf{b}_z), \tag{1.9a}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_{rx}\mathbf{x}_t + \mathbf{W}_{rc}\mathbf{c}_{t-1} + \mathbf{b}_r), \tag{1.9b}$$

$$\tilde{\mathbf{c}}_t = \mathbf{h}(\mathbf{W}_{\tilde{c}x}\mathbf{x}_t + \mathbf{W}_{\tilde{c}c}(\mathbf{r}_t \odot \mathbf{c}_{t-1}) + \mathbf{b}_{\tilde{c}}), \tag{1.9c}$$

$$\mathbf{c}_t = (1 - \mathbf{z}_t) \odot \mathbf{c}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{c}}_t \tag{1.9d}$$

where symbols $\mathbf{z}$, $\mathbf{r}$, $\tilde{\mathbf{c}}$, $\mathbf{c}$ are respectively the update gate, reset gate, reset state, and cell state; the $\odot$ operation denotes the point-wise multiplication, and the + operation denotes the point-wise addition. The $\mathbf{W}$ terms denote weight matrices (e.g. $\mathbf{W}_{zx}$ is the matrix of weights from the input vector $\mathbf{x}_t$ to the reset gate). σ is the logistic activation function and $\mathbf{h}$ is a user defined activation function. Here

we use tanh activation function as $\mathbf{h}$. Note that a GRU has two gates (update and reset), while an LSTM has three gates (input, forget, output). GRUs do not have the output gate that is present in LSTMs. Instead, the cell state is taken as the output. The input and forget gates are coupled by an update gate $\mathbf{z}$, and the reset gate $\mathbf{r}$ is applied directly to the previous cell state.

In the above set of equations, we have six matrix-vector multiplications. In the reset and update gates, $\mathbf{W}_{*x}\mathbf{x}_t + \mathbf{W}_{*c}\mathbf{c}_{t-1}$ can be combined/fused in one matrix-vector multiplication by concatenating the matrix and vector as $\mathbf{W}_{*(xc)}[\mathbf{x}_t^T, \mathbf{c}_{t-1}^T]^T$. Furthermore, the reset and update gate matrices can also be concatenated and calculated through one matrix-vector multiplication as $\mathbf{W}_{(rz)(xc)}[\mathbf{x}_t^T, \mathbf{c}_{t-1}^T]^T$. In this way, we compute the above equations with three matrix-vector multiplications, i.e. $\mathbf{W}_{(rz)(xc)}[\mathbf{x}_t^T, \mathbf{c}_{t-1}^T]^T$, $\mathbf{W}_{\tilde{c}x}\mathbf{x}_t$, and $\mathbf{W}_{\tilde{c}c}(\mathbf{r}_t \odot \mathbf{c}_{t-1})$.

1.6 Contributions

This thesis studies the inference acceleration for modern machine learning models with high accuracy performance. We investigate three computing paradigms to explore the efficient machine learning model acceleration. The organization and contributions of this thesis are concluded as the following.

1. Cogenet confabulation based models on the text recognition system have been investigated and optimized in [77, 98, 99] to offer the state-of-the-art quality. In Chapter 2, we used Chinese language sentence completion problem as a case study to describe our exploration on the cogent confabulation based text recognition models. The exploration and optimization

of the cogent confabulation based models have been conducted through various comparisons.

2. In Chapter 3, we develop a multi-processing system for cogent confabulation models on sentence completion problems [78]. We propose a parallel framework for the confabulation recall algorithm. The parallel implementation reduced runtime, improve the recall accuracy by breaking the fixed evaluation order and introducing more generalization, and maintain a balanced progress in status update among all neurons. A lexicon scheduling algorithm was presented to further improve the model performance.

3. In Chapter 4, the Stochastic Computing (SC) based efficient inference framework for the deep convolutional neural networks (DCNNs) are designed [104, 79, 70]. We firstly describe how we apply the stochastic computing paradigm to DCNNs followed by a detailed partitioning of DCNN components [71, 136, 72]. The joint optimizations among the DCNN components are discussed from the perspective of SC [103, 80, 76, 75]. Finally we propose the hardware-level optimization on the complete SC based system. The synthesis results have shown that our proposed framework achieves remarkable low hardware cost and low power and energy consumption. The comparisons with latest peer works are provided.

4. In Chapter 5, we introduced the structured matrices based acceleration of the neural network. Inspired by previous work [18], we propose block-circulant matrices [140] based weight matrices formatting, where a weight matrix is partitioned into several blocks each of which is a circulant matrix. A circulant matrix can be represented using a single vector so that the matrix is compressed. In this way, the compression of the weight matrix is controllable. With the help of Fourier Transform based equivalent compu-

tation, the inference of the deep neural network can be accelerated energy efficiently on the FPGAs [124, 74]. We also offer the optimization for the training algorithm for block circulant matrices based neural networks to obtain a high accuracy after compression.

5. Finally in Chapter 6, the works contributing this thesis are reviewed and summarized. Potential directions to improve the studies are proposed.

CHAPTER 2

COGENT CONFABULATION ASSISTED CHINESE SENTENCE COMPLETION

2.1 Introduction

As an important part of text recognition, sentence completion and prediction, which stands for the capability of filling missing words in an incomplete sentence, has attracted much attention. The first step of sentence completion is syntactic parsing of the input text. Among different languages, Chinese is a great challenge due to its linguistically isolating. Each Chinese character generally corresponds to exactly one morpheme and multiple semantic meanings. Moreover, there has been a strong tendency in the Chinese language family over the last 2000 years for single morpheme words to develop into compounds of two or more morphemes [120], which makes Chinese language linguistically more flexible and complex. All of the above makes Chinese sentence completion extremely difficult.

In our previous research [102, 100, 101, 132], a cogent confabulation based sentence completion framework is developed. A sentence is represented by a set of lexicons corresponding to its words, word pairs, and part-of-speech tags. The conditional probability between neighboring lexicons are learned from training corpus. During recall, the missing information (including unknown word and part-of-speech tags for both unknown and given words) is selected that maximizes the likelihood of observed information (i.e. those words already given in the input sentence). Due to the difference between linguistic structures, this framework has to be modified for Chinese sentences. First of all, each Chinese

character, which is represented as a 3-byte UTF-8 code, is analogy to an English word. In the rest of the chapter, we use character and word interchangeably, as they are the same in Chinese. Secondly, the part-of-speech tagging of Chinese is usually associated with each multi-character compound. Correct segmentation is essential to syntactic parsing of the sentence.

We improve previous cogent confabulation model and apply it to Chinese sentence completion. Besides integrating *parts-of-speech (POS)* tagging that identifies the function of each word, in the Chinese sentence confabulation, segmentation label for multi-character compound is added, which identifies word compound consisting of $1 \sim 4$ Chinese characters. This work focuses on developing, optimizing and evaluating a confabulation model for Chinese sentence completion with high accuracy. It has three major contributions:

1. We extend the original sentence confabulation model to consider linguistic properties of Chinese language. Segmentation labels and beginning of sentence markers are specifically added to the model. *Knowledge links (KL)* are shared to reduce complexity and improve performance as well. Experiment results shows that the extended Chinese sentence confabulation model achieve 76.9% sentence recall accuracy with reduced memory and computing complexity.
2. We analyze the mutual information between source and target lexicons of each knowledge link in the confabulation model and assign weight to these knowledge links accordingly. Compared to the original model, the model with weighted knowledge link has 9% higher recall accuracy.
3. The mutual information of KLs is also exploited to find the best training set size, which gives the best trade-offs between training effort and recall

Table 2.1: Penn Treebank tag list.

Tag	Function	Examples
VA	Predicative Adjective	很(very), 雪白(snow white), ...
VC	Copula	(be), (not be), ...
VE	有 (have) as the main verb	有(have), 没有(have not), 无(not have), ...
VV	Other verb	想(want to), 走(walk), 喜欢(like), ...
NR	Proper Nouns (location, newspaper, ...)	北京(Beijing), 纽约时报(New York Times), ...
NT	Temporal Nouns	一月(Janurary), 汉朝(Han Dynasty), ...
NN	All other Nouns	书(book), 房子(house), ...
PN	Pronoun	我(I), 你(you), 这(this), ...
...	...	...

accuracy.

2.2 Formulation of Chinese Completion using Cogent Confabulation

2.2.1 Processing the training text

Our training text is segmented and tagged using Standford Part-of-speech (POS) tagger [39]. It is one of the most matured Natural Language Processing software based on probabilistic tagging systems. First, the Chinese training sentences are segmented using Stanford Chinese word Segmenter, which is based on a linear-chain *conditional random field (CRF)* model. The tool partitions sentence into compound words consisting of single or multiple Chinese characters. And then Stanford POS Tagger takes segmented sentence as input and assigns a part-of-speech tag to each compound. Stanford POS Tagger for Chinese Language exploits 33 word level Chinese tags specified by the Penn Treebank Tagging System [129]. Table. 2.1 lists some examples of these Tags. The information of POS tags and segments will be built into the knowledge

base during training. However, the POS tagger cannot be used to process sentences with missing words. Therefore during recall, we cannot use POS tagger for syntactic analysis. Our solution is to rely on the confabulation model to recall the segments and tags during the same time when the missing words are filled in. The basic idea is to assume that all tags and segment partitions are possible at the beginning, and gradually eliminate the ambiguity during the recall process. This approach is feasible since the number of tags and possible segment partitions is limited. Our experimental results show that considering tags and segmentations at the same time helps to improve the accuracy of sentence completion.

2.2.2 Chinese Sentence Confabulation

Basic confabulation framework

Inheriting from original sentence confabulation framework [101], we assume that the maximum length of a sentence is 20 words and sentence with more than 20 words will be truncated. We pad the sentence that has less than 20 words with special character [`] to represent the end of a sentence. Anything beyond the end of sentence will be ignored during training and recall.

Original Sentence confabulation framework has two levels of lexicons – word and word pair. Lexicons 0 to 19 correspond to single English word at location 0 to 19 in a sentence. Lexicons 20 to 38 correspond to 19 word pairs combining word from lexicon $0 \sim 19$ and its right adjacent neighbor. Each lexicon stores tremendous number of symbols (words or word pairs) that appears in the corresponding location.

In original framework, a KL is created between any two lexicons. In training process, we build all KL matrices to form knowledge base. And during recall, observed symbols will be set active in each lexicon. When there is no ambiguity in observation, only one symbol in a lexicon will be set active. Multiple symbols in the same lexicon will be set active because of ambiguous observation. They are referred as candidates. When a lexicon is not observable, all possible symbols will be set active to indicate the highest ambiguity. The excitation level of each candidate in the lexicon with ambiguity will be calculated and the symbols that is least excited will be suppressed. This procedure repeats until there is only one symbol left in each lexicon.

Chinese sentence confabulation model

Each Chinese character is encoded using 3 bytes of UTF-8 code. As mentioned before, we regard each Chinese character as a "word" and they occupy the word level lexicons in the confabulation framework. Modern Chinese language is based on word compound, which consists of $1 \sim 4$ single Chinese characters. These word compounds are not delimited, however, they can be found with the help of tools, such as the Stanford POS tagger. We label each Chinese character based on its position in a word compound, and refer this as segmentation label. For example, in a two character word compound 书籍 (book), 书 (book) is located at the first position of the two character word compound, therefore, it is marked as 1IN2, and 籍 (book) is marked as 2IN2. In this work, ten segmentation labels are used. They are: 1IN1, 1IN2, 2IN2, 1IN3, 2IN3, 3IN3, 1IN4, 2IN4, 3IN4, and 4IN4. Please note that segmentation label is only needed in Chinese sentence confabulation. This is a major difference between Chinese and western

languages. In the next section, we will show the necessity of including segmentation label in the confabulation model.

In the improved confabulation model, new lexicons are created for tags and segmentation labels. Moreover, instead of having lexicons for two adjacent words, we create lexicons for three adjacent words in order to adapt to semantic compounds of multiple Chinese characters. Therefore, lexicons in the new confabulation model can be divided into four levels: lexicons $0 \sim 19$ correspond to single Chinese word; lexicons $20 \sim 37$ correspond to Chinese word triplets; lexicons $38 \sim 57$ correspond to POS tags and lexicons $58 \sim 77$ correspond to segmentation labels.

The original sentence confabulation framework has a knowledge link between any two lexicons. Therefore, the size of knowledge base increases exponentially with the number of lexicons. In this way, $78 \times 77 = 6006$ KLs will be generated for the Chinese sentence confabulation model, which takes tremendous resources. To reduce the complexity of our computational model, two actions are jointly taken. First is to share KL matrix between lexicons that have the same relative position in sentence. For example, the distance from lexicon 0 to lexicon 1 is the same as the distance from lexicon 1 to lexicon 2, so the KLs between $0 \sim 1$ and $1 \sim 2$ are merged and shared.

The second action is to only create KLs between lexicons within N-neighborhood in the same lexicon level or across lexicon levels. In [88], experimental results show that considering words with low correlation in speech recognition making the performance poor. Empirically, Five-neighborhood is a best trade-off for accuracy and complexity. Therefore, we only generate knowledge links between two lexicons whose horizontal distance is within -5 to 5.

We refer to the new sentence confabulation model with these two changes as circular model as the knowledge links are circulated among lexicons.

Segmented and tagged training text is used during training. Characters, tags and segment labels are placed in corresponding lexicons. KLs are established not only between two lexicons in the same level, but also between lexicons in different levels, as long as their distance is less than 5. However, there is no KL between tag and segmentation label lexicons, because tags and segments are derivatives of the Chinese characters, and Stanford tools are not able to ensure 100% accuracy in tagging. Keeping KL between tag and segment lexicons will introduce noise in the confabulation procedure. A test sentence with missing characters will be given during recall. For those lexicons that are partially observable, a set of candidates that compliant with the partial observation is activated. If a lexicon is completely unobservable, then all possible symbols are activated as potential candidates. Since the test sentence originally is provided without tags and segmentation labels, the confabulation model automatically activates all tags and segmentation labels as possible candidates for each tag and segmentation label lexicon respectively.

2.3 Training and Recall Algorithm with Case Study

Given the confabulation model, the training and recall procedures are developed. The training process establishes knowledge base on tagged and segmented text. Taking following sentence “国 王#NN (The king) 有#VE (has) 两#CD个#M (two) 儿子#NN (sons)” as example, the corresponding tagged and segmented training text is as follows, and the confabulation model constructed

Figure 2.1: Lexion Structure of confabulation model.

based on the training text is given in Figure 2.1.

As shown in Figure 2.1, a special symbol [^] is assigned to the first lexicon in each level. Those words that frequently appear at the beginning of a sentence will have strong link with this special symbol. The indication of beginning of sentence is especially important for circular model, because its knowledge base only contains relative position information. The beginning of sentence symbol acts as anchors that provide absolute position information. We can also see from Figure 2.1 that the sentence is extended to 20 characters that are symbols assigned to lexicons 0 to 19 respectively. Those 20 characters will generate 18 three-word triplets and be assigned to lexicons 20 ~ 37, 20 tags and 20 segmentation labels will enter lexicons 38 ~ 57 and lexicons 58 ~ 77 respectively. At the end of training, the system will calculate the symbol to symbol conditional probability to fill in the KL matrix entry. For example, P("国"|"王") will be stored as an entry in the KL connecting lexicons 1 and 2, and P("国"|"NN") will be stored as an entry in the KL connecting lexicons 1 and 39.

During recall, sentences with missing characters will be given. Taking the same sentence in Figure 2.1 as example, Figure 2.2 gives a simple explanation

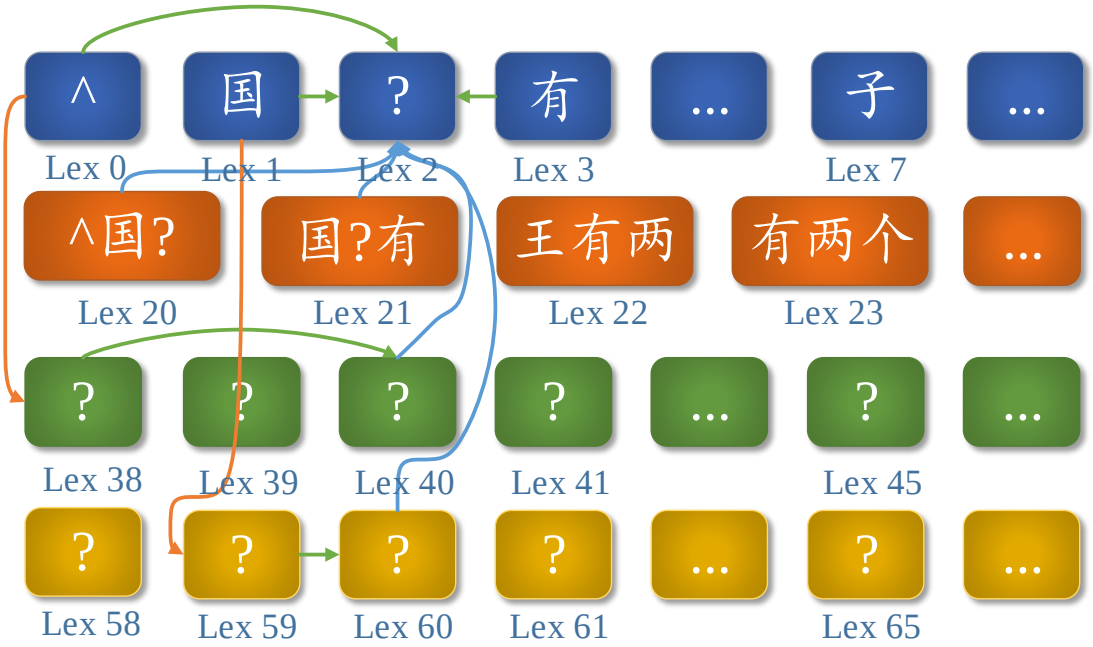


Figure 2.2: Lexion Structure of confabulation model (Any arrow is from source lexicon to target lexicon. Orange arrows represents Knowledge Links from observable lexicons to unobservable or partially observable lexicons; Green arrows represents Knowledge Links between lexicons in same level; Blue arrows represents Knowledge Links from unobservable or partially observable lexicons to observable lexicons).

how the model works. Assume that the third character "王(king)" is partially observable, and the ambiguous observation gives two candidates: "'王(king)" and "工(labor)". Symbols in lexicons are activated according to the observation. Hence lexicon 2 has two symbols "王(king)", "工(labor)" activated. And since no tags and segmentation labels are provided for the test sentence, all tags and segmentation labels are activated in tag and segmentation label lexicons. The lexicons with only one candidate are regarded as known lexicons and others are regarded as unknown lexicons. Through KLs, active symbols in source lexicons will excite candidate symbols in target lexicons. Each candidate's excitation level is calculated based on equation 1.1. The least excited one is eliminated from candidate list and others are set to be active. It is noted that

no matter a source lexicon is known or not, as long as its candidates are set to be active, the active symbols will always excite the symbols in unknown lexicons. In this example, [^] in lexicon 0 will excite tag candidates in lexicon 38, and active symbols in lexicon 38 will then excite tag candidates in lexicon 40, while the active symbols in lexicon 40 excite candidates, "王(king)" , "工(labor)" respectively in unknown lexicon 2. This procedure iterates so that unknown character will be determined gradually by eliminating weak candidates in unknown tag lexicons, segmentation lexicons and word triplet lexicons. Finally only one candidate is left in each lexicon and the candidate will be chosen as the most likely result and "王(king)" is recalled for the missing character.

2.3.1 Knowledge Link Weighting

In the basic confabulation model, the excitation level of a candidate is the sum of contributions from active symbols in other lexicons. Intuitively, however, different source lexicons do not contribute equally to a target lexicon. For example, the lexicon right next to an unknown word obviously gives more information in determining the unknown word than the lexicon that is five words away. This motivates us to weight KL's contribution during recall.

The basic idea is to weight the contribution of each KL based on the *Mutual information (MI)* [133] between its source and target lexicons. Mutual information of two random variables is a measure of variables' mutual independence. In our work, mutual information is calculated as

$$I(A;B) = \sum_{b\in B}\sum_{a\in A} p(a,b)\log(\frac{p(a,b)}{p(a)p(b)}) \tag{2.1}$$

where A is the source lexicon and a represents symbols in A; B is the target

lexicon and b represents symbols in B. $p(a, b)$ is the joint probability of symbol a and b; $p(a)$ and $p(b)$ are the margin probability of symbol a and b respectively. $I(A; B)$ is nonnegative. The value of $I(A; B)$ will increase when the correlation of symbols in lexicon A and B get stronger. Because each knowledge link has its source and target lexicons, in the rest of the chapter when we say the MI of a KL we refer to the MI of the source and target lexicons of that KL.

2.4 Evaluations

In this section, we compare the performance of different models and show how the analysis of mutual information can help to improve the efficiency of the confabulation modeling and recall. We train the Chinese confabulation model with a corpus of 10 sets of collected fairy and folk tales. We choose Chinese version of worldwide fairy tales such as Hans Christian Andersen's Fairytales, Grimm's Fairy Tales and also Chinese folk tales, because those works use vivid and common language, which will lead to a statistically meaningful knowledge base. The training set includes $364,709$ sentences and $3,232,600$ words, and is chunked into 1328 small files with equal size. The test document includes 91 sentences extracted from elementary school textbook on Chinese language art. Each test sentence has $1 \sim 4$ randomly picked missing Chinese words. For each missing word, $2 \sim 5$ possible candidates will be given. Accuracy is measured as the rate of successfully confabulated sentences, which must be identical to the original sentences.

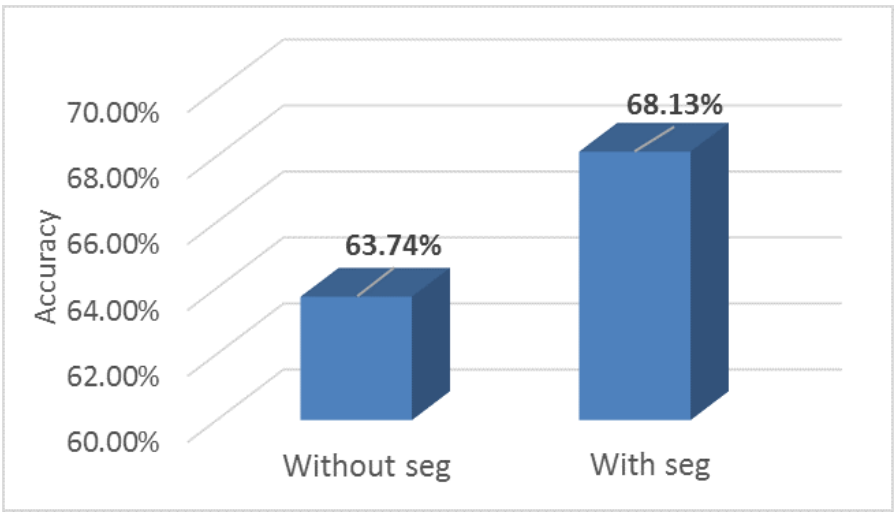


Figure 2.3: Recall accuracy of sentence confabulation model with/without segmentation label .

2.4.1 Necessity of incorporating segmentation labels and circular Knowledge Base

The first thing we want to show is the importance of including segmentation labels in the confabulation model. In this experiment, all knowledge links have the equal weight. We compare the recall accuracy of confabulation models with and without segmentation labels. The result is shown in Figure 2.3. As we can see, adding segmentation label improves recall accuracy by 4.4%.

Another experiment compares the training time, recall time and accuracy between non-circular model and circular model. Table 2.2 shows that non-circular model takes about four times training effort more than the circular model, and 17.5% more recall time, but gives 13% lower recall accuracy.

Table 2.2: Comparison of non-circular and circular model.

	Non-circular	Circular	Improvement(%)
Training time(s)	489,180	144,540	70.45%
Recall time(s)	6,317.22	5,207.83	17.56%
accuracy	54.95%	68.13%	13.18%

2.4.2 Analysis of mutual information

In the second set of experiments, we demonstrate how the change of mutual information (MI) relates to the effectiveness of the training process. We continuously monitor the mutual information of each KL as the training process progresses. The 1328 files of the training corpus is processed one by one. The MI of each KL is calculated each time after a training file is processed.

The line chart in Figure 2.4 shows the change of MI for four selected KLs as the number of processed training files increases. The blue line gives the MI of KL0, which connects two word lexicons of immediate neighbor. We can see that as more files are trained; the MI of KL0 gets smaller. The grey and yellow lines in the figure give the MI of KL72 and KL94 respectively. They are the knowledge links between a single word lexicon and its corresponding tag lexicon. The MI of these KLs fluctuate within a very small range at the beginning of training. When more training files are processed, they converge to a stable value. This is because every Chinese character has its specific semantic and syntactic function and the relation between tags (or segmentation labels) and words are relatively fixed. Very few new character-tag or character-segment relationship will be learned after certain amount of training. In other words, the knowledge base becomes saturated at certain point.

The orange line gives the MI of KL50 that connects between single word lex-

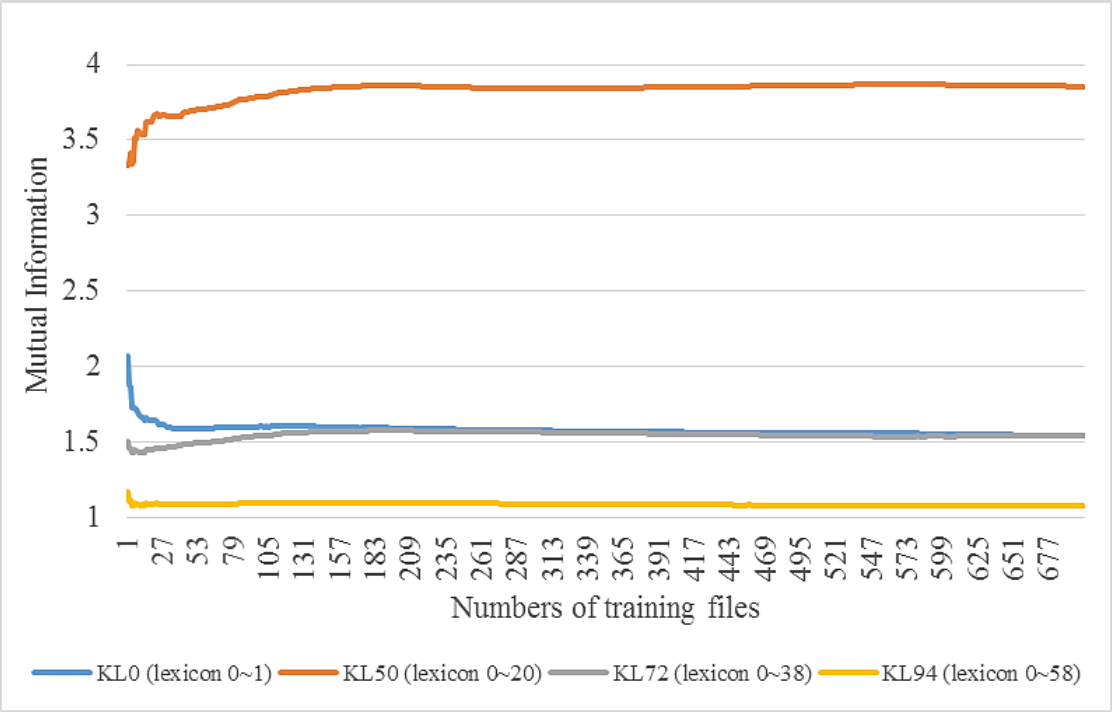


Figure 2.4: Mutual information trend chart for 4 kinds of Knowledge Links.

icon and its corresponding word triplet lexicon. Our results show that there is a strong correlation between a word and its corresponding word triplet. This means a character always co-occur with a limited number of word triplets. Similar to the MI of KL0, 72 and 94, when more training files are processed, the MI of KL50 increases and then gradually saturates to a stable value.

The convergence of MI of KLs indicates that adding more training files will not necessarily increase the learned knowledge. At certain point, the knowledge acquiring speed slows down and further learning will not be as effective as before. It is the time that we should either stop the training or switch to another set of training text that has significantly different style.

Our experimental data show that most Knowledge Links' mutual information will reach ±5% and ±3% of its stable value after the model is trained with

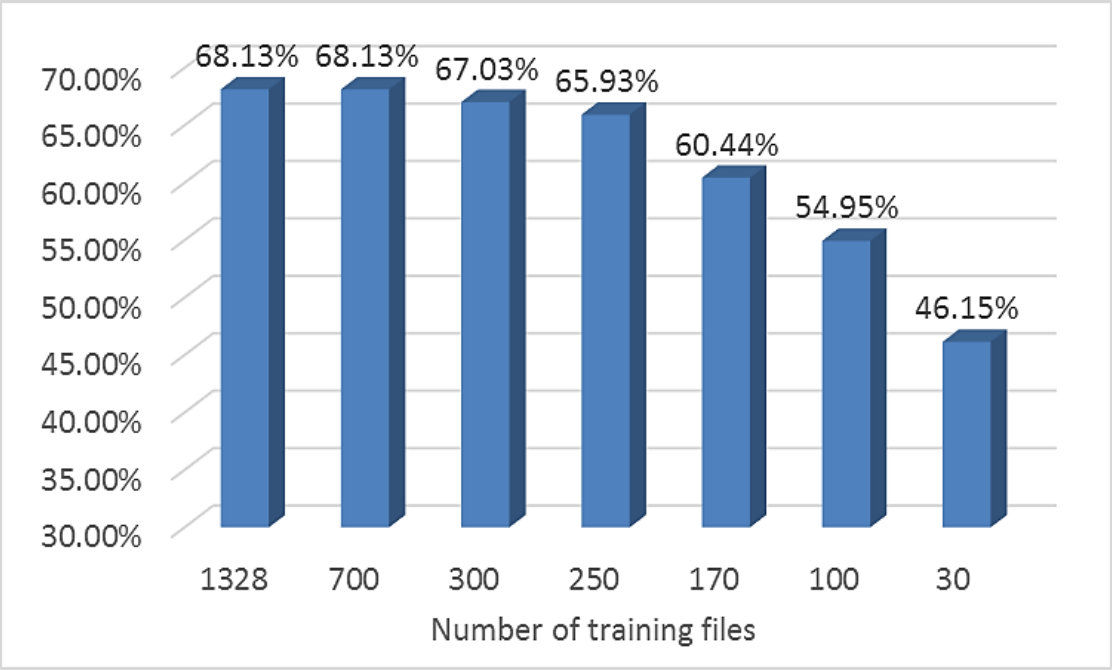


Figure 2.5: Recall accuracy of different training set size.

30 and 100 files respectively. We take the knowledge base generated at different stages of training and apply them to sentence completion test. Their recall accuracy is given in Figure 2.5. The X-axis gives the number of training files used to generate the knowledge base and the Y-axis give the recall accuracy. The graph shows that when training set size exceeds 300, the recall accuracy has stabilized at around 68%, and when the training set size reaches 170, the recall accuracy is already close to its peak. However, if the training set size is too small, the recall quality is not acceptable. This result agrees with Figure 2.4, which shows that the MI of knowledge links starts to converge after 170 training files and becomes very stable after 300 training files. Based on the above discovery, we set the saturation threshold of the training set size at 300. Our previous work [101] shows that the training time is linearly proportional to the size of training data. Limiting the training set size to the saturation threshold can sharply reduce training time with very little sacrifice of accuracy.

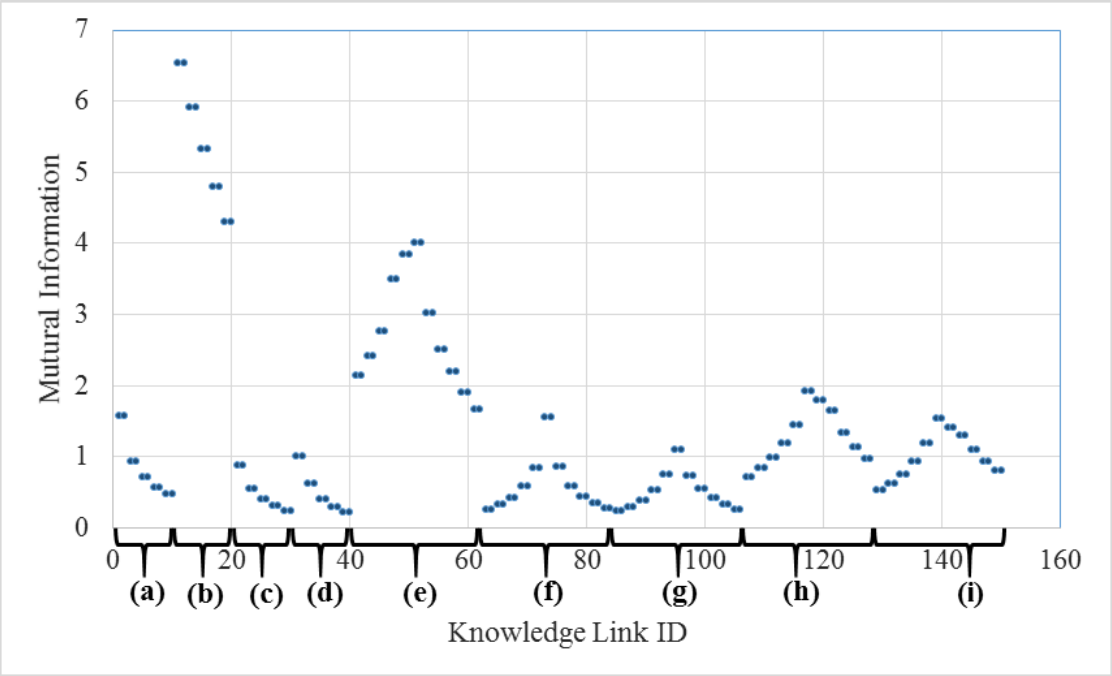


Figure 2.6: Knowledge Links' mutual information.

2.4.3 Quantified Knowledge Link Weighting scheme

The goal of next experiment is to find a systematic way to assign KL weight. Previous works [132] have shown that weighing the contribution of KLs based on their significance can improve recall accuracy, however, their weight is assigned only in an ad-hoc way. We believe that the significance of a KL can be measured by the mutual information between its source and target lexicons and therefore the MI of a KL should decide its weight.

Figure 2.6 shows the mutual information of all knowledge links. Based on their connections, the KLs are divided into 9 groups. The group division is described in Table 2.3. and labeled in Figure 2.6 underneath the X-axis.

We can see from Figure 2.6 that, from left to right, the MI of the KLs in the same group are clustered together. And as the distance between the source and

Table 2.3: Knowledge link grouping.

KL group	KL IDs	Connection
(a)	0~9	Between word lexicons
(b)	10~19	Between word triplet lexicons
(c)	20~29	Between tag lexicons
(d)	30~39	Between segmentation label lexicons
(e)	40~61	Between word and word triplet lexicons
(f)	62~83	Between word and tag lexicons
(g)	84~105	Between word and segmentation label lexicons
(h)	106~127	Between word triplet and tag lexicons
(i)	128~149	Between word triplet and segmentation label lexicons

target lexicon of the KL increases, the MI of the KL decreases. For example, KL0 and KL8 belong to the same group, therefore, they have similar MI. However, since KL0 connects between two immediate neighboring lexicons while KL8 connects between two word lexicons that are 4 words apart from each other, the MI of KL0 is slightly greater than KL8. This agrees with our intuitions that adjacent characters have stronger correlations. Second, the KLs connecting to word triplet lexicons always give more information than others, therefore they should be weighed as the biggest during the recall.

We assign the weight of a KL as a linear function of its mutual information. We then compare the recall accuracy of confabulation models with and without weighted KL. Figure 2.7 shows the recall accuracy of the two sets of confabulation models. For each set of models, the Bandgap is varied from 1 to 1000. As we can see, when bandgap value is 10 or less, assigning weight to KL provides little improvement. However, when the bandgap value exceeds 100, assigning weight to KLs brings visible benefits; it improves accuracy by more than 4%. We also observe that, without weighted KL, changing the bandgap value has almost no impact on the recall accuracy. However, with weighted KL, increasing the bandgap value from 1 to 10 and 100 can increase recall accuracy from

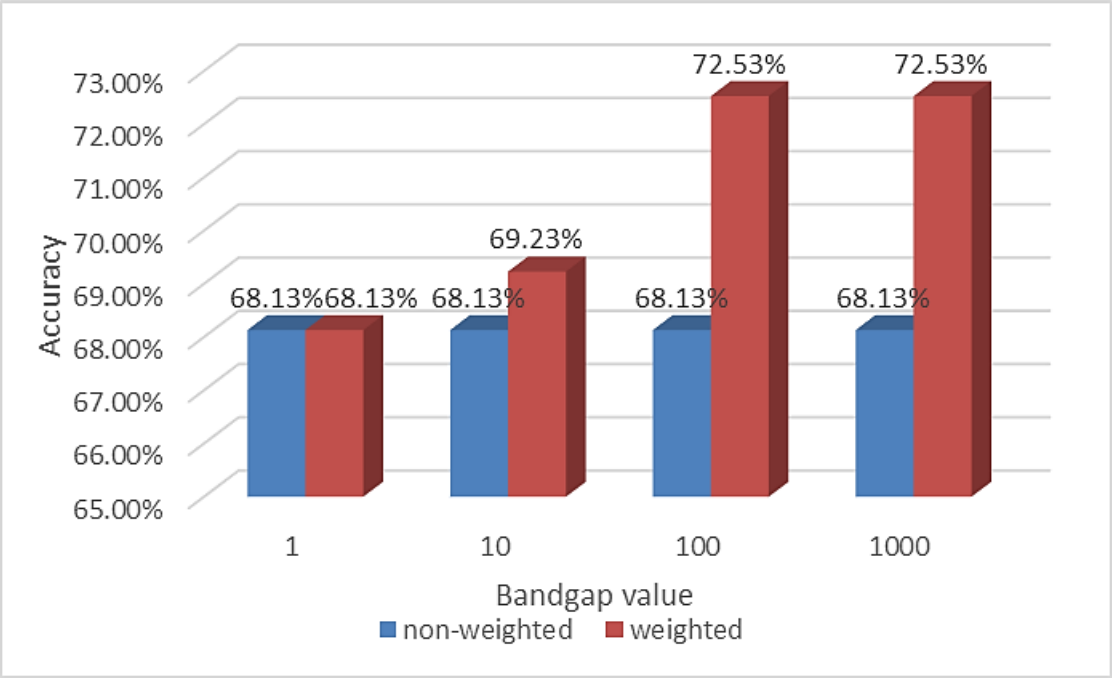


Figure 2.7: Recall Accuracy of basic confabulation model of different bandgap value with/without weighting.

68.13% to 69.23% and 72.53% respectively. The recall accuracy becomes saturated after the bandgap exceeds 100. Thus for the rest of the experiments, we fix the bandgap value to be 100.

2.4.4 Confabulation model optimization

Word pair lexicons increases accuracy

Modern Chinese language is based on word compounds that consists of two or three single character words. Considering only word triplets in the confabulation model will lose information of two-word compound. Thus we add one more level of lexicons for adjacent word pairs. This brings the confabulation lexicon structure to five levels: words, word pairs, word triplets, tags, and segmen-

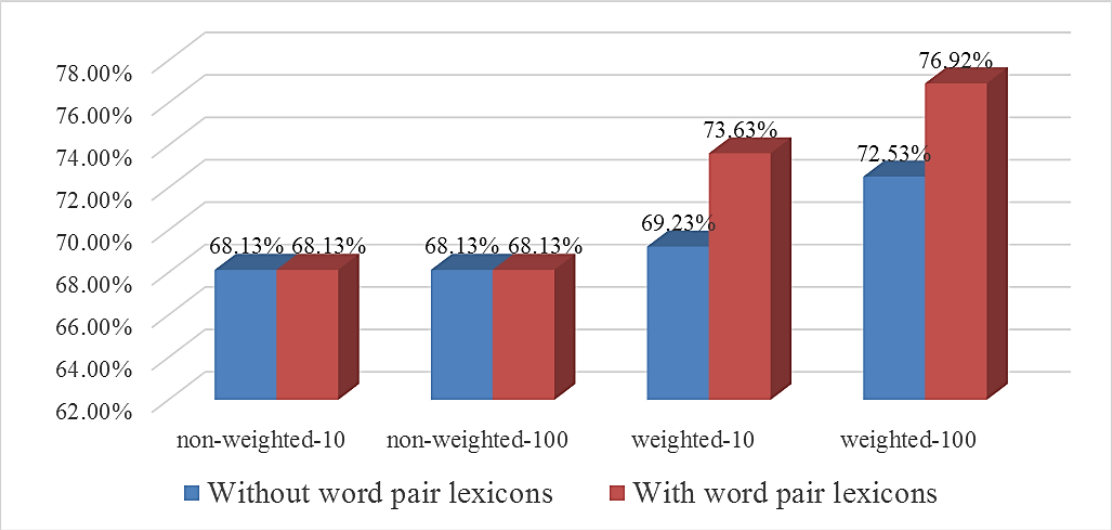


Figure 2.8: Recall Accuracy of confabulation model with/without word pair lexicons (non-weighted-10 represents recall accuracy with bandgap value of 10 and without weighting scheme; non-weighted-100 represents recall accuracy with bandgap value of 100 and without weighting scheme; Weighted-10 represents recall accuracy with bandgap value of 10 and weighting scheme; Weighted-100 represents recall accuracy with bandgap value of 100 and weighting scheme).

tation labels. We then repeat the previous experiments to assign KL weights and evaluate the recall accuracy. Figure 2.8 shows two sets of recall accuracy. The blue bars give the recall accuracy of the original 4-level confabulation model and the red bars give the recall accuracy of the new 5-level model. Both models are evaluated with and without KL weight and with two different bandgap values. The results show that adding one more layer of lexicon does not improve the recall accuracy when the KLs are not weighted and the improvement is limited if bandgap is small. However, it does make visible differences when KLs are properly weighted and bandgap is set large enough. The overall recall accuracy can be 76.9%, which is about 9% higher than the basic model without weighted KL

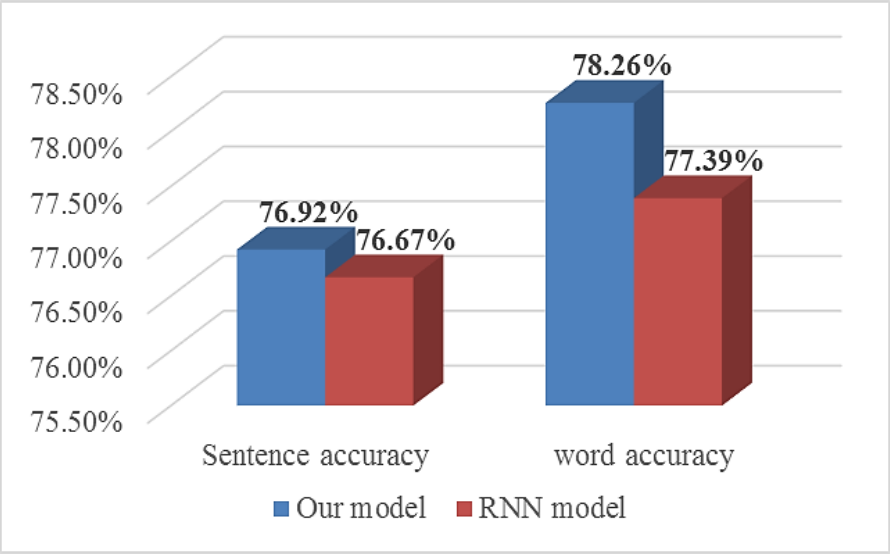


Figure 2.9: Recall accuracy between different models(sentence accuracy is evaluated by the amount of sentences recalled identically to original sentences; word accuracy is evaluated by the amount of missing Chinese charaters recalled identically to the original).

Comparison with RNN model

As a reference, we compare the confabulation model with a recurrent neural network (RNN) model [88]. Please note that the RNN model identifies the missing word from the list of candidates by evaluating the probability of the sentence that they could make. Therefore, it has to create a sentence for each combination of the candidates and calculate its probability. The complexity of the RNN is an exponential function of the number of missing words, while the complexity of confabulation model is a linear function of the number of missing words. Figure 2.9 compares the recall accuracy of the RNN model and confabulation model. It shows that these two has comparable recall accuracy and the confabulation model has slightly better word recall accuracy.

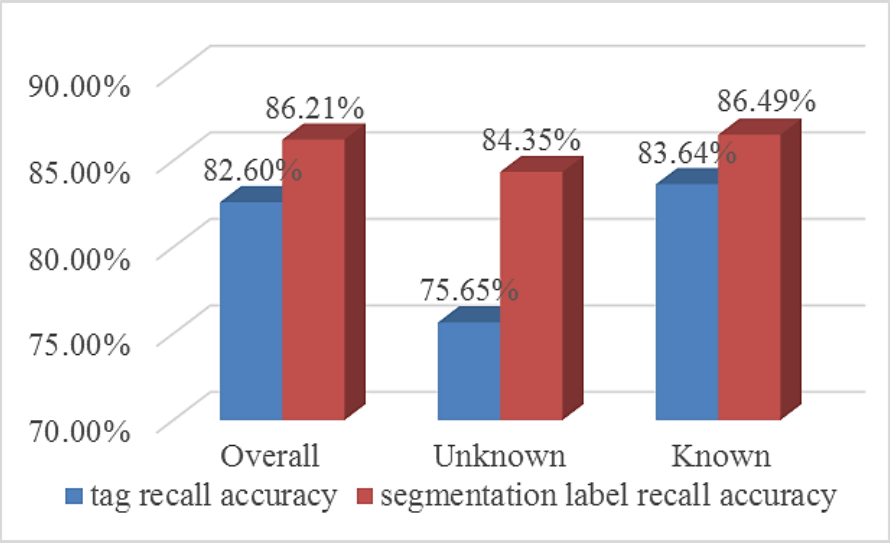


Figure 2.10: Recall accuracy of tags and segmentation labels(Overall denotes accuracy for all characters in sentences; Unknown denotes accuracy for unknown missing characters, Known denotes accuracy for known characters).

Syntactic parsing performance

One of the advantages of using the confabulation model is that it performs syntactic parsing at the same time of sentence completion. It does not only fill in the missing characters, but also finds out the tags and segmentation labels for all words in the sentence. Figure 2.10 gives the tag and segmentation label recall accuracy. Overall, 82.6% of the words are tagged correctly with POS tagging and 86.2% of the words are correctly labeled with their segmentation information. We can see that even those unknown characters are tagged and labeled with quite high accuracy. Their tagging accuracy is 75.6% and segmentation accuracy is 84.3%.

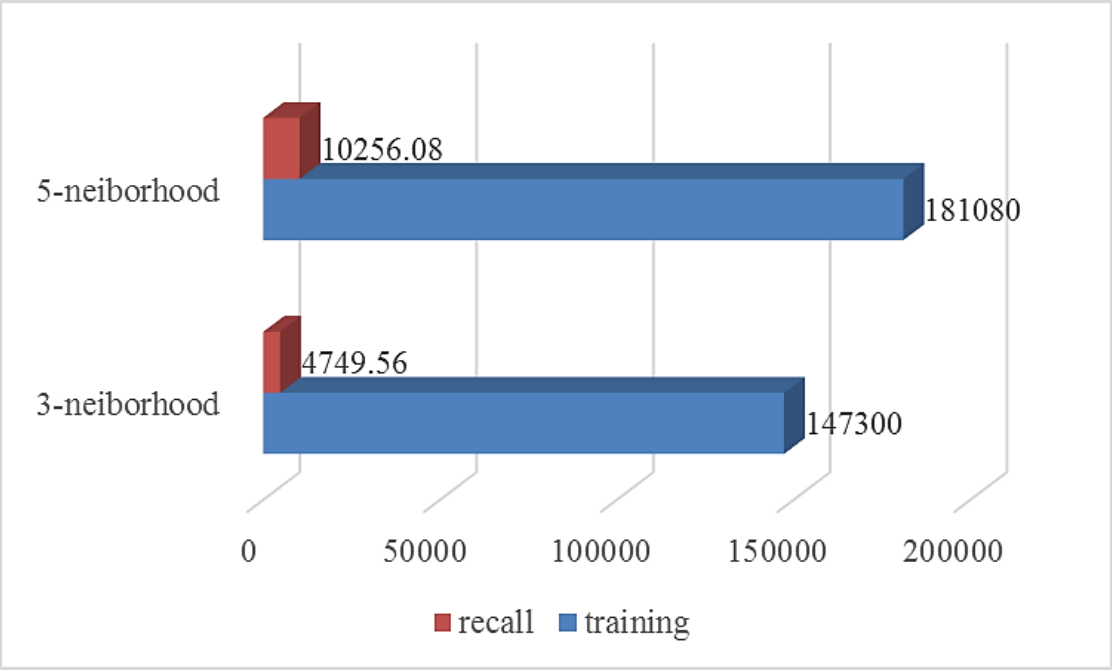


Figure 2.11: Training time and recall time of different KL structure.

Mutual information guided parameter selection

Observing the mutual information, we realize that the 4^{th} and 5^{th} neighbors of a lexicon provide far less information than other closer neighbors. The MI of these KLs is approximately 20 ~ 60% of the average of other KLs. This motivates us to use KLs only up to the 3^{rd}-neighborhood. This simplification does not only reduce KB size but also save training and recall time. Figure 2.11 compares the training and recall time of the original model (5-neighbor) and simplified model (3-neighbor). Experiments show that these two models give the same recall accuracy, which is 76.9%. However, the training time is decreased by 18.6% and the recall time is decreased by 53.7%. This is because the KB size is reduced from 226 to 142 by removing KLs connecting between lexicons and their 4th and 5th neighbors. These data show that, the mutual information of KLs does not only help us to assign weight to KLs for better recall accuracy, but also facilitate the

Table 2.4: Example of confabulated sentences.

Original	王明负责(be in charge of)检查卫生工作
Basic	王明责任(responsibility)检查卫生工作
Optimized	王明负责(be in charge of)检查卫生工作
Original	锤炼得更坚强(temper it to be stronger)
Basic	锤炼的更坚强(tempered stronger)
Optimized	锤炼得更坚强(temper it to be stronger)
Original	口号特别震撼人心(Slogan excites people's mind)
Basic	口号特别振撼人心(Slogan shakes people's mind)
Optimized	口号特别震撼人心(Slogan excites people's mind)

decision on removing KLs with small contribution for lower model complexity without significantly sacrificing the accuracy.

2.4.5 Qualitative Results

All above results are based on the restriction that only sentences identical to the original are considered as correct. Many recalled sentences are actually syntactically correct and semantically close to the original sentence. For example, for the original sentence: 妈妈是一个很温(柔)的人 (Mother is a gentle person), our recalled sentence is 妈妈是一个很温(和)的人 (Mother is a gentle and mild person), in which 温柔 (gentle) and 温和 (gentle and mild) are synonyms. Even though the recalled sentence is not identical to the original sentence, it has very close meaning. If we treat all grammatically correct recalled sentences as successful recall, the accuracy will increase to 80%.

Table 2.4 lists some examples of recalled sentences. The rows labeled as "Original" give the correct sentences; the rows labeled as "Basic" give the recall sentence from the original Chinese confabulation model with only word

triplet lexicons and without KL weights; and the rows labeled as "Optimized" give the recall results from the optimized model, which has both word triplet and word pair lexicons as well as MI directed KL weights. The text in bold highlights the difference between the recall results. We can see that the optimized model improves the recall results semantically and syntactically.

2.5 Conclusion

We proposed a Chinese sentence confabulation model by refining and modifying the English sentence confabulation model. The proposed model exploits semantic information including POS tags and segmentation labels, as well as optimized method such as circular knowledge storage, marking the start of sentence and N-neighborhood lexicon link, to successfully complete Chinese sentences with missing characters. Based on the mutual information analysis, a saturation threshold is set to the size of training set. This can sharply reduce the training time with little sacrifice of accuracy. We also found that MI directed KL weights could amplify the effect of other optimization actions, such as increasing the bandgap value and adding word pair lexicons. All together they can improve the recall accuracy by 9%. Finally, the MI analysis helps us to simplify the model and reduces the overall training and recall time by 18.6% and 53.7% respectively.

CHAPTER 3

PARALLEL IMPLEMENTATION OF ASSOCIATIVE INFERENCE FOR COGENT CONFABULATION

3.1 Introduction

Human perception and cognition involve two steps, sensing and association. The association area is by far the most developed part of the cerebral cortex. The associative inference is related to brain activities in routine, repetitive situations and well-precedent problems. It has been used to explain the perception of sensory input [8] and language learning [56]. The superb of human cognizance comes from its extensive highly associative memory. For example, it is easy for human to correct errors and recover the damages in document images or speech; while this has always been a difficult task for conventional OCR or speech recognition tools.

Many associative memory models have been proposed. They include attractor network associative memories, bidirectional associative memories, and sequence associative memories, etc. In this work, we focus on the cogent confabulation model, which is a belief network with recurrent connections. Cogent confabulation arranges neurons into lexicons. The input of the cogent confabulation is the initial status of the neurons, which corresponding to noisy and ambiguous observations or lack of information. The recall process is an iterative excitation and inhibition among neurons. The feedback connections also loop back the neural activities and gradually refine the memory until at the end only the set of the most highly associated neurons are active. This model has been applied in text image recognition and sentence construction in previous

works [101, 100, 132, 77, 98, 99].

Associative inference in the neocortex system is a concurrent process, where neurons update their states simultaneously. During this procedure, there is no deterministic order among neurons. In all previous works, cogent confabulation is implemented as a sequential process, where neurons are updated based on a prefixed order. The status of the neurons in one lexicon depends on the output of neurons in some other lexicons. Due to feedback connections, the data dependency is cyclic. None of the fixed evaluation order can satisfy all precedent constrains.

In this work we aim at parallelizing the algorithm on a multicore processor. The parallel framework consists of a thread pool, where each thread evaluates the status of specific lexicons. In addition to reduced computation time, we found that the parallel implementation also improves recall accuracy. The racing among threads breaks the fixed processing order in sequential processing and introduces randomness. The parallel architecture allows neurons to use the most up-to-date information of their neighbors.

The number of lexicons to be processed in a confabulation model is usually much greater than the number of active threads that can be supported by a general purpose CPU. Each thread needs to process multiple lexicons. We design a lexicon scheduling algorithm to further add randomness in the lexicon processing order and at the same time ensure a balanced progress in status update among neurons. Based on the algorithm, a thread switches from one lexicon to another when the neuron activity of the former has reached to a state that is informational.

The contributions of this work can be summarized as follows:

1. A parallel implementation for cogent confabulation is developed using multi-threading and blocking queue techniques.
2. We demonstrate that the parallel implementation not only reduces runtime, but also improves recall accuracy by breaking the fixed evaluation order and maintain a balanced progress in status update among all neurons.
3. A lexicon scheduling algorithm is presented that provides further improvements.

The experimental results show that up to 93.4% reduction in runtime and 5% increase in recall accuracy can be achieved using the proposed parallel implementation and scheduling algorithm.

3.2 Sequential Cogent Confabulation algorithm

The input of the recall function is a set of activated symbols A_l for each lexicon l. These symbols are referred as *candidates*. They correspond to a noisy observation of the target. In this observation, some features are observed with great ambiguity, therefore multiple symbols are activated in the corresponding lexicons, i.e. $|A_l| \geq 1$. In extreme cases, no observation is obtained for certain features, therefore, all symbols in those lexicons are activated as potential candidates. The goal of the recall process is to resolve the ambiguity and select the set of symbols that are most highly associated with each other using the statistical information obtained during the learning. At the end of the recall process,

we obtain a refined activation set A_l^*, and $|A_l^*| = 1$, $\forall l$. This is achieved a sequence of iterative excitation and inhibition among neurons as described in the following.

Each neuron in a target lexicon receives an excitation from neurons of other lexicons through KLs, which is the weighted sum of its incoming excitatory synapses. Let l denote a lexicon, F denote the set of lexicons that have knowledge links going into lexicon l, and S_l denote the set of symbols that belong to lexicon l. The excitation $el(t)$ of a symbol t in lexicon l is calculated by summing up all incoming knowledge links:

$$el(t) = \sum_{k \in F} \{ \sum_{s \in S_k} [I(s) w_{kl} \ln(\frac{P(s|t)}{p_0})] + B \}, t \in S_l \tag{3.1}$$

$$I(s) = \frac{el(s)}{\sum_{j \in S_k} el(j)}, s \in S_k \tag{3.2}$$

$I(s)$ is the normalized excitation level across all actives in the same lexicon. The parameter p_0 is the smallest meaningful value of $P(s_i|t_j)$. w_{kl} is the weighting factor which is a linear function of mutual information [7] of KL from lexicon k to lexicon l [77]. The parameter B is a positive global constant called the *bandgap*. The purpose of introducing B in the function is to ensure that a symbol receiving N active knowledge links will always have a higher excitation level than a symbol receiving $(N-1)$ active knowledge links, regardless of their strength. As we can see, the excitation level of a symbol is actually its log-likelihood given the observed attributes in other lexicons.

Among neurons in the same lexicon, those that are least excited will be suppressed and the rest will fire and become excitatory input of other neurons. Their firing strengths are normalized and proportional to their excitation levels. As neurons gradually being suppressed, eventually only the neuron that has

the highest excitation remains to fire in each lexicon and the ambiguity is thus resolved.

Algorithm 1: Baseline sequence confabulation recall algorithm

Data: The activation set A_l ($|A_l| \geq 1$) of symbols in each lexicon l, predefined *maxAmbiguity*, *maxIteration*
Result: The refined activation set A_l^* ($|A_l^*| = 1$)
$N \leftarrow maxAmbiguity$
Initialize the set of unknown lexicons $L_u = \{l : |A_l| > 1\}$
while $N > 1$ **do**
  $converged \leftarrow false$
  $iterationCount \leftarrow 0$
  while $\neg converged$ **do**
    for *each* $l_u \in L_u$ **do**
      for *each symbol* $s \in A_{l_u}$ **do**
        calculate $el(s)$
      end
      sort(A_{l_u}) based on $el(s), \forall s \in A_{l_u}$
      set the first N symbols in A_{l_u} as active
      for *each symbol* $s \in A_{l_u}$ **do**
        calculate $I(s)$
      end
    end
    $iterationCount \leftarrow iterationCount + 1$
    if *active symbol set unchanged* $\vee$ $iterationCount \geq maxIteration$ **then**
      $converged \leftarrow true$
    end
  end
  for *each* $l_u \in L_u$ **do**
    $A_{l_u} \leftarrow$ the first N symbols in A_{l_u}
  end
  $N \leftarrow N - 1$
end
$A_l^* \leftarrow A_l, \forall l$
output $A_l^*, \forall l$

Algorithm 1 shows the recall function as a sequential process. The candidates' excitation levels are calculated lexicon by lexicon in series. Due to the recurrent connections between lexicons, the evaluation needs to iterate several times to ensure that changes in the excitation level of lexicons propagate

through the network before we prune one symbol from every lexicon. It is not optimal to prune all lexicons at the same time, because the decision is made upon the excitation level calculated using the old symbol status. After a symbol in one lexicon is pruned, the excitation levels of symbols in other lexicons are likely to change. Therefore, it is possible that we are going to prune something that should be kept. However, because it is expensive to evaluate the excitation level sequentially, we cannot afford to update the excitation level each time after pruning a symbol in one lexicon. And due to the fixed processing order, some lexicons always have the privilege of making decision on more update-to-date information than the others.

3.3 Parallel Implementation of Associative Inference

To fully explore the computation power of multi-core multi-thread processors, we investigate parallel implementation of the recall function of cogent confabulation. Starting from a naïve implementation that simply duplicates multiple copies of the recall function and run them in parallel, we improve the architecture by adopting finer grained parallelism and better-controlled scheduling algorithms. In Section 3.4, the experimental results will show that these additions not only reduce the runtime but also improve the recall accuracy.

3.3.1 Request Level Parallelization

The intuitive design of parallelization is to run multiple copies of the recall functions in Algorithm 1 using multiple threads. Each thread processes an indepen-

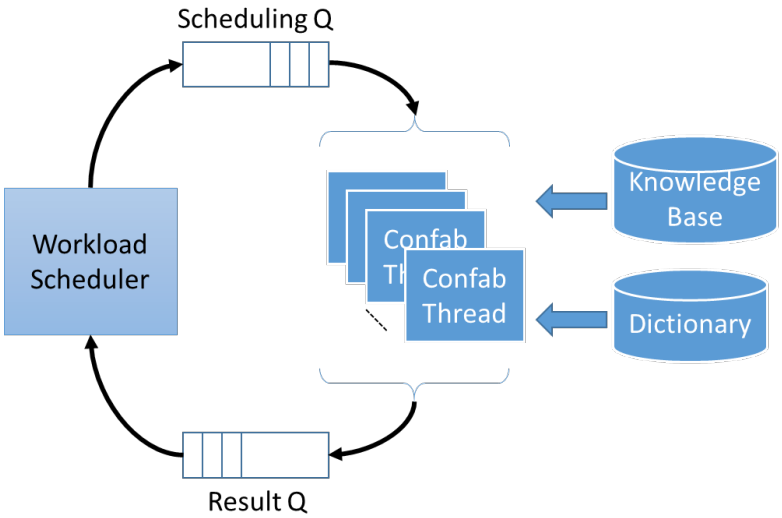


Figure 3.1: Intuitive parallel confabulation architecture

dent confabulation recall, and all threads share the same knowledge base. As shown in Figure 3.1, for each incoming confabulation request, the workload scheduler will collect symbol candidates to assemble the initial activation set as the input of the recall function. Once the initial activation set is assembled, it will be placed in the scheduling queue. And each confab thread will de-queue one request and run Algorithm 1.

At the end of recall, the refined activation set will be sent to the result queue. Please note that in this design, scheduling queue and result queue are both thread safe blocking queues, which enable an automatic load balancing among multiple threads. This design can simultaneously process as many recall functions as the number of confab threads. However, within a confab thread, the processing is still sequentially. All the previously mentioned limitations of the sequential implementation, such as fixed processing order, the need of iterative evaluation, and the possibility of pruning some symbol without sufficient information, still exists in this naïve implementation.

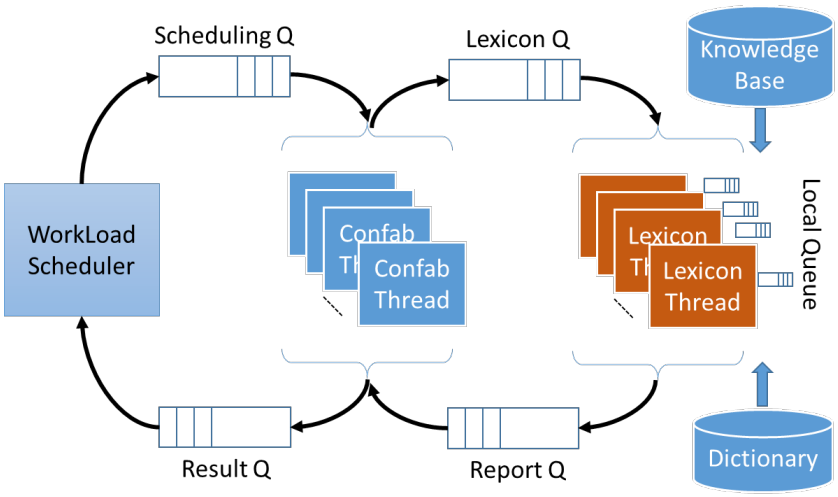


Figure 3.2: Parallel confabulation architecture on lexicon level

3.3.2 Lexicon Level Parallelization

To emulate the parallel structure in a biological neural system, we investigate finer grained parallelism where lexicons are processed in parallel. A set of lexicon threads are created. Once a confabulation thread receives a request from the scheduling queue, it divides the input to multiple lexicons. It places those lexicons with ambiguous or unknown information (i.e. the lexicons l_u with more than one active candidates) with their activation set (i.e. A_{l_u})) into a lexicon queue. Each lexicon thread will fetch its input from the lexicon queue, iteratively performs excitation level calculation and pruning the inactive symbols until there is only one active symbol. The lexicon together with the refined activation set is put into a report queue, which will be read by the confabulation thread. After the confabulation thread collected the report for all the lexicons belonging to the same recall request, it will forward the result to the result queue.

The new architecture is depicted in Figure 3.2, except the round-robin queues, which are not needed to achieve the lexicon level parallelism. They will be discussed in the next section to enable lexicon scheduling. In Figure 3.2,

scheduling queue, lexicon queue, result queue and report queue are blocking queues. We use blocking queues for them because they are accessed by multiple threads and this makes sure that the requests and reports are read and written thread safely. Furthermore, when a thread is blocked, it will not consume CPU resource, hence making the system more efficient. Lexicon threads reside in fixed-size thread pool so that we can control the pooling effort.

Algorithm 2: Lexicon thread confabulation recall algorithm

```
Data: Primary lexicon scheduling queue LexiconQ
Result: Status report queue RepQ
while True do
    l_u ←dequeue(LexiconQ)
    N ← |A_{l_u}|
    while N > 1 do
        for each symbol s ∈ A_{l_u} do
            calculate el(s)
        end
        sort(A_{l_u}) based el(s), ∀s ∈ A_{l_u}
        A_{l_u} ← the first N symbols in A_{l_u}
        for each symbol s ∈ A_{l_u} do
            calculate I(s)
        end
        N ← N − 1
    end
    A*_{l_u} ← A_{l_u}
    ReqQ ← enqueue(l_u)
end
```

Algorithm 2 shows the function of a lexicon thread. Because all lexicons update their status simultaneously, it is easier for a lexicon to obtain its neighbors most recent status and the status change of one lexicon can propagate through the network faster than the sequential implementation. In the parallel implementation each lexicon updates its excitation level based on neighbor information and prune inactive symbols asynchronously in a distributed manner.

3.3.3 Lexicon scheduling for intermittent pruning

The lexicon thread in Algorithm 2 picks a lexicon from lexicon queue and keeps on processing it until there is only one active symbol. Its effectiveness is based on an assumption that all other lexicons are simultaneously being processed, and the thread has the most up-to-date information on its neighbor status. For many applications, the number of lexicons is greater than the number of active lexicon threads that can run simultaneously on a processor. If the status of the lexicons that are currently being computed depends on the status of lexicons that have not been processed, then all the calculation and pruning are based on stale information. Again, due to the recurrent knowledge link connections, lexicons have mutual dependencies, and we are not able to find an evaluation order for the lexicons to satisfy all precedence constraints.

Instead of oversubscribing the hardware and using a very large pool of lexicon threads to accommodate all lexicons, and requesting OS to schedule those threads, we limit ourselves to a small thread pool and create our own lexicon scheduling algorithm to share each thread among multiple lexicons. The main idea is to process a lexicon until it reaches a state that its status is informational to its neighbors, then yield the computation resource (i.e. the lexicon thread) to another lexicon. Whether a lexicon status is informational is measured by the normalized difference between the highest and second highest excitation level of the lexicon, and we refer it as the *confidence level (cl)*. It is calculated as the following:

$$cl = min(1, \frac{|el(0) - el(1)|}{max(el(0), el(1))}) \tag{3.3}$$

where $el(0)$ and $el(1)$ are the highest and second highest excitation level in that lexicon.

As shown in Figure 3.2, a local non-blocking queue is attached to each lexicon thread to process lexicons in a round-robin way. Lexicon thread keeps popping up one lexicon from its local queue to run confabulation algorithm. If current lexicon's confidence level exceeds the average *cl* among all lexicons in this recall, then the lexicon will yield the thread and be put back into the local queue for future access, and a new lexicon from the local queue will be fetched for processing. When all lexicons in the local queue have been processed, the thread will move a new lexicon from the lexicon queue to its local queue and repeat the above procedures.

The revised parallel implementation processes each lexicon in an intermittent manner; therefore, we refer it as lexicon level parallel with *intermittent-pruning*. Intuitively if lexicons are pruned too slowly, as they were in the sequential implementation, then there are too many active symbols in the network and they will introduce noise to the recall. On the other hand, if lexicons are pruned too fast, as they were in the simple lexicon level parallel implementation, then there is not enough time for the status change to propagate through the network. The intermittent-pruning finds a balance between these two. It maintains a balanced progress among all lexicon evaluations and by putting a lexicon back to the local queue it gives some time for the lexicon's new status to propagate through the network before working on it again.

Interestingly, intermittent pruning also helps to improve the performance, as we will show in the experimental results. This is probably because it reduces the memory contention. Since all running lexicon threads are using shared data, cache coherence [113] needs to be maintained. It takes a lot of time updating the L1/L2 Cache for each CPU core when the excitation levels are read and writ-

ten by different threads running on different cores simultaneously. Intermittent pruning reduces the frequency that a lexicon is updated, therefore, it reduces the chance of memory access contentions.

3.4 Experimental Results

We take sentence reconstruction as an example to evaluate the proposed parallel implementation of associative inference. We implemented our proposed algorithm as a standalone module in ITRS [100, 98, 99]. It receives ambiguous word candidates from noisy scan input, then run confabulation recall to resolve the ambiguity and select appropriate words to reconstruct a sentence. In sentence confabulation model, we assume that the maximum length of a sentence is 20 words and sentences with more than 20 words will be truncated. As Figure 3.3 shows, we have three layers of lexicons: words, adjacent word pairs, and *Part-of-speech(POS)* tags [123, 122, 132]. Lexicon 0 to 19 correspond to single English word at location 0 to 19 in a sentence. Lexicons 20 to 38 correspond to 19 word pairs combining word from lexicon 0 to 19 and its right adjacent neighbor. Lexicon 39 to 58 correspond to the POS tag in accordance with each word. Each lexicon stores a tremendous number of symbols (words, word pairs or tags) that appears in the corresponding location. We define intra-level KLs as KLs from one lexicon to another in the same lexicon layer while inter-level KLs are KLs from one lexicon to another lexicon in a different lexicon layer. KLs are created between two inter-level or intra-level lexicons that are less than N-neighborhood (empirically $N = 5$) far away from each other and shared among lexicon pairs that have the same relative position in a sentence.

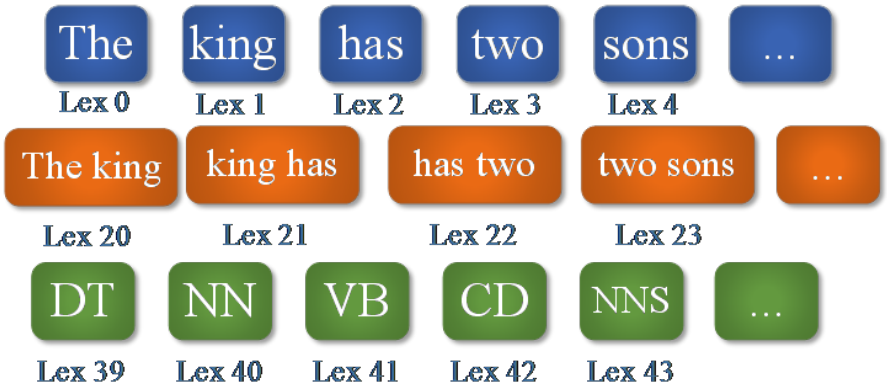


Figure 3.3: Sentence lexicon model example

We took three pages of occluded scanned documents, which are not included in the training corpus of sentence confabulation model. The documents consist of 315 sentences, totally 2241 words scanned from a printed paper. About 10% of the words are fully or partially occluded. The noise causes great ambiguity in the input of sentence confabulation. In average there are 5 candidate words at each word location of a sentence, and we need to resolve the ambiguity to recover the sentences.

We measure the compute time as the duration from the end of system initialization to the time when the last sentence has been confabulated. The recall accuracy is measured by sentence accuracy, which specifies the percentage of sentences that are reconstructed exactly same as the original sentence. The tests are implemented on a Linux-2.6.32 based machine with two 4-core CPUs (Intel® Xeon® W5580@3.20GHz with 48GB RAM). The machine has totally 8 cores and 16 logical processors (16 simultaneous threads).

The first group of tests compares the *request level parallelism (RLP)* to *lexicon level parallelism (LLP)*. No intermittent pruning is enabled. For the parallel design, we set lexicon thread pool size to 5 and vary the number of confabulation threads from 1 to 16. When there is only 1 confabulation thread, the RLP im-

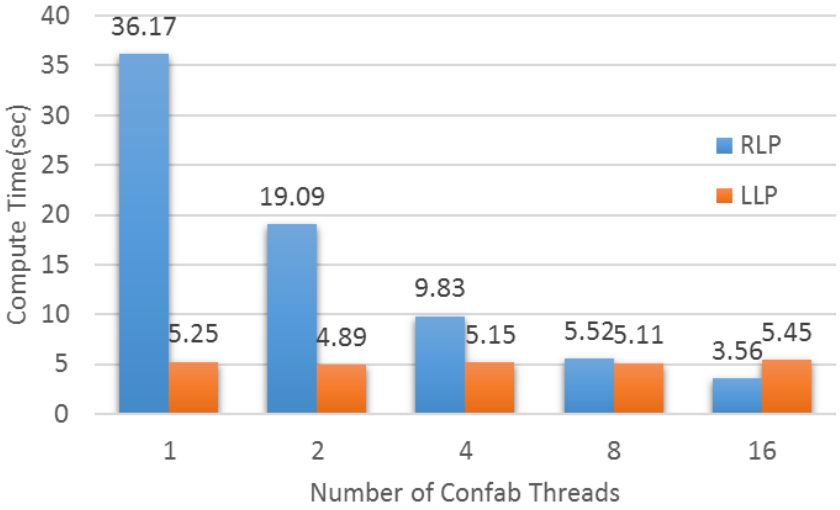


Figure 3.4: Compute time for RLP and LLP w/o intermittent pruning

plementation reduces to serial implementation. Figure 3.4 and Figure 3.5 show the compute time and recall accuracy of these two implementations. We can observe that as the number of confab thread increases, the compute time of RLP decrease linearly, and the sentence accuracy of RLP is consistently 69.21% for all five configurations. This is because each confab thread in RLP processes independent recall requests serially. Increasing the number of threads will not affect how the recall function is evaluated. Meanwhile, the compute time of LLP is relatively independent of the number of confabulation threads, because its computation resources are the lexicon threads, whose size is constant in this experiment. Furthermore, because LLP no longer has the overhead of convergence check and it prunes inactive symbols asynchronously, running with 5 lexicon threads in LLP is faster than running with 8 confab threads in RLP. It is interesting to see that using LLP the recall accuracies are also improved visibly. Because parallel confabulation introduces randomness of computation and overcomes the error due to the fixed execution order.

In the second group of tests, we compare lexicon level parallel implementa-

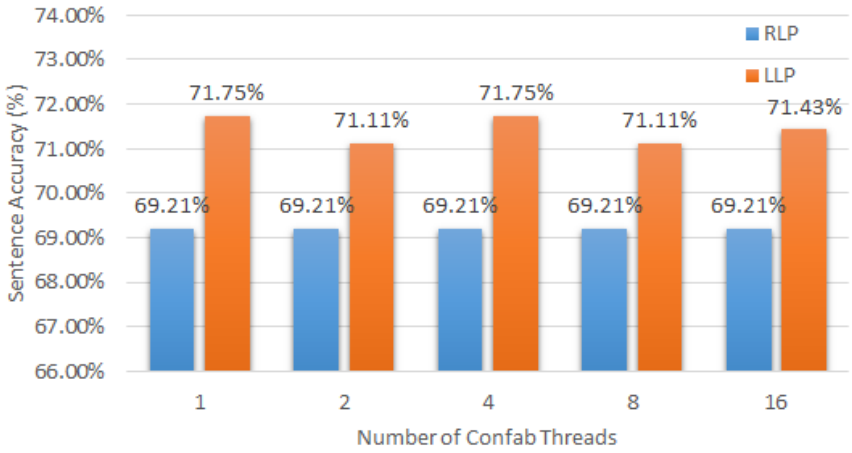


Figure 3.5: Sentency accuracy for RLP and LLP w/o intermittent pruning

tion without intermittent pruning (LLP w/o Itm) with parallel implementation with intermittent pruning (LLP w. Itm). Again we set lexicon thread pool size to 5 for both of them. Figure 3.6 shows that the compute time of both implementations are independent to the number of confabulation threads. However, LLP with intermittent pruning is 7.2% faster than LLP without intermittent pruning. As we explained before, we attribute this phenomenon to less memory contention. Pausing will spread the processing of a lexicon over longer period of time, and hence reduce the chance of memory contention due to simultaneous read and write by different threads, and alleviate the cache coherence overhead.

As for the accuracies, with intermittent-pruning, sentence accuracy improved by about 1%. In the second group compared to the other group. As Figure 3.7 shows, the sentence accuracy ranges from 72.38% to 73.02%, which is higher than the other group's accuracies ranging from 71.11% to 71.75%. This is because with intermittent pruning, more lexicons can be loaded to lexicon threads; therefore, it maintains a more balanced progress of status update among different lexicons. Furthermore, a lexicon is pruned through multiple pruning runs instead of one processing. This gives some time for changes to

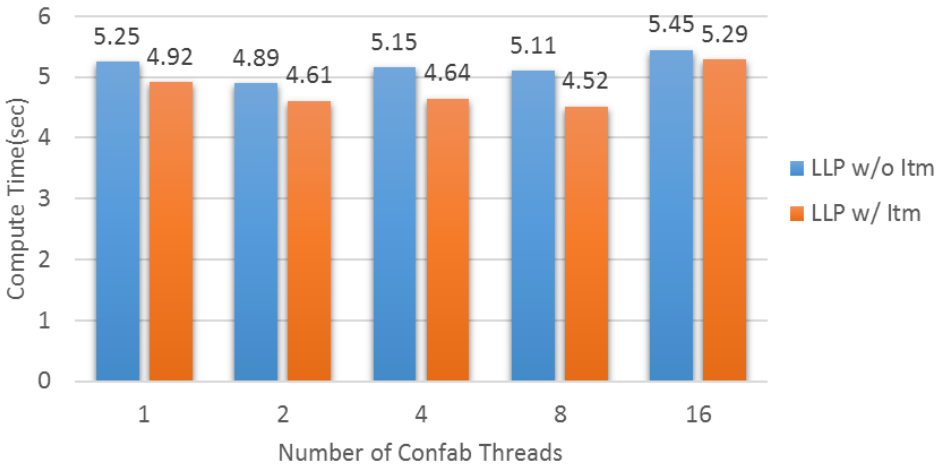


Figure 3.6: Compute time for LLP w/o and w/ intermittent pruning

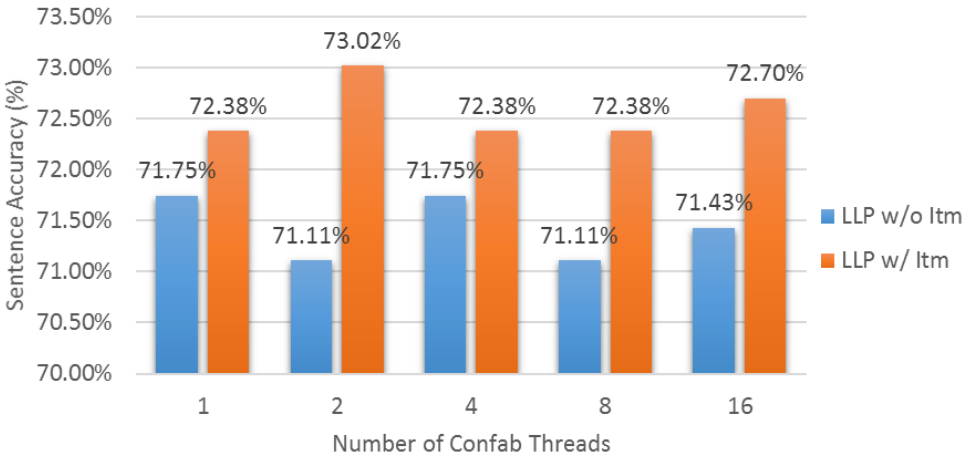


Figure 3.7: Sentency accuracy for LLP w/o and w/ intermittent pruning

propagate through the network.

In the third group of tests, we compare performance and accuracy of different configurations of LLP with intermittent pruning. The sizes of lexicon pool are set to 5, 10, and 20. We can see that in Figure 3.8, given the same number of confab thread, increasing number of lexicon threads improves performance. Moreover, more lexicon threads not only speed up computation, but also increase the accuracies as shown in Figure 3.9. In general, higher parallelism introduces more randomness in processing order and more vividly resemble a real

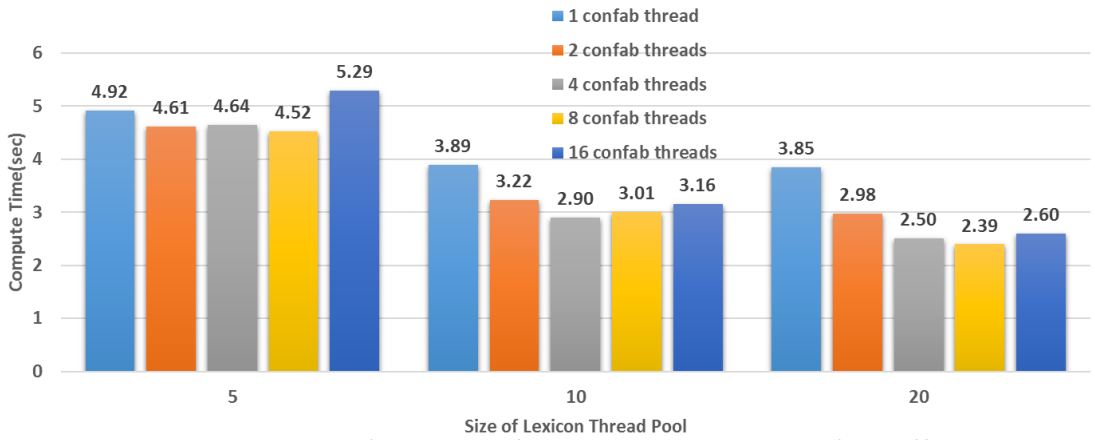


Figure 3.8: Compute time for LLP w/ intermittent pruning for different lexicon thread pool sizes

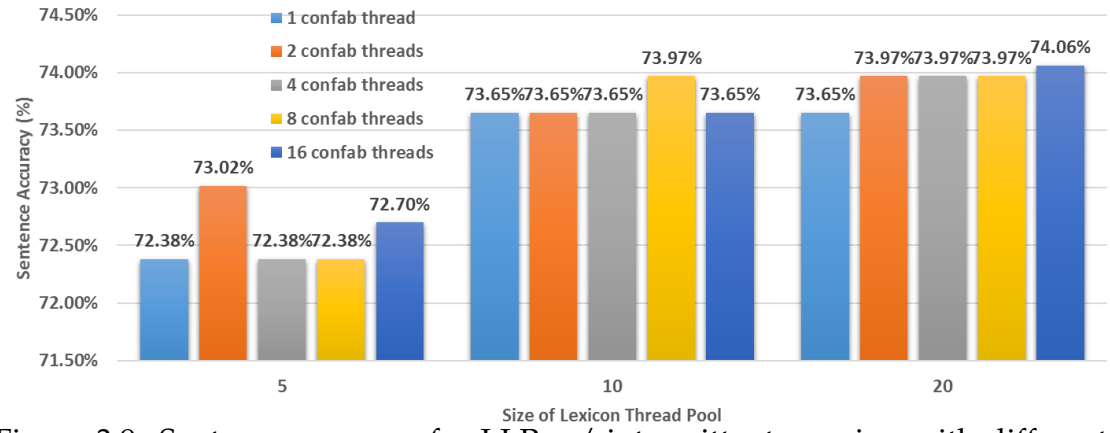


Figure 3.9: Sentence accuracy for LLP w/ intermittent pruning with different lexicon thread pool sizes

biological neural network. However, the improvement is not significant when we increase lexicon pool size from 10 to 20. This is probably because in average each lexicon is connecting to 10 neighbors (N = 5). We also found that increasing the number confabulation thread does not have noticeable impact on recall accuracy. It does not help to reduce computing time either when the lexicon pool is small. However, when the lexicon pool size increases to 20 threads, increasing the number of confabulation thread can give up to 60% reduction in computing time. This is because we need more confabulation threads to generate inputs and analyze outputs for the large number lexicon threads.

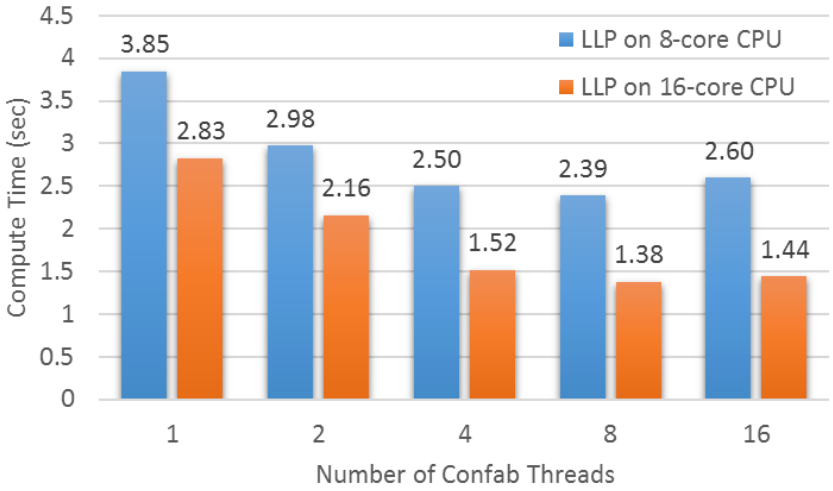


Figure 3.10: Compute time for LLP w/ intermittent pruning on 8-core CPU machine and 16-core CPU machine with 20 lexicon threads

In the last group of tests, to investigate the impact of over-subscription, we run LLP with intermittent pruning on another Linux-2.6.32 based machine with two 8-core CPUs (Intel® Xeon® E5-2690@2.90GHz with 192GB RAM). The machine has totally 16 cores and 32 logical processors (32 simultaneous threads). We compare LLP-Itm running on the 16-core CPU with that running on the 8-core CPU. We set lexicon thread pool size as 20 so that we will oversubscribe the 8-core CPU. As shown in Figure 3.10, it is not surprising to see that 16-core CPU is faster than 8-core CPU. What is interesting is that the system on 16-core CPU benefits more from increasing the number of confab threads. When the confabulation thread increases from 1 to 2, it brings approximately 30% compute time reduction in both systems. When we increase the number of confabulation thread to 8, it leads to 60% improvement in the 8-core system but more than 100% improvement in the 16-core system. It receives more than 100% reduction in compute time. This is because not all 20 lexicon threads are active in the 8-core system, and its throughput saturates. However, the accuracy of both systems are very close. This is because oversubscription randomly schedules the

lexicon thread to be processed, the randomness of the execution order remains.

3.5 Conclusion

We proposed a parallel sentence confabulation framework inspired by concurrent association phase of human cognitive processing. The proposed framework exploits multi-threading to build a parallel structure to process lexicons in sentences. We optimized the proposed framework by using intermittent pruning, to overcome the compute speed overhead due to cache coherence and this also improves the accuracy performance of the framework. Compared to request level parallelization, the proposed finer grained parallelization reduces the recall time by up to 93.4%, and increase the sentence accuracy by 5%.

CHAPTER 4

HIGHLY-SCALABLE DEEP CONVOLUTIONAL NEURAL NETWORK USING STOCHASTIC COMPUTING

4.1 Introduction

Deep learning (or deep structured learning) has emerged as a new area of machine learning research, which enables a system to automatically learn complex information and extract representations at multiple levels of abstraction [28]. *Deep Convolutional Neural Network (DCNN)* is recognized as one of the most promising types of artificial neural networks taking advantage of deep learning and has become the dominant approach for almost all recognition and detection tasks [67]. Specifically, DCNN has achieved significant success in a wide range of machine learning applications, such as image classification [112], natural language processing [22], speech recognition [108], and video classification [57].

Currently, the high-performance servers are usually required for executing software-based DCNNs since software-based DCNN implementations involve a large amount of computations to achieve outstanding performance. However, the high-performance servers incur high power (energy) consumption and large hardware cost, making them unsuitable for applications in embedded and mobile IoT devices that require low-power consumption. These applications play an increasingly important role in our everyday life and exhibit a notable trend of being "smart". To enable DCNNs in these application with low-power and low-hardware cost, the highly-parallel and specialized hardware has been designed using General-Purpose Graphics Processing Units (GPGPUs), Field-

Programmable Gate Array (FPGAs), and Application-Specific Integrated Circuit (ASICs) [65, 114, 138, 89, 5, 118, 3, 116, 49, 130, 17, 43]. Despite the performance and power (energy) efficiency achieved, a large margin of improvement still exists due to the inherent inefficiency in implementing DCNNs using conventional computing methods or using general-purpose computing devices [53, 58].

We consider *Stochastic Computing (SC)* as a novel computing paradigm to provide significantly low hardware footprint with high energy efficiency and scalability. In SC, a probability number is represented using a bit-stream [33], therefore, the key arithmetic operations such as multiplications and additions can be implemented as simple as AND gates and multiplexers (MUX), respectively [12]. Due to these features, SC has the potential to implement DCNNs with significantly reduced hardware resources and high power (energy) efficiency. Considering the large number of multiplications and additions in DCNN, achieving the efficient DCNN implementation using SC requires the exploration of a large design space.

In this chapter, we propose *SC-DCNN*, the first comprehensive design and optimization framework of SC-based DCNNs, using a bottom-up approach. The proposed SC-DCNN fully utilizes the advantages of SC and achieves remarkably low hardware footprint, low power and energy consumption, while maintaining high network accuracy.

We then present *HEIF* (i.e. **H**ighly **E**fficient **I**nference **F**ramework) with broad applications including (but not limited to) *LeNet5 and AlexNet*, that achieves high energy efficiency and low area/hardware cost. The HEIF overcome the limitations in SC-DCNN in two aspects, *1)* SC-DCNN suffers from the

degraded overall accuracy because it utilizes the easy-to-implement hyperbolic tangent (tanh) function instead of ReLU function. *2)* SC-DCNN is not sufficiently optimized which leads to the difficulty to maintain the high application-level accuracy due to the stochastic nature of SC components.

This chapter made the following key contributions:

1. **Applying SC to DCNNs.** We are the *first* (to the best of our knowledge) to apply SC to DCNNs. This approach is motivated by *1)* the potential of SC as a computing paradigm to provide low hardware footprint with high energy efficiency and scalability; and *2)* the need to implement DCNNs in the embedded and mobile IoT devices.
2. **Basic function blocks and hardware-oriented max pooling.** We propose the design of *function blocks* that perform the basic operations in DCNN. Specifically, we present a novel hardware-oriented max pooling design for efficiently implementing (approximate) max pooling in SC domain. We propose the *first* (to the best of our knowledge) SC-based ReLU activation function and corresponding optimization to catch up with recent software advances and mitigate degradation on application-level accuracy. The pros and cons of different types of function blocks are also thoroughly investigated.
3. **Basic function block redesign.** We re-design the *Approximate Parallel Counter (APC)* proposed in [59] and optimize stochastic multiplication, which is utilized in the inner product calculations of DCNN, to achieve a smaller footprint and higher energy efficiency without sacrificing any precision.

4. **Joint optimizations for feature extraction blocks.** We propose four optimized designs of *feature extraction blocks* which are in charge of extracting features from input feature maps. The function blocks inside the feature extraction block are *jointly optimized* through both analysis and experiments with respect to input bit-stream length, function block structure, and function block compatibilities.

5. **Weight storage schemes.** The area and power (energy) consumption of weight storage are reduced by comprehensive techniques, including efficient filter-aware SRAM sharing, effective weight storage methods, and clustering method considering the effects of hardware imprecision on the overall application-level accuracy.

6. **Overall network-level optimization.** We conduct holistic optimizations of the overall SC-DCNN and HEIF architecture with the cascade structural connection of function blocks, the pipelining technique, and the bit-stream length reduction. It significantly improves the energy efficiency without compromising application-level accuracy requirements.

7. **Remarkably low hardware footprint and low power (energy) consumption.** Overall, the proposed SC-DCNN achieves the *lowest* hardware cost and energy consumption in implementing LeNet5 compared with reference works. Moreover, HEIF could achieve very high energy efficiency of $1.2M$ Images/J and $1.3M$ Images/J, and high throughput of $3.2M$ Images/s and $2.5M$ Images/s, along with very small area of 22.9 mm^2 and 24.7 mm^2 on LeNet-5 and AlexNet, respectively. HEIF outperforms SC-DCNN by throughput of 4.1×, by area efficiency of up to 6.5× and achieves up to 5.6× energy improvement.

4.2 Related Works

Authors in [65, 114, 63, 54] leverage the parallel computing and storage resources in GPUs for efficient DCNN implementations. FPGA-based acceleration [138, 89] is another promising path towards the hardware implementation of DCNNs due to the programmable logic, high degree of parallelism and short develop round. However, the GPU and FPGA-based implementations still exhibit a large margin of performance enhancement and power reduction. It is because *1)* GPUs and FPGAs are general-purpose computing devices not specifically optimized for executing DCNNs, and *ii)* the relatively limited signal routing resources in such general platforms restricts the performance of DCNNs which typically exhibit high inter-neuron communication requirements.

ASIC-based implementations of DCNNs have been recently exploited to overcome the limitations of general-purpose computing devices. Three representative state-of-the-art works on ASIC-based implementations are Eyeriss [16], EIE [43], and the DianNao family, including DianNao [15], DaDianNao [17], ShiDianNao [30], and PuDianNao [84]. Eyeriss [16] is an energy-efficient reconfigurable accelerator for the large CNNs with various shapes. EIE [43] focuses specifically on the fully-connected layers of DCNN and achieves high throughput and energy efficiency. The DianNao family [15]-[84] is the series of hardware accelerators designed for a variety of machine learning tasks (especially the large-scale DCNNs) with a special emphasis on the impact of memory on accelerator design, performance, and energy.

To significantly reduce hardware cost and improve energy efficiency and scalability, novel computing paradigms need to be investigated. We consider

SC-based implementation of neural network an attractive candidate to meet the stringent requirements and facilitate the widespread of DCNNs in embedded and mobile IoT devices. Although not focusing on deep learning, [110] proposes the design of a neurochip using stochastic logic. [53] utilizes stochastic logic to implement a radial basis function-based neural network. In addition, a neuron design with SC for deep belief network was presented in [58]. Despite the previous application of SC, there is no existing work that investigates comprehensive designs and optimizations of SC-based hardware DCNNs including both computation blocks and weight storing methods.

4.3 Overview of DCNN Architecture and Stochastic Computing

4.3.1 DCNN Architecture Decomposition

The concept of "neuron" is widely used in the software/algorithm domain. In the context of DCNNs, a neuron consists of one or multiple basic operations. In this chapter, we focus on the basic operations in hardware designs and optimizations, including: inner product, pooling, and activation. The corresponding SC-based designs of these fundamental operations are termed *function blocks*. Figure 4.1 illustrates the behaviors of function blocks, where x_i's in Figure 4.1(a) represent the elements in a receptive filed, and w_i's represent the elements in a filter. Figure 4.1(b) shows the average pooling and max pooling function blocks. Figure 4.1(c) shows the activation function block (e.g. hyperbolic tangent function). The composition of an inner product block, a pooling

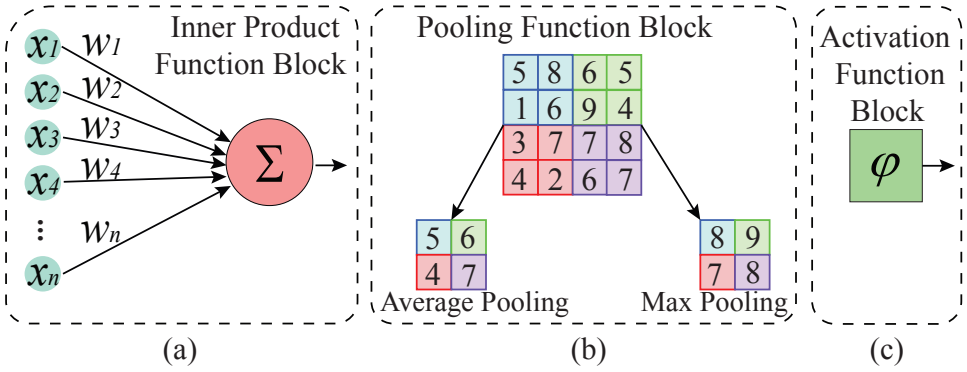


Figure 4.1: Three types of basic operations (function blocks) in DCNN. (a) Inner Product, (b) pooling, and (c) activation.

block, and an activation function block is referred to as the *feature extraction block*, which extracts features from feature maps.

4.3.2 Stochastic Computing (SC)

Stochastic Computing (SC) is a paradigm that represents a probabilistic number by counting the number of ones in a bit-stream. For instance, the bit-stream 0100110100 contains four ones in a ten-bit stream, thus it represents $P(X = 1) = 4/10 = 0.4$. In addition to this unipolar encoding format, SC can also represent numbers in the range of [-1, 1] using the bipolar encoding format. In this scenario, a real number x is processed by $P(X = 1) = (x + 1)/2$, thus 0.4 can be represented by 1011011101, as $P(X = 1) = (0.4 + 1)/2 = 7/10$. To represent a number beyond the range [0, 1] using unipolar format or beyond [-1, 1] using bipolar format, a pre-scaling operation [135] can be used. Furthermore, since the bit-streams are randomly generated with stochastic number generators (SNGs), the randomness and length of the bit-streams can significantly affect the calculation accuracy [105]. Therefore, the efficient utilization of SNGs and the trade-off between the bit-stream length (i.e. the accuracy) and the resource consumption

need to be carefully taken into consideration.

Compared to the conventional binary computing, the major advantage of stochastic computing is the significantly lower hardware cost for a large category of arithmetic calculations. The abundant area budget offers immense design space in optimizing hardware performance via exploring the tradeoffs between the area and other metrics, such as power, latency, and parallelism degree. Therefore, SC is an interesting and promising approach to implementing large-scale DCNNs.

Multiplication. Figure 4.2 shows the basic multiplication components in SC domain. A unipolar multiplication can be performed by an AND gate since $P(A \cdot B = 1) = P(A = 1)P(B = 1)$ (assuming independence of two random variables), and a bipolar multiplication is performed by means of a XNOR gate since $c = 2P(C = 1) - 1 = 2(P(A = 1)P(B = 1) + P(A = 0)P(B = 0)) - 1 = (2P(A = 1) - 1)(2P(B = 1) - 1) = ab$.

Addition. We consider four popular stochastic addition methods for SC-DCNNs. *1)* OR gate (Figure 4.3 (a)). It is the simplest method that consumes the least hardware footprint to perform an addition, but this method introduces considerable accuracy loss because the computation "logic 1 OR logic 1" only generates a single logic 1. *2)* Multiplexer (Figure 4.3 (b)). It uses a multiplexer, which is the most popular way to perform additions in either the unipolar or the bipolar format [12]. For example, a bipolar addition is performed as $c = 2P(C = 1) - 1 = 2(1/2P(A = 1) + 1/2P(B = 1)) - 1 = 1/2(2P(A = 1) - 1) + (2P(B = 1) - 1)) = 1/2(a + b)$. *3)* Approximate parallel counter (APC) [59] (Figure 4.3 (c)). It counts the number of 1s in the inputs and represents the result with a binary number. This method consumes fewer logic gates compared with the

conventional accumulative parallel counter [59, 95]. *4)* Two-line representation of a stochastic number [121] (Figure 4.3 (d)). This representation consists of a magnitude stream $M(X)$ and a sign stream $S(X)$, in which 1 represents a negative bit and 0 represents a positive bit. The value of the represented stochastic number is calculated by: $x = \frac{1}{L}\sum_{i=0}^{L-1}(1-2S(X_i))M(X_i)$, where L is the length of the bit-stream. As an example, -0.5 can be represented by $M(-0.5) : 10110001$ and $S(-0.5) : 11111111$. Figure 4.3 (d) illustrates the structure of the two-line representation-based adder. The summation of A_i (consisting of $S(A_i)$ and $M(A_i)$) and B_i are sent to a truth table, then the truth table and the counter together determine the carry bit and C_i. The truth table can be found in [121].

Hyperbolic Tangent (tanh). The tanh function is highly suitable for SC-based implementations because *i)* it can be easily implemented with a K-state finite state machine (FSM) in the SC domain [12, 70] and costs less hardware when compared to the piecewise linear approximation (PLAN)-based implementation [64] in conventional computing domain; and *ii)* replacing ReLU or sigmoid function by tanh function does not cause accuracy loss in DCNN [63]. Therefore, we choose tanh as the activation function in SC-DCNNs in our design. The diagram of the FSM is shown in Figure 4.4. It reads the input bit-stream bit by bit, when the current input bit is one, it moves to the next state, otherwise it moves to the previous state. It outputs a 0 when the current state is on the left half of the diagram, otherwise it outputs a 1. The value calculated by the FSM satisfies $Stanh(K, x) \cong tanh(\frac{K}{2}x)$, where $Stanh$ denotes stochastic tanh.

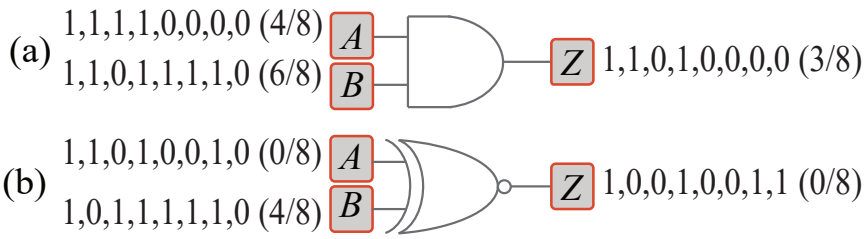


Figure 4.2: Stochastic multiplication. (a) Unipolar multiplication and (b) bipolar multiplication.

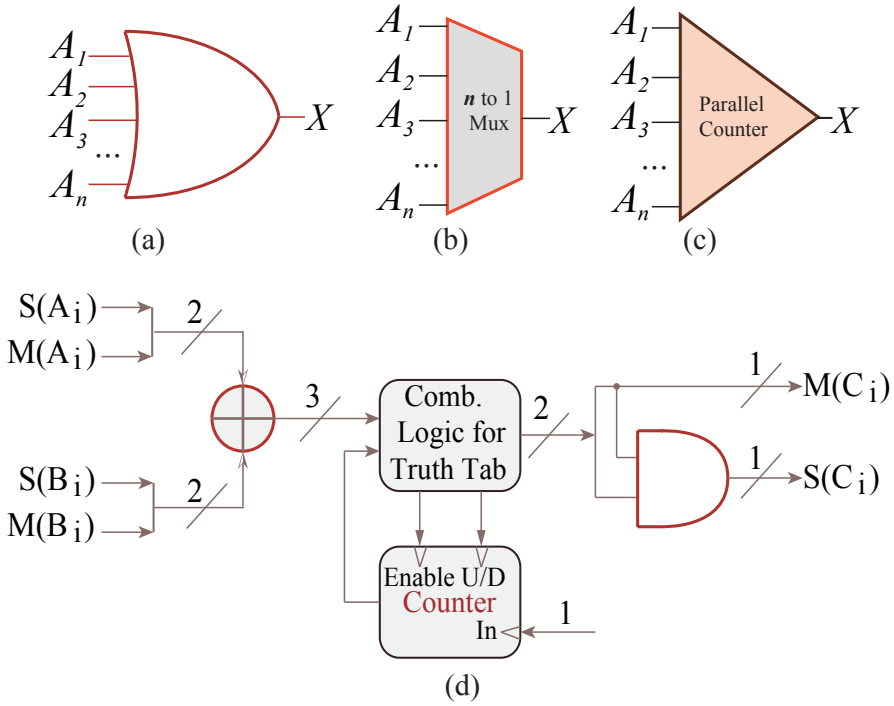


Figure 4.3: Stochastic addition. (a) OR gate, (b) MUX, (c) APC, and (d) two-line representation-based adder.

4.3.3 Application-level vs. Hardware Accuracy

The overall network accuracy (e.g., the overall recognition or classification rates) is one of the key optimization goals of the SC-based hardware DCNN. Due to the inherent stochastic nature, the SC-based function blocks and feature extraction blocks exhibit certain degree of hardware inaccuracy. The network accuracy and hardware accuracy are different but correlated, — the high accuracy in each

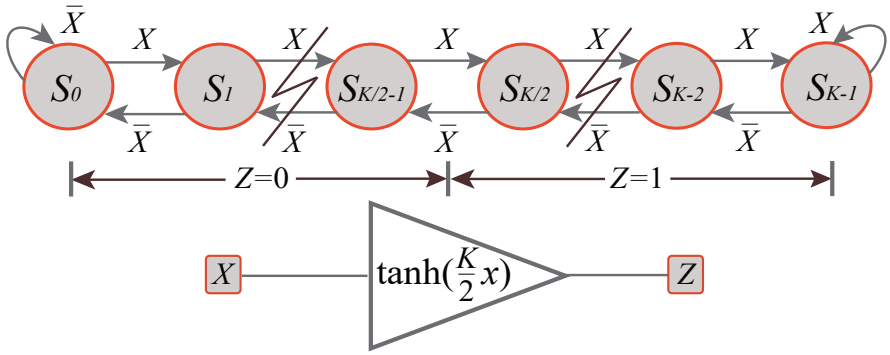


Figure 4.4: Stochastic hyperbolic tangent.

function block will likely lead to a high overall network accuracy. Therefore, the hardware accuracy can be optimized in the design of SC-based function blocks and feature extraction blocks.

4.4 Design for Function Blocks and Feature Extraction Blocks

In this section, we first perform comprehensive designs and optimizations in order to derive the most efficient SC-based implementations for function blocks, including inner product/convolution, pooling, and activation function. The goal is to reduce power, energy and hardware resource while still maintaining high accuracy. Based on the detailed analysis of pros and cons of each basic function block design, we propose the designs of feature extraction blocks in SC-DCNN and HEIF through both analysis and experiments.

4.4.1 Inner Product/Convolution Block Design

As shown in Figure 4.1 (a), an inner product/convolution block in DCNNs is composed of multiplication and addition operations. In DCNNs, inputs are

distributed in the range of [-1, 1], we adopt the bipolar multiplication implementation (i.e. XNOR gate) for the inner product block design. The summation of all products is performed by the adder(s). As discussed in Section 4.3.2, the addition operation has different implementations. To find the best option for DCNN, we replace the summation unit in Figure 4.1 (a) with the four different adder implementations shown in Figure 4.3.

OR Gate-Based Inner Product Block Design. Performing addition using OR gate is straightforward. For example, $\frac{3}{8} + \frac{4}{8}$ can be performed by "00100101 OR 11001010", which generates "11101111" ($\frac{7}{8}$). However, if first input bit-stream is changed to "10011000", the output of OR gate becomes "11011010" ($\frac{5}{8}$). Such inaccuracy is introduced by the multiple representations of the same value in SC domain and the fact that the simple "logic 1 OR logic 1" cannot tolerate such variance. To reduce the accuracy loss, the input streams should be pre-scaled to ensure that there are only very few 1's in the bit-streams. For the unipolar format bit-streams, the scaling can be easily done by dividing the original number by a scaling factor. Nevertheless, in the scenario of bipolar encoding format, there are about 50% 1's in the bit-stream when the original value is close to 0. It renders the scaling ineffective in reducing the number of 1's in the bit-stream.

Table 4.1 shows the average inaccuracy (absolute error) of OR gate-based inner product block with different input sizes, in which the bit-stream length is fixed at 1024 and all average inaccuracy values are obtained with the most suitable pre-scaling. The experimental results suggest that the accuracy of unipolar calculations may be acceptable, but the accuracy is too low for bipolar calculations and becomes even worse with the increased input size. Since it is almost

Table 4.1: Absolute Errors of OR Gate-Based Inner Product Block

Input Size	16	32	64
Unipolar inputs	0.47	0.66	1.29
Bipolar inputs	1.54	1.70	2.3

Table 4.2: Absolute Errors of MUX-Based Inner Product Block

Input size	Bit stream length			
	512	1024	2048	4096
16	0.54	0.39	0.28	0.21
32	1.18	0.77	0.56	0.38
64	2.35	1.58	1.19	0.79

impossible to have only positive input values and weights, we conclude that the OR gate-based inner product block is not appropriate for SC-DCNNs.

MUX-Based Inner Product Block Design. According to [12], an n-to-1 MUX can sum all inputs together and generate an output with a scaling down factor $\frac{1}{n}$. Since only one bit is selected among all inputs to that MUX at one time, the probability of each input to be selected is $\frac{1}{n}$. The selection signal is controlled by a randomly generated natural number between 1 and n. Taking Figure 4.1 (a) as an example, the output of the summation unit (MUX) is $\frac{1}{n}(x_0w_0 + ... + x_{n-1}w_{n-1})$.

Table 4.2 shows the average inaccuracy (absolute error) of the MUX-based inner product block measured with different input sizes and bit-stream lengths. The accuracy loss of MUX-based block is mainly caused by the fact that only one input is selected at one time, and all the other inputs are not used. The increasing input size causes accuracy reduction because more bits are dropped. However, we see that sufficiently good accuracy can still be obtained by increasing the bit-stream length.

APC-Based Inner Product Block. The structure of a 16-bit APC is shown in Figure 4.5. $A_0 - A_7$ and $B_0 - B_7$ are the outputs of XNOR gates, i.e., the products

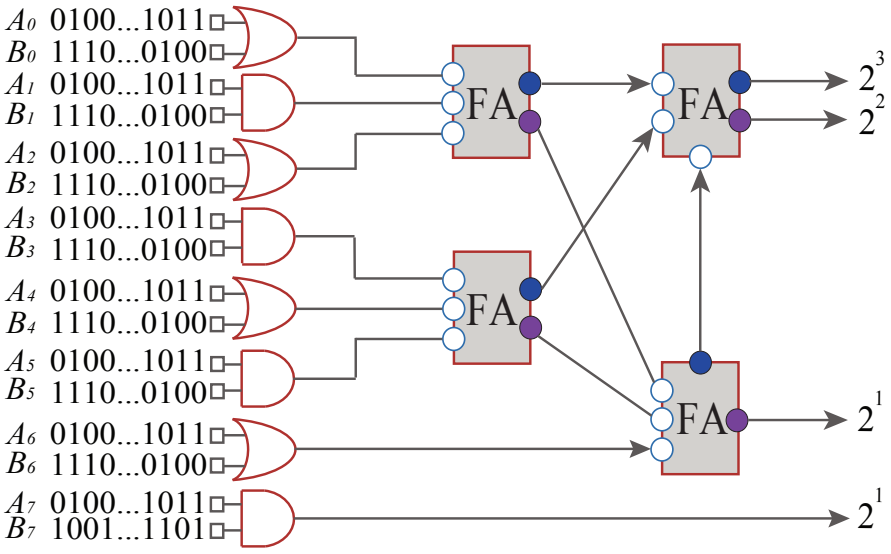


Figure 4.5: 16-bit Approximate Parallel Counter.

of inputs x_i's and weights w_i's. Suppose the number of inputs is n and the length of a bit-stream is m, then the products of x_i's and w_i's can be represented by a bit-matrix of size $n \times m$. The function of the APC is to count the number of ones in each column and represent the result in the binary format. Therefore, the number of outputs is $\log_2 n$. Taking the 16-bit APC as an example, the output should be 4-bit to represent a number between 0 - 16. However, it is worth noting that the weight of the least significant bit is 2^1 rather than 2^0 to represent 16. Therefore, the output of the APC is a bit-matrix with size of $\log_2 n \times m$.

From Table 4.3, we see that the APC-based inner product block only results in less than 1% accuracy degradation when compared with the conventional accumulative parallel counter, but it can achieve about 40% reduction of gate count [59]. This observation demonstrates the significant advantage of implementing efficient inner product block using APC-based method, in terms of power, energy, and hardware resource.

Two-Line Representation-Based Inner Product Block. The two-line

Table 4.3: Relative Errors of the APC-Based Compared with the Conventional Parallel Counter-Based Inner Product Blocks

Input size	Bit stream length			
	128	256	384	512
16	1.01%	0.87%	0.88%	0.84%
32	0.70%	0.61%	0.58%	0.57%
64	0.49%	0.44%	0.44%	0.42%

representation-based SC scheme [121] can be used to construct a non-scaled adder. Figure 4.3 (d) illustrates the structure of a two-line representation-based adder. Since A_i, B_i, and C_i are bounded as the element of $\{-1, 0, 1\}$, a carry bit may be missed. Therefore, a three-state counter is used here to store the positive or negative carry bit.

However, there are two limitations for the two-line representation-based inner product block in hardware DCNNs: *i)* An inner product block generally has more than two inputs, the overflow may often occur in the two-line representation-based inner product calculation due to its non-scaling characteristics. This leads to significant accuracy loss; and *ii)* the area overhead is too high compared with other inner product implementation methods.

4.4.2 Pooling Block Designs

Pooling (or down-sampling) operations are performed by pooling function blocks in DCNNs to significantly reduce *i)* inter-layer connections; and *ii)* the number of parameters and computations in the network, meanwhile maintaining the translation invariance of the extracted features [1]. Average pooling and max pooling are two widely used pooling strategies. Average pooling is straightforward to implement in SC domain, while max pooling, which exhibits

higher performance in general, requires more hardware resources. In order to overcome this challenge, we propose a novel hardware-oriented max pooling design with high performance and amenable to SC-based implementation.

Average Pooling Block Design. Figure 4.1 (b) shows how the feature map is average pooled with 2×2 filters. Since average pooling calculates the mean value of entries in a small matrix, the inherent down-scaling property of the MUX can be utilized. Therefore, the average pooling can be performed by the structure shown in Figure 4.3 (b) with low hardware cost.

Hardware-Oriented Max Pooling Block Design. The max pooling operation has been recently shown to provide higher performance in practice compared with the average pooling operation [1]. However, in SC domain, we can find out the bit-stream with the maximum value among four candidates only after counting the total number of 1's through the whole bit-streams, which inevitably incurs long latency and considerable energy consumption.

To mitigate the cost, we propose a novel SC-based hardware-oriented max pooling scheme. The insight is that once a set of bit-streams are sliced into segments, the globally largest bit-stream (among the four candidates) has the highest probability to be the locally largest one in each set of bit-stream segments. This is because all 1's are randomly distributed in the stochastic bit-streams.

Consider the input bit-streams of the hardware-oriented max pooling block as a bit matrix. Suppose there are four bit-streams, and each has m bits, thus the size of the bit matrix is $4 \times m$. Then the bit matrix is evenly sliced into small matrices whose size are $c \times m$ (i.e., each bit-stream is evenly sliced into segments whose length are c). Since the bit-streams are randomly generated,

ideally, the largest row (segment) among the four rows in each small matrix is also the largest row of the global matrix. To determine the largest row in a small matrix, the number of 1s are counted in all rows in a small matrix in parallel. The maximum counted result determines the next c-bit row that is sent to the output of the pooling block. In another word, the currently selected c-bit segment is determined by the counted results of the previous matrix. To reduce latency, the c-bit segment from the first small matrix is randomly chosen. This strategy incurs zero extra latency but only causes a *negligible accuracy loss* when c is properly selected.

Figure 4.6 illustrates the structure of the hardware-oriented max pooling block, where the output from *max_output* approximately is equal to the largest bit-stream. The four input bit-streams sent to the multiplexer are also connected to four counters, and the outputs of the counters are connected to a comparator to determine the largest segment. The output of the comparator is used to control the selection of the four-to-one MUX. Suppose in the previous small bit matrix, the second row is the largest, then MUX will output the second row of the current small matrix as the current c-bit output.

Table 4.4 shows the result deviations of the hardware-oriented max pooling design compared with the software-based max pooling implementation. The length of a bit-stream segment is 16. In general, the proposed pooling block can provide a sufficiently accurate result even with large input size.

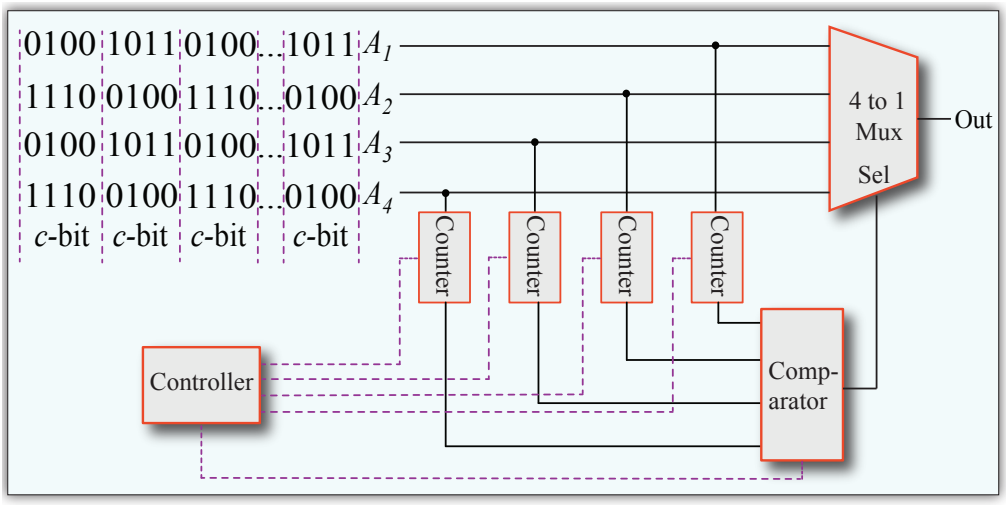


Figure 4.6: The Proposed Hardware-Oriented Max Pooling.

Table 4.4: Relative Result Deviation of Hardware-Oriented Max Pooling Block Compared with Software-Based Max Pooling

Input size	Bit-stream length			
	128	256	384	512
4	0.127	0.081	0.066	0.059
9	0.147	0.099	0.086	0.074
16	0.166	0.108	0.097	0.086

4.4.3 Activation Function Block Designs

Stanh. [12] proposed a K-state FSM-based design (i.e., Stanh) in the SC domain for implementing the tanh function and describes the relationship between Stanh and tanh as $Stanh(K, x) \cong tanh(\frac{K}{2}x)$. When the input stream x is distributed in the range [-1, 1] (i.e. $\frac{K}{2}x$ is distributed in the range $[-\frac{K}{2}, \frac{K}{2}]$), this equation works well, and higher accuracy can be achieved with the increased state number K.

However, $Stanh$ cannot be applied directly in SC-DCNN for three reasons. First, as shown in Figure 4.7 and Table 4.5 (with bit-stream length fixed at 8192), when the input variable of Stanh (i.e. $\frac{K}{2}x$) is distributed in the range [-1, 1],

Table 4.5: The Relationship Between State Number and Relative Inaccuracy of Stanh

State Number	8	10	12	14	16	18	20
Relative Inaccuracy (%)	10.06	8.27	7.43	7.36	7.51	8.07	8.55

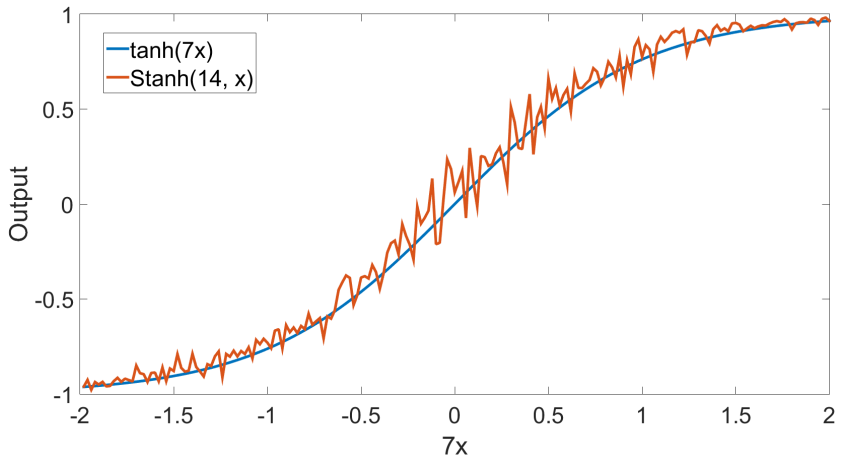


Figure 4.7: Output comparison of Stanh vs tanh.

the inaccuracy is quite notable and is not suppressed with the increasing of K. Second, the equation works well when x is precisely represented. However, when the bit-stream is not impractically long (less than 2^{16} according to our experiments), the equation should be adjusted with a consideration of bit-stream length. Third, in practice, we usually need to proactively down-scale the inputs since a bipolar stochastic number cannot reach beyond the range [-1, 1]. Moreover, the stochastic number may be sometimes passively down-scaled by certain components, such as a MUX-based adder or an average pooling block [81, 79]. Therefore, a *scaling-back* process is imperative to obtain an accurate result. Based on the these reasons, the design of Stanh needs to be optimized together with other function blocks to achieve high accuracy for different bit-stream lengths and meanwhile provide a scaling-back function. More details are discussed in Section 4.4.4.

Btanh. Btanh is specifically designed for the APC-based adder to perform a

scaled hyperbolic tangent function. Instead of using FSM, a saturated up/down counter is used to convert the binary outputs of the APC-based adder back to a bit-stream. The implementation details and how to determine the number of states can be found in [58].

ReLU. Rectified Linear Unit (ReLU) has become the most popular activation function in state-of-the-art DCNNs, however, only hyperbolic tangent/sigmoid functions have been implemented in the SC domain in previous works [12, 60]. Therefore, it is important to have the design of SC-based ReLU block [75] in order to accommodate the SC technique in the state-of-the-art large-scale DCNNs, such as AlexNet [63] for ImageNet applications.

The mathematical expression of ReLU is $f(x) = max(0, x)$, i.e., when input x is less than 0, the activation result is 0, otherwise the activation result is x itself. This characteristic of ReLU gives rise to a challenge for SC-based designs. Since x is represented by a stochastic bit-stream in SC with length m, we can only intuitively determine its sign and value through a counter using m clock cycles. This straightforward implementation of ReLU function in SC domain undoubtedly leads to a significant extra delay and energy overhead. On the other hand, the bit-stream-based representation in SC restricts the number it represents within the range [-1, 1], and as a result, the output of SC-based ReLU block should be clipped to 1. The clipped ReLU in the SC domain is expressed as $f(x) = min(max(0, x), 1)$.

Figure 4.8 illustrates the proposed architecture of SC-based ReLU. The input of SC-based ReLU is accumulated, and the accumulation result is compared with a reference number (half of the passed clock cycles). The comparator output is used as an input and also the control signal of the multiplexer. If the

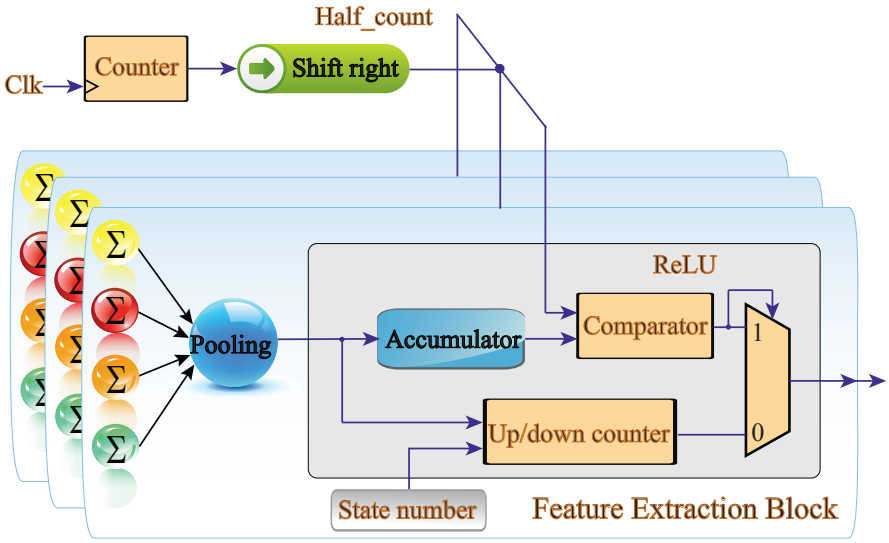


Figure 4.8: Diagram of the proposed ReLU block.

accumulation result is less than the reference number, the comparator outputs a 1 and is selected by the multiplexer as the output of SC-based ReLU block. Otherwise, the output is determined by the FSM inside the SC-based ReLU block. Please note that the proposed SC-based ReLU will not incur any extra latency.

The algorithm of the proposed SC-based ReLU is illustrated in Algorithm 3. Please note that the *Positive* signal is used to adjust the SC-based ReLU for different types of APCs. When the outputs of APCs represent the number of 1's among inputs, the normal logic is assigned to the output of SC-based ReLU. When the outputs of APCs represent the number of 0's, the inverted logic is assigned. The purpose is to make the output of SC-based ReLU (and thereby the whole FEB) not affected by the types of APCs.

Algorithm 3: Proposed SC-based ReLU hardware.

input : $BitMatrix$ is the output of the previous pooling block
each column of the matrix is a binary vector
$Cycle_{half}$ is the half of the passed clock cycles
S is the FSM state number
N is the input size of a feature extraction block
m is the length of a stochastic bit-stream
$Positive$ indicates whether APC's output represents the number of 1's

output: Z is a bit-stream output by ReLU

$S_{max} = S$; //upper bound of the state
$S_{half} = S/2$;
$State = S_{half}$; //$State$ is used to record the state history
$Accumulated = 0$; //to accumulate each column of $BitMatrix$
if $Positive == 1$ **then**
  $ActiveBit = 1$; $InactiveBit = 0$;
else
  $ActiveBit = 0$; $InactiveBit = 1$;
end
for $i++ < m$ **do**
  $BinaryVec = BitMatrix[:i]$; //current column
  $State = State + BinaryVec * 2 - N$; //update current state
  //accumulate current column of the input
  $Accumulated = Accumulated + BinaryVec$;
  if $Accumulated < Cycle_{half}$ **then**
    $Z[i] = ActiveBit$;
    //enforce the output of ReLU to be greater than or equal to 0,
    //otherwise the output is determined by the following FSM
  else
    if $State > S_{max}$ **then**
      $State = S_{max}$;
    else
      if $State < 0$ **then**
        $State = 0$;
      end
    end
    if $State < S_{half}$ **then**
      $Z[i] = ActiveBit$;
    else
      $Z[i] = InactiveBit$;
    end
  end
end

4.4.4 Feature Extraction Block Designs

In this section, we propose an optimized feature extraction blocks. Based on the previous analysis and results, we select several candidates for constructing feature extraction blocks shown in Figure 4.9, including: the MUX-based and APC-based inner product/convolution blocks, average pooling and hardware-oriented max pooling blocks, and Stanh and Btanh blocks.

In SC domain, the parameters such as input size, bit-stream length, and the inaccuracy introduced by the previous connected block can *collectively* affect the overall performance of the feature extraction block. Therefore, the isolated optimizations on each individual basic function block are insufficient to achieve the satisfactory performance for the entire feature extraction block. For example, the most important advantage of the APC-based inner product block is its high accuracy and thus the bit-stream length can be reduced. On the other side, the most important advantage of MUX-based inner product block is the low hardware cost and the accuracy can be improved by increasing the bit-stream length. Accordingly, to achieve good performance, we cannot simply compose these basic function blocks, instead, a series of joint optimizations are performed on each type of feature extraction block. Specifically, we attempt to fully making use of the advantages of each of the building blocks.

In the following discussion, we use MUX/APC to denote the MUX-based or APC-based inner product/convolution blocks; use Avg/Max to denote the average or hardware-oriented max pooling blocks; use Stanh/Btanh to denote the corresponding activation function blocks. A feature extraction block configuration is represented by choosing various combinations from the three components. For example, MUX-Avg-Stanh means that four MUX-based inner prod-

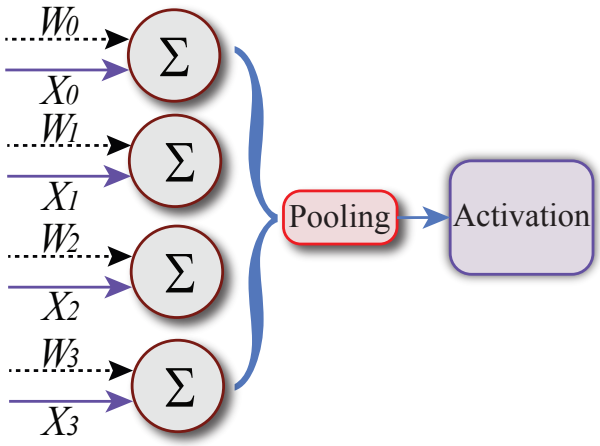


Figure 4.9: The structure of a feature extraction block.

uct blocks, one average pooling block, and one Stanh activation function block are cascade-connected to construct an instance of feature extraction block.

MUX-Avg-Stanh. As discussed in Section 4.4.3, when Stanh is used, the number of states needs to be carefully selected with a comprehensive consideration of the scaling factor, bit-stream length, and accuracy requirement. Below is the empirical equation that is extracted from our comprehensive experiments to obtain the approximately optimal state number K to achieve a high accuracy:

$$K = f(L, N) \approx 2 \times \log_2 N + \frac{\log_2 L \times N}{\alpha \times \log_2 N}, \tag{4.1}$$

where the nearest even number to the result calculated by the above equation is assigned to K, N is the input size, L is the bit-stream length, and empirical parameter $\alpha = 33.27$.

MUX-Max-Stanh. The hardware-oriented max pooling block shown in Figure 4.6 in most cases generates an output that is slightly less than the maximum value. In this design of feature extraction block, the inner products are all scaled down by a factor of n (n is the input size), and the subsequent scaling back func-

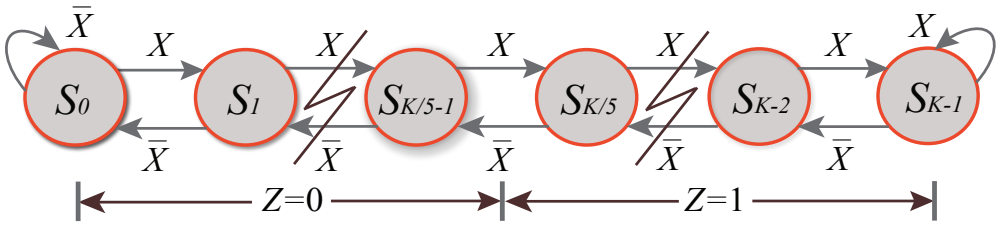


Figure 4.10: Structure of optimized Stanh for MUX-Max-Stanh.

tion of Stanh will enlarge the inaccuracy, especially when the positive/negative sign of the selected maximum inner product value is changed. For example, 505/1000 is a positive number, and 1% under-counting will lead the output of the hardware-oriented max pooling unit to be 495/1000, which is a negative number. Thereafter, the obtained output of Stanh may be -0.5, but the expected result should be 0.5. Therefore, the bit-stream has to be long enough to diminish the impact of under-counting, and the Stanh needs to be re-designed to fit the correct (expected) results. As shown in Figure 4.10, the re-designed FSM for Stanh will output 0 when the current state is at the left 1/5 of the diagram, otherwise it outputs a 1. The optimal state number K is calculated through the following empirical equation derived from experiments:

$$K = f(L, N) \approx 2 \times (\log_2 N + \log_2 L) - \frac{\alpha}{\log_2 N} - \frac{\beta}{\log_5 L}, \tag{4.2}$$

where the nearest even number to the result calculated by the above equation is assigned to K, N is the input size, L is the bit-stream length, $\alpha = 37$, and empirical parameter $\beta = 16.5$.

APC-Avg-Btanh. When the APC is used to construct the inner product block, conventional arithmetic calculation components, such as full adders and dividers, can be utilized to perform the averaging calculation, because the output of APC-based inner product block is a binary number. Since the design of

Btanh initially aims at directly connecting to the output of APC, and an average pooling block is now inserted between APC and Btanh, the original formula proposed in [58] for calculating the optimal state number of Btanh needs to be re-formulated as:

$$K = f(N) \approx \frac{N}{2}, \tag{4.3}$$

from our experiments. In this equation N is the input size, and the nearest even number to $\frac{N}{2}$ is assigned to K.

APC-Max-Btanh. Although the output of APC-based inner product block is a binary number, the conventional binary comparator cannot be directly used to perform max pooling. This is because the output sequence of APC-based inner product block is still a stochastic bit-stream. If the maximum binary number is selected at each time, the pooling output is always greater than the actual maximum inner product result. Instead, the proposed hardware-oriented max pooling design should be used here, and the counters should be replaced by accumulators for accumulating the binary numbers. Thanks to the high accuracy provided by accumulators in selecting the maximum inner product result, the original Btanh design presented in [58] can be directly used without adjustment.

Structural Exploration of Feature Extraction Blocks. Extended in [81], we also explored the alternative structure of FEB. In software-level design, a FEB of DCNN is formed by convolution neurons, pooling neurons and activation function in order.This is reasonable, because intuitively the order of pooling before activation can save 3/4 computation resources to do the activation. However, in hardware design, due to the cross-dependency of components and their effects on calculation precision, another arrangement of neurons (pooling after

Table 4.6: The designs of FEBs and corresponding optimization functions

No.	Design	Optimization Function	Parameters
1	MUX-Avg-Stanh	$K = f(L,N) \approx 2\log_2(N) + N\log_2(L)/\alpha\log_2(N)$	$\alpha = 33.27$
2	MUX-Stanh-Avg	$K = f(L,N) \approx \alpha\log_2 N + N\log_5 L/\beta\log_2 N$	$\alpha = 1.3$ $\beta = 8.74$
3	MUX-Max-Stanh	$K = f(L,N) \approx 2(\log_2 N + \log_2 L) - \alpha/\log_2 N - \beta/\log_5 L$	$\alpha = 37$ $\beta = 16.5$
4	MUX-Stanh-Max	$K = f(L,N) \approx -\gamma\sqrt{N}\log_2 N/L + \alpha\log_2 N + L/\beta\log_2 L$	$\alpha = 1, \beta = 5$ $\gamma = 5.2$
5	APC-Avg-Btanh	$K = g(N) \approx \alpha N$	$\alpha = 0.5$
6	APC-Btanh-avg		$\alpha = 2$
7	APC-Max-Btanh		
8	APC-Btanh-Max		

activation) in an FEB must be investigated [79]. In this section, we summarize two different arrangements. Eight designs of FEBs are analyzed and optimized listed in Table 4.6 by permuting MUX-based inner-product calculation block (in short, *MUX*), APC-based inner-product calculation block (*APC*) Average pooling (*Avg*), Max Pooling (Max), FSM-based Stanh (*Stanh*) and Binary-based Btanh (*Btanh*). For each design, we extract empirical functions by regression of exhaustive data samples, which is shown in Table 4.6. The enormous data samples are generated randomly and the expected outputs are regarded as golden references for the regression functions respectively. The functions are obtained by minimizing the difference between golden reference and the calculated value of a design with a specific K.

As shown in Table 4.6, given the input size N (and bit-stream length L for MUX based designs), an optimal number of K is the nearest even number of the result calculated by the corresponding function. Please note in No. 5, each set of output of the average pooling block is a binary number instead of stochastic

a bit-stream, the formula proposed by [58] to determine the optimal state number should be modified because of the existence of the average pooling block. Thus the parameter α is adjusted to 0.5. In No. 6 and 8, the Btanhs are directly connected with APCs just as they are originally designed, thus the formula proposed in [58] to determine the optimal state number is not modified. However, since we do not conduct the pre-scaling, the scaling factor s in [58] is 1, α is adjusted to 2. In No. 7, since the Max pooling block for binary numbers is accurate (implemented by accumulators), APC is regarded to connect with Btanh directly, then the same parameter of $\alpha = 2$ is used.

4.5 Further Optimization for Function Blocks and Feature Extraction Blocks in HEIF

4.5.1 Transmission Gate Based Multiplication

As discussed before, the multiplications are implemented with XNOR gates in bipolar SC. Generally, an XNOR gate costs at least sixteen transistors if it is implemented in static CMOS technology, and its simplest structure in gate-level is shown in Figure 4.11 (a). However, if the XNOR gate is implemented with transmission gates, only eight transistors are needed, leading to 50% savings in hardware. The main drawback of potential voltage degradation of a transmission gate does not cause latent errors for three reasons: *i)* the multiplication operations are only performed in the first sub-layer of each network layer, so any latent voltage degradation will not be significant; *ii)* the following APCs and activation blocks are implemented with static CMOS technology, so any minor

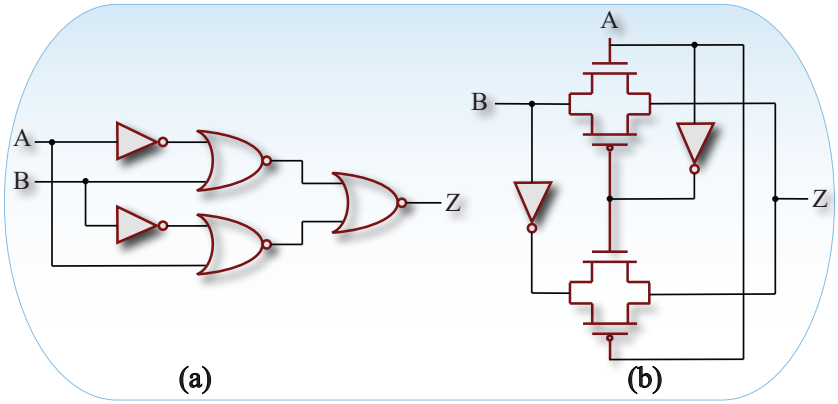


Figure 4.11: XNOR gate implementations. (a) Static CMOS design, (b) Transmission gate design.

voltage degradation introduced by transimission gates will be compensated; *iii)* SC itself is soft error resilient, i.e., a soft error at one single bit has a negligible impact on the whole bit-stream. The structure of the transmission gate based XNOR gate is illustrated in Figure 4.11 (b).

4.5.2 APC Optimization

Approximate Parallel Counter (APC) [59] has been designed for efficiently performing addition with a large number of inputs in SC domain. More specifically, it efficiently counts the total number of 1's in each "column" of the input stochastic bit-streams and the output is represented by a binary number, as shown in Figure 4.3 (c). The APC consists of two parts: approximate units (AU), implemented by a combination of simple two-input gates such as AND/OR gate, and an accurate Parallel Counter (PC) with size significantly reduced. The PC circuit consists of a network of full adders for precisely counting the total number of 1's among the input bit-streams. Although the literature [59] presented the operation principle of APC, there is no existing work targeting at

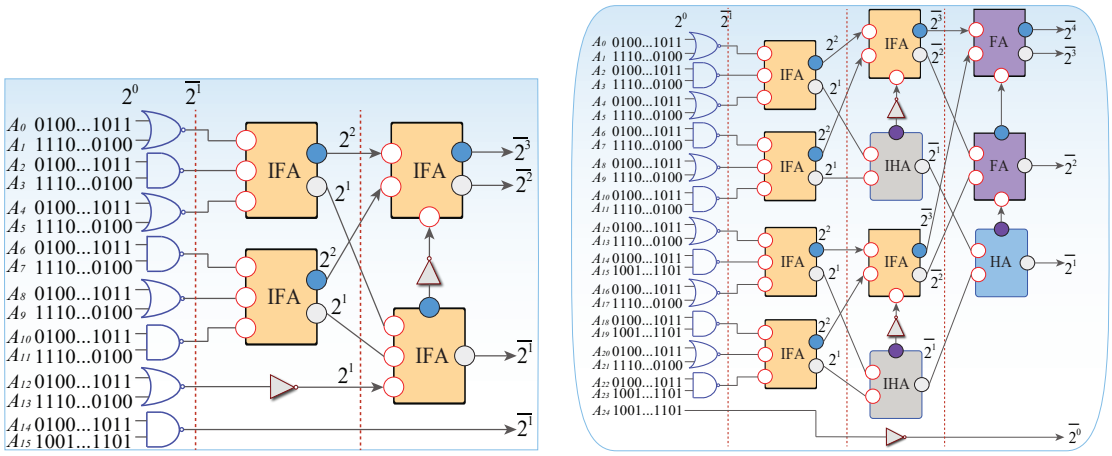


Figure 4.12: (a) Redesigned 16-input APC structure, (b) Redesigned 25-input APC structure.

optimization of the performance and energy efficiency. We mitigate this limitation by presenting a holistic optimization framework of APC in the following.

First, we investigate the design optimization of adder trees in PC to refine APC design. A conventional PC uses full adders and half adders to calculate the number of active inputs (the total number of 1's). Each adder reduces a set of three inputs (for full adder) or two inputs (for half adder) with weight 2^n into an output line with weight 2^n and another output with weight 2^{n+1}, which correspond to the summation and output carry, respectively. To reduce the area and power & energy consumption of APC, we design adder tree using *inverse mirror full adders* [128], i.e. mirror full adders without output inverters, whose outputs are the logical inversion of summation and carry out bits. Compared to a full adder synthesis results (from Synopsys Design Compiler) requiring 32 transistors, an inverse mirror full adder only costs 24 transistors. An adder tree design is available for the PC using inverse full adders, in which the odd layer (of adders) outputs the inverse values of summation and output carry, representing the number of inactive inputs (the total number of 0's). The results

are inverted back in the subsequent even layer of adders. Inspired by the same idea of using inverse logic, NAND/NOR gates can be used to construct the AU layer instead of AND/OR gates, to achieve further delay/area reductions.

Depending on the input size, the output of the proposed APC can either represent the number of 1's among the input bit-streams, or the number of 0's. Please note that the activation function needs to be modified if the APC output represents the number of 0's as discussed in the ReLU block design. As an example, the proposed 16-input APC design is shown in Figure 4.12 (a).

Next, we discuss the APC designs for input size that is not a power of two. An example of the proposed 25-input APC is shown in Figure 4.12 (b). Two modifications are needed compared with the previous case. First, *arithmetic inverse half adders* are required to calculate the number of inactive inputs (number of 0's among inputs). In addition, in this case, the final output of APC should be the non-inverted value compared with the inputs to the adder tree. In other words, if the inputs of adder tree represent the number of 0's (inactive inputs), then the APC output must also be the number of 0's. The reason is as follows: the summation of the number of 0's and the number of 1's should be equal to the input size (e.g., 25 as shown in Figure 4.12 (b)), whereas the inverse operation in adders assumes that their summation is 2^{N+1}, where N is the number of bits in the output binary number. Thus, the final layer of adder tree should use either adders or inverse adders to generate non-inverted results compared with the inputs.

Table 4.7 shows the comparison of inner-product blocks before and after optimization using the 1024-bit-stream. After applying the optimization on the inner-product blocks, the hardware performance in terms of clock period, area,

Table 4.7: Comparison of inner-product blocks before and after optimization using 1024-bit-stream.

Input Size	Approach	Optimization	Delay(ns)	Area(μm^2)	Energy (fJ)
16	SC	before	0.57	51.1	26.2
		after	0.49	26.6	22.8
	binary	-	2.02	2759.4	4775.4
32	SC	before	0.88	134.3	133.9
		after	0.78	82.7	122.3
	binary	-	2.15	5589.7	10618.2
64	SC	before	1.24	253.5	328.3
		after	1.12	147.1	294.3
	binary	-	2.38	11279.9	24095.1
128	SC	before	1.46	597.4	1069.7
		after	1.32	380.9	996.2
	binary	-	2.61	22664.7	53492.0
256	SC	before	1.78	1177.6	2652.3
		after	1.62	740.3	2450.6
	binary	-	2.84	45438.7	117201.1

and energy are all reduced, especially the area. Table 4.7 also demonstrates the advantages of SC over conventional binary computing. We can observe that the SC delay/area/energy are much smaller than binary's, this is because SC based inner-product blocks taking multiple input bit-streams in a parallel manner with simple gate logic, while the binary logic compute equivalent binary numbers bit by bit with complex gate logic.

4.5.3 Weight Storage Optimization

The main computing task of an inner-product block is to calculate the inner-products of x_i's and w_i's. x_i's are inputs of neurons, while w_i's are weights obtained during training, stored, and used in the hardware-based DCNNs. The number of weights is skyrocketing as the structure of DCNNs becomes much deeper and more complex. For example, LeNet-5 [68] includes 431k parame-

ters, AlexNet [63] has around 61M parameters, and VGG-16 [112] contains over 138M parameters. It is urgent to explore the techniques to store the tremendous parameters efficiently. In convolutional layers, weights are shared within filter domain, while in fully connected layers, the number of weights is enormous and independent. Thus the weights need to be either shared or reduced. The reduction of weights has been explored in many previous works such as [43, 31], however, weight sharing lacks the discussion. In this section, we present a simple weight reduction method and a clustering based weight sharing optimization. The methods presented can be combined with weight reduction/pruning methods in related works.

We use *Static random access memory (SRAM)* for weight storage due to its high reliability, high speed, and small area. The specifically optimized SRAM placement schemes and weight storage methods are imperative for further reductions of area and power (energy) consumptions.

Efficient Filter-Aware SRAM Sharing Scheme. Since all receptive fields of a feature map share one filter (a matrix of weights), all weights functionally can be separated into filter-based blocks, and each weight block is shared by all inner product/convolution blocks using the corresponding filter. Inspired by this fact, we propose an efficient filter-aware SRAM sharing scheme, with structure illustrated in Figure 4.13. The scheme divides the whole SRAM into small blocks to mimic filters. Besides, all inner product blocks can also be separated into feature map-based groups, where each group extracts a specific feature map. In this scheme, a local SRAM block is shared by all the inner product blocks of the corresponding group. The weights of the corresponding filter are stored in the local SRAM block of this group. This scheme significantly reduces the routing

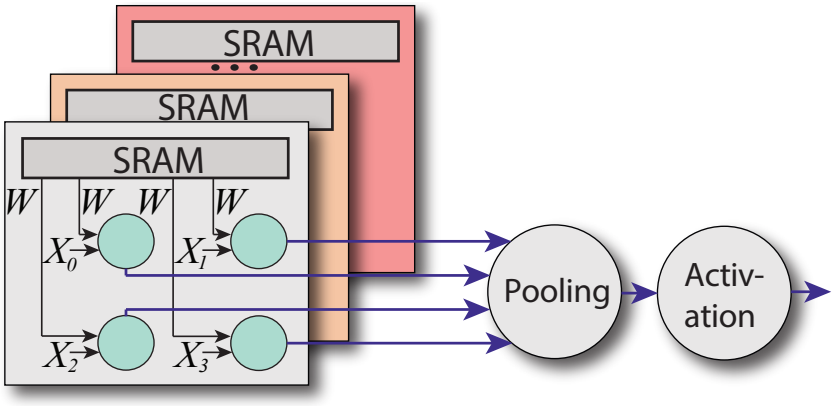


Figure 4.13: Filter-Aware SRAM Sharing Scheme.

overhead and wire delay.

Low Precision Weight Storage Method In general, DCNN will be trained with single floating point precision. Thus on hardware, up to 64-bit SRAM is needed for storing one weight value in the fixed point format to maintain its original high precision. This scheme can provide high accuracy as there is almost no information loss of weights. However, it also brings about high hardware consumptions in that the size of SRAM and its related read/write circuits is increasing with the increasing of precision of the stored weight values.

According to our software-level experiments, many least significant bits far from the decimal point only have a very limited impact on the overall application-level accuracy, thus the number of bits for weight representation in the SRAM block can be significantly reduced. We adopt a mapping equation that converts a weight in the real number format to the binary number stored in SRAM to eliminate the proper numbers of least significant bits. Suppose the weight value is x, and the number of bits to store a weight value in SRAM is w (which is defined as the *precision* of the represented weight value in this paper),

then the binary number to be stored for representing x is:

$$y = \frac{Int(\frac{x+1}{2} \times 2^w)}{2^w} \tag{4.4}$$

where $Int()$ means only keeping the integer part. Please note that the binary numbers stored in SRAMs are fed into efficient *Random Number Generators (RNGs)* to generate stochastic numbers at runtime. For instance, a 6-bit binary number can be used to generate a stochastic number with 1024-bit length through RNG. Hence, there is no need to store the entire 1024 bit stochastic number in SRAM. The overhead of RNGs is also taken into account in our experiments. Therefore, this weight storage method can significantly reduce the size of SRAMs and their read/write circuits through decreasing the precision. The area saving achieved by this method based on estimations from CACTI 5.3 [119] is 10.3×.

Weight Clustering. As mentioned before, a state-of-the-art DCNN contains millions of weights. A large amount of SRAM will be consumed for storing all these weights. In fact, many weight values can be rounded to a neighboring value without significant accuracy loss according to our experiments. Therefore, we investigate the k-means based weight clustering method that clusters all weights into clusters and rounds the weights in each cluster to one centroid value. Consequently, only a part of weight values need to be stored in SRAM. A multiplexer is used to select a weight from a SRAM block for each w_i of an inner product block, and the selection signals are stored in SRAM block as well. Suppose the filter size is $p \times p$, each weight occupies n bits, and storing one bit consumes t units hardware resources on average (including read/write circuits). Accordingly, the size of an SRAM block before clustering is $p^2 \times n \times t$. After clustering, only s weights are needed, thus the size of an SRAM block for storing weights is $s \times n \times t$. Since an inner product block has p^2 weight val-

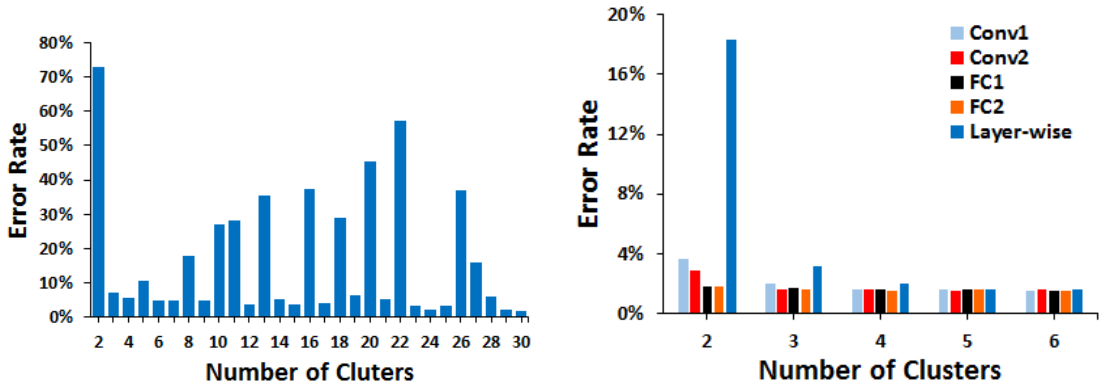


Figure 4.14: Application-level error rates for (a) clustering through all layers, (b) clustering within each layer and layer-wise clustering.

ues, p^2 multiplexers are required for each inner product block, and $p^2 \times log_2 s \times t$ units hardware resources are needed for storing the selection signals. Suppose the size of a multiplexer is m units hardware resources, and there are q inner product blocks for extracting a feature map. The area saving achieved by the clustering method is $p^2 \times n \times t - (s \times n \times t + p^2 \times log_2 s \times t + p^2 \times m \times q)$ for each feature map.

As shown in Figure 4.14(a), when the clustering is performed on all weights of the network, the application-level error rate vibrates obviously with the change of the clustering number, and the error rates in many cases exceed 10%. It indicates that the clustering on all weights is not practicable.

Then we perform the clustering on weights within each single layer to explore the application-level accuracy performance. As illustrated in Figure 4.14(b), when the clustering is performed on each layer from Conv1 to FC2, desirable application-level accuracy can be obtained while the number of clusters is more than three. Inspired by the experimental results, we investigate the application-level accuracy when the clustering is performed on the whole network but each layer is individually clustered (called *layer-wise*, and different

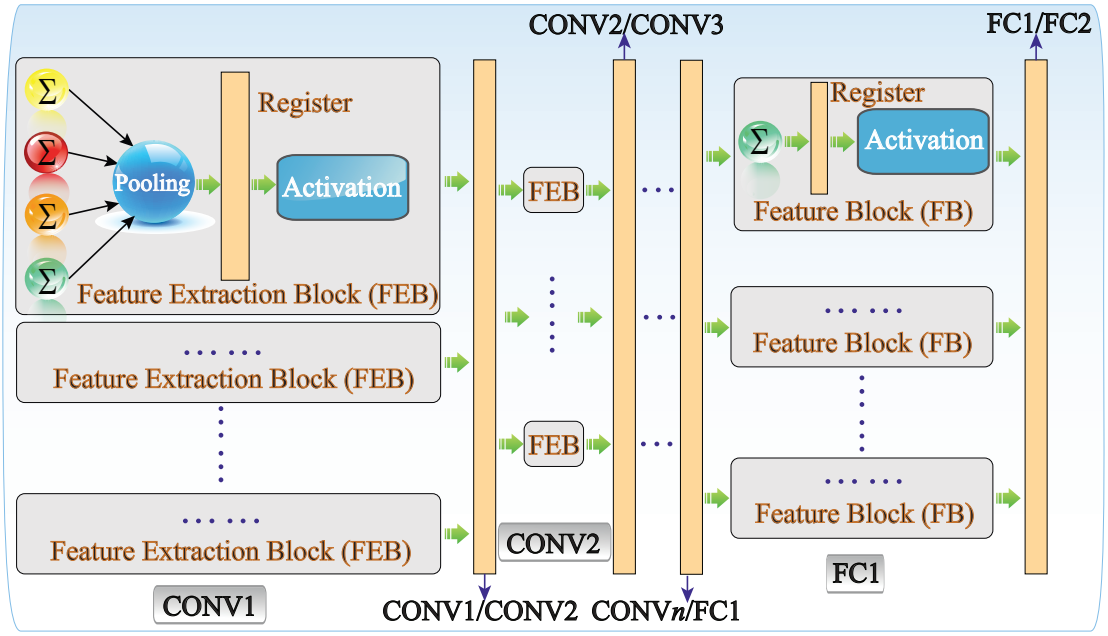


Figure 4.15: Two-tier pipeline design in the framework.

layers may have the different number of clusters). When all layers are individually clustered into five or more clusters, the application-level error rate is less than 2%.

4.5.4 Pipeline Based DCNN Optimizations

We propose a two-tier pipeline based network optimization as shown in Figure 4.15.

The first-tier pipeline is placed in between different convolutional and fully-connected layers, i.e., inserting DFFs between consecutive layers to hold the temporary results, which enables pipelining across the deep layers of DCNNs. The second-tier pipeline is placed within a layer which is inspired by [111]. More specifically, based on the delay results of inner product, pooling and ReLU blocks, we insert DFFs between the pooling unit and ReLU block in order to fur-

Table 4.8: Hardware performance of FEBs with the different input sizes using 1024-bit-stream w/ and w/o pipeline based optimization

Optimization	Pipelining				Non-pipelining			
Input size	16	32	64	128	16	32	64	128
Clock Period (ns)	1.74	1.82	2.08	2.16	2.2	2.51	2.67	2.79
Area (μm^2)	910.8	1162.4	1569.4	2305.2	904.4	1102.3	1453.9	2149.5
Power (μW)	556.6	771.2	928.4	1409.4	421.5	490.3	659.9	973.5
Energy (fJ)	968.4	1403.5	1931.0	3044.2	927.3	1230.8	1762.0	2716.1

ther reduce the system clock period. We place the pooling unit in the first stage. Because after pooling, the output size is reduced so that we can use less DFFs to save area & power & energy. To show the effectiveness of pipelining within a layer, we also evaluate the hardware costs for FEBs without pipelining in the right section in Table 4.8. Comparing the results in Table 4.8, we observe that the pipelining optimization significantly reduce the delay (clock period) by about 22% in average with slight area & power & energy increase by DFFs. An additional key optimization knob is the bit-stream length. A smaller bit-stream length in SC can almost improve the energy efficiency in a proportional manner. However, we must ensure that the overall application-level accuracy is maintained when the bit-stream length is reduced, and therefore, a joint optimization is required. In this procedure, we first optimize the accuracy of each function block, i.e., APC, max pooling and ReLU, to reduce the imprecision within an FEB. Furthermore, we conduct co-optimization through FEB to find the best configuration of each unit inside one FEB, in order to mitigate the propagation of imprecision and maintain the overall application-level accuracy.

4.6 Overall Evaluation

In this section, we present optimizations of feature extraction blocks along with comparison results with respect to accuracy, area/hardware footprint, power (energy) consumption, etc. Based on the results, we perform thorough optimizations on the overall SC-DCNN and HEIF to construct LeNet5 structure, which is one of the most well-known large-scale deep DCNN structure. The goal is to minimize area and power (energy) consumption while maintaining a high network accuracy. We present comprehensive comparison results among *i)* SC-DCNN and HEIF designs with different target network accuracy, and *ii)* existing hardware platforms. The hardware performance of the various SC-DCNN and HEIF implementations regarding area, path delay, power and energy consumption are obtained by: *i)* synthesizing with the 45nm Nangate Open Cell Library [2] using Synopsys Design Compiler; and *ii)* estimating using CACTI 5.3 [119] for the SRAM blocks. The key peripheral circuitries in the SC domain (e.g. the random number generators) are developed using the design in [60] and synthesized using Synopsys Design Compiler.

4.6.1 Optimization Results on Feature Extraction Blocks

Figure 4.16 shows the imprecisions of eight FEB designs. Each evaluation is given $10,000$ sets of random inputs ranging from -1 to 1 with different input sizes, i.e. 16, 32, 64, 128, 256 and bit-stream lengths, i.e. 256, 512, 1024. The average absolute error is used as the measure of imprecision, which is the mean of the absolute difference between the expected results and the observed results for the same test cases. We observed for each design that (i) as a bit-stream gets

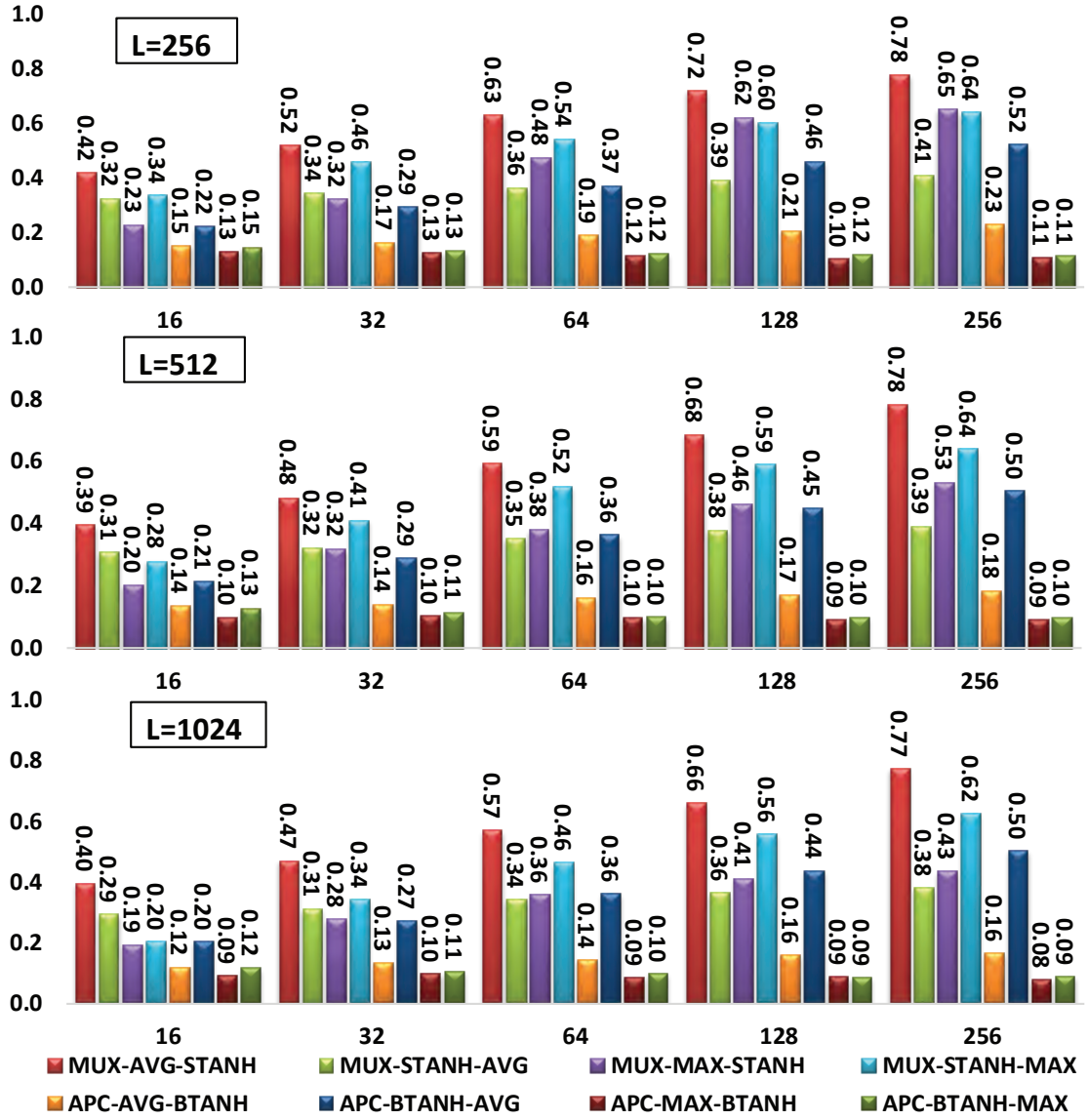


Figure 4.16: Imprecision for Optimized FEBs with bit-stream length $L = 256, 512, 1024$

longer, the absolute error decreases for the same input size. Because a number can be represented more precisely with longer bit-streams. But the improvements are not significant from the observation of each group of bars in Fig. 4.16. (ii) With the same length of bit-stream, additional inputs result in increase of imprecision.

From MUX-Avg-Stanh and MUX-Stanh-Avg, it is observed that hardware-

Table 4.9: Comparison among various hardware-based and software-based DCNNs

No.	Software Configuration	Validation Error (%)	Test Error (%)	Area (mm^2)	Power (W)	Delay (ns)	Energy (μJ)	Harware Configuration	Bit-stream Length	Validation Error (%)	Test Error (%)	Area (mm^2)	Power (W)	Delay (ns)	Energy (μJ)
				CPU				MUX-Avg-Stanh	256	10.54	11.55			332.8	1.1
1	Avg-tanh	1.41	1.34	160	41.4	876062	36268.97		512	9.80	9.96	6.62	3.3	665.6	2.2
				GPU					1024	8.69	8.64			1331.2	4.4
2				148	54	39910	2155.14	APC-Avg-Btanh	256	1.72	1.69			1280.0	4.0
				Binary-ASIC					512	1.61	1.54	13.98	3.1	2560.0	7.9
				769.30	587.5	4.2	2.44		1024	1.48	1.50			5120.0	15.8
				CPU				MUX-Stanh-Avg	256	7.91	8.01			332.8	2.7
3	tanh-Avg	1.01	1.02	160	41.4	1069806	44289.99		512	6.86	7.38	11.42	8.1	665.6	5.4
				GPU					1024	6.11	6.59			1331.2	10.8
4				148	54	40969	2212.32	APC-Btanh-Avg	256	2.16	1.95			1280.0	7.9
				Binary-ASIC					512	1.64	1.75	28.67	6.2	2560.0	15.8
				4334.64	603.1	4.2	2.03		1024	1.67	1.60			5120.0	31.5
				CPU				MUX-NMAX-Stanh	256	7.92	8.12			332.8	2.1
5	Max-tanh	0.93	0.96	160	41.4	865169	35818.04		512	4.78	4.88	10.12	6.4	665.6	4.3
				GPU					1024	3.18	2.96			1331.2	8.5
6				148	54	30178	1629.59	APC-NMAX-Btanh	256	1.11	1.08			1280.0	7.2
				Binary-ASIC					512	1.15	1.04	28.29	5.6	2560.0	14.3
				770.81	444.2	5.6	3.48		1024	1.02	0.96			5120.0	28.6
				CPU				MUX-Stanh-NMAX	256	11.19	11.11			332.8	4.4
7	tanh-Max	0.94	0.96	160	41.4	1059372	43858.03		512	7.84	8.18	17.51	13.4	665.6	8.9
				GPU					1024	3.99	4.08			1331.2	17.8
8				148	54	40119	2166.46	APC-Btanh-NMAX	256	1.11	1.21			1280.0	8.7
				Binary-ASIC					512	1.05	1.12	34.94	6.8	2560.0	17.5
				1067.88	590.6	5.46	3.9		1024	1.02	1.06			5120.0	35.0

oriented design, where pooling blocks follows activation function, halves that absolute error at its best effort. Generally, APC-based FEBs are more accurate than MUX-based ones, for APCs precisely count and sum 1s in the input bit-streams. However, as a special case, from MUX-Stanh-Avg and APC-Btanh-Avg, we can observe that when the input size reaches 64 or more, the MUX-based design is more accurate than the APC-based one. For both MUX-based and APC-based designs, near-max pooling designs gives competitive and better results than average pooling designs. Because in our average pooling design, we use multiplexers to randomly select bits from one of the inputs as the average of all inputs while in near-max pooling, our selection of the maximum bit-stream from inputs is based on local statistics. This greedy selection mechanism guarantees the precision of designs using max pooling.

In Table. 4.9, we conclude the software-based DCNNs as the reference on the left side and shows the performance for corresponding proposed hardware-based DCNNs on the right side. There are totally four reference software-based models. Avg-tanh represents a software-based LeNet-5 model with FEBs in

which average pooling is followed by an activation function of hyperbolic tangent. Similarly, we named other three models as tanh-Avg, Max-tanh, tanh-Max. Correspondingly, each software reference model has two hardware implementations listed on the right side. Each DCNN with one hardware FEB design is evaluated with different lengths of bit-streams, i.e. 256, 512, 1024. Both network accuracies (validation error rate and test error rate) and hardware performance are analyzed. The delay and energy are measured for one run of DCNN inference.

It is observed that the DCNNs using MUX-based inner-product blocks (No. $1, 3, 5, 7$) provide smaller footprints while the DCNNs with APC-based inner-product blocks (No. $2, 4, 6, 8$) achieve better network accuracies and lower powers. However, APC-based designs have longer path delays than MUX-based designs with the same bit-stream length correspondingly, which makes APC-based designs' energy consumptions are much higher. Average pooling based designs (No. $1-4$) exploit smaller footprints than max pooling based designs (No. $5-8$), for its simplicity in the hardware implementation (multiplexers only). For the same reason, average pooling based designs show a lower power/energy than max pooling based designs. The arrangements of pooling neurons and activation functions reflects two different facts that for some designs like No. 1 and No. 3, hardware-oriented modified designs perform better network accuracy (smaller validation and test error), whereas, for some designs like No. 5 and No. 7, SC implementations on software-based structure provide better accuracy and hardware performance.

The proposed SC-based hardware designs of DCNN are much more area efficient, with improvements up to 24.17×, 22.36×, and 776.61× compared to

CPU, GPU, and synthesized Binary-ASICs respectively. Besides, compared with CPU and GPU, proposed designs of SC-based DCNN outperform in the aspects of energy efficiency and power efficiency. They achieve up to 13.41× power and 32,835.32× energy improvements over CPU implementations as well as up to 17.49× power and 1,951.11× energy improvements over GPU implementations. Regarding the synthesized Binary-ASICs, the proposed designs provide as high as 190.27× power efficiency improvements and up to 2.21× energy efficiency improvement. Please note in each Binary-ASIC synthesis, we implemented an ideal full-parallel pipelined structure where SC-based components are replaced with binary-based components and we used 8-bit fix-point numbers for the implementation. Thus the Binary-ASICs conducted the inference of a DCNN much faster, which made the energy efficiency improvement by SC-based designs not significant. But in reality, the power ($400 \sim 600W$) and area ($700 \sim 4000mm^2$) of the Binary-ASIC syntheses are not acceptable. With the sequential logic to reduce the area and power, the delay and energy consumption of the Binary-ASICs will be considerably increased, so that the SC-based DCNNs have the potential to achieve more substantial energy improvements. The proposed SC-based DCNNs can achieve as low as 1.02% validation error rate (No. 8) and 0.96% test error rate (No. 6) which are extremely close to corresponding software-based model results. In those cases, the power, area, and energy of proposed SC-based designs are very small compared to CPU and GPU implementations. Meanwhile, those SC-based designs are more power/area efficient than synthesized Binary-ASICs.

4.6.2 Overall Results on the DCNNs

Using the design strategies presented so far, we perform holistic optimizations on the overall SC-DCNN to construct the LeNet 5 DCNN structure. The (max pooling-based or average pooling-based) LeNet 5 is a widely-used DCNN structure [66] with a configuration of 784-11520-2880-3200-800-500-10. The frameworks are evaluated with the MNIST handwritten digit image dataset [27], which consists of $60,000$ training data and $10,000$ testing data.

The baseline error rates of the max pooling-based and average pooling-based LeNet5 DCNNs using software implementations are 1.53% and 2.24%, respectively. In the optimization procedure, we set 1.5% as the threshold on the error rate difference compared with the error rates of software implementation. In another word, the network accuracy degradation of the SC-DCNNs cannot exceed 1.5%. We set the maximum bit-stream length as 1024 to avoid excessively long delays. In the optimization procedure, for the configurations that achieve the target network accuracy, the bit-stream length is reduced by half in order to reduce energy consumption. Configurations are removed if they fail to meet the network accuracy goal. The process is iterated until no configuration is left.

Table 4.10 displays some selected typical configurations and their comparison results (including the consumption of SRAMs and random number generators). Configurations No.1-6 are max pooling-based SC-DCNNs, and No.7-12 are average pooling-based SC-DCNNs. It can be observed that the configurations involving more MUX-based feature extraction blocks achieve lower hardware cost. Those involving more APC-based feature extraction blocks achieve higher accuracy. For the max pooling-based configurations, No.1 is the most area efficient as well as power efficient configuration, and No.5 is the most en-

Table 4.10: Comparison among Various SC-DCNN Designs Implementing LeNet 5

No.	Pooling	Bit Stream	Configuration			Performance				
			Layer 0	Layer 1	Layer 2	Inaccuracy (%)	Area (mm^2)	Power (W)	Delay (ns)	Energy (μJ)
1	Max	1024	MUX	MUX	APC	2.64	19.1	1.74	5120	8.9
2			MUX	APC	APC	2.23	22.9	2.13	5120	10.9
3		512	APC	MUX	APC	1.91	32.7	3.14	2560	8.0
4			APC	APC	APC	1.68	36.4	3.53	2560	9.0
5		256	APC	MUX	APC	2.13	32.7	3.14	1280	4.0
6			APC	APC	APC	1.74	36.4	3.53	1280	4.5
7	Average	1024	MUX	APC	APC	3.06	17.0	1.53	5120	7.8
8			APC	APC	APC	2.58	22.1	2.14	5120	11.0
9		512	MUX	APC	APC	3.16	17.0	1.53	2560	3.9
10			APC	APC	APC	2.65	22.1	2.14	2560	5.5
11		256	MUX	APC	APC	3.36	17.0	1.53	1280	2.0
12			APC	APC	APC	2.76	22.1	2.14	1280	2.7

Table 4.11: Application-level performance and hardware cost of LeNet-5 implementation using the proposed HEIF.

Bit Stream	ReLU Clipped		Area (mm^2)	Power (W)	Delay (ns)	Energy (μJ)
	Validation	Test				
1024	1.07%	0.88%	22.9	2.6	2498.6	6.4
512	1.12%	0.87%			1249.3	3.2
256	1.13%	0.91%			624.6	1.6
128	1.18%	0.93%			312.3	0.8
software	0.94%	0.83%		-		
highest software accuracy in the literature [21]						0.23%

ergy efficient configuration. With regard to the average pooling-based configurations, No.7, 9, 11 are the most area efficient and power efficient configurations, and No.11 is the most energy efficient configuration.

Table 4.11 concludes the performance and hardware cost of the proposed HEIF on LeNet-5 implementation. One can observe that the proposed HEIF can realize the entire LeNet-5 with only 0.10% accuracy degradation compared to the software accuracy of our software-based implementations. Table 4.12 compares the performance and hardware cost of the proposed HEIF with the existing hardware platforms on the MNIST dataset. For SC-DCNN, the configuration No.6 and No.11 are selected to compare with software implementation on CPU server or GPU. No.6 is selected because it is the most accurate max pooling-based configuration. No.11 is selected because it is the most energy efficient average pooling-based configuration. It can be observed that compared with

Table 4.12: Comparison with existing hardware platforms for handwritten digit recognition using the MNIST [27] dataset

Platform	Network Type	Year	Platform Type	Clock (MHz)	Area (mm^2)	Power (W)	Accuracy (%)	Throughput (Images/s)	Area Efficiency (Images/s/mm^2)	Energy Efficiency (Images/J)
2×Intel Xeon W5580	CNN	2009	CPU	3200	263	156	99.17	656	2.5	4.2
Nvidia Tesla C2075	CNN	2011	GPU	1150	520	202.5	99.17	2333	4.5	3.2
Minitaur [90]	ANN[1]	2014	FPGA	400	N/A	≤1.5	92.00	4880	N/A	≥3253
SpiNNaker [115]	DBN	2015	ARM	150	N/A	0.3	95.00	50	N/A	166.7
TrueNorth [31]	SNN[2]	2015	ASIC	Async	430	0.18	99.42	1000	2.3	9259
SC-DCNN (No.6)[103]	CNN	2016	ASIC	200	36.4	3.53	98.26	781250	21439	221287
SC-DCNN (No.11)[103]	CNN	2016	ASIC	200	17.0	1.53	96.64	781250	45946	510734
HEIF(128bit)	CNN	2016	ASIC	410	22.9	2.6	99.07	3203125	139874	1231971

[1]ANN: Artificial Neural Network; [2]SNN: Spiking Neural Network

the other platforms, the proposed HEIF yields the highest throughput, area efficiency and energy efficiency while approaching the highest software accuracy, i.e. 99.77%, demonstrating the effectiveness of the SC technology and our proposed holistic optimization procedure. Compared with the high-performance version of SC-DCNN in [103], the proposed method achieves up to 0.81% accuracy increase, and 4.1×, 6.5× and 5.5× improvement in terms of throughput, area efficiency and energy efficiency, respectively. Compared with the low-power version of SC-DCNN, the proposed HEIF achieves improved accuracy due to the overall optimization on the cascade connection of function blocks and the novel ReLU design, whereas the area, power and energy efficiency gain are mainly achieved through APC optimization, pipelining technique, bit-stream length reduction, and weight storage optimization.

Next we present the results of HEIF on the large-scale AlexNet applications. We trained AlexNet using ImageNet training set by our own configurations. To follow the stochastic computing paradigm, we use scaled pixel values within [0,1] instead of original range [0,255]. Because data pre-processing first deducts the mean value of each image from each pixel value, the input then ranges in [-1,1]. Moreover, we use clipped ReLU to restrain the activation output to be [0,1]. We also move pooling units before ReLU so that we can save resource of ReLU in the aspect of hardware cost. The trained network achieves top-1 and

Table 4.13: List of existing hardware platforms for image classification using (part of) the AlexNet [63] on ImageNet [26] dataset

Platform	Year	Platform Type	Memory Type	Area (mm^2)	Power (W)	Throughput (Images/s)	Area Efficiency (Images/s/mm^2)	Energy Efficiency (Images/J)
2×Intel Xeon W5580	2009	CPU	DRAM	263	156	139	0.5	0.9
Nvidia Tesla C2075	2011	GPU	DRAM	520	202.5	573	1.1	2.8
DaDianNao [17]	2014	ASIC	eDRAM	67.7	15.97	147938	2185	9263
Eyeriss [16]	2016	ASIC	DRAM	12.25	0.28	35	2.8	125
EIE-64PE [43]	2016	ASIC	SRAM	40.8	0.59	81967	2009	138927
EIE-256PE [43]	2016	ASIC	SRAM	63.8	2.36	426230	6681	180606
HEIF(128bit)	2016	ASIC	SRAM	24.7	1.9	2520161	102030	1326400

top-5 accuracies of 56.56% and 80.48% on the test set, respectively. To the best of our knowledge, the existing hardware platforms either implemented one computation layer of the AlexNet [43], built a reconfigurable circuit to accelerate each layer separately [16], or designed a reconfigurable system that can be connected in a chip system to deal with large computation tasks [17]. Table 4.13 lists the existing hardware platforms for AlexNet implementation. As EIE [43] provided the results on the fully-connected FC7 layer of AlexNet, we evaluate the proposed HEIF on the same FC7 layer of AlexNet. We apply the same weight compression technique in [45], making a fair comparison. Note that Table 4.13 is a list of existing platforms instead of a strict comparison table, because the implementation scales and method of different works are not the same (and some are not discussed in details in papers). One can observe from Table 4.13 that the proposed HEIF has the smallest footprint due to the small footprint of each stochastic computing component, and achieves the best performance in terms of throughput, area efficiency and energy efficiency.

4.7 Conclusion

In this chapter, we propose SC-DCNN, the first comprehensive design and optimization framework of SC-based DCNNs. Then we present HEIF, the opti-

mized version of SC-DCNN, a highly efficient SC-based inference framework of the large-scale deep convolutional neural networks, with broad applications on (but not limited to) both LeNet-5 and AlexNet, in order to achieve ultra-high energy efficiency and low area/hardware cost. SC-DCNN and HEIF fully utilizes the advantages of SC and achieves remarkably low hardware footprint, low power and energy consumption, while maintaining high network accuracy.

We fully explore the design space of different components to achieve high power (energy) efficiency and low hardware footprint. First, we investigated various function blocks including inner product calculations, pooling operations, and activation functions. Then we propose four designs of feature extraction blocks, which are in charge of extracting features from input feature maps, by connecting different basic function blocks with joint optimization. Moreover, three weight storage optimization schemes are investigated for reducing the area and power (energy) consumption of SRAM.

Besides, we re-design the Approximate Parallel Counter and optimize stochastic multiplication while proposing for the first time SC-based Rectified Linear Unit (ReLU) activation function to track with the recent advances in software models.

Experimental results demonstrate that the SC-DCNN achieves low hardware footprint and low energy consumption. It achieves the throughput of $781,250$ images/s, area efficiency of $45,946$ images/s/mm^2, and energy efficiency of $510,734$ images/J. The HEIF framework achieves very high energy efficiency of $1.2M$ Images/J and $1.3M$ Images/J, and high throughput of $3.2M$ Images/s and $2.5M$ Images/s, along with very small area of 22.9 mm^2 and 24.7 mm^2 on LeNet-5 and AlexNet respectively. HEIF outperforms previous SC-

DCNN by the throughput of 4.1×, by area efficiency of up to 6.5× and achieves up to 5.6× energy improvement.

CHAPTER 5

ENABLING EFFICIENT RECURRENT NEURAL NETWORKS USING STRUCTURED COMPRESSION TECHNIQUES ON FPGAS

5.1 Introduction

Recurrent Neural Networks (RNNs) represent an important class of machine learning techniques that are specialized for processing sequential data [38]. RNNs have wide applications in speech recognition, natural language processing, scene and semantic understanding, time series analysis, etc. Many of these applications require efficient and real-time implementations. The two major types of RNNs with the broadest applications and highest performance are the *Long Short-Term Memory* (LSTM) unit [48] and the *Gated Recurrent unit* (GRU) [19]. LSTM and GRU RNNs are computationally intensive but can effectively overcome *vanishing* and *exploding* gradient problems [96] of traditional RNNs. However, the significant recognition accuracy improvement comes at the cost of increased computational complexity of larger model size [36]. Therefore, customized hardware acceleration is increasingly important for LSTM/GRUs, as exemplified by recent works on employing GPUs [24, 82], FPGAs [42, 73] and ASICs [32] as accelerators to speedup LSTM RNNs.

Among the numerous platforms, FPGA has emerged as a promising solution for hardware acceleration as it provides customized hardware performance with flexible reconfigurability. By creating dedicated pipelines, parallel processing units, customized bit width, and etc., application designers can accelerate many workloads by orders of magnitude using FPGAs [107]. More importantly, High-level Synthesis (HLS) has greatly lowered the programming

hurdle of FPGAs and improved the productivity by raising the programming abstraction from tedious RTL to high-level languages such as C/C++ [23] and OpenCL [125].

While the benefits of FPGAs is clear, it is still challenging to design efficient designs for LSTMs on FPGAs mainly for two reasons. On one hand, the capacity of the FPGA on-chip memory (a few or tens of Mb on-chip memory) is usually not large enough to store all the weight matrices of a standard LSTM inference model (e.g. hundreds of Mb). Although the previous work ESE [42] proposes to use the parameter pruning based compression technique to compress the dense weight matrices in the LSTM model into sparse ones, the sparse matrices need extra storage and processing units to store and decode the indices of the non-zero data, respectively. The skewed distribution of the data is likely to cause unbalanced workloads among parallel compute units. Therefore, the benefits of unstructured model compression is diminished by the sparsity of weight matrices. On the other hand, the computational complexity among the operators of the LSTMs is highly skewed and the data dependencies between operator are complicated. So, it is difficult to evenly allocate computing resources under the FPGA resource constraints while guaranteeing the complex data dependencies.

In this chapter, we propose to compress the weight matrices in the RNN inference model in a structured manner by using block-circulant matrix [94]. The circulant matrix is a square matrix, of which each row (column) vector is the circulant reformat of the row (column) vector. Any matrix could be transformed into a set of circulant submatrices a.k.a. block-circulant matrices. Therefore, by representing each block-circulant matrix with a vector, the storage requirement could be reduced from $\mathcal{O}(k^2)$ to $\mathcal{O}(k)$ if the block (vector) size is k. Since

the compressed weight matrices are still dense, the block-circulant matrix based compression is amenable to hardware acceleration on FPGAs. In order to further speed up the computation of LSTMs, we propose to accelerate the most computation-intensive circulant convolution operator by applying Fast Fourier Transform (FFT) algorithm to reduce the computational complexity from $\mathcal{O}(k^2)$ to $\mathcal{O}(k\log k)$.

After the model is compressed, we propose an automatic optimization and synthesis framework called E-RNN to port efficient RNN designs onto FPGAs. The framework is composed of model training and implementation flows. The former one is in charge of iteratively training the compressed RNN model and exploring the trade-offs between compression ratio and prediction accuracy. As for the model implementation, it mainly consists of two parts which are (1) template generation and (2) automatic RNN synthesis framework. For the former part, after analyzing a wide range of RNN algorithms, we generalize a suite of RNN primitive operators which is general enough to accommodate even the most complicated RNN variant [109]. Then, a suite of highly optimized C/C++ templates of the primitive operators are manually generated by walking through a series of optimizations such as datapath and activation quantization, DFT-IDFT decoupling and etc. As for the latter part, the well-trained RNN inference model is first analyzed and transformed into a directed acyclic dependency graph, where each node represents an operator and each edge indicates the associated data dependency between two operators. Secondly, we propose a specialized pipeline optimization algorithm considering both coarse-grained and fine-grained pipelining schemes to schedule the operators into appropriate stages. In the third step, we use an accurate performance and resource model to enable a fast design space exploration for optimal design parameters. Lastly,

the scheduling results and optimization parameters are fed to code generator and backend toolchain as to implement the optimized RNN accelerator design on FPGAs.

Overall, the contributions of this chapter are listed as:

- We employ the block-circulant matrices based structured compression technique for RNNs which largely reduces the computation complexity and memory footprint without incurring any computation and memory access irregularities. This method results in both compression and acceleration of the RNN models.We use ADMM-based training for deriving block-circulant matrix-based RNN representation. ADMM-based training provides an effective means to deal with the structure requirement in weight matrices, thereby enhancing accuracy and training speed.
- We develop a general RNN optimization and synthesis framework E-RNN to enable automatic and efficient implementations of a wide range of RNN variants on FPGAs. The framework mainly consists of a suite of highly optimized C/C++ based templates of primitive operators and an automatic RNN synthesis flow.
- We present efficient implementations of RNNs which achieve up to 37.4× gains in energy efficiency compared with the state-of-the-art. The proposed implementations incur very small accuracy degradation.

5.2 Related Work

Recently, FPGA has emerged as a promising hardware acceleration platform for DNNs as it provides high performance, low power and reconfigurability. A lot of FPGA based accelerators have been proposed for convolutional neural networks (CNNs) to overcome the computing and energy efficiency challenges. [127] proposes to utilize systolic array based convolution architecture to achieve better frequency and thus performance for CNNs on FPGAs. [86] employs the Winograd algorithm to reduce the multiplication operators as to save DSP resources and accelerate matrix multiplication in CNNs. [131] proposes to take advantage of the heterogeneous algorithms to maximize the resource utilization for convolutional layers on FPGAs. Some studies also propose to transform the CNN models to frequency domains and then exploit FFT algorithms for further acceleration [61]. The FFT based acceleration scheme used in the CNN model is completely different from this work, in which we target on a totally different LSTM based RNN model and the FFT algorithm is applied to the circulant convolution operators instead of the convolution layers of CNNs.

There are also a lot of works on implementing RNN accelerators for FPGAs [40, 73, 91]. [91] designs an accelerator for the gated recurrent network (GRU) which embodies a different architecture from the LSTM based RNNs. [40] and [73] focus on LSTM based RNNs but none of these works utilize compression techniques to reduce the model size. The most relevant study to this work is ESE [42], which proposes a software and hardware co-design framework to accelerate compressed sparse LSTM model obtained by parameter pruning [44]. The performance and energy efficiency gains achieved by ESE is very promising compared with CPU and GPU based implementations.

However, due to the irregular computation and memory accesses caused by the sparse weight matrices of the compressed model, the computing power of the FPGA is not fully exerted by ESE. In order to deal with this problem, this work proposes to employ a structured compression technique as to completely eliminate the irregularities of computation and memory accesses. Moreover, a suite of highly efficient optimization techniques is enabled by an automatic synthesis framework to generate RNN accelerators with much higher performance and energy efficiency under the same conditions.

5.3 Structured Compression

Deep neural networks (DNNs) bear a significant amount of redundancy [44] and thus model compression is a natural method to mitigate the computation and memory storage requirements for the hardware implementations on FPGAs. In this section, we propose to employ a structured compression technique to compress the weight matrices of RNN model by using block-circulant matrices. We first introduce the block-circulant matrix and then integrate it with the inference and training algorithms of LSTMs. In the last, we explore the trade-offs between compression ratio and prediction error rate.

5.3.1 Block-Circulant Matrix

The circulant matrix is a square matrix whose each row (or column) vector is the circulant reformat of the row (or column) vectors [18, 94]. Any matrix could be transformed into a set of circulant submatrices (blocks) and we define the

transformed matrix as a block-circulant matrix. For example, Figure 5.1 shows that the 3×9 weight matrix (on the bottom) is reformatted into a block-circulant matrix containing three 3×3 circulant matrices (on the top). Since each row vector of the circulant submatrix is a reformat of the first row vector, we could use a row vector to represent a circulant submatrix. Therefore, the first obvious benefit of the block-circulant matrix is that the number of parameters in each weight matrix is reduced by a factor of the block size $\mathcal{O}(k)$. As for the example in Figure 5.1, the 3×9 weight matrix (on the bottom) holding 27 parameters is reduced to three 3×3 circulant matrices (on the top) containing only 9 parameters, which easily leads to $3\times$ model size reduction.

Intuitively, the model compression ratio is determined by the block size of the circulant submatrices: larger block size leads to higher compression ratio and vice versa. However, high compression ratio may degrade the prediction accuracy. Specifically, a larger block size should be selected to achieve a higher compression ratio but lower accuracy and the smaller block size provides higher accuracy but less compression ratio. The block size is 1 if no compression is utilized. It is necessary to note that block-circulant matrix based DNNs have been proved to asymptotically approach the original networks in accuracy with mathematical rigor [140]. Therefore, if the compression ratio is selected properly, the accuracy loss would be negligible. The design exploration from the perspective of compression ratio and predication accuracy is discussed in Section 5.4

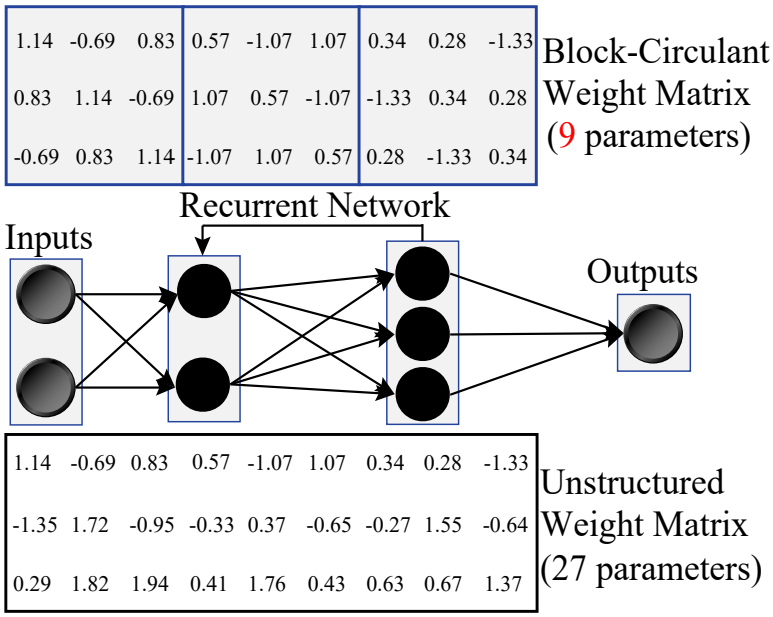


Figure 5.1: Block-circulant matrices for weight representation.

5.3.2 Inference and Training Algorithms

Forward Propagation. The primary idea of block-circulant matrix-based RNN is to represent the original arbitrary weight matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ with an array of equal-size square sub-matrices (i.e., *blocks*), where each sub-matrix is a *circulant* matrix. Assume there are $p \times q$ blocks after partitioning the matrix $\mathbf{W}$, where $p = \frac{m}{L_b}$ and $q = \frac{n}{L_b}$. Here L_b is the *block size*. Then $\mathbf{W} = [\mathbf{W}_{ij}]$, $i \in \{1 \dots p\}$, $j \in \{1 \dots q\}$.

Each circulant matrix $\mathbf{W}_{ij}$ can be defined by a vector $\mathbf{w}_{ij}$. More specifically, $\mathbf{w}_{ij}$ is the first row vector of $\mathbf{W}_{ij}$; the second row vector of $\mathbf{W}_{ij}$ is a circulation of the first row vector, and so on. Figure 5.2 provides an example of circulant matrix $\mathbf{W}_{ij}$. The storage complexity of a block-circulant weight matrix is significantly reduced since we only need to store one vector $\mathbf{w}_{ij}$ for each circulant matrix $\mathbf{W}_{ij}$. As a result, we have the ability to store all the weights matrices

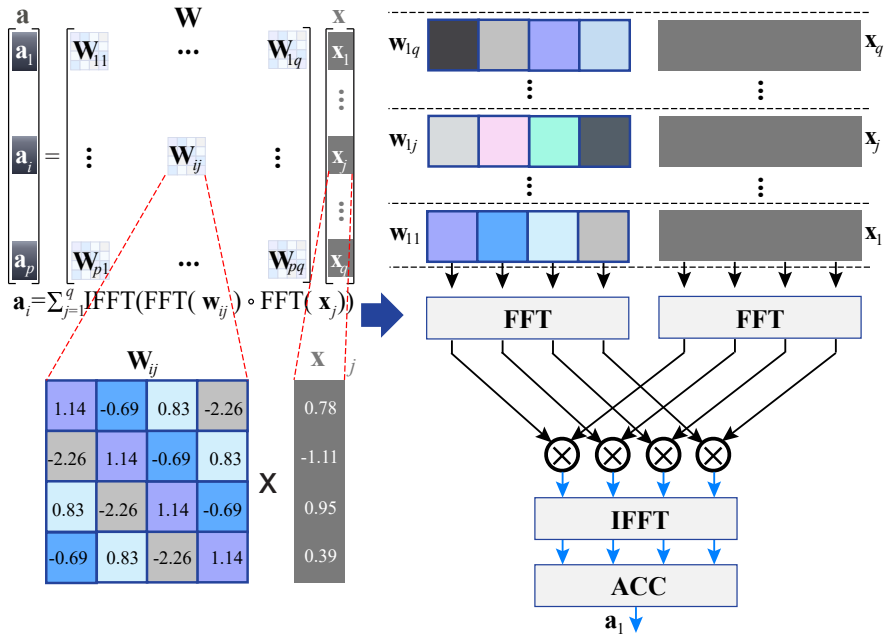


Figure 5.2: An illustration of FFT-based calculation in block-circulant matrix multiplication.

(i.e., $\mathbf{W}_{*(xr)}$) and the projection matrix $\mathbf{W}_{ym}$ in block RAM (BRAM), thereby significantly improving the FPGA performance. Additionally, the input feature $\mathbf{x}$, bias $\mathbf{b}$ ($\mathbf{b}_i$, $\mathbf{b}_f$, and $\mathbf{b}_o$), and diagonal matrices $\mathbf{W}_c$ ($\mathbf{W}_{ic}$, $\mathbf{W}_{fc}$, and $\mathbf{W}_{oc}$) can also be stored in BRAM due to a small quantity of corresponding parameters.

Since a weight matrix $\mathbf{W}$ is now partitioned into $p \times q$ blocks, correspondingly, the input $\mathbf{x}$ is also partitioned as $\mathbf{x} = [\mathbf{x}_1^T, \mathbf{x}_2^T, \ldots, \mathbf{x}_q^T]^T$, $\mathbf{x}_j \in \mathbb{R}^{L_b}$. Then, the *forward propagation* process in the inference phase is given by (with bias and activation function omitted):

$$\mathbf{a} = \mathbf{W}\mathbf{x} = \begin{bmatrix} \sum_{j=1}^{q} \mathbf{W}_{1j}\mathbf{x}_j \\ \sum_{j=1}^{q} \mathbf{W}_{2j}\mathbf{x}_j \\ \cdots \\ \sum_{j=1}^{q} \mathbf{W}_{pj}\mathbf{x}_j \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \cdots \\ \mathbf{a}_p \end{bmatrix}, \tag{5.1}$$

where $\mathbf{a}_i \in \mathbb{R}^{L_b}$ is a column vector. We can see the calculation of $\mathbf{W}\mathbf{x}$ is reduced to the calculation of $\mathbf{W}_{ij}\mathbf{x}_j$'s. Then according to the *circulant convolution theorem*

[94, 10], the calculation of $\mathbf{W}_{ij}\mathbf{x}_j$ can be performed as

$$\mathbf{a}_i = \sum_{j=1}^{q} \mathcal{F}^{-1}[\mathcal{F}(\mathbf{w}_{ij}) \odot \mathcal{F}(\mathbf{x}_j)], \tag{5.2}$$

where $\mathcal{F}(\cdot)$ is the Discrete Fourier Transform (DFT) operator, $\mathcal{F}^{-1}(\cdot)$ is the inverse DFT (IDFT) operator, and $\odot$ is the element-wise multiply operator. The computational complexity of $\mathbf{Wx}$ is reduced from $O(n^2)$ by direct matrix-vector multiplication to $O(pqL_b \log L_b)$ by the "FFT→element-wise multiplication→IFFT" procedure in Eqn. (5.2), which is equivalent to $O(n \log n)$ for small p, q values. As a result, the simultaneous acceleration and model compression compared with the original RNN can be achieved for the inference process.

Backward Propagation. The backward propagation process in the training phase can also be implemented using block-circulant matrices. Here we use a_{il} to denote the l-th output element in $\mathbf{a}_i$, and L to represent the loss function. Then by using the chain rule we can derive the backward propagation process as follows:

$$\frac{\partial L}{\partial \mathbf{w}_{ij}} = \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}} \frac{\partial a_{il}}{\partial \mathbf{w}_{ij}} = \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}, \tag{5.3}$$

$$\frac{\partial L}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}} \frac{\partial a_{il}}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}. \tag{5.4}$$

where $\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ are proved to be block-circulant matrices [140]. Thus, $\frac{\partial L}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ can be calculated similarly as Equation (5.2) with the same computational complexity. The details of the training procedure for a fully-connected layer in DNNs are presented in [29, 126] and also applicable to the LSTM based RNNs.

5.3.3 Alternating Direction Method of Multipliers (ADMM) Based Training

Consider an optimization problem $\min_{\mathbf{x}} f(\mathbf{x})$ with combinatorial constraints. This problem is difficult to solve directly using optimization tools [139]. Through the application of ADMM [11, 55], the original optimization problem is decomposed into two subproblems, and will be iteratively solved until convergence. The first subproblem is $\min_{\mathbf{x}} f(\mathbf{x}) + q_1(\mathbf{x})$ where $q_1(\mathbf{x})$ is a differentiable, quadratic term. This subproblem does not have combinatorial constraints and can be solved using traditional optimization method, e.g., SGD for RNN training. The second subproblem is $\min_{\mathbf{x}} g(\mathbf{x}) + q_2(\mathbf{x})$, where $g(\mathbf{x})$ corresponds to the original combinatorial constraints and $q_2(\mathbf{x})$ is also quadratic. For special types of combinatorial constraints, including structured matrices, quantization, etc., the second subproblem can be optimally and analytically solved, as shown in the following discussions.

Consider an RNN model with N layers. The collection of weights in layer l is denoted by $\mathbf{W}_l$. The loss function is denoted by $f(\{\mathbf{W}_l\}_{l=1}^N)$. Let $(\mathbf{W}_l)_{ij}$ with dimension $L_b \times L_b$ denote the ij^{th} block in the structured matrix that $\mathbf{W}_l$ should be mapped to.

We introduce auxiliary variables $\mathbf{Z}_l$ and $\mathbf{U}_l$, which have the same dimensionality as $\mathbf{W}_l$. Through the application of ADMM[1], the original structured training problem can be decomposed into two subproblems, which are iteratively solved until convergence. In each iteration k, the first subproblem is

[1]The details of the ADMM algorithm are discussed in [11, 139].

$$\underset{\{\mathbf{W}_l\}}{\text{minimize}} \quad f(\{\mathbf{W}_l\}_{l=1}^N) + \sum_{l=1}^{N} \frac{\rho_l}{2} \|\mathbf{W}_l - \mathbf{Z}_l^k + \mathbf{U}_l^k\|_F^2, \tag{5.5}$$

where $\mathbf{U}_l^k$ is the dual variable updated in each iteration, $\mathbf{U}_l^k := \mathbf{U}_l^{k-1} + \mathbf{W}_l^k - \mathbf{Z}_l^k$. In the objective function of (equation 5.5), the first term is the differentiable loss function of RNN, and the second quadratic term is differentiable and convex. As a result, this subproblem can be solved by stochastic gradient descent and the complexity is the same as training the original RNN. A large number of contraints are avoided here. The result of the first subproblem is denoted by $\mathbf{W}_l^{k+1}$. Proven in [11], the global optimal solution of the second subproblem is to find a Euclidean mapping of $\mathbf{W}_l^{k+1} + \mathbf{U}_l^k$ to the closest structured (circulant) matrix format. The result of the second subproblem is denoted by $\mathbf{Z}_l^{k+1}$.

For better illustration, let $(\mathbf{W}_l^{k+1} + \mathbf{U}_l^k)$ denote a specific matrix to be mapped, and let $(\mathbf{Z}_l^{k+1})$ denote the corresponding structured format. For the ij^{th} block,the elements $(1,1)$, $(2,2)$,..., (L_b, L_b) of $(\mathbf{Z}_l^{k+1})_{ij}$ should be equal. For Euclidean mapping, we have:

$$\begin{aligned} &(\mathbf{Z}_l^{k+1})_{ij,(1,1)} = (\mathbf{Z}_l^{k+1})_{ij,(2,2)} = ... = (\mathbf{Z}_l^{k+1})_{ij,(L_b,L_b)} \\ &= \frac{(\mathbf{W}_l^{k+1} + \mathbf{U}_l^k)_{ij,(1,1)} + ... + (\mathbf{W}_l^{k+1} + \mathbf{U}_l^k)_{ij,(L_b,L_b)}}{L_b} \end{aligned} \tag{5.6}$$

Similarly the other entries in $(\mathbf{Z}_l^{k+1})_{ij}$ can be calculated. We have proved that this is the optimal analytical solution of the second subproblem. Figure 5.3 illustrates an example of the Euclidean mapping by applying Eqn. (5.6).

The overall procedure of ADMM-based structured matrix training is shown in Figure 5.4. Essentially speaking, it iteratively (i) map $\mathbf{Z}_l^{k+1}$ to the structured format in the optimal manner, and (ii) use the mapped $\mathbf{Z}_l^{k+1}$ as a dynamic regularization target for weight training. Upon convergence the RNN weights will

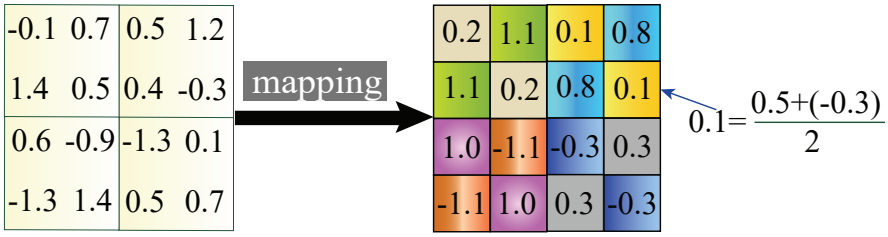


Figure 5.3: Euclidean mapping for a 4×4 matrix with block size of 2.

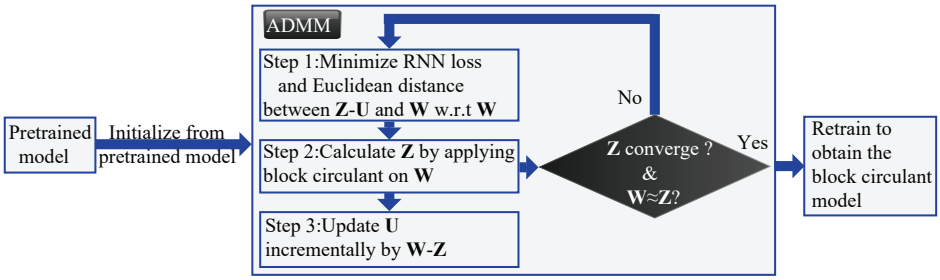


Figure 5.4: The overall procedure of ADMM-based structured matrix training.

converge to the structured format. The proposed method effectively overcomes the limitation of combinatorial constraints and achieves higher training accuracy compared with the prior work, as shall be seen in experimental results.

5.4 RNN Model Design Exploration: A software view

In this section, we perform RNN model design exploration at the algorithm level, in order to shed some light on RNN training trial reductions. More specifically, we provide an analysis of the effect of model type (LSTM or GRU), layer size, and block size on the overall accuracy. The design variable with the least impact on the overall accuracy should be given priority in design optimization. We focus on TIMIT benchmark, the most widely utilized benchmark for ASR applications. In the following, we will provide a detailed discussion on the data set, RNN models, and results and observations.

Dataset. The TIMIT dataset [35] contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences, totally $6,300$ utterances. The TIMIT corpus includes time-aligned orthographic, phonetic and word transcriptions as well as a 16-bit, 16kHz speech waveform file for each utterance.

RNN Models. The RNN models utilized in the design exploration are summarized in Table 5.1 and Table 5.2. We stack multiple RNN layers to build our network. The number of layers and layer sizes (dimensionality of $\mathbf{c}_t$) are listed in the tables. For an LSTM cell, $256 - 256 - 256$ means that the network has three layers of LSTM cells with 256 hidden neurons in $\mathbf{c}_t$. The block sizes (as a power of 2) are listed in the same format as layer sizes correspondingly. "–" means that we do not apply (block-)circulant matrix on the network, which is the baseline model for that specific network structure. The baseline model with layer size $1,024$ is the same as the baseline in ESE [42]. We also list the configuration options like "peephole" and "projection". The performance is evaluated by *phone error rate* (PER) or *word error rate* (WER) and degradations compared to the corresponding baseline model. The smaller the PER or WER, the better of the corresponding RNN model.

Results Discussion and Observations. From Table 5.1 and Table 5.2, we can observe that the block-circulant matrix-based framework results in very small accuracy degradation compared with the baseline model. More specifically, when the block size is 4 ($4\times$ parameter reduction) or smaller, there is in general no accuracy degradation compared with the corresponding baseline. When the block size is 8 ($8\times$ parameter reduction), the accuracy degradation is negligible, around 0.1%-0.15%. When the block size is 16, the accuracy degra-

Table 5.1: Comparison among LSTM based RNN models.

ID	Layer Size	Block Size	Peep-hole	Projection (512)	Phone Error Rate (PER) %	PER degra-dation (%)
1	256 – 256 – 256	–	×	×	20.83	–
2	256 – 256 – 256	2 – 2 – 2	×	×	20.75	–0.08
3	256 – 256 – 256	4 – 4 – 4	×	×	20.85	0.02
4	512 – 512	–	√	×	20.53	–
5	512 – 512	4 – 4	√	×	20.57	0.04
6	512 – 512	4 – 8	√	×	20.85	0.28
7	512 – 512	8 – 4	√	×	20.98	0.41
8	512 – 512	8 – 8	√	×	21.01	0.48
9	1024 – 1024	–	√	√	20.01	–
10	1024 – 1024	4 – 4	√	√	20.01	0.00
11	1024 – 1024	4 – 8	√	√	20.05	0.04
12	1024 – 1024	8 – 4	√	√	20.10	0.09
13	1024 – 1024	8 – 8	√	√	20.14	0.13
14	1024 – 1024	8 – 16	√	√	20.22	0.21
15	1024 – 1024	16 – 8	√	√	20.29	0.28
16	1024 – 1024	16 – 16	√	√	20.32	0.31

Table 5.2: Comparison among GRU based RNN models.

ID	Layer Size	Block Size	Phone Error Rate (PER) %	PER degradation (%)
1	256 – 256 – 256	–	20.72	–
2	256 – 256 – 256	4 – 4 – 4	20.81	0.09
3	256 – 256 – 256	8 – 8 – 8	20.88	0.16
4	512 – 512	–	20.51	–
5	512 – 512	4 – 4	20.55	0.04
6	512 – 512	4 – 8	20.73	0.22
7	512 – 512	8 – 4	20.89	0.38
8	512 – 512	8 – 8	20.95	0.44
9	1024 – 1024	–	20.02	–
10	1024 – 1024	4 – 4	20.03	0.01
11	1024 – 1024	4 – 8	20.08	0.06
12	1024 – 1024	8 – 4	20.13	0.11
13	1024 – 1024	8 – 8	20.20	0.18
14	1024 – 1024	8 – 16	20.25	0.23
15	1024 – 1024	16 – 8	20.31	0.29
16	1024 – 1024	16 – 16	20.36	0.33

dation is still only around 0.3%. As discussed before, the baseline model with layer size 1,024 is the same as the baseline in ESE [42]. Then we can conclude that the block-circulant matrix-based framework outperforms ESE in terms of model compression. This is because ESE achieves 9× parameter reduction with 0.3% accuracy degradation. This parameter reduction even does not account for the indices, which are needed at least one for each parameter in the network structure after pruning. We will observe in the hardware experimental results that the performance and energy efficiency gains are even more significant compared with ESE, thanks to the regularity in this framework.

Moreover, the above design exploration procedure provides observations on the RNN model selection and optimization, which could shed some lights on training trial reductions. We can observe that changing from LSTM to GRU or using a block size of 4 or smaller will not result in accuracy degradation. Therefore, if the accuracy requirement is very tight for the target application, we can in general change to GRU and/or using a block size of 4. In this way the amounts of computation and storage are reduced, which is directly related to the performance and energy consumption in hardware implementations, with zero accuracy degradation. If a small amount of accuracy degradation is allowed, then the top priority is using a block size of 8 or 16 compared with a smaller LSTM/GRU RNN model (i.e., a smaller layer size). This is because that the block-circulant matrix based framework, as shown in the two tables, results in smaller amount of accuracy loss and greater computation/storage reduction compared with a smaller LSTM/GRU RNN model. For ASR applications, a block size of 8 or 16 will make the whole RNN model easily accommodated by the on-chip BRAM of FPGAs. This observation validates the effectiveness of the block-circulant framework.

5.5 FPGA Acceleration

In this section, we start by introducing a set of FPGA optimization techniques for circulant convolution operator and then apply quantizations to activation and element-wise operators. In the last, we propose an operator scheduling algorithm to generate the whole RNN pipeline with the help of performance and resource models. In this section, we use LSTM as the case study as it is the harder case than GRU.

5.5.1 FFT/IFFT Decoupling

Since the FFT based circulant convolution operator in the form of Equation (5.2) is the most computation-intensive operator in the LSTM inference model, we propose three techniques to further reduce the computational complexity by reducing the number of DFT and IDFT operator calls, and the redundant arithmetic operations of its complex number multiplication operators.

In order to reduce the number of IDFT calls in the circulant convolution operator, we propose the DFT-IDFT decoupling technique. Since DFT and IDFT are linear operators [92], we could decouple the DFT and IDFT operators in Equation 5.2 and move the IDFT operator $\mathcal{F}^{-1}(\cdot)$ outside the accumulation operator $\sum$ as following,

$$\mathbf{a}_i = \mathcal{F}^{-1}\Big[\sum_{j=1}^{q} \mathcal{F}(\mathbf{w}_{ij}) \odot \mathcal{F}(\mathbf{x}_j)\Big], \tag{5.7}$$

where the number of IDFT operator calls for each circulant convolution operator is reduced from q to 1 and the numbers of the other operator calls are kept the

same as before.

According to Equation (5.7), the number of calls of DFT operator $\mathcal{F}(\cdot)$ in a circulant convolution operator is $2q$, and q is the number of weight vectors $\mathcal{F}(\mathbf{w}_{ij})$ and input vectors $\mathcal{F}(\mathbf{x}_j)$. Since the weight vectors $\mathbf{w}_{ij}$ are fixed when the training process is done, we could precalculate the $\mathcal{F}(\mathbf{w}_{ij})$ values and store them in the BRAM buffers of FPGAs and fetch the required values when needed instead of computing the associated DFT values at runtime. This method completely eliminates the DFT operator $\mathcal{F}(\cdot)$ calls for weight vectors and reduces the number of calls from $2qk$ to qk for each circulant convolution operator. The BRAM buffer size, however, would be doubled since the outputs of DFT values $\mathcal{F}(\mathbf{w}_{ij})$ are complex numbers whose both real and imaginary parts need to be stored. In order to alleviate the BRAM buffer overhead, we propose to exploit the complex conjugate symmetry property of DFT output values, where almost half of the conjugate complex numbers could be eliminated [92, 106]. Therefore, there is only negligible BRAM buffer overhead to store the DFT results of weight vectors $\mathcal{F}(\mathbf{w}_{ij})$.

The element-wise multiplication $\odot$ between two complex number vectors $\mathcal{F}(\mathbf{w}_{ij})$ and $\mathcal{F}(\mathbf{x}_j)$ requires $4k$ multiplications and $3k$ additions. Due to the complex conjugate symmetry property of DFT $\mathcal{F}(\cdot)$ results, about half of the multiplications and additions could be eliminated. Overall, Figure 5.5 illustrates the implementations of the original and optimized circulant convolution operators when the block size is 8.

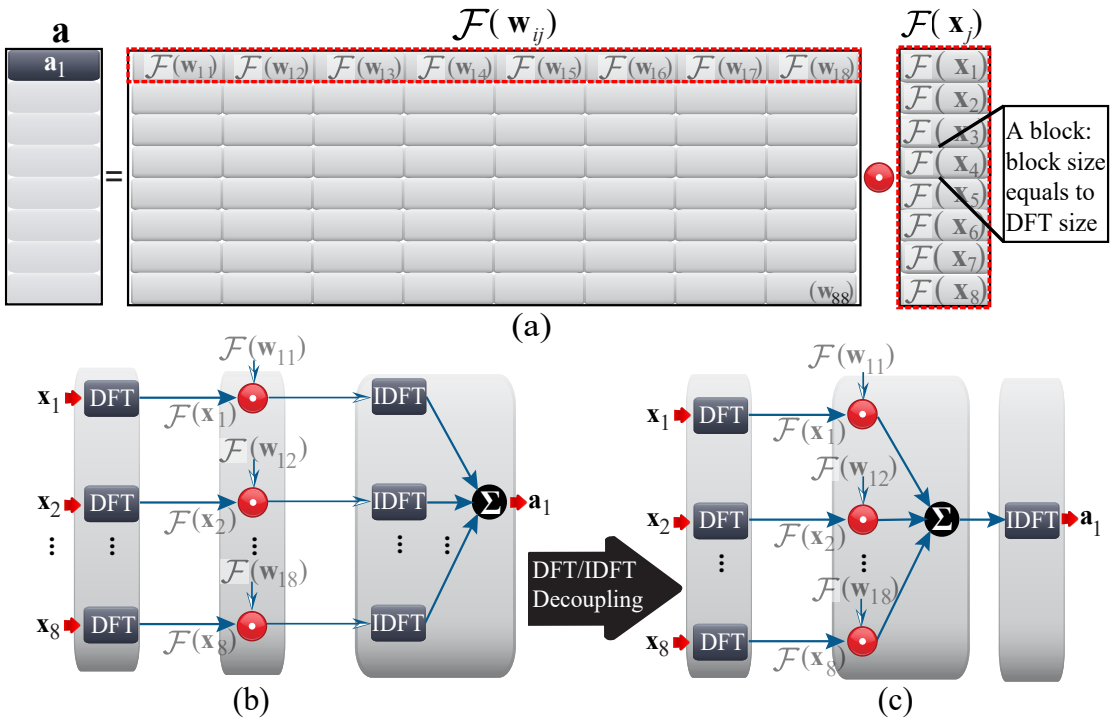


Figure 5.5: An illustration of the (a) circulant convolution operator; (b) its original implementation; (c) and the optimized implementation.

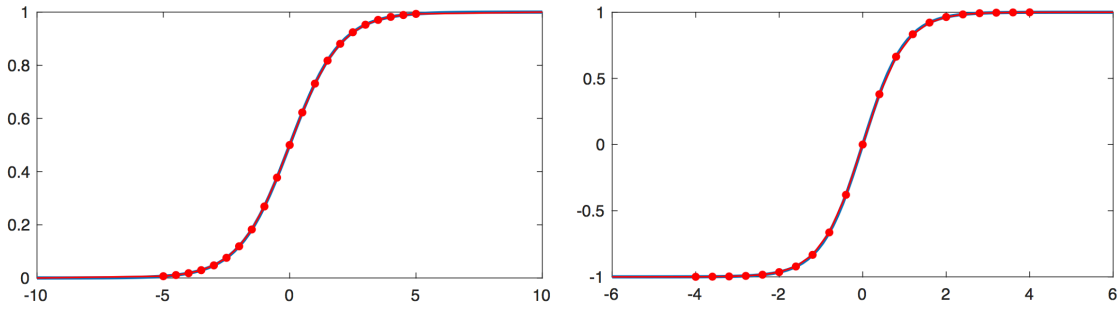
5.5.2 Datapath and Activation Quantization

The LSTM model size could be further compressed without accuracy degradation if the datapath of LSTM implementation on FPGA is carefully quantized into shorter bitwidth. We design a bit-accurate software simulator to study the impact of the bitwidth of datapath on the prediction accuracy. We first analyze the numerical range of the trained weights in the LSTM, and then determine the bitwidth of integer and fractional parts to avoid data overflow and accuracy degradation. We observe that 16-bit fixed point is accurate enough for implementing the LSTM inference model on FPGAs.

In order to alleviate accuracy degradation problem caused by the data truncation and overflow problems in the architecture of the proposed circulant con-

volution operator. It is observed that the output data of IDFT are first divided by the block size (or IDFT input size) k, which is implemented as right shifting the numbers by $\log_2 k$ bits, and then output in the last stage of IDFT pipeline. However, the more bits are right shifted, the more fractional bits are truncated and thus degrading the overall accuracy. In order to deal with the accuracy loss caused by the data truncation, we propose to evenly distribute the shift operations inside the stages of the IDFT pipeline based on the observation that right shifting one bit at a time achieves better accuracy than right shifting multiple bits at once. As for the data overflow problem, it is most likely to occur in the accumulation stage of circulant convolution operator since multiple values are summed here. We propose to move the evenly distributed right shifting operations from stages of IDFT pipeline to the ones of DFT. Since the DFT is processed before accumulation operator and right shifting makes the number to be smaller and, it is less likely to cause overflow in accumulation stage.

The activation functions in LSTMs are all transcendental functions whose implementations on FPGA are very expensive with respect to resource utilization. In order to achieve a balance between accuracy and resource cost, we propose to utilize quantized piece-wise linear functions to approximate them. Figure 5.6 shows that the sigmoid and tanh functions are approximated using piece-wise linear functions with 22 segments. As we can see from the figure, the approximated and the original functions are almost the same and the error rate is less than 1%. Since the linear function could be represented in the slope-intercept form like $y = ax + b$, we only need to store the associated slope a and intercept b for each piece of linear function. In the real implementation, the computational complexity of activation functions only involves a simple comparison to index the associated pair of slope and intercept and one 16-bit fixed



(a) sigmoid (b) tanh

Figure 5.6: Piece-wise linear activation functions.

point multiplication followed by an addition. It is necessary to note that, according to our experimental results, the piece-wise linear approximation incurs negligible accuracy degradation for LSTMs.

5.5.3 Operator Scheduling

We use LSTM as example to explain the operator scheduling in this section. The recurrent nature of LSTM enforces strict data dependency among operators inside the LSTM module. In order to accommodate the complicated interactions of LSTM primitive operators, we propose a graph generator to transform the LSTM algorithm specification in the form of the equations like Equation (1.8) to a directed acyclic data dependency graph. Figure 5.8 (a) shows the generated LSTM directed operator graph from the LSTM descriptions, where each node is an LSTM primitive operator and the edge represents the data dependency between two operators. It is necessary to note that the generated graph is acyclic because we deliberately remove the feedback edges from cell output $\mathbf{c}_t$ to the LSTM module output $\mathbf{y}_t$. Since the backward edges are taken care of by the double-buffer mechanism, this practice would never harm the correctness and efficiency of the final LSTM accelerator design.

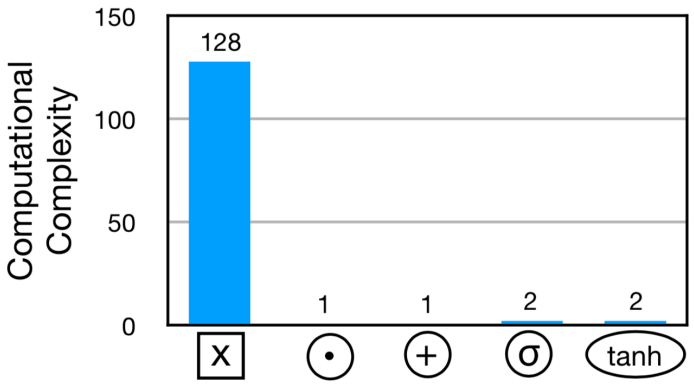
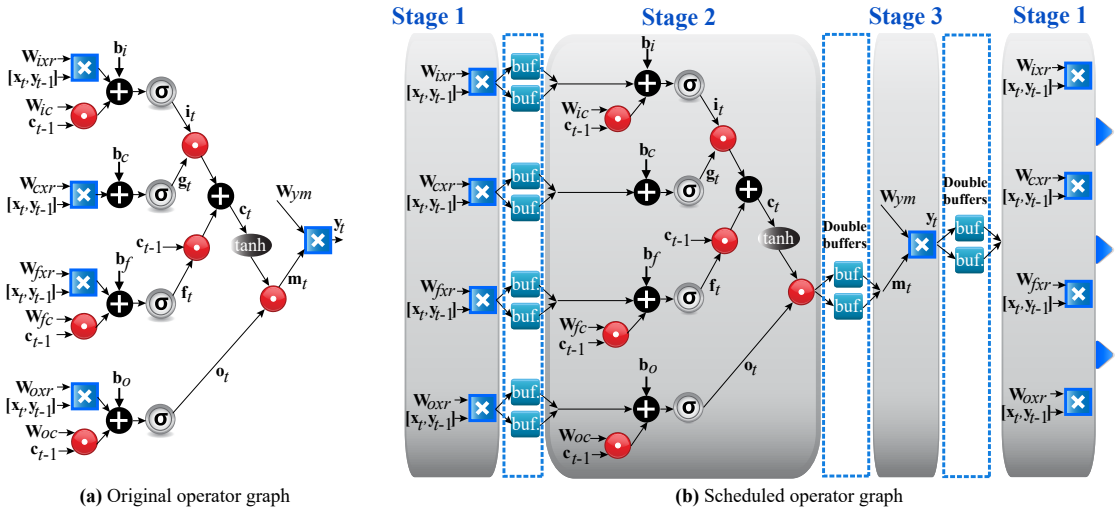


Figure 5.7: Computational complexity of LSTM operators.



(a) Original operator graph **(b)** Scheduled operator graph

Figure 5.8: Illustration of operator scheduling on data dependency graph. The circle represents the element-wise operator, and the square represents the circulant convolution operator.

LSTMs exhibit a highly skewed distribution of computation complexity among the primitive operators. Figure 5.7 shows the normalized computational complexity of the five primitive operators of the Google LSTM [109] studied in this work. The computational complexity gap between the circulant convolution operator and element-wise multiply operator ⊙ is as large as 128 times. So, if we want to pipeline these two operators we must either boost the parallelism of the former operator or make the latter operator wait (idle) for the former one. However, the reality is that the limited on-chip resources of FP-

GAs generally cannot sustain sufficient parallelism and the idle operators make the design inefficient. Therefore, pipelining a complex LSTM algorithm as a whole, such as the Google LSTM [109] shown in 5.8(a), is very inefficient on FPGAs. In order to deal with this problem, we propose to break down the orig-

Algorithm 4: Operator Scheduling Algorithm

Input: operator graph $G = (V, E)$, operator weight set $W(V)$, and priority set $P(V)$;
Output: operator subgraph of each stage $G_k = (V_k, E_k)$;
Traverse $G = (V, E)$ and compute priority set $P(V)$;
$k \leftarrow 0, N(V) \leftarrow \{\mathbf{1}\}$;
foreach $v_i \in V$ *in decreasing order of* $P(v)$ **do**
  if $k = 0$ **then**
    $k \leftarrow k + 1$;
    $G_k \leftarrow v$; `// add the operator to a new stage`
  else
    foreach $N'(v_j) \in G_k$ **do**
      $N'(v_j) \leftarrow N(v_j) \cdot \lceil \frac{W(v_j)}{W(v_i)} \rceil$;
    end
    if *resource constraints are satisfied* **then**
      $G_j \leftarrow v$; `// add the operator to current stage`
      $N(V) \leftarrow N'(V)$; `// update operator parallelisms`
    else
      $k \leftarrow k + 1$;
      $G_k \leftarrow v_i$; `// add the operator to a new stage`
    end
  end
end
$K \leftarrow k$;
Enumerate $R(G_k)$ values to maximize throughput and fully utilize FPGA resource;
return $N(V)$, $\{G_1, G_2, ..., G_K\}$, *and* $\{R(G_1), R(G_2), ..., (G_K)\}$;

inal single pipeline into several smaller coarse-grained pipelines and overlap their execution time by inserting double-buffers for each concatenated pipeline pair. For example, the original operator graph of Google LSTM [109] in 5.8(a) is divided into three stages in 5.8(b), where each stage will be implemented as a coarse-grained pipeline on FPGAs. The double-buffers added between

stages are used to buffer the data produced/consumed by the previous/current stage. However, scheduling the operators to different stages in an efficient way is still a problem. We propose an operator scheduling algorithm shown in Algorithm 4 to tackle this problem. The algorithm takes the original operator graph $G = (V, E)$, operator weight set $W(V)$, and operator priority set $P(V)$ as input and outputs several operator subgraphs G_k. For original operator graph $G = (V, E)$, each vertex $v_i \in V$ represents an operator and the edge e_{ij} represents the data dependency between v_i and v_j. Each vertex v_i has a weight $W(w_i)$ which is the associated arithmetic computational complexity. The algorithm first traverses down the graph from the source vertex computing the priority of each vertex by

$$P(v_i) = \begin{cases} W(v_i) + \max\limits_{v_j \in Succ(v_i)} P(v_j), & v_i \neq v_{sink} \\ W(v_{sink}), & \text{otherwise} \end{cases} \tag{5.8}$$

Since $P(v_i)$ is accumulated with the maximum value of successors $P(v_j)$ as shown in Equation (5.8), priority set $P(V)$ is topologically ordered, which means that it is guaranteed that all predecessor operators are scheduled before scheduling a new operator [69]. After the prioritization, the algorithm selects the operator with the highest priority value and then determines the parallelism of the operator $N(v_j)$ and whether it should be added to the current or a new stage according to the resource utilization of FPGAs. Then, the operator subgraphs G_k and the operator parallelism set $N(V)$ are output by this algorithm, where each stage represents a corresponding LSTM execution stage that will be implemented as a coarse-grained pipeline on FPGAs. Since the overall throughput of this coarse-grained pipeline design is constrained by the slowest stage, we need to further determine the pipeline replication factor $R(G_k)$ for each stage. To fully utilize

the resources of a certain FPGA chip, we also need to take into account of the resource utilization of each stage, and thus we propose to enumerate pipeline replication factor $R(G_k)$ to get the optimal setting with the help of our analytical performance and resource models which are presented in Section 5.5.4.

5.5.4 Performance and Resource Models

Since the throughput of the proposed coarse-grained pipeline design is constrained by the slowest stage, the analytical performance model is built as following,

$$FPS = \frac{Frequency}{\max\{T_1, T_2, ..., T_h, ...T_K\}}, \tag{5.9}$$

where FPS is the number of frames per second of E-RNN accelerator, T_k represents the number of execution clock cycles of stage k, and K is the total number of stages. T_k is calculated by considering the parallelism and input data size of each stage as following,

$$T_k = \lceil \max_{v_i \in G_k} \frac{Q(v_i)}{N(v_i)} / R(G_k) \rceil + D_k \tag{5.10}$$

where $Q(v_i)$ is the workload of operator v_i and D_k is the pipeline depth of stage k. It is necessary to note that, the compression ratio of the block-circulant matrices based technique is large enough to store the whole LSTM model on BRAMs of FPGAs, and for each frame, we only need to load very limited size of input data which makes computation time of LSTM to be overlapped with data loading.

The resource model of the highly optimized primitive operator templates is

very straightforward because the linear model with respect to the associated operator parallelism $N(v_i)$ and stage parallelism $R(G_k)$ is accurate enough to guide the design space exploration for energy-efficient designs. The models are shown in the following,

$$DSP = \sum_{k=1}^{K} R(G_k) \cdot \sum_{v_i \in V,} \Delta DSP(v_i) \cdot N(v_i), \tag{5.11}$$

$$BRAM = \sum_{k=1}^{K} R(G_k) \cdot \sum_{v_i \in V,} \Delta BRAM(v_i) \cdot N(v_i), \tag{5.12}$$

$$LUT = \sum_{k=1}^{K} R(G_k) \cdot \sum_{v_i \in V,} \Delta LUT(v_i) \cdot N(v_i), \tag{5.13}$$

where $\Delta DSP(v_i)$, $\Delta BRAM(v_i)$, and $\Delta LUT(v_i)$ are obtained by profiling the resource consumption values for operator v_i on the FPGA using the manually optimized operator template.

5.6 Hardware Design

5.6.1 E-RNN Hardware Architecture

Figure 5.9 demonstrates the E-RNN hardware architecture. A CPU and a host memory communicate with the FPGA chip through PCI-Express (PCIE) bus. They can transmit the input voice vector to the FPGA and receive the computation results from the accelerator on FPGA. The host memory initially stores all the parameters (weight matrices and biases) and input voice vectors, which will be further loaded into on-chip memories (BRAM) of FPGA for online inference.

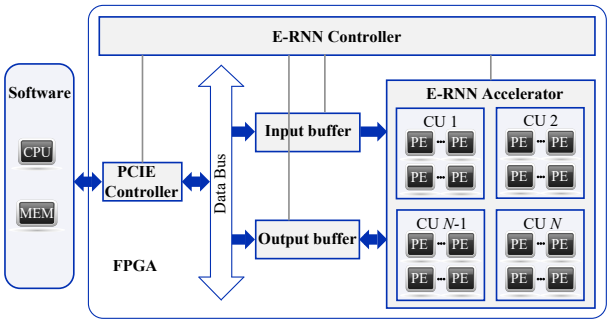


Figure 5.9: The overall E-RNN hardware architecture.

In the FPGA chip, we implement the E-RNN controller, E-RNN accelerator, PCIE controller, and input/output buffer. The E-RNN accelerator comprises a group of *processing elements* (PEs). PEs are the basic computation block for one set of input voice vectors with the corresponding weights and are primarily responsible for the computing tasks in LSTM and GRU. A handful of PEs and their peripheral components are bundled as a *compute unit* (CU). Each CU implements the LSTM/GRU model and computes one input voice vector sequence independently. The E-RNN controller takes charge of the process of data fetching of the PCIE controller. Most importantly, it determines the computation pipeline flow of the whole LSTM/GRU network. The on-chip input buffer and output buffer have the data ready for PEs and collect the output results from the accelerator. The E-RNN accelerator fetches parameters and input voice vectors from on-chip BRAM and collects the results and writes back to BRAM.

5.6.2 PE Design

As shown in Figure 5.10, a PE consists of two FFT operators, M multipliers, a conjugation operator, $\log_2 N$ right shifting registers, and an accumulator. The accumulator is an adder tree with N inputs (same as the FFT size). Due to the

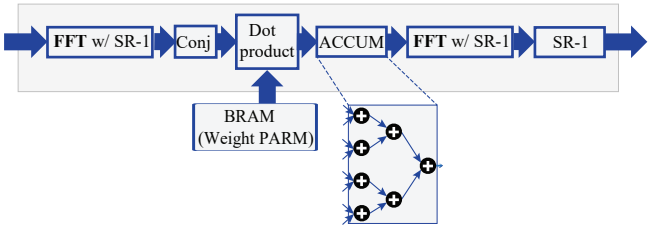


Figure 5.10: The PE design in FPGA implementation.

resource limitation on FPGAs, we need to let PEs operate using time-division multiplexing (TDM) for different blocks. Suppose the DSP and LUT usage of one PE are ΔDSP and ΔLUT, respectively. The number of PEs can be expressed as: $\#PE = min\{\lfloor \frac{DSP}{\Delta DSP} \rfloor, \lfloor \frac{LUT}{\Delta LUT} \rfloor\}$, where DSP, LUT are the total resources of DSP and LUT, respectively.

5.6.3 Compute Unit (CU) Implementation

CU implementation of LSTM

The proposed CU architecture for LSTM model described in Eqn. (1.8) can be implemented using above designs, shown in Figure 5.11. The architecture consists of multiple PEs, sigmoid/tanh, double buffers, and multiplier-adder block. There are five BRAM blocks. BRAM 1 stores input features. The weights matrices ($\mathbf{W}_{*(xr)}$ and $\mathbf{W}_c$) are stored in BRAM 2, 3. BRAM 4 stores bias vectors $\mathbf{b}$ and the projection matrix $\mathbf{W}_{ym}$ is stored in BRAM 5. Of course these weight matrices are stored with compression in the block-circulant framework. Based on data dependency of the LSTM model, we propose to adopt *multi-stage coarse-grained pipelining* (abbreviated as CGPipe) techniques, to achieve maximum performance under the resource constraints. The first CGPipe stage is responsible for multiplication of weights matrices (i.e.,$\mathbf{W}_{*(xr)}$) and input vectors

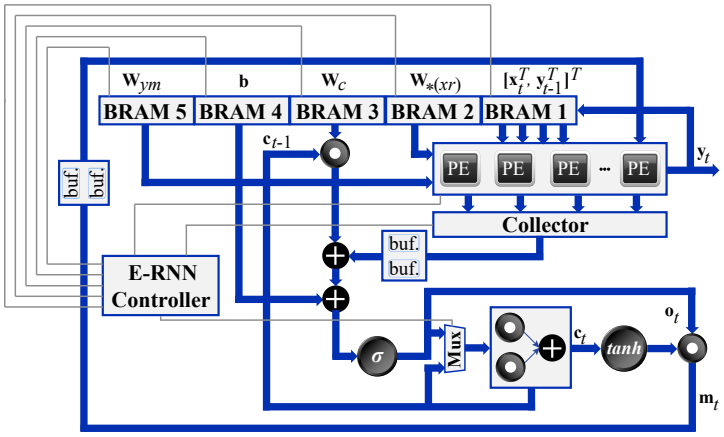


Figure 5.11: One compute unit (CU) with multiple processing elements (PEs) of LSTM.

$[\mathbf{x}_t^T, \mathbf{y}_{t-1}^T]^T$. The second CGPipe stage is in charge of non-matrix vector multiplications such as diagonal matrix-vector multiplication, bias addition, and activation functions. The third CGPipe stage processes the matrix-vector multiplication for projection matrix $\mathbf{W}_{ym}$ and projected output $\mathbf{y}_t$. A double buffer is inserted among each CGPipe stage to shorten the idle time. Fine-grained pipelining (abbreviated as FGPipe) methodology is utilized to schedule the associated sub-operations for each CGPipe stage. In our designs, double buffers are only used between each pair of concatenated coarse-grained pipelining stages and only 3 coarse-grained stages are used. Double buffers are not used for weights. Because the inputs/intermediate results of LSTM/GRU do not have high dimension (with dimension of 1,024, as example), the double buffers only account for a very small portion of BRAM resource.

The intermediate results ($\mathbf{c}_t$ and $\mathbf{m}_t$) are initialized to zero. To explain the mechanism of the architecture, we take the computation of forget gate $\mathbf{f}_t$ as a demonstration. As shown in Figure 5.11, input feature vectors $[\mathbf{x}_t^T, \mathbf{y}_{t-1}^T]^T$ fetched from BRAM 1 and weight matrices $\mathbf{W}_{f(xr)}$ fetched from BRAM 2 are prepared for PEs for the purpose of calculating $\mathbf{W}_{fx}\mathbf{x}_t$ and $\mathbf{W}_{fr}\mathbf{y}_{t-1}$ in CGPipe stage 1. $\mathbf{W}_{fc}\mathbf{c}_{t-1}$

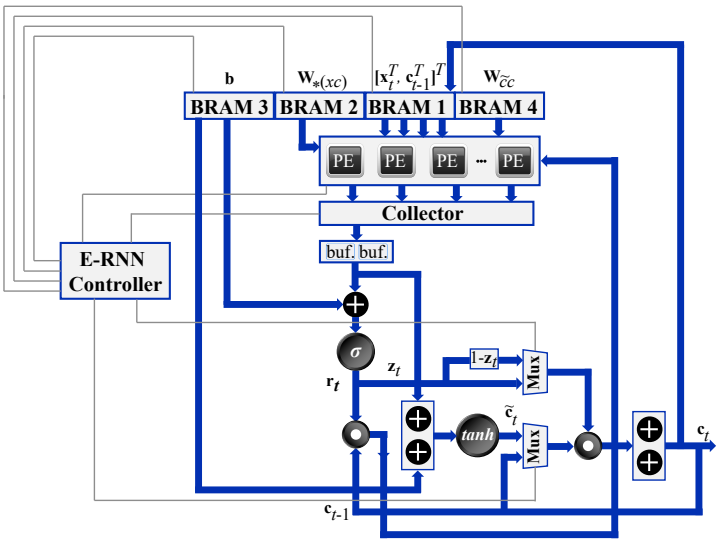


Figure 5.12: A compute unit (CU) with multiple processing elements (PEs) of GRU.

is generated by point-wise multiplication (a group of multipliers) in the first phase of CGPipe stage 2. Adder trees accumulate $\mathbf{W}_{fx}\mathbf{x}_t$, $\mathbf{W}_{fr}\mathbf{y}_{t-1}$, $\mathbf{W}_{fc}\mathbf{c}_{t-1}$, and bias $\mathbf{b}_f$ in the second phase of CGPipe stage 2. After passing the intermediate data through the activation function σ, E-RNN produces the result $\mathbf{f}_t$. The computations of other gates are implemented similarly. In the third phase of CGPipe stage 2, the computed gate outputs ($\mathbf{i}_t$, $\mathbf{g}_t$, and $\mathbf{f}_t$) are then fed into the multiplier-adder block. By multiplying $\mathbf{o}_t$ with the intermediate result from *tanh* activation, E-RNN produces the projected output $\mathbf{m}_t$. Output $\mathbf{y}_t$ will be written back to BRAM 1 and replace $\mathbf{y}_t$ for the next recurrent process ($\mathbf{y}_{t-1} \leftarrow \mathbf{y}_t$) after CGPipe stage 3.

CU Implementation of GRU

The CU of GRU model described in Eqn. (1.9) can also be implemented using above design. The proposed architecture for GRU is shown in Figure 5.12,

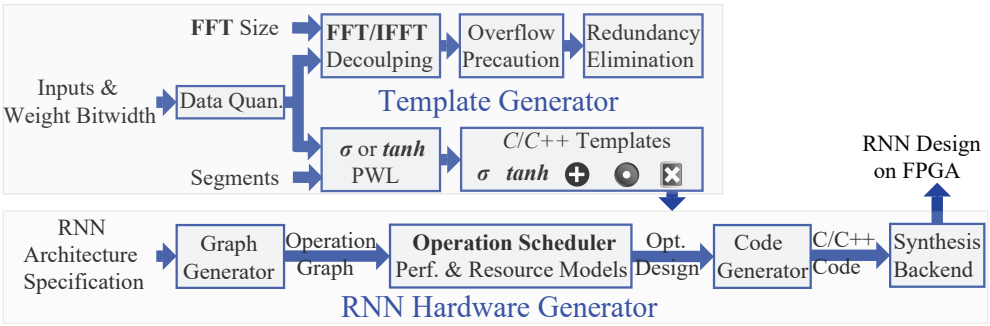


Figure 5.13: Overview of high level synthesis framework.

which contains multiple PEs, double buffer, sigmoid/tanh, adder tree, and element-wise multiplier. GRU architecture has four BRAM blocks, in which input feature vectors $[\mathbf{x}_t^T, \mathbf{c}_{t-1}^T]^T$ are stored in BRAM 1. Weight matrix $\mathbf{W}_{*(xc)}$ is stored in BRAM 2. Bias values (including $\mathbf{b}_z$, $\mathbf{b}_r$, and $\mathbf{b}_{\tilde{c}}$) are stored in BRAM 3, and weight matrix $\mathbf{W}_{\tilde{c}x}$ is stored in BRAM 4.

Multi-stage CGPipe techniques are utilized based on data dependency of the GRU model, to separate the timing and resource-consuming matrix-vector operations. In GRU, the first CGPipe stage takes charge of multiplication of $\mathbf{W}_{*(xc)}[\mathbf{x}_t^T, \mathbf{c}_{t-1}^T]^T$. The second CGPipe stage computes the multiplication of $\mathbf{W}_{\tilde{c}c}(\mathbf{r}_t \odot \mathbf{c}_{t-1})$ ($\mathbf{r}_t$ calculated in the first CGPipe stage) and $\mathbf{W}_{\tilde{c}x}\mathbf{x}_t$. The third CGPipe stage is responsible for the point-wise multiplication, activation functions, and summation operations. In the proposed GRU architecture, CGPipe stage 1 and CGPipe stage 2 can be implemented using the same hardware resource of FPGA with TDM method.

High-Level Synthesis (HLS) Exploration

We have developed an HLS framework for automatically converting high-level descriptions of RNNs into FPGA implementations, with the framework

overview shown in Figure 5.13. This is a template-based framework for design automation of RNN implementations, based on the above described optimizations. The HLS framework consists of two parts which are the *primitive operation templates generator* and the *RNN hardware design generator*. More details are provided as follows:

Template Generator: We develop the C/C++ based template for each of the primitive operations in RNNs, e.g., *tanh*, sigmoid σ, point-wise vector addition, point-wise multiplication, and "FFT→element-wise multiplication →IFFT" procedure.

Graph Generator: In order to extract the complicated interactions among primitive operations in an RNN model, we design a graph generator that produces a directed acyclic data dependency and operation graph unrolling the computations in RNNs. We deliberately remove the feedback edges of $\mathbf{c}_t$ and $\mathbf{y}_t$, which are taken care of by the *double-buffer mechanism*, and therefore do not harm the correctness and efficiency of the RNN.

Operation Scheduler: The computational complexities of the primitive operations in RNN exhibit a highly skewed distribution. For example, the complexity of matrix-vector multiplication $[\mathbf{W}_{*x} \ \ \mathbf{W}_{*r}][\mathbf{x}_t^T, \mathbf{y}_{t-1}^T]^T$ is 128× as that of point-wise multiplication $\mathbf{W}_{ic} \odot \mathbf{c}_{t-1}$. Therefore, we develop an automatic operation scheduler to generate a pipeline scheme given the data dependency and operation graph from the graph generator. The objective is to maximize throughput under hardware resource constraints.

Code Generator and Synthesis Backend: The code generator takes the operation scheduling result as input and generates the final C/C++ code automat-

Table 5.3: Comparison of two selected FPGA platforms

FPGA Platform	DSP	BRAM	LUT	FF	Process
ADM-PCIE-7V3	3,600	1,470	859,200	429,600	28nm
XCKU060	2,760	1,080	331,680	663,360	20nm

ically by integrating the involved primitive operations. The generated C/C++ code for RNN is then fed to an off-the-shelf commercial synthesis backend to generate the FPGA implementation.

5.7 Experiment Evaluation

Experiment Platform. We use two FPGA platforms for evaluating the proposed E-RNN framework for LSTM and GRU RNNs: Alpha Data's ADM-PCIE-7V3 and Xilinx KU060. The ADM-PCIE-7V3 board, comprising a Xilinx Virtex-7 (690t) FPGA and a 16GB DDR3 memory, is connected to the host machine through PCIE Gen3 $\times$ 8 I/O Interface. Xilinx KU 060 is a Kintex UltraScale serial FPGA with two 4GB DDR3 memory. The host machine adopted in our experiments is a server configured with multiple Intel Core i7-4790 processors. The detailed comparison of on-chip resources of the two FPGA platforms is presented in Table 5.3. We use Xilinx SDX 2017.1 as the commercial high-level synthesis backend to synthesize the high-level (C/C++) based RNN designs on the selected FPGAs. The E-RNN framework of FPGA implementation of (LSTM and GRU) RNNs are operating at 200MHz on both platforms, which is configured to be the same as the prior works ESE [42] and C-LSTM [124] for fair comparisons.

We evaluate the performance on both FPGA platforms for LSTM and GRU RNNs using the same TIMIT dataset, which is the same dataset utilized in the

Table 5.4: Detailed comparisons for different (LSTM and GRU) RNN designs on FPGAs (ours, ESE, and C-LSTM).

	ESE [42]	**C-LSTM FFT8 [124]** (Block size: 8)		**E-RNN FFT8** (Block size: 8)		**E-RNN FFT16** (Block size: 16)		**E-RNN FFT8** (Block size: 8)		**E-RNN FFT16** (Block size: 16)	
RNN Cell	LSTM-1024 w/ projection-512 [109, 42]							GRU-1024			
Matrix Size (#Params of top layer)	0.73M	0.41M				0.20M		0.45M		0.23M	
Quantization	12bit fixed	16bit fixed		12bit fixed							
Matrix Compression Ratio	4.5 : 1[a]	7.9 : 1[c]				15.9 : 1		8.0 : 1		15.9 : 1	
Platform	KU060	7V3		KU060	7V3	KU060	7V3	KU060	7V3	KU060	7V3
DSP (%)	54.5	74.3		95.4	85.6	96.4	79.6	79.0	62.1	79.5	64.3
BRAM (%)	87.7	65.7		88.1	78.5	90.3	65.2	90.8	88.2	81.2	79.5
LUT (%)	88.6	58.7		77.6	74.0	76.5	59.4	81.2	78.8	72.5	67.4
FF (%)	68.3	46.5		61.2	52.3	65.1	55.3	72.4	73.2	65.2	60.3
Frequency (MHz)	200										
PER Degradation	0.30%	0.32%		0.14%		0.31%		0.18%		0.33%	
Latency (μs)	57.0	16.7		13.7	12.9	7.4	8.3	10.5	10.5	6.7	6.5
Frames per Second (FPS)	17,544[b]	179,687		231,514	240,389	429,327	382,510	284,540	284,463	445,167	464,582
Power (W)	41	22		-	24	-	25	-	22	-	29
Energy Efficiency (FPS/W)	428	8,168		-	10,016	-	15,300	-	12,930	-	16,020

[a] This estimation considers both weights and indices (there is at least one index per weight after compression in ESE). However, this is a pessimistic estimation for ESE because indices can use fewer bits for representation than weights.
[b] We use ESE's theoretical computation time to calculate FPS, the real computation time is larger than theoretical one which leads to smaller FPS.
[c] We measure the compression ratio by the number of parameters in matrices. As the network architectures are identical in C-LSTM and E-RNN, their matrix compression ratios are the same.

prior works ESE and C-LSTM. The latencies of E-RNN framework implementation are measured by the total number of clock cycles (N_{CC}) multiplied by the clock period T (5 ns) from the Xilinx SDx tools, and power/energy consumptions are from actual power measurements. For platform KU060, since we do not have the physical platform for power measurement, we leave the power and energy efficiency values to be blank in Table 5.4. As shown in Table 5.4 with detailed comparison results, we explore on both LSTM and GRU, with two different block sizes 8 and 16, on both selected FPGA platforms. The bit length is optimized to be 12 bits, which is validated to result in no additional accuracy degradation due to quantization. We use the same baseline LSTM model with ESE. (i) We present a comparison between E-RNN with block size 8 and ESE, in which case the compression ratio will be similar. The comparison aims to demonstrate the lower accuracy degradation and higher performance achieved by E-RNN; (ii) we present a comparison between E-RNN with block size 16 and ESE, in which case the accuracy degradation will be similar. The comparison

aims to demonstrate that E-RNN achieves better performance and energy efficiency under the same accuracy degradation; (iii) we compare the performance and energy efficiency between E-RNN and C-LSTM using the same block size (both are based on the block-circulant matrix-based framework), to illustrate the effectiveness of the design optimization framework; (iv) we provide the results of E-RNN based on GRU model, for further enhancement on performance and energy efficiency.

Comparison with ESE

When the block size is 8, the compression ratio of E-RNN is similar compared with ESE. The comparison results, as shown in the first and third columns of Table 5.4, are both on the KU060 FPGA platform. We could observe that the E-RNN achieves lower accuracy degradation compared with ESE (0.14% vs. 0.30%), demonstrating the effectiveness of the block-circulant framework in terms of accuracy. We can also observe that E-RNN achieves 13.2× performance improvement, with an energy efficiency improvement of 23.4× using actual measurement results on the ADM-PCIE-7V3 board. It is necessary to note that as shown in Table 5.3, the manufacturing process of XCKU060 FPGA is 20nm while the process of Virtex-7 is 28nm, which means the energy efficiency gain reported here is even conservative.

Although the compression ratios are similar, the significant efficiency improvement is because of the following two reasons. First, the block-circulant framework results in a regular network structure, and therefore a significantly higher degree of parallelism. As an illustrative example, we can implement in parallel 16 FFTs, each with 16 inputs, in parallel in FPGA. In contrast, it will

be especially difficult for ESE to operate in parallel $16 \times 16 = 256$ inputs when the network is stored in the irregular structure (one weight indexing another). The second reason is the efficient implementations of tanh and sigmoid activation functions. Our piecewise linear approximation method can support activation implementation only using on-chip resources. In contrast, the ESE implements activations in look-up tables, and therefore requires off-chip DDR storage if enough parallelism is required (although it is possible to store all weight parameters of ESE on-chip). The latter reason accounts for more than 2× energy efficiency gain and the majority is attributed to the regularity benefit. As a side evidence, the LUT and FF utilizations of E-RNN are lower than ESE, which shows that E-RNN has less boolean and numeric nodes due to the regularity.

With block size 16, the accuracy degradation of E-RNN (using LSTM model) is similar as ESE. As shown in the first and fifth column of Table 5.4, the E-RNN achieves 24.47 × performance improvement, with a energy efficiency improvement of 35.75 × using ADM-7V3 platform compared with ESE. The results are at least 50% higher than results of E-RNN with block size 8.

Comparison with C-LSTM

We applied ADMM to well trained RNN models to train the block circulant matrices. As ADMM does not hurt the original model performance theoretically, but only convert the matrices to block circulant format, the accuracy degradation is smaller than C-LSTM. As a result, E-RNN achieves lower PER degradation than C-LSTM when given the same block size (0.14% vs. 0.32% with block size of 8). We compare the performance and energy efficiency between E-RNN and C-LSTM using the same block size 8 (both are based on the block-

circulant matrix-based framework). We can observe that E-RNN achieves 1.33× performance improvement with a block size of 8, with an energy efficiency improvement of 1.22× using the same ADM-PCIE-7V3 board. The similar observation is also obtained from comparison using block size of 16: E-RNN (using LSTM) achieves 1.32× performance and 1.06× energy efficiency improvement compared with C-LSTM. These improvements are attributed to the design optimization framework, including hardware system design, PE optimization, and quantization.

Among the three, the first two components are more effective compared to quantization: reducing from 16 bit to 12 bit only accounts for less than 10% performance improvement. Compared to C-LSTM, E-RNN has a systematic architecture including PE and CU for both LSTM and GRU. In addition, the optimization target of E-RNN is in the bottom level, i.e., PE level. The seemingly counterintuitive observation is because the same number of DSP blocks are utilized in FPGA (on the other hand, BRAM does not account for a large portion of energy consumption in FPGA).

Experimental Results on GRU

As shown in the right four columns of Table 5.4, compared with ESE, C-LSTM, and E-RNN with LSTM, we can observe that the E-RNN with GRU model achieves 26.48×, 2.59×, and 1.21× performance improvement under the same accuracy degradation, respectively. For the perspective of energy efficiency, the E-RNN with GRU model can achieve 37.4×, 2.0×, and 1.05× improvement, respectively. Experimental results show that the design optimization framework E-RNN with GRU model can have the best performance and energy efficiency.

We verify that if the accuracy requirement can be satisfied, it is desirable to shift from LSTM to GRU because of less computation and storage.

5.8 Conclusion

In this work, we employ a structured compression technique (block-circulant matrices) to compress the RNN model small enough to be fitted on BRAMs of FPGA. Besides the reduced model size, the irregular computation and memory accesses have been completely eliminated by the regular structure of the block-circulant matrices. Moreover, an efficient FFT based fast circulant convolution is applied to accelerate the RNN computation by reducing both the computational and storage complexities. we use ADMM-based training for deriving block-circulant matrice-based RNN representation to achieve high accuracy. In order to accommodate a wide range of RNN variants, we also propose an automatic optimization and synthesis framework. We explore on both LSTM and GRU using the proposed E-RNN and we provide comprehensive comparisons. Experimental results demonstrate the effectiveness of the proposed framework E-RNN compared with the prior works.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we studied three computing paradigms to accelearate the machine learning inference. We offered better model to improve the performance. Meantime, the inference framework achieves higher energy efficiency and faster speed.

In Chapter 2, we discussed the cogent confabulation based sentence completion algorithm. Using Chinese language as a case study, we developer better model for sentence completion problem through mathematical analysis. With incremental comparison experiments, we proved the optimization for the sentence confabulation model is efficient to improve the model performance.

In Chapter 3, we deployed the sentence confabulation model on a multiprocessing system. The cogent confabulation model was implemented and modified in a parallel manner. The system exploited multi-threading to build a parallel structure to process lexicons in sentences. We optimized the system by using intermittent pruning to overcome the compute speed overhead due to cache coherence and this also improved the accuracy performance of the framework.

In Chapter 4, We applied an approximate computing paradigm, stochastic computing to the deep convolutional neural networks. By decomposing the DCNNs, we built fuctional blocks using SC components and joint optimization along the datapath was conducted to main a high inference accuracy with

ultra low hardware footprint and energy/power cost. We presented two SC based frameworks, SC-DCNN and HEIF. HEIF was an optimized version of SC-DCNN. Both frameworks achieved high energy efficiency, throughputs, and area efficiency for the DCNNs with less accuracy loss.

In Chapter 5, we investigated the a structured compression technique using block-circulant matrices to compress the RNN model small enough to be fitted on BRAMs of FPGA. An efficient FFT based fast circulant convolution is applied to accelerate the RNN model computation by reducing both the computational and storage complexities. we used ADMM-based training for deriving block-circulant matrice-based RNN representation to achieve high accuracy. In order to accommodate a wide range of RNN variants, we also propose an automatic optimization and synthesis framework called E-RNN. We explored on both LSTM and GRU using the proposed E-RNN and Comprehensive experimental results demonstrate the effectiveness of the proposed framework E-RNN compared with the prior works.

6.2 Future Directions

6.2.1 Applying Structure Matrices to DCNN

We presented a holistic framework for energy efficient high-performance highly-compressed DNN hardware design in [83]. This design can be applied to the fully-connected layers in the modern neural networks. Meanwhile, in [126, 29], we proposed an CirCNN architecture, a universal DNN inference engine that can be implemented in various hardware/software platforms with

configurable network architecture. The inference engine can not only accelerate the computation for the fully-connected layer (including RNNs), but also accelerate the computation of convolution layers by reshaping the convolution tensors. Till now, all the components can be accelerated using block circulant matrices. As the deep learning evolves fast, modern deep learning models utilize deeper architectures with lots of convolution layers. We propose to evaluate our block circulant matrices based acceleration method on modern neural network structures like ResNet [46], wide ResNet [137], etc..

6.2.2 Accelerating structured matrices on GPGPU

The structure matrices have been fully proven to support an energy-efficient fast inference for deep neural networks based on our previous discussion. However, the structural acceleration on the GPU has not been demonstrated. We measured the wall time of block-circulant DCNN on GPU platform but do not observe obvious speedup. We believe such phenomenon results from the insufficient low-level optimization for block-circulant operations on GPU such tensor padding/reshaping and FFT. Unlike matrix operation, FFT operation is far less efficiently implemented and optimized on GPU. For instance, computation of block-circulant DCNN requires a fine-grained parallel design consisting of both FFT/IFFT operations and summations over divided blocks. However, block-level synchronization on GPU has been a difficult problem. Instead, Nvidia specifically optimizes GPU micro-architecture to accelerate matrix multiplication. Thus, a low-level customized implementation and joint optimization for the block circulant operations on GPU are urgently needed.

BIBLIOGRAPHY

[1] Stanford cs class, cs231n: Convolutional neural networks for visual recognition, 2016.

[2] Nangate 45nm Open Library, Nangate Inc., 2009.

[3] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla, Nabil Imam, Yutaka Nakamura, Pallab Datta, Gi-Joon Nam, Brian Taba, Michael Beakes, Bernard Brezzo, Jente B. Kuang, Rajit Manohar, William P. Risk, Bryan Jackson, and Dharmendra S. Modha. Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(10):1537–1557, 2015.

[4] James A Anderson. *An introduction to neural networks*. MIT press, 1995.

[5] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. *arXiv preprint arXiv:1606.05487*, 2016.

[6] Arash Ardakani, François Leduc-Primeau, Naoya Onizawa, Takahiro Hanyu, and Warren J Gross. Vlsi implementation of deep neural network using integral stochastic computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2688–2699, 2017.

[7] Roberto Battiti. Using mutual information for selecting features in supervised neural net learning. *Neural Networks, IEEE Transactions on*, 5(4):537–550, 1994.

[8] David Bennett and Christopher Hill. *Sensory integration and the unity of consciousness*. MIT Press, 2014.

[9] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*. Citeseer, 2011.

[10] Dario Bini, Victor Pan, and Wayne Eberly. Polynomial and matrix computations volume 1: Fundamental algorithms. *SIAM Review*, 38(1), 1996.

[11] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, et al. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine learning*, 3(1):1–122, 2011.

[12] Bradley D Brown and Howard C Card. Stochastic neural computation. i. computational elements. *IEEE Transactions on computers*, 50(9):891–905, 2001.

[13] Robert Burbidge, Matthew Trotter, B Buxton, and Sl Holden. Drug design by machine learning: support vector machines for pharmaceutical data analysis. *Computers & chemistry*, 26(1):5–14, 2001.

[14] Bryan Catanzaro. Deep learning with cots hpc systems. 2013.

[15] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[16] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263. IEEE, 2016.

[17] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[18] Yu Cheng, Felix X Yu, Rogerio S Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu Chang. An exploration of parameter redundancy in deep networks with circulant projections. In *ICCV*, 2015.

[19] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[20] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[21] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.

[22] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[23] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*, 2011.

[24] Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. An accurate gpu performance model for effective control flow divergence optimization. In *IPDPS*, 2012.

[25] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

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

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

[28] Li Deng and Dong Yu. Deep learning. *Signal Processing*, 7:3–4, 2014.

[29] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408. ACM, 2017.

[30] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[31] Steve K Esser, Rathinakumar Appuswamy, Paul Merolla, John V Arthur, and Dharmendra S Modha. Backpropagation for energy-efficient neuromorphic computing. In *Advances in Neural Information Processing Systems*, pages 1117–1125, 2015.

[32] Steven K Esser, Paul A Merolla, John V Arthur, Andrew S Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J Berg, Jeffrey L McKinstry, Timothy Melano, Davis R Barch, et al. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 2016.

[33] Brian R Gaines. Stochastic computing. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 149–156. ACM, 1967.

[34] Brian R Gaines. Stochastic computing systems. In *Advances in information systems science*, pages 37–172. Springer, 1969.

[35] John S Garofolo, Lori F Lamel, William M Fisher, Jonathon G Fiscus, and David S Pallett. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *NASA STI/Recon technical report n*, 93, 1993.

[36] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*, 2000.

[37] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[39] The Stanford Natural Language Processing Group. Chinese natural language processing and speech processing. *Available: http://nlp.stanford .edu/projects/chinese-nlp. shtml*.

[40] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *ASP-DAC*, 2017.

[41] Nils Y Hammerla, Shane Halloran, and Thomas Ploetz. Deep, convolu-

tional, and recurrent models for human activity recognition using wearables. *arXiv preprint arXiv:1604.08880*, 2016.

[42] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *FPGA*, 2017.

[43] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528*, 2016.

[44] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[45] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[47] Robert Hecht-Nielsen. *Confabulation theory: the mechanism of thought*. Springer Heidelberg, 2007.

[48] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8), 1997.

[49] Miao Hu, Hai Li, Yiran Chen, Qing Wu, Garrett S Rose, and Richard W Linderman. Memristor crossbar-based neuromorphic computing system: A case study. *IEEE transactions on neural networks and learning systems*, 25(10):1864–1878, 2014.

[50] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[51] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[52] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 3d convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.

[53] Yuan Ji, Feng Ran, Cong Ma, and David J Lilja. A hardware implementation of a radial basis function neural network using stochastic logic. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 880–883. EDA Consortium, 2015.

[54] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[55] Rong Jin. Deep learning at alibaba. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 11–16. AAAI Press, 2017.

[56] George Kachergis, Chen Yu, and Richard M Shiffrin. An associative model of adaptive inference for learning word–referent mappings. *Psychonomic bulletin & review*, 19(2):317–324, 2012.

[57] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[58] Kyounghoon Kim, Jungki Kim, Joonsang Yu, Jungwoo Seo, Jongeun Lee, and Kiyoung Choi. Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. In *Proceedings of the 53rd Annual Design Automation Conference*, page 124. ACM, 2016.

[59] Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. Approximate derandomizer for stochastic circuits. *Proc. ISOCC*, 2015.

[60] Kyounghoon Kim, Jongeun Lee, and Kiyoung Choi. An energy-efficient random number generator for stochastic circuits. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 256–261. IEEE, 2016.

[61] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks Using Frequency-Domain Computation. In *DAC*, 2017.

[62] Kishore Reddy Konda, Achim Königs, Hannes Schulz, and Dirk Schulz. Real time interaction with mobile robots using hand gestures. In *Proceedings of the seventh annual ACM/IEEE international conference on Human-Robot Interaction*, pages 177–178. ACM, 2012.

[63] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[64] Daniel Larkin, Andrew Kinane, Valentin Muresan, and Noel O'Connor. An efficient hardware architecture for a neural network activation function generator. In *International Symposium on Neural Networks*, pages 1319–1327. Springer, 2006.

[65] Endre László, Péter Szolgay, and Zoltán Nagy. Analysis of a gpu based cnn implementation. In *2012 13th International Workshop on Cellular Nanoscale Networks and their Applications*, pages 1–5. IEEE, 2012.

[66] Yann LeCun. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 2015.

[67] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[68] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[69] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. Orchestrating multiple data-parallel kernels on multiple devices. In *PACT*, 2015.

[70] Ji Li, Ao Ren, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, and Yanzhi Wang. Towards acceleration of deep convolutional neural networks using stochastic computing. In *The 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE*, 2017.

[71] Ji Li, Zihao Yuan, Zhe Li, Caiwen Ding, Ao Ren, Qinru Qiu, Jeffrey Draper, and Yanzhi Wang. Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 1230–1236. IEEE, 2017.

[72] Ji Li, Zihao Yuan, Zhe Li, Ao Ren, Caiwen Ding, Jeffrey Draper, Shahin Nazarian, Qinru Qiu, Bo Yuan, and Yanzhi Wang. Normalization and dropout for stochastic computing-based deep convolutional neural networks. *Integration, the VLSI Journal*, 2017.

[73] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. Fpga acceleration of recurrent neural network based language model. In *FCCM*, 2015.

[74] Zhe Li, Caiwen Ding, Siyue Wang, Wujie Wen, Youwei Zhou, Chang Liu, Qinru Qiu, Wenyao Xu, Xue Lin, Xuehai Qian, and Yanzhi Wang. E-rnn: Design optimization for efficient recurrent neural networks in fpgas. In *High Performance Computer Architecture (HPCA), 2019 IEEE International Symposium on*. IEEE, 2019.

[75] Zhe Li, Ji Li, Ao Ren, Ruizhe Cai, Caiwen Ding, Xuehai Qian, Jeffrey Draper, Bo Yuan, Jian Tang, Qinru Qiu, et al. Heif: Highly efficient stochastic computing based inference framework for deep neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[76] Zhe Li, Ji Li, Ao Ren, Caiwen Ding, Jeffrey Draper, Qinru Qiu, Bo Yuan, and Yanzhi Wang. Towards budget-driven hardware optimization for deep convolutional neural networks using stochastic computing. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018.

[77] Zhe Li and Qinru Qiu. Completion and parsing chinese sentences using cogent confabulation. In *Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), 2014 IEEE Symposium on*, pages 31–38. IEEE, 2014.

[78] Zhe Li, Qinru Qiu, and Mangesh Tamhankar. Towards parallel implementation of associative inference for cogent confabulation. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–6. IEEE, 2016.

[79] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Yanzhi Wang, and Bo Yuan. Dscnn: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 678–681. IEEE, 2016.

[80] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Bo Yuan, Jeffrey Draper, and Yanzhi Wang. Structural design optimization for deep convolutional neural net-

works using stochastic computing. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 250–253. European Design and Automation Association, 2017.

[81] Zhe Li, Ao Ren, Ji Li, Qinru Qiu, Bo Yuan, Jeffrey Draper, and Yanzhi Wang. Structural design optimization for deep convolutional neural networks using stochastic computing. 2017.

[82] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. Efficient gpu spatial-temporal multitasking. *TPDS*, 26(3), 2015.

[83] Siyu Liao, Zhe Li, Xue Lin, Qinru Qiu, Yanzhi Wang, and Bo Yuan. Energy-efficient, high-performance, highly-compressed deep neural network design using block-circulant matrices. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 458–465. IEEE Press, 2017.

[84] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 369–381. ACM, 2015.

[85] Y Liu and KK Parhi. Lattice fir digital filters using stochastic computing. In *Proceedings of 2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*.

[86] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. In *FCCM*.

[87] Frederic Maire, Luis Mejias, and Amanda Hodgson. A convolutional neural network for automatic analysis of aerial imagery. In *Digital lmage Computing: Techniques and Applications (DlCTA), 2014 International Conference on*, pages 1–8. IEEE, 2014.

[88] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE, 2011.

[89] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural

networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 575–580. IEEE, 2016.

[90] Daniel Neil and Shih-Chii Liu. Minitaur, an event-driven fpga-based spiking network accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628, 2014.

[91] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. Accelerating recurrent neural networks in analytics servers: comparison of fpga, cpu, gpu, and asic. In *FPL*, 2016.

[92] Alan V Oppenheim. *Discrete-time signal processing*. Pearson Education India, 1999.

[93] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[94] Victor Pan. *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media, 2012.

[95] Behraoz Parhami and Chi-Hsiang Yeh. Accumulative parallel counters. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 2, pages 966–970. IEEE, 1995.

[96] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[97] Weikang Qian, Xin Li, Marc D Riedel, Kia Bazargan, and David J Lilja. An architecture for fault-tolerant computation with stochastic logic. *IEEE Transactions on Computers*, 60(1):93–105, 2011.

[98] Qinru Qiu, Zhe Li, Khadeer Ahmed, Hai Helen Li, and Miao Hu. Neuromorphic acceleration for context aware text image recognition. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014.

[99] Qinru Qiu, Zhe Li, Khadeer Ahmed, Wei Liu, Syed Faisal Habib, Hai Helen Li, and Miao Hu. A neuromorphic architecture for context aware text image recognition. *Journal of Signal Processing Systems*, pages 1–15, 2015.

[100] Qinru Qiu, Qing Wu, Martin Bishop, Robinson E Pino, and Richard W

Linderman. A parallel neuromorphic text recognition system and its implementation on a heterogeneous high-performance computing cluster. *Computers, IEEE Transactions on*, 62(5):886–899, 2013.

[101] Qinru Qiu, Qing Wu, Daniel J Burns, Michael J Moore, Robinson E Pino, Morgan Bishop, and Richard W Linderman. Confabulation based sentence completion for machine reading. In *Computational Intelligence, Cognitive Algorithms, Mind, and Brain (CCMB), 2011 IEEE Symposium on*, pages 1–8. IEEE, 2011.

[102] Qinru Qiu, Qing Wu, and Richard Linderman. Unified perception-prediction model for context aware text recognition on a heterogeneous many-core platform. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1714–1721. IEEE, 2011.

[103] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. *ACM SIGOPS Operating Systems Review*, 51(2):405–418, 2017.

[104] Ao Ren, Zhe Li, Yanzhi Wang, Qinru Qiu, and Bo Yuan. Designing reconfigurable large-scale deep learning systems using stochastic computing. In *Rebooting Computing (ICRC), IEEE International Conference on*, pages 1–7. IEEE, 2016.

[105] Ao Ren, Zhe Li, Yanzhi Wang, Qinru Qiu, and Bo Yuan. Designing reconfigurable large-scale deep learning systems using stochastic computing. In *2016 IEEE International Conference on Rebooting Computing*. IEEE, 2016.

[106] Anamitra Bardhan Roy, Debasmita Dey, Bidisha Mohanty, and Devmalya Banerjee. Comparison of FFT, DCT, DWT, WHT compression techniques on electrocardiogram and photoplethysmography signals. In *IJCA*, 2012.

[107] Kyle Rupnow, Yun Liang, Yinan Li, Dongbo Min, Minh Do, and Deming Chen. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *FPT*, 2011.

[108] Tara N Sainath, Abdel-rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep convolutional neural networks for lvcsr. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8614–8618. IEEE, 2013.

[109] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[110] Shigeo Sato, Ken Nemoto, Shunsuke Akimoto, Mitsunaga Kinjo, and Koji Nakajima. Implementation of a new neurochip using stochastic logic. *IEEE Transactions on Neural Networks*, 14(5):1122–1127, 2003.

[111] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the International Symposium on Computer Architecture*, 2016.

[112] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[113] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[114] GEORGE VALENTIN STOICA, RADU DOGARU, and CE Stoica. High performance cuda based cnn image processor, 2015.

[115] Evangelos Stromatias, Daniel Neil, Francesco Galluppi, Michael Pfeiffer, Shih-Chii Liu, and Steve Furber. Scalable energy-efficient, low-latency implementations of trained spiking deep belief networks on spinnaker. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2015.

[116] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 453(7191):80–83, 2008.

[117] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[118] Masakazu Tanomoto, Shinya Takamaeda-Yamazaki, Jun Yao, and Yasuhiko Nakashima. A cgra-based approach for accelerating convolutional neural networks. In *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*, pages 73–80. IEEE, 2015.

[119] S Thoziyoor, N Muralimanohar, JH Ahn, and N Jouppi. Cacti 5.3. *HP Laboratories, Palo Alto, CA*, 2008.

[120] Bill Tong. Linguistic features of the chinese language family. *Available: http://www.oakton.edu/user/4/billtong/chinaclass/Language/linguistics.htm*.

[121] SL Toral, JM Quero, and LG Franquelo. Stochastic pulse coded arithmetic. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 1, pages 599–602. IEEE, 2000.

[122] Kristina Toutanova, Dan Klein, Christopher Manning, William Morgan, Anna Rafferty, Michel Galley, and John Bauer. Stanford log-linear part-of-speech tagger, 2000.

[123] Kristina Toutanova and Christopher D Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the 2000 Joint SIGDAT conference on Empirical methods in natural language processing and very large corpora: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 13*, pages 63–70. Association for Computational Linguistics, 2000.

[124] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, pages 11–20. ACM, 2018.

[125] Shuo Wang, Yun Liang, and Wei Zhang. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *DAC*, 2017.

[126] Yanzhi Wang, Caiwen Ding, Geng Yuan, Siyu Liao, Zhe Li, Xiaolong Ma, Bo Yuan, Xuehai Qian, Jian Tang, Qinru Qiu, and Xue Lin. Towards ultra-high performance and energy efficiency of deep learning systems: an algorithm-hardware co-optimization framework. In *AAAI*, 2018.

[127] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *DAC*, 2017.

[128] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[129] Fei Xia. The part-of-speech tagging guidelines for the penn chinese treebank (3.0). *IRCS Technical Reports Series*, page 38, 2000.

[130] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Xiling Yin, Wenqin Huangfu, Pai-Yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, and Huazhong Yang. Mnsim: Simulation platform for memristor-based neuromorphic computing system. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 469–474. IEEE, 2016.

[131] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *DAC*, 2017.

[132] Fan Yang, Qinru Qiu, Morgan Bishop, and Qing Wu. Tag-assisted sentence confabulation for intelligent text recognition. In *Computational Intelligence for Security and Defence Applications (CISDA), 2012 IEEE Symposium on*, pages 1–7. IEEE, 2012.

[133] Zheng Rong Yang and Mark Zwolinski. Mutual information theory for adaptive mixture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(4):396–403, 2001.

[134] Bo Yuan, Yanzhi Wang, and Zhongfeng Wang. Area-efficient error-resilient discrete fourier transformation design using stochastic computing. In *Proceedings of the 26th edition on Great Lakes Symposium on VLSI*, pages 33–38. ACM, 2016.

[135] Bo Yuan, Chuan Zhang, and Zhongfeng Wang. Design space exploration for hardware-efficient stochastic computing: A case study on discrete cosine transformation. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 6555–6559. IEEE, 2016.

[136] Zihao Yuan, Ji Li, Zhe Li, Caiwen Ding, Ao Ren, Bo Yuan, Qinru Qiu, Jeffrey Draper, and Yanzhi Wang. Softmax regression design for stochastic computing based deep convolutional neural networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 467–470. ACM, 2017.

[137] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[138] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional

neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[139] Tianyun Zhang, Shaokai Ye, Kaiqi Zhang, Jian Tang, Wujie Wen, Makan Fardad, and Yanzhi Wang. A systematic dnn weight pruning framework using alternating direction method of multipliers. *arXiv preprint arXiv:1804.03294*, 2018.

[140] Liang Zhao, Siyu Liao, Yanzhi Wang, Zhe Li, Jian Tang, and Bo Yuan. Theoretical properties for neural networks with weight matrices of low displacement rank. In *International Conference on Machine Learning*, pages 4082–4090, 2017.

VITA

NAME OF AUTHOR: Zhe Li

PLACE OF BIRTH: Changchun, Jilin, China

DATE OF BIRTH: April 16, 1989

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

- M.Sc. 2014, Syracuse University, US
- B.Sc. 2012, Beijing University of Posts and Telecommunications, China

PROFESSIONAL EXPERIENCE:

- Software Engineer Intern in Research, 2018, Google, Mountain View, US
- Software Engineer Intern in Research, 2017, Google, Sunnyvale, US
- Research Assistant, 2014 ~ 2018, Syracuse University, Syracuse, US
- Software Engineer Intern, 2013, Software Center, Bank of China, Beijing, China
- Software Engineer Intern, 2012, China Unicom, Changchun, China